

Strategies for Minimizing Context Switch Times in  
Large Register Set Environments with  
Primary Focus on the PowerPC® Architecture with  
Floating Point and AltiVec™ Extensions  
[aka SMCSTLRSEPFPAPAE]

by Bill Dittmann  
Quadros Systems, Inc., Chief Engineer

Abstract: A common performance metric for processors used with real time environments is context switch time. While the industry defines "context switch time" in a variety of different ways, it is commonly agreed that that a processor with a larger register set will have a slower context switch than a processor with a smaller register set. When there are more registers to swap, more overhead is involved in switching the processor context from one task to another task. This paper will review the basic context switch problem, discuss methods and strategies that can be used by a real-time operating system (RTOS) to minimize average context switch times in processor environments with large register sets, and finally present how those strategies can be used in the PowerPC architecture with Floating Point and AltiVec extensions.

Introduction.....2  
    *Factors Affecting Context Switching*..... 2  
        Ease of Access ..... 2  
        ISA (Instruction Set Architecture) ..... 3  
        Register Volatility ..... 3  
        Register Access Width ..... 4  
    *PowerPC Architecture* ..... 4  
RTOS Implications ..... 6  
    Floating Point Context ..... 7  
    Reduced Context Strategy ..... 7  
    AltiVec Context ..... 8  
    Register Tracking Strategy ..... 9  
    On Demand Strategy ..... 11  
    Lazy Swap Strategy ..... 11  
    Hybrid Strategy ..... 12  
Summary ..... 12

### **Introduction**

In the 8-bit microcontroller space, processor register sets are small consisting of from 4 to 8 registers. Common microprocessor registers include a status register (SR), a program counter (PC), one or more accumulators, one or more index registers, and a stack pointer (SP). In 16- and 32-bit microprocessors, the register sets consist generally of 8 to 20 registers. In addition to SR, PC, and SP, the register set is usually made up of 8 to 16 general purpose registers (GPRs), 3 to 8 index registers, and 3 to 8 accumulators. More advanced RISC processors like PowerPC and MIPS® processors often have 32 or more GPRs, along with the standard set of PC, SR, and SP.

Classic and more modern DSP processors are made up of groups of more specialized registers such as from one to thirty-two 20-, 24-, 40-, or 48-bit accumulators, up to eight modulo address registers, up to eight offset registers, up to four pairs of registers used for zero overhead hardware loops (typically the start address and loop count value), one or more dedicated loop counter registers, and on-chip hardware stacks made up of from 16 to 256 entries.

With the convergence of many applications that include both control and DSP components, many classic control processors like Motorola ColdFire®, Infineon C16x, Hitachi SH3-DSP, and PowerPC have added hardware support to address digital signal processing problems. Likewise, modern DSP processors have evolved and added better general purpose register support. The result is an ever-growing conglomerate with the combined register architecture characteristics of 32-bit microprocessors and classic DSP. These new evolving and converging architectures are good for applications and developers. However, they can significantly add to the complexity of the context switch logic handled by the RTOS. The focus of this paper is to review, discuss and devise strategies for minimizing the context switch times for processor architectures with large register sets. In particular, the case study will focus on the PowerPC architecture with Floating Point and AltiVec extensions.

### ***Factors Affecting Context Switching***

Typically, the context switch problem has been defined solely by the raw number of processor registers that are included in the context. Indeed the size of the register set is a critical factor, but there are several not so obvious facets that can also affect the net context switch performance of a system. These factors, ease of access, ISA (Instruction Set Architecture), register volatility and access width, are cited and described below:

### **Ease of Access**

The ease of access to the processor registers is one factor that can affect context switch performance. For example, in some processor architectures only certain registers can be written to memory directly. All other registers must first be transferred to an accumulator (or general purpose register) before being stored to memory, or data must first be loaded to a temporary register and then be transferred to the specific register

when restoring the context. Extra instructions necessary to perform register shuffling can easily double the overhead to save and restore the register set.

### **ISA (Instruction Set Architecture)**

The design of the instruction set and richness of the ISA can also be a factor in improving or restricting context switch performance. For example, some processors have the ability to store multiple registers in a single instruction. One well-known example is the MOVEM (move multiple) instruction found in the classic Motorola M68K CISC and the more modern ColdFire RISC-like architectures. For example, the MOVEM instruction allows for any combination of from 0 to 8 address and 0 to 8 data registers to be saved or loaded from memory in a single instruction instead of executing a long sequence of instructions – one per register.

