Asymmetric Real Time Scheduling on a Multimedia Processor

Alessandro Forin
Andrew Raffman, Jerry Van Aken

February 1998

Technical Report
MSR-TR-98-09

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Asymmetric Real Time Scheduling on a Multimedia Processor

Alessandro Forin, Andrew Raffman, Jerry Van Aken
Microsoft Corporation

Abstract

This paper describes the hardware and system software support for multi-tasking a dual-processor used in a multimedia system. One of the two processors is a conventional RISC processor, the other is a vector processor of new design. There is no commonality in the instruction sets of the two processors. We schedule the former preemptively and the latter cooperatively via check points, resulting in an *asymmetric* scheduling algorithm. Media processing requires predictable scheduling behaviors. Programmers can inform our scheduler of the Real Time requirements of their computations using time-constraints. Scheduling decisions are very fine-grained, and place strong efficiency demands on the system implementation.

Introduction

We have designed and implemented the system support for a new type of processor [Le96] , designed for a high performance multimedia architecture, code-named *Talisman* [To96]. This heterogeneous dual-processor (MSP) contains a general purpose ARM RISC processor, a general purpose Vector Processor (VCP), plus caches and some peripherals all in one chip. The VCP has a very large amount of user-mode state, about 8 KB. Using preemptive scheduling for the VCP would be very inefficient, because of the expensive memory traffic for state save and restore. The ARM processor has a much more limited state size, and is therefore scheduled preemptively.
We have devised a simple way in which an application can cooperate with the system scheduler in saving and restoring the VCP state. The programming model is not affected, it provides the customary multiple processes with multiple threads each. The compiler or the programmer simply inserts a special conditional instruction in the VCP code flow, at points where the active register set is minimal. Normally this instruction does nothing. When the scheduler actually needs to switch the VCP state this instruction becomes a function call to a state save-and-restore routine.
In addition to this *asymmetric* way of dispatching threads on the two processors, the system scheduler provides extremely fine-grained Real Time scheduling support via time constraints [Jo96].

The Talisman [To96] project at Microsoft has designed a new hardware architecture, which achieves state-of-the-art 3D graphics/audio performance and quality at consumer price points. The goal is to make 3D graphics and audio a commodity item, accessible on every personal computer and not just on high-end workstations and/or supercomputers. The Talisman *reference hardware implementation* is the first implementation of the architecture, a single expansion board that plugs into the PCI bus. This single board replaces the functionality that is typically provided by a Windows accelerator board, a 3D accelerator board, an MPEG playback board, a video-conferencing board, a sound board, and a modem. Our OS kernel is resident on the board, and dynamically loads/unloads application modules and device drivers from the host (which runs either the Windows NT or Windows 95 operating systems).

There are two main reasons why our work on Talisman and the MSP processor is of general interest.

Consumer demand for rich media support is focusing software developers on a new set of algorithms and data types, especially in the video and audio domains. Movie playback, 3D graphic intense games and applications, sound spatialization, video conferencing are just a few instances in which data compression and decompression and signal processing techniques have become mainstream computing. Many CPU manufacturers are optimizing their processors for the new algorithms and data types [In96]. The MSP design accommodates the new needs by associating a special (vector) coprocessor to an existing general-purpose architecture. This approach minimizes the impact on existing system and application software. Vector coprocessors have large register files, but media processing demands limited latencies. The problems we faced on the MSP are therefore likely to surface again [Pa97].

Embedded systems often employ specialized digital signal processors (DSP), but their programmers long for a more general-purpose instruction set. Recently, hardware manufacturers have started to address this dilemma by packaging a general-purpose CPU core together with the DSP components. Our heterogeneous processor case fits in this new category, and indeed the MSP is expected to be widely used in embedded applications and consumer devices. We demonstrate in this paper a general and economic model for structuring the system support for this new generation of more flexible DSPs.

