Real-Time RISC Processing

Chip Weems Steve Dropsho

Computer Science Department
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003

April 25, 1995

Traditional and Modern Real-Time Processing Systems

In real-time computing the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced [1]. That is, the completion of the computation has a deadline, and failure to meet the deadline may have catastrophic effects. One example of such a real-time application is found in advanced variable-cycle jet engines that can *explode* if correct control inputs are not applied every 20-50 ms. Timing constraints such as these are known as **hard real-time deadlines.** The correct operation of a system with hard deadlines requires that the computations be guaranteed to meet these deadlines under all conditions. In other applications, failure to meet a deadline may simply result in a degradation of performance in the system under control. For example, in an automobile ignition controller, failure to meet deadlines may result in reduced fuel economy and increased pollution, but the engine will continue to operate within a normal range of tolerances. These less critical constraints are known as **soft real-time deadlines**. Many applications involve combinations of hard and soft deadlines. For example, the jet engine controller may also be required to report fuel-flow to a cockpit display -- a non critical function that can miss a deadline by a small amount without affecting safety. Other examples of real-time applications can be found in the control of laboratory experiments, control of electrical power distribution networks, nuclear power plants, process control plants, flight control systems, spacecraft and aircraft avionics, digital telecommunication and robotics.

The traditional approach that is employed in the development of real-time systems is to manually reduce variability of timing by employing processors that have predictable performance and algorithms that are designed to complete with reasonable results prior to their deadlines. Digital signal processors (DSP) are employed in many real-time systems, even those that do not involve signal processing, because they have simple architectures for which it is easy to predict the time required to execute an algorithm. Algorithms are developed that either have known execution times, or the ability to incrementally refine a result in such a manner that they can be terminated after any iteration and still produce a usable result. These algorithms are then carefully assembled into a call graph that is shown to always satisfy the system's deadlines while minimizing processing requirements. The algorithms are then assigned to a predetermined memory map that minimizes memory requirements while preserving the correctness of the system.

The traditional approach results in solutions that are effective but inflexible. For example, if one of the algorithms is enhanced in a manner that alters its timing, then the entire call graph must be re-evaluated and the system may have to be completely reorganized to compensate. Or, if an enhancement changes the memory requirements of an algorithm, a similar redesign must be carried out. Thus, once a real-time system has been completed with the traditional method, it is very difficult and costly to make incremental improvements. The usual practice is to identify all of the desired enhancements and include them in the specification of the next generation of the system.

Further inflexibility is seen in the limitations of this design methodology. The designers must foresee every possible combination of events that require a response by the system and then design the system to handle these. The result is that for applications which operate in a dynamic environment, if an unforeseen combination of events should occur, a catastrophic failure may result. There is little ability to respond to the situation in a similarly dynamic manner. Clearly both military and commercial real-time applications would benefit greatly from increased flexibility that allows them to respond more quickly to changing requirements and to operate with fewer constraints.

The modern approach to real-time system development tries to guarantee deadlines by predicting the computation time of individual algorithms and dynamically scheduling them for execution. There is no predetermined call graph or memory map. Instead, the processing is managed by an operating system that takes into account the resource requirements of the algorithms, including execution time and memory, and assigns the available resources as needed to meet the application's deadlines. The system developers are thus freed from the burden of manually assembling the software components, and the system has the

flexibility to allow incremental improvements to individual components with minimal cost. Further, because the operating system responds dynamically to the current set of processing requests, it has the ability to handle unexpected situations gracefully. The system can recognize when it will be impossible to meet certain deadlines and it can make reasonable choices about how to respond to the situation, rather than doggedly pursuing a predetermined course of action that may be inappropriate.

Such flexibility comes the price of greater overhead and system complexity. The operating system requires resources in addition to the basic computations for the application. Thus, system memory and processing power must increase to support this overhead. The architecture of the processor must also support the protection mechanisms that keep application tasks from damaging each other or the operating system, and must support the virtual memory mapping mechanisms that enable the operating system to dynamically allocate space to the application tasks. For some applications it may even be necessary to augment the computational capabilities of the hardware with a coprocessor to accelerate scheduling of application tasks.

Unfortunately, DSPs are not appropriate for modern real-time systems. They do not support the protection and memory mapping mechanisms required by an operating system. They also have complex instruction sets that are difficult to exploit with compilers (many of their features are designed to support specific filtering algorithms that are normally coded in assembly language), and their performance on applications other than specific signal processing algorithms lags substantially behind commodity microprocessors. This disparity in general purpose performance is expected to grow as economic pressures tend to favor lower cost DSPs for high volume markets such as toys, modems, and so on.

In recent years, most developers of modern real-time systems have come to favor the Intel i860 microprocessor. This RISC design somewhat serendipitously supports real-time processing by providing the necessary features to run an operating system while also enabling the user to selectively bypass certain features that would otherwise lead to unacceptable levels of variability in execution time. It also has vector processing features that make it attractive as a replacement for DSPs. The i860 is not perfect, in particular its interrupt structure has been criticized as costly and inflexible, but its users have been able to work around its problems while gaining levels of performance and flexibility that approach commodity microprocessors. Unfortunately, Intel has decided to discontinue the i860 product line and has been unwilling to license the design.

Most of the current i860 users seem to be turning to the PowerPC architecture as a replacement, but this is not entirely satisfactory. The PowerPC, like most other RISC architectures has a complex system of caches, deep pipelines, multiple functional units, and dynamic instruction scheduling that enhance overall performance but make it very difficult to predict execution time. Essentially these designs trade a higher level of variability in execution time for an increase in average performance. In a critical real-time application, however, this variability is unacceptable and it becomes necessary to schedule resources on the basis of worst case assumptions in order to guarantee hard deadlines. Worst case performance can be an order of magnitude lower than the average, so it is possible that 90 percent of system capacity could go unutilized.

What is needed at this time is a real-time enhanced RISC processor that trades back a modest amount of its performance for greatly increased execution time predictability. This can be accomplished by allowing the user to selectively bypass or constrain some of the architectural features that introduce the greatest variability, and the capability to quickly change modes from highest performance to highest predictability. The operating system can thus take greater advantage of the available resources by scheduling soft-deadline or background tasks to execute in high-performance mode and switching to high-predictability mode when executing hard-deadline tasks. In the next section we further describe the architectural enhancements that must be made to a commodity RISC microprocessor in order to make it suitable for modern real-time processing.

Architectural Enhancements for Real-Time Processing

To provide predictability, a real-time processor would ideally have zero variance for any given operation. Previous real-time processors attained a high level of predictability by having a single stage or short instruction pipeline, eliminating caching, and prohibiting the use of virtual addressing (e.g. as is done in DSPs). While this removes the major sources of execution variability, it results in a slow processor that makes it difficult to develop large applications.