It is important to understand that due to the bus width limitations (16- or 32-bit in the Motorola M68K / ColdFire case) that the actual register transfer to memory operation still requires multiple cycles. In ColdFire terms, an N register save operation requires N+1 processor cycles (ignoring memory wait states). In contrast, N individual register store instructions would require 2N processor cycles. The substitution of a single, optimized MOVEM instruction for an otherwise normal sequence of MOVE instruction can reduce raw cycle overhead by almost 50 percent [a 15 registers save operation consumes 16 (N+1) cycles versus 30 (N\*2)]. An even more important performance improvement comes from the fact that the write (and read) operations can often be “burst” to memory. The bursting can avoid individual external memory setup and arbitration cycles, and drastically reduce context switch times.

A second and compounded performance improvement from an ISA with multiple register load / store support results from the fact that only a single instruction needs to be fetched and executed. In other words, fetching fewer instructions from memory means fewer cycles and less power. In environments where cache or on-chip memory is available for code storage, the more compact the context switch code, the more efficient the use of these precious resources.

A similar result to the Motorola M68K / ColdFire MOVEM instruction example can be seen when using the PowerPC equivalent of a “move multiple register to / from memory” instruction. In non-Harvard architectures (shared code and data busses), the smaller instruction sequence reduces bus contention between register writes to memory and subsequent instruction fetches. In Harvard architectures with separate code and data busses, the smaller instruction sequence allows for fore-fetching more instructions thus avoiding potential pipeline stalls due to starvation.

### **Register Volatility**

Another key factor in context switch performance is the volatility of register contents as defined by the processor’s ABI (Application Binary Interface) specification. A non-volatile register must be preserved across a C/C++ function call. In contrast, a function is free to change the contents of a volatile register without first saving it. Roughly speaking, a non-volatile register can be considered to be “callee” saved, while a volatile

register is “caller” saved. The specification of the argument passing scheme used for standard C/C++ functions is considered by some to be one of the most valuable contributions of the ABI. However, equally important for performance is the logical breakdown of the processor register set into volatile and non-volatile components.

In order to better appreciate the intricacies of the register assignment issue, we will look closely at two real world implementations. First, consider the PowerPC architecture primarily made up of 32 GPRs. The PowerPC EABI divides the GPR set into 14 volatile and 18 non-volatile registers. This is in sharp contrast to the register allocation in the StarCore™ EABI. The StarCore architecture, also basically 32 GPRs, is divided into 28 volatile and 4 non-volatile registers. Performing the process GPR context switch for PowerPC requires a minimum of 18 GPRs to be restored while StarCore requires only 4 GPRs to be handled. This is not the whole story on processor context switch performance, but it can sure lead to some very interesting vendor-provided benchmarks.

### **Register Access Width**

The width of the data bus can have a major affect on the performance of a system. Likewise, the access width of the register load / store ISA can impact context switch performance. For example in the Motorola DSP56300 architecture, performance doubling can be achieved by executing a single write operation of the 48-bit X (or Y) accumulator register by splitting the operation across the 24-bit wide X: and Y: data busses. In effect, two registers, X high and X low, can be saved in the same instruction (and memory) cycle.

Working our way up the technology ladder, the save or restore of the four, non-volatile, 32-bit StarCore registers (cited in the Register Volatility example above) can occur in a single cycle. This feat of digital magic is possible because the StarCore v2 architecture sports dual, independent, 64-bit data busses, and can execute up to 4 register micro-instructions in the same cycle. There is an endless supply of processor-specific low-level “tricks of the trade” for managing processor contexts, but we must move on to a concrete example.

### **PowerPC Architecture**

In order to study some of the real world design decisions that must be made when designing a context switch strategy, we will return our focus to the PowerPC architecture. A minimal embedded PowerPC processor register context set includes thirty-two 32-bit general purpose registers (GPR0 - GPR31), Program Counter (PC), Machine Status Register (MSR), Condition Register (CR), Integer Exception Register (XER), CTR (loop counter), and Link Register (LR).