MSP Overview

Samsung's Media Signal Processor [Le96] is the combination of an established and general purpose RISC CPU core with a new processing architecture based on a SIMD computing model. Additionally, specialized instructions and computing elements enhance the execution speed of algorithms such as MPEG video decode, 3D graphic pipeline, and audio signal transformations. Also on-chip are instruction and data caches, and I/O elements commonly found on microcontrollers, such as audio CODECs, timers and interrupt controller. The RISC core is the ARM-7 from ARM Ltd. It has been used successfully for building personal computers at Acorn, PDAs at Apple and DEC, and in the embedded market for printers and other controllers. This processor is initially expected to be clocked at 40 MHz. Two of the most interesting features of the ARM architecture are the extremely small die size, and the limited code size (even when compared to CISC processors) resulting from the instruction set encoding. From the system point of view, there are a couple of interesting aspects to the ARM. It does not architecturally define support for virtual memory, although at least one MMU has been implemented externally to the CPU core. The ARM supports a number of modes beside the typical *user* and *supervisor* modes. The ARM is otherwise similar to many other RISC processors; it has 16 general-purpose registers in each mode, and the standard load-store memory interface. Sometimes we had to work around errors in the architecture, for instance when implementing our synchronization primitives. But our algorithms and our findings are not tailored to the ARM architecture; any other RISC or CISC core could replace the ARM in what we will describe in the rest of the paper.

We will refer to the SIMD processor circuitry as the Vector CoProcessor (VCP), because that is the way we model it in software. In reality, this is a full general purpose processor in its own right, with independent instruction fetch/decode units, both scalar and vector instructions, load and store instructions, and control transfer instructions. It delegates to the main ARM processor only interrupt and exception handling duties. Another aspect in which the VCP is somewhat enslaved to the ARM is that some of the VCP registers are accessible to the ARM but not vice-versa. Architecturally, it would be easy to define the few additions needed to make the VCP into a truly standalone, general-purpose processor.
The VCP has one bank of 32 scalar registers, which act just like any RISC set of general-purpose registers. It has two banks of 32 vector registers. Each vector register is seen, depending on the instruction operating on it, as a set of N-bits numbers. More specifically, a register could be seen as 32 9-bit integers, 32 8-bit integers, 16 16-bit integers, 8 32-bit integers, and 8 32-bit IEEE 754 single precision format floating-point numbers. There are a few additional control registers that control the behavior of the processor, and the privileged state. This processor is initially expected to be clocked at 80 MHz.

The MSP chip contains two other components intended for use by the VCP. One is a BitStream Processor, designed to efficiently perform certain portions of the MPEG decoding that operate on bitfields, in parallel with the other two processors. The state of this processor is 256 bytes. The other component is the ScratchPad RAM, which is just a portion of the cache that acts as 4 KB of memory. Stores to this memory address range are never written back to memory. The ScratchPad is intended as a high-speed memory buffer, to complement the register files.

The VCP is clearly a complex engine, but from the operating system point of view the most important aspect of it is the size of the state that would have to be stored and loaded from memory during a full context switch. The grand total is 8600 x 2 = 17,200 bytes. Our techniques should be useful on any other coprocessor of similar complexity.

The VCP Model

The VCP is faster at signal processing than the ARM, and the overall performance of the system is very dependent upon efficient utilization and scheduling of the VCP resource. We made the following requirements and assumptions while developing a scheduling model for the VCP:

1. The VCP scheduling mechanism must allow developers to implement tightly coupled threads between the ARM processor and the VCP. Some programmers want to use both processors together as a single unit to get the most performance out of the system. This cannot be done by treating the ARM processor and the VCP as totally asynchronous processors.
2. The mechanism must also allow independent computations on the two processors. Some programmers will simply start the VCP on a long computation and block the ARM processor waiting for termination. In these instances we want the ARM processor to be available for other threads.
3. Most VCP algorithms will be somewhat batch oriented, or will otherwise have locations in their code paths at which they maintain relatively little state. Therefore, software developers can design their code to give up the processor at well defined places, and even (in some cases) restart a set of operations that was aborted.
4. VCP algorithms will be single-threaded. The only synchronization is with the ARM counterpart, where all the explicit multi-threading takes place. Consequently, VCP code has no access to synchronization primitives such as condition variables and mutexes, or to time-constraints. There is no separate VCP scheduling code. Indeed, there is little or no kernel code at all compiled for the VCP.