New real-time applications are demanding greater processing speed than ever before. To be able to provide that performance real-time processors must incorporate the performance enhancing architectural features common in high speed microprocessors today. However, these features must be added in a manner that makes the benefits temporally deterministic *a priori*. These features include: Translation Lookaside Buffer (TLB), cache memory, memory write buffer, out-of-order instruction execution, branch prediction, deep piplines and multiple functional units. We will address each of these and the changes needed to incorporate them into real-time processing.

Translation Lookaside Buffer

Translation of virtual addresses to physical addresses is a large source of execution time variance. In large software systems virtual addressing is used by the applications to allow the object code to be relocatable. This remapping of addresses gives the operating system the freedom to place an application anywhere in physical memory that is convenient. Virtual memory has proven an indispensable feature in multitasking systems. Without it applications must be 'hard wired' to a particular physical memory location and cannot be placed anywhere else. Without virtual memory, if two applications overlap in their physical address spaces they cannot both be resident in memory at the same time. Also, if a memory page in RAM is detected as faulty and cannot be used, any application using that page cannot be executed. Virtual addressing alleviates these restrictions and allows the important capability of protected memory access.

However, application code must ultimately be placed in physical memory and therefore must have a physical address. To allow applications to use virtual addresses, operating systems maintain mappings of process virtual pages to physical memory pages. On *every* access to memory the virtual address of the access must be translated via the system mapping tables to a physical address.

A typical translation involves the following steps: The translation starts with a virtual address from a memory reference by the processor. The mapping of the virtual address to the physical address is found in the **page map tables**. The tables that map the entire 2^{32} (or 2^{48} for some systems) byte virtual address space are quite large, so to save memory only a fraction of the page map tables are resident in memory at a given time. To support this capability the tables are organized in a hierarchical fashion. A three level design is typical. The root of the tree is kept in a processor register that points to the physical address of the **segment table** that is always resident in memory. The segment table contains pointers, each of which points to a **page table**. A page table contains pointers to the physical address mapping of a virtual page. Unlike the segment table, page tables are not guaranteed to be resident in memory and can be swapped out if not used, to make space available for applications.

The address translation process begins with a memory reference to the segment table via the pointer in the register. The offset into the segment table is determined by a subset of the bits in the virtual address. A second memory access is made to the page table determined from the segment table entry. The offset into the selected page table is determined by another subset of bits in the virtual address. The value returned by the second memory access provides the physical page for the original memory reference. The low order bits of the virtual address are the offset into the physical page for the particular data item. At this point a third and final memory reference can be made for the data item sought.

Assuming all map tables are in memory so no disk IO is required, each memory request by an application requires three memory references and multiple operations to merge bit fields. This occurs reasonably quickly (a few clock cycles) as long as the tables are in cache, but can slow execution by nearly an order of magnitude if any of the references miss in the cache. To reduce the time of the translation process, an Translation Lookaside Buffer (TLB) is usually added that caches the most recently used translations for quick access. When a translation is stored, its virtual address is stored along with it as an identifier. On a memory access from the processor the virtual address is compared to the entries in this cache, usually with a fully associative lookup. If a match is found then the corresponding translated address is used as the physical address of the request, saving two memory fetches. If a match is not found then the procedure described above is followed with the translation result being stored into the TLB for future use. Because the TLB is often a fully associative store, it is expensive in terms of silicon real estate, and thus it is usually quite small (64 entries is typical on current systems). Because of the speed with which the TLB must operate, only simple replacement policies such as random replacement or FIFO replacement can be used. Thus, it is difficult to know in advance whether a reference will be a hit or a miss in the TLB, even when it is to a recently accessed location.

Predicting the behavior of the TLB is further complicated by interrupts. When an interrupt occurs, there is an initial penalty for executing the service routine because none of its references hit in the TLB and all must go through the translation process. However, if the service routine has a reasonable level of spatial locality it soon accelerates by filling the TLB with entries for its own locations. Unfortunately, when the routine returns to the point where the interrupt occurred, the interrupted task is forced to endure the same initial slowdown because it must refill the TLB with its own working set by a series of TLB misses. Thus, the task is slowed down as a side effect of the interrupt -- a side effect that cannot be predicted in advance.

It should be obvious how the probabilistic nature of the TLB adds significant uncertainty to the execution time of an application since delays such as this must be accounted for on every memory reference. For a hard real-time deadline, a worst case of three memory reference times must be assumed for every access because it is nearly impossible to predict whether an entry will be in the TLB when it is needed. An inexpensive solution is to allow software explicit control over loading and invalidating TLB entries. Real-time systems have in-depth knowledge of the run-time conditions and can employ many techniques to speed the loading of the TLB to ensure predictable performance. Explicit control over the TLB should be software controllable so applications that can tolerate variance can simplify their design by letting the hardware automatically manage the contents of the TLB. Thus, for example, the real-time operating system might lock the TLB during interrupts when a hard deadline is approaching. It would thus be able to precisely calculate the time penalty caused by the interrupt and decide whether to service or postpone it.

An alternative that would be fairly costly would be to provide a shadow TLB or a tagged TLB. The shadow TLB would be used by interrupts and the operating system, while the main TLB would be left to the user. It would also be possible to ping-pong the TLBs, with the operating system preloading the shadow TLB with the working set of the next task to execute, just prior to starting it. Thus, the task would begin with the TLB warmed up and not have to suffer the initial slowdown due to TLB misses. A slightly less costly alternative would be to allow certain entries in the TLB to be tagged as belonging to the current working set, and replacement during interrupts would be constrained to entries in the TLB that are not tagged.

Another feature useful for any system is the ability to define variable page sizes. This can significantly reduce the number of TLB entries required by an application and reduce contention for the limited number of entries. Several modern RISC designs already incorporate this feature.

Cache Storage

Caches have the greatest potential benefit for performance and, thus, are the greatest source of variability in execution times. References to memory initiate a lookup in a primary cache that is simply high speed static RAM with an access time of one cycle. An identification tag associated with the virtual address is stored with each data item in the cache. If a match is found between a tag and the virtual address (a **hit**) then the corresponding data item is returned to the processor. If a match is not found (a **miss**) then a physical address is generated via the TLB and the request is sent to the second level cache (if there is one). If there is a hit in the second level cache the data item is returned; otherwise, the request is forwarded to main memory where the information is guaranteed to be found. In the absence of a secondary cache, requests are forwarded directly to main memory on a primary cache miss.

The access times of the different levels in the memory hierarchy often differ by an order of magnitude. For example, in the R4400 based SGI Onyx system, a primary cache access, a secondary cache access, and a main memory access are 1, 10, and 130 cycles, respectively. In a real-time system, this can cause two orders of magnitude difference between the best case analysis and worst case.