The PowerPC EABI specifies register assignments as follows: GPR0 (reserved for temporary use), GPR1 (Stack Pointer), GPR2 (aka SDA2 - Small Data Area for constants), GPR3 - GPR10 (used for passing arguments), GPR11 - GPR12 (reserved for temporary use), GPR13 (aka SDA - Small Data Area for data variables), and CTR (loop counter) are considered volatile. The non-volatile set includes GPR14 - GPR31. [Note

## Strategies for Minimizing Context Switch Times

that there are slight differences between the EABI (Embedded ABI) and Desktop PowerPC ABI with respect to SDA usage.]

PowerPC architectures, such as but not limited to Motorola MPC826x, MPC5xx, and MPC74xx families, that include hardware floating point support add an additional thirty-two 64-bit Floating Point Registers (FPR0 - FPR31), a Floating Point Status / Control Register (FPSCR), and a Floating Point Exception Cause Register (on MPC5xx only) to the processor context. In a similar fashion to GPR allocations in the PowerPC EABI, the floating point register set is divided with FPR14 - FPR31 deemed as non-volatile, and FPR0 - FPR13 as volatile registers.

PowerPC architectures, such as but not limited to IBM POWER, Apple G4 and G5, and the Motorola MPC74xx family, that include AltiVec extensions add another thirty-two 128-bit AltiVec registers (VR0 - VR31) and a vector status / control register (VRSCR) to the PowerPC processor context. The PowerPC EABI allocates the AltiVec register set into VR20 - VR31 as non-volatile, and VR0 - VR19 as volatile registers.

Now that all of the details of the PowerPC EABI and register architecture are presented, let's get down to some context switch timing analysis and design. First, consider the PowerPC general purpose register set (GPR0 - GPR31). Some, but not all variants of the PowerPC ISA include a multiple general purpose register move to / from memory operation via STMW / LDM instructions, respectively. Unfortunately, the PowerPC move multiple register instruction itself is not general purpose. The STMW / LMW instruction encoding is designed such that the register set of interest is defined using a starting GPR number up to and including the last numbered GPR. In other words, the multiple move operation begins at a specified register and continues through GPR31. The use of the multiple register operation is desirable for some of the same reasons cited above related to explanation of the Motorola 68K / ColdFire MOVEM instruction.

Note that some PowerPC environments, e.g., little-endian mode on Motorola MPC8xx family, do not support the use of STMW / LDM instructions. In such restricted environments, developers must resort to using single register store / load instructions for context switching the GPRs.

Regardless of the history, the PowerPC register set restrictions imposed by the store / load multiple register instructions correlates well with the reason why the original PowerPC ABI team allocated registers GPR14 - GPR31 as non-volatile registers. In layman's term, GPR14 - GPR31 must be preserved across function calls. During code generation, the PowerPC compiler allocates registers for working variables from GPR14 to GPR31 - in reverse order. In this manner, the compilers can easily save and restore working register sets in the function prologues and epilogues using efficient STMW and LMW instructions.

## RTOS Implications

In the preceding sections, we have been looking at context switch overhead from the perspective of synchronous function calls to explain the various characteristics of a processor that can affect context switch performance. Operating systems face similar challenges when dealing with the asynchronous interrupt domain. The setting for this paper is the asynchronous interrupt domain – not the more mundane world of synchronous function calls. However, it helps to understand the nature of both breeds as there are many similarities.

We will now change our attention to saving and restoring register contexts in the more challenging asynchronous domain, e.g., interrupts, handled by the RTOS. In an interrupt handler, the powerful PowerPC load and store multiple register instructions can essentially save and restore the entire GPR context in a single instruction referencing GPR0 thru GPR31. This appears to be a good thing, but unfortunately this is not always quite what you need. For example, GPR1 is the stack pointer and saving its value on the stack seems a little odd and redundant. In many embedded RTOS environments, GPR2 and GPR13 are dedicated as global SDA (Small Data Area or base) registers for constant and data variable space. Since the values in these two SDA registers are fixed (constant) for all processes, there is no need to save and restore them on every interrupt or context switch.

With this refined register use specification, the minimum GPR set to be saved becomes GPR0, GPR3 - GPR12, and GPR14 - GPR31. Unfortunately, this isn't a very convenient register pattern for use by the sequential register restricted STMW instruction. For the sake of execution speed, code size, simplicity, and overall convenience, GPR13 is often saved anyway in forming a single STMW instruction saving GPR3 - GPR31 leaving the saving of GPR0 as a single GPR save operation. The alternative "ugly" solution is to save GPR0, GPR3 through GPR12 individually, and then GPR14 - GPR31 using STMW.