The VCP is a resource that must be shared among the threads that execute on the ARM processor. The method used to facilitate this sharing is to *virtualize* the VCP: each thread that uses the VCP can assume that it has exclusive use of the VCP.
Although each thread assumes that it has exclusive use of the VCP, the program that runs on the VCP is, in fact, treated by the operating system as a task that can be context-switched in and out of the VCP. Associated with each ARM thread that uses the VCP, therefore, is a single VCP task that is managed behind the scenes by the kernel. As long as a thread accesses the VCP through the supported primitives and the VCP application programmer follows a few simple rules, the possibility of interactions with other threads that are also using the VCP can be ignored.

The first and most basic rule is that an ARM thread must bracket access to the VCP hardware between calls to the *VCPTake*() and *VCPRelease*() functions. This indicates that the programmer's accesses to the VCP are intentional; any VCP access outside a block that begins with *VCPTake()* and ends with *VCPRelease()* is considered an error and generates an exception.
The second rule is that tasks running on the VCP must be ready to voluntarily release the processor, saving the necessary state in memory and reloading it when restarted. This is done synchronously; the code stream is interspersed with special instructions to mark *check points*. At these points the amount of state to be saved is much less than the worst case, typically just a handful of scalar registers or even null. The

compiler (or the programmer) generates the VCP function to save and restore state, or uses a pessimistic default function.
The last rule is that check points must occur “frequently enough”. The worst-case latency with which a thread can regain ownership of the VCP is the system-wide longest code sequence between any two check points. Presently, programmers might have to help the compiler to make sure this rule is obeyed.

Preemptive Scheduling of the Main Processor

By design, the MSP is an unbalanced multiprocessor. One of the two processors is meant to execute the more system-intensive tasks, while the other is meant for the compute-intensive ones. This division of labor is not absolute. There can be codes for which the instruction set of the intended processor is sub-optimal and the programmer certainly has the freedom to allocate a task where it best performs. Since interrupts are all routed to the main ARM processor, that is the one that runs the kernel, the device drivers, and the DMA engines’ code. In addition, all the system code for inter-thread synchronization and data dispatching is run on the main processor, to take advantage of the tighter integration possible with the scheduler.

Threads running on the main processor are preemptively context-switched to support a multitasking environment. When a thread is switched out of the main processor, the kernel saves all the registers and other internal state of that thread into an area of memory. This state is later restored when the thread is switched back into the main processor. In this way, each thread has full access to all of the registers of the machine and does not need to worry about sharing those registers with other threads. Preemptive context switching minimizes the latency with which we can respond to asynchronous events. Saving and restoring the whole register set (16 words) is feasible on the ARM because it represent only a small percentage of the total time required to schedule a thread.
A thread can be preempted either because of a timer interrupt, or because of a device interrupt. When the scheduler selects a new thread to run it also decides how long the thread should be allowed to use the CPU. This time interval is then programmed into the system timer before dispatching the thread. When the timer expires the scheduler is invoked and a new thread is selected for execution. Note that we reprogram the interval timer on each context switch instead of using a periodic interrupt. Using a periodic interrupt would generate a large number of useless interrupts. To provide fine-grain scheduling control it would have to run at a very high frequency, causing excessive overhead. The precision and granularity of the hardware timer directly affects the scheduler's precision and accuracy.

An interrupt from a device other than the timer does not necessarily cause any scheduling activity. In an Interrupt Service Routine (ISR) the device driver’s writer is entitled to use one (and only one) system primitive: signaling of a condition variable. Should this primitive be invoked, upon return from the ISR the kernel code might find it necessary to invoke the scheduler. This in turn might preempt the current thread in favor of another one that has now become more urgent. Ideally, the ISR will just do the minimal amount of work to dismiss the cause of the interrupt, and wakeup some thread to do the actual work of handling the interrupt. For instance, a very simple ISR approach is to disable further interrupts from that particular device, by clearing its interrupt enable bit in some control register. It will be the thread waiting on the device's condition variable that will do all the work that would normally be done in the ISR of a non-Real Time OS kernel. It is up to this thread to then re-enable the device's interrupts.