The solution is to allow explicit control over loading and invalidating entries in the cache. A real-time compiler can use this control to generate code for explicitly preloading a complete application or a portion so cache hits and misses can be calculated a priori to run-time. The cache might also have an additional tag field that would enable the operating system to partition it into areas that are under explicit control (hard deadline tasks) and those that are more tolerant of the variability found in normal cache operation (soft deadline tasks). Another useful addition is to have a programmable field in each TLB entry that can be set to trap on a cache miss, to trap on a cache hit, or be ignored. Incorporating the flag into the TLB lets a separate policy be set for each page. This would be used as a debug feature in the development of real-time applications for detecting unexpected cache activity.

Write Buffer

A write buffer is temporary storage for processor stores to memory. Most processors support the **weakly ordered memory model**. In this model the processor can dynamically reorder memory references (loads and stores) to improve performance. Since processors must stall on loads but not stores, stores are often buffered at the bus interface and loads issued after any pending stores are prioritized in accessing memory. This minimizes the time the processor is stalled.

The situation of pending stores can arise when a series of stores are quickly issued. The lower bandwidth of the memory bus compared to the internal bandwidth of the processor can result in a backup at the interface. This can cause timing variability in both loads and stores. Under these conditions a load will be prioritized over the pending stores, but must wait for any store currently in progress. Variability in stores can arise if the rate of issued stores is so great that the write buffer becomes full and must stall the pipeline until an entry empties. Note that stores which hit in the primary cache under a write back policy are held in cache and are not forwarded to the memory bus interface so do not contribute to this problem.

The performance benefit of the write buffer can be especially significant in applications with large data sets where accesses frequently miss in cache (e.g., machine vision). However, the potential for variability in execution time is also quite high. The solution is to provide an explicit control mechanism to empty the write buffer. In conjunction with the features for controlling caches, an intelligent real-time compiler has enough information to calculate the state of the write buffer prior to each store or load in many cases. Points where the information is not clear are entry points to blocks that can be entered from multiple locations (e.g., at the beginning of a function or at the end of a multiway branch construct). In these cases two options are available: (1) the worst case entry path can be assumed (that which burdens the write buffer the most) and an appropriate delay inserted such that at the end of the delay the write buffer is empty; (2) before jumping to an entry point the compiler can insert a delay that allows the write buffer to

empty. The second option reduces the delay to a minimum but adds code in multiple locations. Adding an instruction to the instruction set (often called a **sync** instruction) that forces the write buffer to empty would reduce the delay code to a single instruction.

Another possible approach would be a programmable status bit that forced processor stalls on writes that missed in primary cache. This inserts an implicit sync instruction after every store. The operating system could switch to sync mode for tasks with hard deadlines and enable opportunistic use of the write buffer for tasks with soft deadlines. Current processors only offer the explicit sync instruction.

Out-of-Order Instruction Execution

In a processor with multiple functional units, a significant performance improvement can be attained if instructions are allowed to execute in an order different than that in which they are issued. This technique is commonly used in today's superscalar processors that can issue two, three, four, or more instructions simultaneously. In [2], Smith et al. show that out-of-order execution can boost performance by over 20%.

A simple and common example of out-of-order execution is the issue of an integer multiply followed by integer addition operations on a machine with two integer units. In today's pipelined general purpose processors multiplication takes multiple cycles while addition requires only one; for example the PowerPC 604 requires four cycles for multiply and one for add. During the four cycles for the multiplication, multiple additions can be completed and their results written to result registers *before the prior multiplication has finished*. Clearly, by allowing out-of-order execution the multiple execution units can be better utilized resulting in a significant performance boost for this set of operations. The problem with out-of-order execution is when exceptions are encountered. An exception is any branch that is not foreseen. Current RISC processors encounter a branch roughly once in every six instructions. Branch prediction, described below, can successfully foresee roughly 90 percent of these branches, so exceptions are encountered approximately every 60 instruction cycles.

To simplify the software model the programmer sees, precise exceptions are maintained. In this model, when an exception occurs the programmer is guaranteed that all instructions that would have been issued by an **in-order execution** processor have successfully completed and have modified the machine state accordingly. This has proven to be a very useful model in the programming community and is important to preserve. The programmer is also guaranteed that the machine state has not been affected by any instructions that would be issued by an in-order processor *after* the instruction experiencing the exception. The effects of the exception on the machine state have been precisely narrowed to a single instruction. The machine state includes among other things the program counter, condition codes, and the registers.

When an exception arises, an out-of-order processor must undo the effects of any instructions that followed the instruction experiencing the exception but finished in advance of it. While processors maintain additional state information to perform this undo operation, designers optimize the process so the amount of time for the undo is dependent on how much needs to be undone. With these optimizations it is possible to shave cycles off the average time required to restore the previous state, but they add variance to the cost of handling an exception.

To enhance the predictability of handling exceptions this variance must be removed. The solution is to ensure that the undo operation requires a fixed amount of time under all conditions, which can be easily added to any processor design by inserting delay cycles when an undo requires less than the maximum time. This feature is also one that can be implemented as an option to be selected dynamically through software. Thus, tasks that can tolerate variance can opt for potential higher performance at the cost of less predictable bounds on execution time.

Branch Prediction

To increase clock rates today's processors are heavily pipelined. That is, the work to process an instruction is broken down into many short but fast stages. In an assembly line fashion, as an instruction leaves one stage another enters. The key to high performance is that during any given cycle many instructions are being processed simultaneously in this fashion.

Conditional branches represent points of control flow change that are data dependent, and can therefore cause 'bubbles' in the pipeline because the next instruction to fetch depends on the result of the conditional test which is data dependent. To always correctly fetch the next instruction, the CPU must delay issue of the next instruction until the branch instruction has progressed to the stage in the pipeline where the result of the test becomes known. Since branch instructions are frequent (up to 17% in some applications [3]) inserting bubbles into the pipeline after each branch significantly impacts performance.

A common method to minimize the impact of branches is to add hardware for predicting the most likely result of the test and immediately issue the appropriate instruction. The prediction is checked against the actual result when it becomes available. If the prediction was correct then no corrective action needs to be taken and no pipeline efficiency has been lost. If the prediction was incorrect then the processor cancels the incorrectly issued instructions by clearing them from the pipeline. It then fetches the correct instruction and starts it through the pipelines. The canceled instructions result in a **bubble** in the pipeline. The number of canceled instructions due to the incorrect branch is called the branch penalty.

For example, in the MIPS R4400 the branch penalty is potentially two instructions because it requires two cycles to correct the branch and it issues only one instruction per cycle. The PowerPC 604, on the other hand, needs only a single cycle to correct a branch but can issue up to 4 instructions per cycle (to its multiple functional units) for a potential penalty of four instructions. The effective penalties seen by non-real-time codes are much smaller because current techniques for branch prediction have a success rate up to 90%, so the penalty is only paid on 10% of the branches.