Up to this point in this paper, we have dealt only with the PowerPC general purpose register (GPR) set. Equally important is maintaining the PowerPC SPRs (Special Purpose Registers) such as link register (LR = SPR8), counter register (CTR = SPR9), condition register, and integer extended register (XER = SPR1). Unfortunately, the PowerPC ISA does not allow for SPRs to be directly written to or read from memory. Consequently, at least 2 operations per SPR are necessary to save or load an SPR. One instruction is necessary to transfer an SPR to a GPR, and then another one to write the GPR to memory. In order to minimize the total time needed to save the SPRs in the context, the set of SPRs are often individually transferred to a consecutive run of high-numbered GPRs, and then written using a single store multiple register instruction. The following code fragment is typical of this method:

```
    mfspr  r28,lr
    mfspr  r29,ctr
    mfcrl  r30
    mfspr  r31,xer
    stmw   r28,128(r1)
```

Note that the complete register context of a synchronous interrupt, e.g., caused by a TRAP or a function call, is much smaller than the interrupt context. Discounting the stack pointer (GPR1), fixed SDA registers (GPR2 and GPR13), volatile registers (GPR3 - GPR10), and temporary registers (GPR0, GPR11, and GP12), the minimum GPR context shrinks to just GPR14 - GPR31 which can be saved and restored using only a single instruction.

### **Floating Point Context**

We have discussed several of the issues of managing the PowerPC GPR set. Now, we will move on to the more challenging floating point register set made up of FPR0 - FPR31. Unfortunately, there is no composite instruction available in the basic PowerPC ISA to move multiple FP registers to / from memory. Consequently, 32 individual instructions must be executed to save or load the entire thirty-two 64-bit FP register file.

From the architectural description of the floating point register set, it should be obvious that the addition of the hardware floating point unit (FPU) can greatly increase the overall context switch time for a system. Note that each 64-bit wide FPR has twice as much raw context as each 32-bit wide GPR. The limited bus width would be a concern in a system designed with only a 32-bit external data bus since two memory cycles would be required for each FPR access. It is not unusual to expect to more than double the context switch time with the addition of a FPU to a system.

The addition of a FPU can add tremendous number-crunching capacity to a system, but the cost to the overall system performance isn't free. The RTOS design challenge becomes, "How can the FPU context switch overhead be reduced?" The first option is quite simple, but is not always obvious.

### **Reduced Context Strategy**

One solution to the problem involves simply making a design decision to not to use all of the 32 FPRs in the application. Indeed, it is a rare application in which the compiler requires the use of all 32 FPRs in a function. For full FPU register set utilization to occur, an algorithm would have to require 32 working floating point variables to be active (or live) at the same time. With today's ever improving compiler technology with smart register allocators, sophisticated variable lifetime analysis, register coloring, deep flow analysis, and runtime feedback from code profiling, it is often possible to generate very efficient code even with a reduced FPR set. In the rare case where the requirement for the complete FPR context is required, often the specific algorithm can be limited to running in a single execution thread. If the additional FPRs are utilized only by a single process, then the additional FPRs do not have to be swapped, and the reduced register set approach still meets the design goals.

In order to remain largely compliant with the PowerPC EABI (and integrate most easily with existing compiler technology), the unused FPRs should be allocated beginning at FPR14 toward FPR31. Note that there is no standard technique for reducing

the PowerPC FPR set size, and any solution that is devised should be considered to be highly compiler and tool chain dependent.

In one popular tool suite, compiler switches are used to simply remove individual FP registers for use by the compiler. This is considered as “reserving” the registers. In contrast, when using the Metrowerks Embedded PowerPC C/C++ compiler, the creation of a reduced register set environment can be accomplished by declaring global C register variables at file scope, and dedicating the variables to specific physical FPRs. A code fragment demonstrating the Metrowerks-specific C syntax for dedicating two FPRs to global variables is shown below:

```
register double reg_variable_f14 asm ("f14");  
register double reg_variable_f15 asm ("f15");
```

To complete the FPR diet, the global scope floating point `reg_variable(s)` are never actually referenced by any application code. The net result is a system that never uses the dedicated FPRs. An environment consisting of a reduced floating point register set – which is just what we need.