The scheduler is admittedly vulnerable to the user-written ISRs. The scheduler does not account for any time spent in an ISR, for efficiency reasons. We also decided not to employ any interrupt nesting or prioritizing scheme. Doing so would only provide the illusion of low latency for one device (the highest priority one) at the cost of a more costly dispatching of interrupts in the normal, no collision case. It is *never* the case that the performance of the system depends exclusively on one device’s ISR. Therefore our ISRs simply run with interrupts disabled. Unfortunately, high-efficiency dispatching of interrupts also means that scheduling is only accurate to the extent that ISR invocations occupy a negligible portion of the

processor's time. Should a programmer install an expensive ISR the whole system will visibly suffer. We did not install any of the many possible safeguards against this danger.

A thread can also be rescheduled non-preemptively upon invocation of one of the various synchronization primitives. A typical case is a thread that signals a condition waited upon by a thread with a more stringent time constraint. This happens, for instance, in a small data-dispatching package we wrote to support media streaming. A thread that is starving for data, and approaching its deadline, will most likely wakeup and preempt the thread that just provided the data and signaled the non-empty condition.

Cooperative Scheduling of the Vector CoProcessor

Virtualizing the VCP in the same way that the main processor is virtualized would require preemptively context-switching the entire state of the VCP when switching from one VCP thread to another. The system scheduler lacks any information as to which VCP resources are actually in use, and therefore must make pessimistic assumptions[1]. The amount of state information associated with the VCP is quite large, and the overhead of preemptive context switching would severely degrade the operation of the system. We decided to adopt a hybrid approach: when the ARM-based kernel needs to context switch the VCP, it initiates a context-switch request to the VCP. This request will be honored when the VCP thread reaches a designated point in its execution, one at which its active context is much smaller than the worst case. In other words, we adopt the software convention that VCP context switches, as well as synchronization between a VCP task and its associated main processor thread, can occur only at these designated check points within a VCP program.

We have identified three types of check points that the model must support: **sync points, clean points**, and **exit points**. The VCP hardware includes special instructions to assist in implementing these check points. The following sections document the implementation of these primitives using VCP instructions.

Clean Points

Clean points are check points at which only a small amount of VCP state information must be saved and restored in the event of a VCP context switch. The scheduler requests a context switch when some other VCP task needs to preempt the currently executing VCP task. To avoid the cost of a full VCP context switch, however, the scheduler does not forcefully initiate a context switch. Instead, it allows the VCP task to continue to execute until it reaches a clean point. At a clean point, it is understood that the VCP state to be saved is much smaller than the worst-case 17,200 bytes, and possibly null.
The responsibility for identifying clean points within a particular VCP task belongs to the application programmer, not the OS. The OS has no knowledge of which registers may be in use by the application at the time the OS requests a context switch.

The application programmer must also ensure that clean points occur frequently enough within his VCP program, so that a VCP task is never forced to wait "too long" for a context switch. The meaning of the words "too long" in this context is system-dependent and varies with the nature of the applications that must share the VCP.
In the Talisman architecture, the rule-of-thumb is that the time between successive clean points in a VCP program should be no longer than 100 microseconds. This figure is somewhat arbitrary. The nature of the clean-point method for performing context switches, however, does require the establishment of a maximum clean-point latency to which all VCP programs must conform.

[1] The hardware could be designed to provide such information, in the form of in-use bits. Each bit corresponds to a given resource. It is set by the hardware when the resource is accessed, and reset by software upon context switch. Note that hardware cannot tell if a resource that has been accessed will infact be needed again in the future. This scheme is therefore sub-optimal.

A clean point in a VCP program is implemented by the Vector Coprocessor Context Switch (VCCS) instruction:

VCCS <*label*>

where <*label*> is a subroutine label. The target subroutine performs the context switch, perhaps with help from the main processor. The subroutine must be customized to save and restore any VCP registers that are dirty (contain data still in use by the program) at the given clean point. The term "clean point" implies that only a small portion of the VCP's register state is dirty.

The VCCS instruction tests the Context Switch Enable (CSE) bit in the interrupt mask register (VIMSK), which is controlled by the ARM. If CSE = 1 the VCP jumps to the target subroutine, otherwise it continues with the next instruction. The OS requests a VCP context switch by simply setting the CSE bit to 1. This instruction is implemented assuming the jump is unlikely to be taken— the subroutine is never prefetched.