The probabilistic nature of branch prediction causes the variability of branch instructions to be high. In the most optimistic case all predictions would be correct so every branch instruction would require only one cycle. However, in the most pessimistic case all predictions would be wrong so the branch penalty would be paid on every conditional branch.

Since real-time systems do analysis based on worst case execution times, minimizing the variability of conditional branches is important. However, the frequency of branches requires that the cost of each branch be small for performance reasons. There are a number of potential solutions. The first is to reduce the branch penalty to zero by using a Y-pipe design [4]. This is a hardware intensive solution that duplicates the instruction fetch logic to allow both paths of a branch be followed simultaneously. Depending on the result of the branch test, the execution unit selects its instructions from one fetch unit or the other.

A second solution is to always delay the issue of instructions following the branch until the test result is known. This avoids execution time variability, but impacts performance -- effectively eliminating the benefits of branch prediction. This impact can be mitigated if the delay is a feature that can be quickly enabled or disabled via software. Then the penalty does not have to be paid in some special, but common cases. The PowerPC 604 has capabilities that demonstrate how this can be exploited.

The PowerPC 604 allows the compiler to provide hints on how to make predictions. This feature can be used to advantage by a real-time compiler for analyzing the timing of loops in the code. Having a data independent prediction policy and given a constant number of iterations (N), the number of correct predictions for the branch at the end of the loop is known (N-1) as is the number of incorrect predictions (one, at end of iteration count). The effective total penalty will be one branch penalty amortized over N

branches. By making the delay dynamically programmable, such loops can enjoy both zero variance and effective low branch penalties.

Other Real-Time Support Features

There are additional features for real-time support that are orthogonal to the base design and can be added easily for selective use via software control. These features include high speed timers for accurately measuring small delays. These should have single cycle resolution and be accessible with low latency. Boundary timer/counters are desirable so that approaching deadlines can be signaled. Inter processor synchronization support will aid in lowering the skew between coordinating processors which will add to better predictions of interaction timings.

Low latency access to a set of synchronous coprocessors should be available to support acceleration of schedule computation, vector processing, real-time I/O, inter processor communication, etc.. If the system bus latency is sufficiently small then attaching coprocessors to the bus and supporting multiprocessing may be sufficient. Because modern real-time systems involve significant levels of task switching, interrupt handling should be as fast a possible and its timing should be controllable via software to tailor interrupt routines to individual circumstances.

Comparison of DSP to General Purpose RISC Processors

Traditionally, digital signal processors (DSP) and general purpose processors are used in very different domains. DSP chips are tailored for tasks that require low-power, low-cost, and high-speed numerical computation on well understood algorithms such as the fast fourier transform (FFT) or the infinite impulse response (IIR) filter. The general purpose processors consume more power, and their inherent flexibility correlates with increased logic, higher cost and lower performance on DSP algorithms.

Today, the differences between the two types of processors are diminishing. This is chiefly due to the general purpose RISC processor manufacturers including many DSP-like features. Smith [5] provides a thorough comparison between the two types of processors and posits that current RISC designs may be more DSP-like than DSP is RISC-like. His perspective is from the DSP application world and how well RISC fits in. He does not address how well DSP addresses general purpose application needs. In his conclusion he suggests that with evolution RISC may displace DSP for the traditional DSP applications. The following sections provide a comparison of the two classes of architecture. While much of the information is from Smith's paper, there are additions.

DSP Characteristic Both RISC and DSP Share

Fast floating point. A trademark of the DSPs has been that they perform floating point operations quickly with their on-chip floating point hardware. This is a distinct advantage over earlier processors that relied on off-chip floating point units. However, current RISC processors all have floating point units on-chip giving them the same base performance as DSPs. The RISCs have another advantage in that their floating point units are pipelined and can retire a floating point operation every clock cycle. Being more heavily pipelined allows RISCs to run at a higher clock frequency than DSPs which traditionally do not have pipelined floating point units. However, RISCs still trail DSP performance in division.

Sufficient Precision. Single precision floating point is sufficient for most DSP applications, however, almost all RISC processors provide double precision capability in addition to single precision. In general, RISC provides higher precision than DSP, but at a higher cost due to the wider data paths and the additional internal logic.

Low loop overhead. Because many loops in DSP applications are contained in well-known algorithms and are extremely small, DSP instruction sets provide **zero-overhead loop control**. Presetting a loop counter and branch test lets the loop control operations be executed in parallel with the arithmetic instructions and avoids repeated fetches. The overall effect is that the overhead to execute the branch is completely hidden. Unfortunately, it is difficult for compilers to take advantage of these specialized instructions, and they are mainly used in hand-coded library subroutines. RISCs now incorporate this basic functionality, but in the reduced instruction spirit, they do not provide separate instructions. Instead, superscalar designs (e.g., PowerPC) that can issue multiple instructions per cycle hide the branch overhead by issuing branches in parallel with other operations. In addition, on small loops the entire loop is likely to fit in the instruction cache so instruction fetches will always hit and thus not use any data cache or main memory bandwidth.

Multiply and accumulate instruction. The multiply and accumulate (MAC) instruction is extremely useful in DSP applications (e.g., forming the dot product). Both DSPs and RISCs offer this functionality, but DSPs offer additional variations that RISCs do not. The variations find uses in the FFT.

Harvard Architecture. DSP designs have traditionally separated the instruction and data memory onto separate buses to enable the processor to fetch an instruction at the same time that it is fetching an operand. Recent RISC implementations have almost universally adopted this sort of scheme in their primary caches (and some in even the secondary cache level, e.g., HP PA7100). In a move toward the more general purpose, recent DSP designs have incorporated on-chip instruction and data buffers (explicitly managed caches) with unified access to external data and instruction memory (and the option to split them if desired).

Low Power. In DSP environments, power use is often a concern. To address this, DSPs have on-chip serial ports and standby power modes to reduce overall operating power consumption. This is traditionally extra hardware for RISC, but power management circuitry is currently being incorporated into designs such as the PowerPC 603 for embedded systems, and in other RISC processors.

Low overhead for address calculations. DSPs often use direct memory access (DMA) units that make memory block moves in parallel with other CPU operations. RISCs, on the other hand, exploit their superscalar designs to calculate addresses in parallel with other operations. On block moves the DSP designs have potential to be more efficient than RISC if other work is available to be done in parallel with the move, while RISCs will be more efficient on small data moves where there is too little time to start other work in parallel on the DSP processor. However, some RISC designs such as the PowerPC and new SPARC architectures also support block transfer operations.

DSP Characteristics RISC Does Not Have

Fast division. RISC designs have not allocated a high proportion of resources to division hardware. In DSPs, division is supported with intensive hardware implementations that result in a small number of cycles, e.g., 1 or 2. However, to reduce costs RISCs use iterative algorithms that typically require 15 to 31 cycles. RISC designs in the future should bring division times in line with DSP processors.