In theory, all 18 non-volatile FPRs should be removable leaving a minimum configuration of 14 FPRs for use by the application. However in this specific example, the compiler requires a minimum of three non-volatile floating point registers for internal use. Consequently, the minimum attainable FPR set size is 17 FPRs made up of: FPR0 - FPR13 and FPR29 - FPR31. This compiler specific optimization will reduce the FPU-specific context switch by 15 FPRs or over 40% compared to the full 32 FPR set version. In addition to the speed improvements, and sometimes just as important in systems with tight RAM budgets, is the reduction in the context save area memory size of 120 bytes per process (15 FPRs \* 8 bytes per register).

In order to successfully utilize a reduced floating point register set environment, every source code module in the system including operating system routines, runtime libraries, and the application must be compiled with the same compiler switches, global register declarations, or header file definitions. Likewise, hand-coded, assembly level modules and possibly low-level details in the operating system, and interrupt handlers will / might require modification to restrict FPR usage to the selected FPR set.

### **AltiVec Context**

Many of the same issues and strategies presented for reducing context switch times for PowerPC FPU are applicable to the AltiVec environment but on even larger scale. Like the PowerPC ISA for loading and saving FPRs, there is no single instruction in the PowerPC ISA to move multiple AltiVec vector registers (VRs) to / from memory. Individual instructions must be executed to save or restore each of the thirty-two 128-bit VRs.

To make matters worse, each VR load / store operation requires 2 instructions. The VR access problem is based on the fact that the PowerPC ISA does not provide a rich

## Strategies for Minimizing Context Switch Times

addressing mode for accessing VRs. Unlike loading / storing SPRs in which 2 instruction sequence were required, a VR can be read or written directly to memory. However, there is no immediate offset component or auto-increment addressing mode in the indirect AltiVec register store / load instructions. Each VR load / store operation must be preceded by an instruction to calculate the offset for the indirect memory access. Fortunately, the extra instructions cost no execution time due to the internal separation of the CPU and the AltiVec units but code space is required. The high level details of a crude AltiVec register save fragment (in Metrowerks mixed C / extended mnemonics assembly language) are as follows:

```
la    r3,pframe->vr20
stvx  vr20,r0,r3

... ; balance of VR registers 21-30 saved here

la    r3,pframe->vr31
stvx  vr31,r0,r3
```

Reducing the size of the non-volatile AltiVec register set, VR20 - VR31, can pay significant dividends in the reduction of context switch time and in AltiVec context save area memory requirements. The Metrowerks Embedded C/C++ compiler requires a minimum of 3 non-volatile VRs for proper code generation. Therefore a maximum of 9 VRs, namely VR20 - VR28, are candidates for removal. A fully reduced VR set can also save 154 bytes (9 registers at 16 bytes / register) of context save area for each process. Simply removing the use of 9 VRs from the system can also remove many, many cycles on each context switch operation. On a 32-bit wide external data bus, the time to save each of the 128-bit AltiVec registers can be significant, and can dwarf even the notoriously slow floating point unit context save time.

### **Register Tracking Strategy**

An alternative strategy to restricting the application to use a reduced AltiVec register set is to use a smart context switch algorithm that saves and restores only those VRs that contain useful information. If you give it a little thought, this strategy should provide the optimal solution. Each process has access to all of the powerful resources of the AltiVec unit, but pays only a context switch overhead proportional to the instantaneous resource usage level.

An architecture in which the minimal context is automatically tracked, identified and saved would be a real performance breakthrough. Unfortunately, current processor technology isn't quite that advanced -- yet. However, thanks to the AltiVec designers who allocated a dedicated special purpose register (SPR256), and the correspondingly clever compiler developers who can dynamically track AltiVec register usage during code generation phase, a reasonable approximation to the ideal world of preserving only "active" AltiVec registers is possible now.

## Strategies for Minimizing Context Switch Times

How does this quasi-magic work? The magic is contained in the AltiVec VRSAVE register which contains one bit corresponding to each of the thirty-two AltiVec registers. Upon entry to an AltiVec™-aware function, the current VRSAVE register is saved in local storage area on the stack, and it is then updated to reflect the register usage within the function. When a function is exited, the saved VRSAVE register is simply restored to its original state. The runtime pushing and popping of the AltiVec register usage information provides an instantaneous view of the AltiVec register utilization that can be used by the RTOS during context switches.