Using this mechanism introduces very little overhead into the system. Clean-point support requires only one VCCS instruction in the code path, conserving code size and instruction cache usage. During normal VCP processing, while the main processor is not trying to context-switch the VCP, the VCCS instruction uses only one cycle to execute and does not flush the pipeline. It is also feasible to eliminate this overhead by placing the VCCS instruction before another instruction that would otherwise be stalled anyway due to pipeline interlocks.

The context-switch function specified by the label is responsible for both saving and restoring the context of the VCP. A VCP thread may use a different context-switch function for each clean-point in its execution path, where each function is specifically optimized for the clean-point associated with it. The compiler uses this approach. Alternatively, the VCP assembly programmer may choose to implement a single general-purpose context-switch function which saves and restores the union of all contexts for the clean-points from which it may be called. Although less efficient in terms of execution time, this solution could reduce the overall code size of the application.

VCP assembly programmers incorporate clean-points into their programs using the macro *VCPMaybeContextSwitch(label)*. This macro simply instructs the assembler to generate the VCCS instruction, but ensures source code compatibility with future revisions of the architecture.

Sync Points

Sync (short for synchronization) points are check points at which the program running on the VCP must synchronize itself to the computation occurring on the main processor. Sync points will occur periodically in applications that require close cooperation between the ARM and the VCP. In contrast, applications that allow the VCP to run independently of the main processor for long periods will use few or no sync points.

The VCP can advance past a sync point in its program only when the main processor thread permits it to do so. The purpose of the sync point is to prevent the VCP task from getting too far ahead of the main processor. Conversely, to ensure that the main processor does not get too far ahead of the VCP, the main processor can wait for the VCP to reach a sync point. In either case, the main processor and the VCP will be synchronized to each other at the moment they both proceed past the sync point. In other words, a sync point is a simple two-party barrier.

As an example of the use of sync points, assume that a main processor thread loads data into buffers, from which a VCP task reads the data for processing. While the VCP is processing one buffer, the main processor may be loading the next buffer. Before reading a new buffer, the VCP must first pass through a sync point. The sync point is necessary to ensure that the VCP does not try to read data from a buffer before the main processor has finished loading the buffer.

The virtual-VCP model supports combined sync-and-clean points, but not simple sync points—- that is, every sync point is also a clean point. Adopting the software convention that every sync point must also be a clean point both streamlines the software implementation and simplifies the underlying hardware architecture.

A combined clean-and-sync point is indicated by the Vector Coprocessor Join (VCJOIN) instruction:

VCJOIN.xx <*label*>

where <*label*> is the label of the context-switch subroutine for the clean point, and *xx* is one of the optional VCP conditionals (GT, EQ, LT, etc.). This is similar to the VCCS instruction, which also specifies the label of a context-switch subroutine. Unlike the VCCS instruction, however, the VCJOIN instruction conditionally halts on the basis of the general processor flags instead of jumping to the target subroutine on the basis of a special control bit from the main processor.

When the VCP reaches a sync point, the VCJOIN instruction conditionally halts the VCP and notifies the associated main processor thread that the VCP program is waiting for the main processor. The ARM processor responds by executing a Start Vector Coprocessor (STARTVC) instruction, which allows the VCP to resume processing. Typically, a main processor thread determines that the VCP has halted at a sync point by using the Test Vector Coprocessor (TESTVC) instruction to poll the idle/running status of the VCP. While the main processor thread is running, the OS facilitates polling by disabling the VCJOIN exception.

The kernel can context-switch the VCP after the VCP has encountered a sync point using the following mechanism: After halting at a VCJOIN instruction, the vector program counter register (VCINS) points to the VCJOIN instruction that caused the halt. To determine the context-switch routine address, the OS must retrieve the 32-bit VCJOIN instruction and extract the PC-relative offset from the 23 LSBs of the instruction. The entry address to the context switch function is then calculated as (VCINS + 4*offset). The OS can force a branch to the context switch function by saving the old VCINS, reloading the VCINS with the address of the context switch function, and restarting the VCP using the STARTVC instruction.