Bit Reverse Addressing. RISCs are not tuned to the peculiarities of some of the DSP algorithms. The FFT is a ubiquitous operation in DSP applications and requires an addressing mode that is most efficiently done by bit operations on the address, such as bit reversing (%11001001 -> %10010011). A RISC processor must emulate this function with multiple instructions. On the other hand, it is difficult for a compiler to take advantage of this highly specialized instruction and so it is primarily used in hand-coded library routines in the DSP application domain.

Distributed Memory Multiprocessing. RISC designs do not include communication ports for easily assembling distributed memory systems. Current high-end DSP designs include up to 6 byte-wide ports, although most merely have one serial port (used in modem applications).

Cost. DSPs are more cost effective than general purpose designs simply due to their being designed for embedded applications with a minimum of support circuitry. Additional functionality increases DSP cost proportionately, as seen with the high-end TMS320C40, which was introduced at a price comparable to contemporary microprocessors. Meanwhile, the entry of RISC designs such as the PowerPC into the personal computer market will increase their volume and decrease their costs to be competitive with DSPs (this shift in production volume is exemplified by the fact that within three months of introducing PowerPC based Macintosh products, Apple became the largest supplier of RISC processors in the world).

RISC Characteristics DSP Does Not Have

Virtual Memory Support. DSP designs ignore virtual memory management. Virtual memory simplifies multitasking and process protection. Process protection is necessary in complex systems to ensure that the integrity of each process in the system is protected from being unexpectedly affected by other processes. This is especially important for critical processes in a real--time system[6].

Shared Memory Multiprocessing. Unlike RISC designs which include cache coherency hardware to automatically manage shared memory multiprocessing over a system bus, DSP designs require software to explicitly manage memory so multiprocessor designs do not accidentally corrupt data. Thus, considerably more overhead must be consumed in a shared memory configuration. Without virtual memory support, sharing must often be implemented via fixed partitions of memory and more of a message-passing style of interaction.

Caching. RISC processors today have on-chip caches of 4KB to 32KB each for instructions and data. In addition, most support secondary caches of up to many megabytes in size. The allocation of space in these caches is done dynamically so that high levels of utilization can be obtained. The caches contribute greatly to the high levels of performance that are obtained with RISC designs. DSP designs do not support caching although they have traditionally supported buffers on the chip that are explicitly loaded with instructions or data -- often so that the chip may run a particular algorithm without any references to external memory. These buffers must have their space painstakingly assigned as part of the application development process, and their utilization is fixed at that time.

Superscalar and Superpipeline Acceleration. Current RISC designs typically have two integer ALUs, one floating point ALU, a Load/Store unit, and a branch unit. It is possible for all of these to be active simultaneously. In addition it is usually the case that the ALUs support simultaneous multiplication and addition operations. This parallelism is further enhanced by long pipelines (6 to 9 stages are typical), so that many instructions are in the process of executing at any moment. Thus, it is possible for a RISC processor to operate at a higher clock rate (60 to 250 MHz is the current range) and to achieve instruction issue rates that are several times the clock rate. DSPs usually have no parallelism other than being able to multiply and add at the same time through explicit instructions, and they have short (e.g. 4 stage) pipelines that limit clock rates to the range of 25 to 75 MHz in most cases.

Recommended Changes to RISC to Support DSP Applications Efficiently

The changes needed are relatively minor. Division hardware should be enhanced and bit addressing should be added if it is not already supported. The bit addressing capabilities can have significant impact on certain DSP applications. To put the overall performance impact of these changes in perspective, in [5] Smith actually compares execution times of common DSP tasks on both DSPs and RISCs. Even without these features the RISC designs *out-performed* the DSP designs, significantly in some cases. The DSP designs used were TI's C25/C30 DSP series (for integer and floating point, respectively) and Motorola's DSP56000/DSP96002. Frequencies ranged from 33 to 50 MHz. The RISC designs that performed best were the i860 and AMD 29050, both at 40 MHz. The benefits came from the large number of internal *general purpose* registers and the pipelining of floating point instructions.

Adding ports for distributed memory multiprocessing should not be a high priority. The same functionality can be included with a separate coprocessor chip that communicates through the system bus. This method allows system designers to customize the network control and make tradeoffs, such as the number of ports to bandwidth per port. The cost may actually be less than that of a single-chip design. Depending on the sizes, two smaller chips can be fabricated for less than one large chip. However, if communication ports are added on-chip the design issues should not be much of an issue because the CPU pipeline design is orthogonal to the communication port control logic.

The other property of DSP processors that must be added to a RISC design is its predictable timing. In a real-time application, being able to guarantee the computation time of an algorithm is often more important than obtaining the maximum performance. DSP designs are straightforward and execute in a predictable deterministic manner. RISC processors trade predictability for increased average performance with a higher degree of variability. If RISC processors are to be used in the same sorts of real-time applications as DSPs, then it must be possible to selectively constrain these mechanisms in order to obtain a desired level of predictability.

Recommended Changes to DSP to Support RISC Applications Efficiently

The changes to DSP for RISC applications are more involved. The major issue is lack of support for virtual addressing which enables process protection. Unfortunately, virtual memory support requires address translation on every memory access. This implies a fundamental change in the pipeline of DSPs. The solution will incorporate a mechanism like RISC's translation lookaside buffer (TLB) to cache translations to minimize the performance impact on accesses.

The surprising results by Smith that some RISCs out-performed DSPs impresses the point that pipelining floating point operations and having a large general purpose register set are important for high performance and a natural path of evolution for DSPs. As the pipeline gets longer, DSPs will require more sophisticated branch management hardware to maintain a high utilization on non-DSP codes where control flow is much less deterministic.

The purpose of pipelining floating point operations is to increase the clock frequency and, equivalently, the rate at which operations can be completed. But increasing the clock frequency will increase the disparity between DSP CPU speed and off-chip main memory. To maintain a high utilization of the pipeline cache memory must be incorporated. (Most DSPs today have on-chip memory that is *not* cache. The difference is that the on-chip memory must be explicitly read from and written to as separate locations from main memory while cache is a logically transparent device to the user between the CPU and memory.) In addition, writes to main memory will take longer so write buffers will be added to decouple the CPU from writes.

Once decoupled from main memory times by the addition of a cache, DSP can try to match or exceed the performance levels of RISC by increasing the clock frequency and/or the number of instructions issued, as in RISC processors. Even so, the directions of the designs will differ somewhat. Where RISC emphasizes "general" applications but has support to run DSP applications fairly efficiently, DSP will emphasize the latter and somewhat efficiently support the former. An example of a difference will be that DSP processors require main memory access latencies to be as short as possible since some DSP applications do not have very high data cache hit rates.