In theory, executing a simple, tight loop testing the contents of the VRSAVE register bit by bit, and then saving the corresponding VR<sub>n</sub> register provides an optimal AltiVec register save solution. However in reality, the additional runtime overhead to fully decode 32-bits in VRSAVE dwarfs the time savings of skipping the VR save or load operation. From experience, I know that the “perfect” (theoretical) solution is worse than the brute force solution of simply saving all thirty-two VRs.

Many possible methods exist to address this problem, but one of the simplest and most effective is to only decode the use of the non-volatile VRs, VR20 - VR31. Using this method and knowledge that VRs are allocated from VR20 toward VR31, a safe assumption can be made that the set of registers beginning at the first bit set beyond VR20 through VR31 are in use and will be saved.

The non-volatile register VRSAVE decode operation is to strip off (zero) the least significant 20 bits representing VR0 - VR19, and then find the first set bit in the region of bits 20 - 31. The determination of bit number can easily be done using the native CNTLZW (Count Leading Zeros) PowerPC instruction. The resulting bit number minus an offset (20) and a multiplier (8 bytes - 2 instruction sequence per VR) is then used in calculating the jump address into a “fall-through” table of LVX or STVX instruction sequences that save or restore increasing AltiVec register numbers.

In the above logic, if one or more non-volatile VRs is determined to be in use, then it is pessimistically assumed that VR0 - VR19 are also in use, and they are saved, too. Indeed in some cases, more than the minimal “active” VR set is saved. However, the odds, and more importantly technology, are in your favor since a modern, smart compiler is likely to first utilize the volatile registers, and then minimize the use of non-volatiles. This simplistic approach usually out performs the more sophisticated “optimal” individual register test approach.

Note that a VRSAVE value of 0 can be used to detect quickly that no AltiVec context at all needs to be preserved. The simple addition of VRSAVE testing for zero logic easily allows for processes with no need for AltiVec resources to pay zero AltiVec context switch overhead.

The combination strategy of using a reduced non-volatile AltiVec register set size in conjunction with the simple decoding of the VRSAVE register can yield compounded time benefits. Also, since the size of the save and restore VR instruction tables are

reduced by the number of reserved (unused) VRs, the total size of the AltiVec context switch code is smaller. Consequently, the total VR context switch code module occupies less memory space when locked in instruction cache, or relocated into fast, on-chip memory. As a general rule, smaller code yields better performance characteristics.

The focus up to this point has been on optimizations related to physically saving and restoring hardware registers. As seen in the case of the use of VRSAVE in the AltiVec environment, often selecting new and better algorithms win out over simply tuning an existing algorithm. The next few sections describe additional strategies to reduce the context switch overhead for extended contexts, e.g., registers associated with Floating Point or AltiVec units.

### **On Demand Strategy**

One available optimization is to only perform extended context switches “on demand” rather than on each process switch. Significant context switch overhead can be eliminated by keeping track of the effective user of each extended context. The extended context only needs to be swapped when the effective user changes. Consider that if process B preempts process A while A is using the AltiVec unit. An AltiVec context switch may not be necessary if process B does not require access to the AltiVec resource. In this example, when process B eventually blocks, the AltiVec register context would still be that of process A. The otherwise AltiVec context switch from process B back to process A would not be necessary. In a high interrupt rate environment with compute bound processes, this scenario is very likely to occur.

In all of the examples presented above, the application developer was relatively, or possibly even totally, unaware of the issue of which processor resources might be used in the application. If we consider the very real case in which an embedded developer is often quite savvy of how his application consumes resources, then still further optimizations may be possible.

For example, if the developer realizes that the floating point hardware is scarcely used and in only a few functions or modules, then it would be seem reasonable to inform the RTOS of periods when the FPR context is relevant. At all other times, the RTOS can know that the FPR context of that process does not need to be saved. The simple act of bracketing floating point code with a pair of system calls to open and close extended context windows can greatly reduce the total overhead of context switches.