Sometimes, a higher-urgency thread will preempt the main processor thread that is currently using the VCP. If the preempting thread does not need to use the VCP, the VCP task associated with the preempted main processor thread continues to run. On the other hand, if the higher-urgency thread requests usage of the VCP, the kernel will need to initiate a context switch request to the VCP. If the VCP is already halted at a sync point, the kernel can immediately revector the VCP based upon the label given in the VCJOIN instruction, and force a context switch. If the VCP was not already at a sync point, the scheduler may choose to block the preempting ARM thread until the VCP reaches a point at which it can be context switched. As the VCP continues to execute, it may reach a sync point (a VCJOIN) and remain halted while it awaits intervention by the preempted thread, which may not be scheduled to run for a long time. VCP processing time is typically too valuable to waste in this fashion.

The scheduler can ensure that it is notified of this situation by enabling the VCJOIN-exception interrupt from the VCP to the main processor. The interrupt is enabled by setting the VIMSK register's VJE bit to 1. When the VCP halts at the VCJOIN instruction, the resulting interrupt informs the scheduler that the VCP is halted awaiting intervention from the associated main processor thread. The scheduler then has the option of increasing the urgency of the preempted main processor thread so that it can more quickly restart the VCP, or to initiate a context-switch.

A main processor thread that is waiting for a sync point that may not occur for some time has the option of immediately blocking. This allows other threads to execute on the main processor while the blocked thread waits. Meanwhile, the OS enables the VCJOIN exception so that the main processor will be interrupted when the VCP reaches the sync point. Upon receiving the interrupt, the OS reschedules the blocked main processor thread to begin executing again.

At the time that the OS requests a VCP context switch, the OS sets both the CSE and VJE bits in the VIMSK register. The VJE bit enables the VCJOIN exception, which causes the VCP to interrupt the main processor when the VCP halts at a VCJOIN instruction. This means that the context switch will occur either at the next simple clean point or at the next combined sync-and-clean point, whichever occurs first.

Note that the context-switch calling mechanism used by clean points (VCCS) differs slightly from the calling mechanism used by sync points (VCJOIN):

- At a clean point, the VCCS instruction actually transfers control to the context-switch routine, pushing the return address onto the stack as it does so. This return address is the address of the instruction after the

VCCS. When returning from a context-switch function, the VCP should resume execution at the instruction following the VCCS by executing a return-from-subroutine instruction (VCRSR).

- At a sync point, the VCJOIN instruction simply halts. In the event that the OS on the main processor is awaiting the opportunity to context-switch the VCP, the main processor responds by simulating a call to the context-switch routine. To do this, the main processor retrieves the offset of the context-switch routine from the VCJOIN instruction, pushes the return address onto the VCP's stack, loads the routine's entry-point address into the VCINS, and restarts the VCP to begin executing the context-switch routine. When returning, the context-switch routine executes a VCRSR instruction, just as it would have if the context switch had been invoked from a clean point rather than from a sync point.

These differences are automatically hidden from the programmer by the kernel. This allows the context switch functions that may be called from clean-points and sync-points to be fully compatible with each other, and a single context-switch function may be shared between the two if desired.

VCP assembly programmers should incorporate sync-points into their programs using the macro *VCPSync(label,condition)*. This macro generates the correct VCJOIN instruction, and ensures source code compatibility with future revisions of the architecture.

Exit Points

An exit point is a check point at which a VCP program terminates. Upon reaching an exit point, the VCP halts and interrupts the main processor. The main processor OS can respond by scheduling another task to run on the VCP, if one is ready to run. There is no context-switch function associated with an exit point. Upon reaching an exit point, it is assumed that the VCP thread has already saved any needed context back to system memory, or that no context needed to be saved.

The assembly programmer designates an exit point in his VCP program by inserting a Vector Coprocessor Interrupt instruction:

VCINT <*value*>

where <*value*> is a 23-bit immediate value. The meaning of this value is system-specific and indicates the reason for the interrupt. A specific value is chosen for exit-point interrupts to distinguish them from other possible future uses of the VCINT instruction.

VCP assembly programmers should incorporate exit-points into their programs using the macro *VCPExit()*. This macro generates the correct VCINT instruction and immediate value, and ensures source code compatibility with future revisions of the architecture.

The VCJOIN and VCINT instructions are similar in operation, but they are used for different purposes. Sync points are check points at which the VCP task halts and awaits intervention from the associated main processor thread. The main processor thread may use polling to determine when the VCP reaches a sync point. Exit points, on the other hand, are check points at which the VCP task halts and interrupts the main processor to alert the OS that the VCP is available to execute another task.