An additional characteristic that should be noted, although it reflects a personal bias of the authors, is that DSP instruction sets are much more complex than RISC instruction sets. This imposes a greater burden on the compiler to recognize instruction sequences that map to the more complex, but more efficient instructions. These specialized instructions have traditionally been an area where compilers have not been successful. In fact, current use of DSPs is largely based on libraries of hand-coded routines to perform common complex operations such as the FFT. To be fair, compilers for DSP-enhanced RISCs will have a similar difficulty but to a lesser degree. The RISCs will benefit from the simpler instruction set

that performs complex DSP operations with multiple instructions and relies on sophisticated instruction issue hardware to maximize performance.

Recommendation on Developing a Real-time Processor

The changes to RISC processors are fairly straight forward and do not impact much of the infrastructure of existing pipelines. The changes appear to be incremental to a RISC pipeline. The effort to modify the DSP appears to be considerably higher. Even the necessary initial step to add virtual address support directly impacts the basic flow of the pipeline. And, to maintain even reasonable performance on general purpose codes additional pipelining of the functional units and a superscalar approach are likely to be necessary. This requires drastic changes to a DSP pipeline and entails considerable work. As an analogy, this is like the difference between turning a race car into a street-legal vehicle by adding the necessary safety and pollution equipment vs. trying to turn a family sedan into a race car, which entails a complete redesign.

Both vision and real--time applications have general purpose and DSP processing. Having to rely on a single type of processor suggests that a RISC be selected. It will be superior on general purpose applications and adequate on DSP applications (and potentially superior here, too). It is not clear that the some DSP changes to the RISC (division, bit addressing) are a high priority. Current RISC performance may be adequate, but the enhancements do suggest areas for improvement. Increasing predictability, however, is a necessity for real-time use.

Almost any modern RISC processor would be a good starting point. However, the RISC processors that best fit the above criteria are superscalar and include the i860, M88110, HP PA7200, MIPS, DEC Alpha, or PowerPC. Unfortunately, both the i860 and the M88110 are no longer supported. The MIPS superscalar R10000 and the Alpha 21164 are very expensive, but the earlier versions R4000 and 21064 might be cost effective. The HP PA7200 requires an external high speed SRAM primary cache which increases the cost of an overall system. However, as VLSI technology improves we expect that HP will bring the primary cache on-chip. In addition, HP has licensed the PA-RISC architecture to Intel, and it is expected that a version of it will be incorporated into some future Pentium successor as a coprocessor.

In advance of the microprocessor survey summary in the next section, we will state that the PowerPC family is an attractive choice. Its main advantages over other processor lines are a physically addressed primary cache, low-power mode, and relatively inexpensive fabrication costs. The physically addressed primary cache allows inexpensive shared bus multiprocessor systems to be constructed. Other RISCs with virtual address caches require translation logic between the cache and the bus. Physical addresses on the bus must check the data block in the virtual address cache (this is called *snooping* on the cache), so the physical address must be translated to a virtual address. This is necessary to maintain cache coherency (i.e., data copy consistency). The PowerPC family does not require this additional logic. Also, the PowerPC 603 has an optional low-power mode so it can be used in systems where power consumption is a concern. Finally, the die of the 603 is relatively small so it can be fabricated less expensively than many of the current high performance processors. The 604, on the other hand, delivers higher performance without going to the power consumption and interface complexity extremes of the PowerPC 620 and DEC Alpha. We have also heard rumors of a 603+ that can switch between a low-power 603 mode and a high performance 604 mode, but we have not been able to obtain any specifics on this design.

One other processor that should be examined, but which we have only been able to obtain high level information about, is the Texas Instruments UltraSparc. However, the Sparc architecture has been steadily falling behind the performance of other RISC architectures and our concern is that even the UltraSparc may not return it to a competitive level. In addition, the UltraSparc is based on version 9 of the Sparc architecture, which is a 64-bit design and thus has a more costly interface and higher precision than is required by most DSP and real-time applications.

Microprocessor Survey Summary

After surveying the field of both general purpose RISC processors and the family of digital signal processors we conclude that the PowerPC family would require the least effort to add the proposed real-time features and each generation is consistently one of the lower cost processors for its performance. Ranked second is the HP PA7xC family that provides many of the features but lags the PowerPC in SPEC performance and cost (it currently requires primary cache in external SRAM and its virtually addressed cache makes multiprocessing more expensive). In third, fourth, and fifth place are the Alpha 21164, UltraSparc, and MIPS T5 (R10000); their precise order is arguable as to which will require the least effort to modify. Making cost the criterion the order settles to UltraSparc, MIPS T5, Alpha 21164. The DSP processors are ranked last due to the major pipeline changes necessary for them to compete favorably in real-time systems that have general purpose processing needs. A brief summary of each processor's interesting features follows.

PowerPC. The newest PowerPC, the 620, is a 64-bit RISC processor with a short 6 stage pipeline and out-of-order execution, a 133 MHz CPU core, a 64 KB on-chip cache, and ratings of 225 SPECint92 and 300 SPECfp92[8]. It can issue up to four instructions per cycle and includes power management features to trim consumption. There are features to hide branch overhead and give limited control of the cache and TLB to the user. The physically addressed primary caches allow low cost shared memory multiprocessing. The PowerPC line has partial implementations of many of the features for a real-time processor and may only need minor adjustments to complete most of the functionality. The 604 [9] is essentially a lower cost 32-bit version of the 620 sharing the same features and may be the better introduction product for real-time processor enhancements. It is also important to note that low-power versions of every processor in the PowerPC family are planned. Currently the only low-power processor in the family is the 603, which is a much simpler design than either the 604 or the 620 -- a fact that may make it attractive as a first target for real-time enhancement.

HP PA71C. The PA71C[12] has a two instruction issue pipeline with a 100 MHz frequency and achieves a SPECint92 of 109 and a SPECfp92 of 168. The interesting features of the PA71C are the capabilities to manually manipulate cache and TLB entries, the low latency floating point division and square root (8 and 15 cycles respectively), multiply/accumulate instructions, pixel operations, and a large virtually addressed *external* primary cache (up to 2 MB). The HP72C will run at 140 MHz with ratings of SPECint92 175 and SPECfp92 250. Like IBM/Motorola with the PowerPC, HP also develops low cost versions of their designs. Were it not for the external cache requirements, this processor might be a first choice for real-time modifications.

Sun UltraSparc. The Sun UltraSparc is a 64-bit RISC processor that emphasizes pixel operations for multimedia applications[7]. The processor has a long 9 stage pipeline, will initially run at 200 MHz, issue up to four instructions per cycle with out-of-order execution, and have a SPECint92 rating of 250-300. It's unique features are a set of pixel operations that can do arithmetic operations on up to 8 pixels simultaneously, high speed block move instructions, and register windows for fast context switching. This design improves the overhead in handling traps by automatically switching to a fresh register set. However, this feature is not desirable in a real-time environment because it can add variability when all register sets are in use and one must be freed by writing its contents to memory. Explicit control of the windowing mechanism must be made available to software, and it is likely that the manipulation of the register windows will be more complicated than managing a basic register set. Floating point divide times are data dependent in the UltraSparc, thus adding variability. The reference did not provide information as to whether the new SPARC-V9 revisions to the SPARC architecture provide for explicit control of the TLB, cache, or register windows. We expect that they do not. In addition, V9 is a 64-bit version of the SPARC architecture, which requires a more costly bus interface and delivers a level of precision that is not usually required in real-time applications. However, the 64-bit extension is a fundamental part of the specialized pixel-vector operations.