### **Lazy Swap Strategy**

A radically different alternative to the strictly, application-controlled method proposed above is an implementation known as “lazy swapping”. In a lazy swapping environment, every process is assumed to require extended processor resources, e.g., FP and AltiVec units. However, the RTOS does not switch the extended context as part of the normal process switch. Using reverse logic, the RTOS makes the assumption that no process will require any extended resources. Then, when a process actually performs an extended operation, the RTOS determines the validity of the operation, and then performs the necessary extended context switch. The current context is saved for the

## Strategies for Minimizing Context Switch Times

previous user, the last known context for the new user is automatically loaded, and the effective user is updated.

In order for lazy swapping to be implemented, assistance by the hardware is required. For example, the successful execution of PowerPC FP and AltiVec unit instructions require dedicated bits in the PowerPC MSR (Master Status Register) to be enabled. If an operation is attempted and the corresponding bits are disabled, then a runtime exception (interrupt) will occur. By manipulating the extended context privilege bits in the MSR, the RTOS can grant or disallow extended instructions on a process by process basis. It is the occurrence of the hardware exception event that allows the RTOS to intercept the implicit passing of the extended processor resource from one process to another, and properly manage the contexts.

The “lazy swapping” method is similar to the “on demand” method but has slightly different runtime characteristics. By purposely disallowing extended operations as a normal part of each process switch, exception faults will occur only when a process actually executes extended operations. The extended context switch is performed by the RTOS as part of the exception handler – not as part of the normal process context switch logic.

The “lazy swapping” method has the advantage over the “on demand” method: in that no explicit application context “tuning” phase is required. The disadvantage is that it is never known exactly when an extended context switch will occur. In some environments, lazy swapping may outperform other extended context switching algorithms. However, in the worst case, there can be significant additional overhead incurred because an exception must be serviced, and privilege (re-)granted each time after resuming an FPU or AltiVec intensive process.

### **Hybrid Strategy**

An even better strategy is to develop a hybrid, extended context management system built with a combination of the “on demand” and “lazy swapping” methods. In such a design, the worst case thrashing scenario described above could be eliminated by opting for compute intensive applications to provide hints to the RTOS of their current use requirements. Most of the other applications could be left as is (untuned), and execute under the lazy swap runtime umbrella. Behind the scenes, the RTOS would not purposely disallow extended operations for registered users during the process switch. This change will remove the extended context thrashing. The resulting behavior would be a system that uses the “on-demand” method for intentional users and “lazy swapping” for the casual user, and yields better average context switch times.

### **Summary**

Several strategies for reducing context switch times have been reviewed. In particular, the focus of the strategies was on architectures with large register sets where the problem is compounded by the sheer volume of the context. Each of the methods and strategies described in this paper and more are in use today in at least one commercially available RTOS. The context switch time remains an important gauge of the performance of an

## Strategies for Minimizing Context Switch Times

RTOS. The bottom line is that the solution with the lowest context switch overhead is a superior real-time offering.

As end-user expectations and the complexity of the underlying processor architectures continue to increase, the challenge will become even greater to reduce the context switch overhead. It is no doubt that incrementally lower context switch times will be produced for next generation runtime environments. However in order to achieve significant improvements in reducing context switch overhead, core designers, software tool vendors, standard committees, RTOS designers, and application developers will need to coordinate their efforts toward the common goal.

Motorola®, AltiVec™ and ColdFire® are trademarks of Motorola, Inc. MIPS® is a registered trademark of MIPS Technologies, Inc. PowerPC® is a registered trademark of IBM Corp. StarCore™ is a trademark of StarCore LLC. All other product or service names are the property of their respective owners.

**Bill Dittmann** is Chief Engineer at Quadros Systems, Inc. (Houston, TX) where his primary responsibility lies with the RTXC Quadros RTOS product line. He has been involved in real-time application and RTOS development since 1978 on processors from Z80s and 8051s to high-end DSPs, and most everything in between. He received his BA degree from Rice University (Houston, TX) in 1979 with focus in Electrical Engineering, Computer Science, and Mathematical Science. He has been an active participant in Embedded ABI standard committees for PowerPC, ColdFire, and StarCore. Other published articles include [Evaluating Real-time Operating Systems for DSP](#), by Tom Barrett and Bill Dittmann, June 1999 in Computer Design's [Electronic Systems](#). He welcomes any questions, comments, or suggestions regarding this topic, and he can be contacted via email at [bill.dittmann@quadros.com](mailto:bill.dittmann@quadros.com).