MIPS T5. The MIPS T5 is a single chip version of the MIPS R8000 multichip module that has a better balance of integer and floating point performance[10]. The initial core speed will be 200 MHz and its (effectively) four instruction issue superscalar design can execute instructions out-of-order to deliver 250 SPECint92 and 350 SPECfp92. The 64 KB of primary cache equally split between the instruction cache and data cache, both two-way set associative, match the PowerPC 620 cache size. The division and square root timings are data dependent. The reference estimates the cost of the chip will be $1000 to $1200. This is expensive compared to the PowerPC series that is positioned to sell well below that. In addition, without falling back to the current R4000 family, there is no option for a 32-bit, low-power, low-cost implementation of the T5. Silicon Graphics is focusing primarily on the workstation market, and thus has not positioned its architecture for lower-cost usage such as in personal computers and embedded applications.

DEC Alpha 21164. The 21164[11] is the highest frequency microprocessor to date. The core clock is 266 MHz, but will be available soon in 300 MHz versions. The four instruction issue, 7 (9) stage integer (floating point) superscalar design boasts a 330 SPECint92 and 500 SPECfp92 rating, making it the fastest processor on the SPEC marks for the near future. Two unique features are the in-order instruction execution that simplifies control (e.g., exception handling) and allows a higher clock frequency and the large on-chip 96 KB second level cache (in addition to 16 KB of primary cache) to decrease the effective memory access time. The Alpha architecture does provide limited capability to manipulate its TLB and cache entries, but only through slow PALcode that traps to special emulation routines. One significant drawback is DEC's pricing policy. At $1865 for 266 MHz or $2669 for 300 MHz DEC is targeting the very high end of the performance market. In addition, the Alpha has extremely high power consumption and unusual cooling requirements. Its interface is also very wide and expensive, as one would expect in order to support this level of performance.

Digital Signal Processing Designs. The digital signal processing (DSP) processors such as the SHARC and TI's C40 are fundamentally different from the general purpose RISC processor family. The distinguishing feature of the DSP family is a streamlined design for low cost, but high performance numerical processing. Thus, they feature very low latency floating point operations including division and square root, a complex instruction set for minimizing branch overhead in tight computational loops, and no virtual memory support since they are usually used in small special purpose systems. For dedicated real-time systems DSP processors have many useful qualities: low latency memory access for streaming data, explicit control of on-chip memory for high speed program execution, timers to generate internal interrupts, and hardware support to handle arithmetic exceptions. And, their target market demands low power and very low cost.

Unfortunately, the streamlining of the DSP to minimize cost makes the processor unsuited for complex real-time environments. Modern real-time environments require virtual memory support (e.g., segment and page tables and a TLB) to simplify the task management of the OS and to provide process protection. In addition, the focus on consumer products with their low margins inhibits performance enhancements that parallel the RISC processors. As a result, the general processors have a significant performance edge (at a higher cost, of course). For DSP processors to attempt to compete in the general real-time application domain architectural features that are common in the RISC designs will have to be incorporated: superscalar pipelines, caches, TLBs, and more sophisticated branching hardware. These are all significant changes to the DSP pipelines.

To summarize, a clear trend for enhancing performance in the RISC designs has emerged. All designs are superscalar (in fact, excluding HP all can issue up to four instructions per cycle), most support out-of-order execution, and all have large caches and sophisticated branching hardware. Thus, a high performance real-time processor must also follow this trend to remain competitive. These requirements preclude DSP processors from being a low cost path to a modern real-time processor. However, real-time systems have both general purpose processing and digital signal processing needs. The RISC PowerPC and HP PA families incorporate a number of features from the DSP families: hidden branch latencies, relatively fast floating point operations, multiply/accumulate functionality, explicit control over on-chip

memory, etc. The processors that will work best should blend the features from both families, however, no processor family has yet to adequately address all the requirements. We feel that modifying either the PowerPC family or the HP PA family will be the least costly path to developing a true high performance real-time processor.

Future Work

While many of the proposed real-time features are easy to incorporate into an existing RISC processor, some are not. Therefore, a study must be undertaken to quantify the benefits from the proposed real-time features so a cost/benefit analysis can be done to justify each feature's inclusion in a RISC design. Specifically, we propose a study to quantify the reduction of the worst case execution times given each of the real-time features individually and in combination. From the results, we will be able to select a set of features that provides maximum benefit at a relatively low cost (i.e., low impact on an existing RISC design).

The purpose of the real-time features is to minimize the difference between the worst case execution time (**WCET**) of an application and the actual time while still providing the performance benefits of architectural features such as caches and branch prediction hardware. The proposed features can be split into two categories. The first category has features that reduce variability within a real-time process. The second category reduces variability in system activity between processes. The features in the first category are explicit control of the TLB, cache, and write buffers and hardware for branching support. The second category includes interrupt handling, high speed timers, boundary timer/counters, inter processor synchronization support, and low latency coprocessor communication. Because the first category of features has the most impact on the WCET of a process and the most potential impact on the processor pipeline design, the proposed study will initially focus on this group.

Scientific Questions

Regarding the features for the TLB, cache, write buffer, and branch prediction, there are many questions to be answered. The questions are of four flavors:

1. What is the best WCET that can be achieved with the real-time feature?

2. How does this compare to the WCET possible if only more intelligent software analysis techniques are used?

3. Given the existence of a real-time feature how does varying the resources it controls affect the WCET?

4. How does the WCET compare to the actual execution time (i.e., what is the variability)?

The questions for the TLB and cache are similar. For the TLB, what is the improvement in WCET if explicit control is possible? How does this compare to the best WCET if only software analysis techniques (used in real-time compilers) are used without the availability of explicit control? Given explicit control of the TLB, what are the effects on the WCET as the TLB size, associativity, or page size are varied? For the cache, the dual of page size is cache line size (16, 32, 64, or 128 bytes).

There are fewer degrees of freedom for questions concerning the write buffer, but they are similar. What is the WCET if the buffer is flushed after every write compared to a minimum number of times determined by analysis? How does this compare to the WCET if only software analysis of the write buffer's affects is used and no explicit control to flush it is available? How does increasing the number of entries in the write buffer affect the WCET?

Branch prediction support to guarantee minimal variance requires a pipeline design similar the Y-Pipe[4] that doubles the instruction fetch units to simultaneously fetch along both possible paths of a branch. This functionality would have significant impact on many of today's RISC pipelines. As a result, we are starting with the hypothesis that the benefits of a Y-Pipe design will not justify the cost. The question to ask then is what is the impact on WCET if a Y-Pipe design is not implemented? Conversely, what is the best WCET that can be achieved if only software analysis techniques are used? By quantifying the benefits we can determine if our hypothesis is correct or not.

Approach

The approach of the study is straight forward and will be based on modifications to well known compiler optimization techniques. We will start with optimized object code output from a commercial compiler for a RISC processor. The machine specific code will be converted into a generic processor-independent RISC assembly code. Next, a program dependence graph (PDG) will be generated that includes data and control flow information of the program.

The PDG forms the basis for several subsequent operations: calculating the WCET of the application, modifying the program via the PDG, or generating generic register transfer level (RTL) code. Operations on the PDG will be iterative. The most common sequence will be to iterate through modifications on the PDG to find a local optimum in the WCET. A final step that writes the PDG out as generic RTL code allows a high level RISC processor simulator to run the program so we can compare the actual execution time to the WCET.

Implementation Steps

There are six components to implement in an environment to answer the posed questions.

1. Generate a simple translator to convert a machine specific RISC code into a generic RISC register transfer level (RTL) code. This is a simple task, but will greatly ease porting the analysis system to another architecture.

2. Develop the program dependence graph generator for the RTL. We will use well known compiler analysis techniques for data and control flow analysis. Our system will be simpler than a compiler's since we are strictly concerned with machine instruction code where high level operations such as array referencing have already been broken down into sequences of simple machine instructions.

3. Develop a timing analysis tool based on a hierarchical design that starts by timing simple blocks, then uses those times to time loops, then procedures, and finally the complete application.

4. Develop algorithms for improving the WCET by modifying the PDG. Operations on the PDG will include aligning code to cache lines to improve locality of reference in simple program blocks (a block is a sequence of instructions that have a single entry point and a single exit point), adding cache and TLB preload code, and relocating procedures to minimize cache conflicts. Clearly, these algorithms will require architectural details as input. This step is the experimental phase and will be the last to be fully implemented.

5. Generate a tool to convert the PDG back into generic RTL code that can be fed into a simulator. This tool will involve a reasonably direct translation process.

6. Develop a high level RISC processor simulator that runs the generic RTL code. For simplicity, we will initially model a single-issue pipelined RISC processor. However, the simulator will model the TLB, cache, and main memory faithfully. Many of the architectural features will be parameterized: TLB size and associativity, cache size and associativity, cache line size, write buffer

size, etc. We anticipate a slowdown factor of about 10,000 to 1 over object code running on a real processor. This will allow applications of over 100 million instructions to be simulated in 8 hours and is of sufficient size to gather results on realistic applications. After this preliminary analysis we will assess whether it is necessary to build a multi-issue simulation and do so if necessary.

Conclusion

The problem to be solved is the design of a high speed, modern real--time processor in a timely fashion with limited resources. Leveraging an existing processor design is a natural conclusion.

Therefore, to minimize the time and cost of development, we recommend pursuing a partnership with a microprocessor manufacturer. We suggest either Motorola/IBM for the PowerPC or Intel/Hewlett-Packard for the PA7xC line. This would involve signing a nondisclosure agreement with them and obtaining models, tools, and a design environment compatible with theirs. We would then proceed to analyze, design, and model (likely at a VHDL level) a real--time version. This would lead to a proof-of-concept prototype to be owned by the manufacturer and hopefully put into production. Another partner could be sought to build a demonstration uniprocessor or multiprocessor system implementation.

The benefits to the research effort are many. Design time and cost are drastically reduced and effort can be focused on the unique features of the real--time processor. Plus, building on an existing processor also makes available a large suite of software applications and debugging tools.

There is also a significant benefit for the manufacturer. The project is a relatively inexpensive path to exploiting the real--time processor market with a unique design for low variability and high performance. This is a growing niche in the real--time system market that has not yet been adequately addressed.

It should be noted that the economic viability of the real--time processor is excellent. Most real--time features will not affect performance or can easily be designed for activation/deactivation via software. This means that the real--time features can be incorporated into every piece of silicon allowing the development costs to be leveraged over sales of all chips, both real--time and non real--time. This is a similar strategy to Intel's which sold two versions of the 80486 chip, in which only one chip design existed however one version had the floating point unit (FPU) turned on and the other did not, thereby allowing the manufacturer to take advantage of chips in which a manufacturing fault occurred in the FPU and increase efective yield.

Of course, compromises may have to be made under such a strategy. Features that significantly impact the base processor design have to be carefully evaluated as to their overall cost and benefit to both real-time systems and general purpose processing, which is why we propose an initial detailed architectural study. Such a study is likely to take between one and two years, with early results allowing VLSI design work to begin at the end of the first year, and first silicon to be released 18 months later. Thus, at the end of three years, a system development partner who has been working from interface specifications, and who already has experience with the base microprocessor, should be able to insert the chip into a working demonstration.

Bibliography

[1] Stankovic, J.A., *A Serious Problem for Next-Generation Systems*. IEEE Computer, 1988. (October 1988).

[2] Smith, J.E. and A.R. Pleszkun, *Implementing Precise Interrupts in Pipelined Processors*. IEEE Transactions on Computers, 1988. **37**(May 1988): p. 562-573.

[3] Hennessy, J.L. and D.A. Patterson, *Computer Architecture A Quantitative Approach*. Second ed. 1990, Palo Alto, CA: Morgan Kaufmann Publishers Inc.

[4] Knieser, M.J. and C.A. Papachristou. *Y-Pipe: A Conditional Branching Scheme Without Pipeline Delays*. in *International Symposium on Microarchitecture*. 1992.

[5] Smith, M.R., *How RISCy Is DSP?* IEEE Micro, 1992. (December): p. 10-23.

[6] Stankovic, J. *The Spring Architecture*. in *EuroMicro Workshop on Real-Time*. 1990.

[7] Wayner, P., *SPARC Strikes Back*. Byte Magazine, 1994. (November): p. 105-112.

[8] Thompson, T. and B. Ryan, *PowerPC 620 Soars*. Byte Magazine, 1994. (November): p. 113-120.

[9] Song, S.P., M. Denman, and J. Chang, *The PowerPC 604 RISC Microprocessor*. IEEE Micro, 1994. (October): p. 8-17.

[10] Halfhill, T.R., *T5: Brute Force*. Byte Magazine, 1994. (November 1994): p. 123-128.

[11] Ryan, B., *Alpha Rides High*. Byte Magazine, 1994. (October): p. 197-198.

[12] Asprey, T., *et al*., *Performance Features of the PA7100 Microprocessor*. IEEE Micro, 1993. **13**(June): p. 22-35.