In order to support both polling for sync points and interrupts for exit points, the interrupts for the VCJOIN and VCINT instructions must be independently maskable. For this purpose, the VIMSK register contains separate bits (VJE and VIE) for enabling the VCJOIN and VCINT interrupts, respectively.

Faulting Unauthorized VCP Accesses

The virtual-VCP model allows the OS to preempt a main processor thread without necessarily preempting the VCP task associated with it. Allowing the VCP task to continue to execute increases parallelism between the main processor and the VCP. By bracketing all accesses to the VCP hardware with *VCPTake()/VCPRelease()*, the OS can track which main processor threads plan to access the VCP and preempt those threads which do not currently "own" the VCP.

A potential danger is that an incorrectly written main processor thread may attempt to access the VCP outside of the take/release pair. In this situation, both the main processor thread and VCP task could be corrupted if such an access was permitted. To solve this problem, the API could force each main processor thread to perform all accesses of the VCP registers through system calls that validate the thread's authorization before each access. The checking would clearly entail a certain software overhead.

This overhead is unnecessary if the hardware performs automatic checking for unauthorized VCP accesses. Hardware support is provided in the form of a VCP-access-enable bit (VAE), which enables a trap on an unauthorized access. If an ARM thread attempts to access the VCP at a time when an unrelated task is running on the VCP, the ARM processor traps. During the trap, the OS intervenes to block the offending thread until its associated VCP task can be context-switched in. A VCP fault is similar to a memory fault in that the faulting instruction can be executed again after the cause of the fault has been corrected.

VCP Scheduling Example

To help illustrate the VCP scheduling mechanisms, Figure 1 shows a series of events during which the VCP is context-switched between two threads. Using the legend below, it illustrates the following steps:

ARM-T1 - Thread 1 running on the main processor
VCP-T1 - Thread 1 running on the VCP
ARM-T2 - Thread 2 running on the main processor
VCP-T2 - Thread 2 running on the VCP

1. ARM-T1 performs a *VCPTake()* to gain control of the VCP. Since the VCP is not currently owned by any other thread, *VCPTake()* returns immediately. ARM-T1 now has full access to the VCP and can start an associated task VCP-T1 on the VCP.
2. ARM-T1 is preempted by ARM-T2. VCP-T1 continues to execute because ARM-T2 has not requested access to the VCP.
3. ARM-T2 performs a *VCPTake()* to get access to the VCP. However, VCP-T1 is still running. ARM-T2 is blocked until the VCP-T1 reaches a sync or clean point. ARM-T1 is allowed to run while ARM-T2 is blocked. Assuming that ARM-T2 is a higher urgency thread than ARM-T1, the OS requests a context switch to the VCP.
4. VCP-T1 reaches a clean point. VCP-T1 is then context-switched out of the VCP. The OS also blocks ARM-T1 to ensure that it does not attempt to access the VCP-T1 after the context switch.
5. ARM-T2 is given access to the VCP. The OS unblocks ARM-T2 so that it can continue execution. ARM-T2 starts an associated task VCP-T2 on the VCP.
6. ARM-T2 is blocked due to some other resource in the system. Note that VCP-T2 continues to execute.
7. VCP-T2 reaches a sync point. However, ARM-T2 is still blocked due to other circumstances. The OS may context switch the VCP back to VCP-T1. ARM-T2 is now also blocked on VCP access.
8. VCP-T1 resumes execution. The OS unblocks ARM-T1 so that it can continue running.
9. ARM-T1 performs a *VCPRelease()* to signal the operating system that it will not attempt to access the VCP.
10. ARM-T2 is unblocked on another resource in the system. However, it still cannot run because VCP-T2 is context-switched out. The OS requests a context switch on the VCP.
11. VCP-T1 reaches a clean point and is context-switched out. Note that ARM-T1 can continue running because it is outside of the *VCPTake()/VCPRelease()* section.
12. VCP-T2 resumes execution. ARM-T2 may now be context-switched in.
13. ARM-T1 is context-switched out, and ARM-T2 resumes execution.

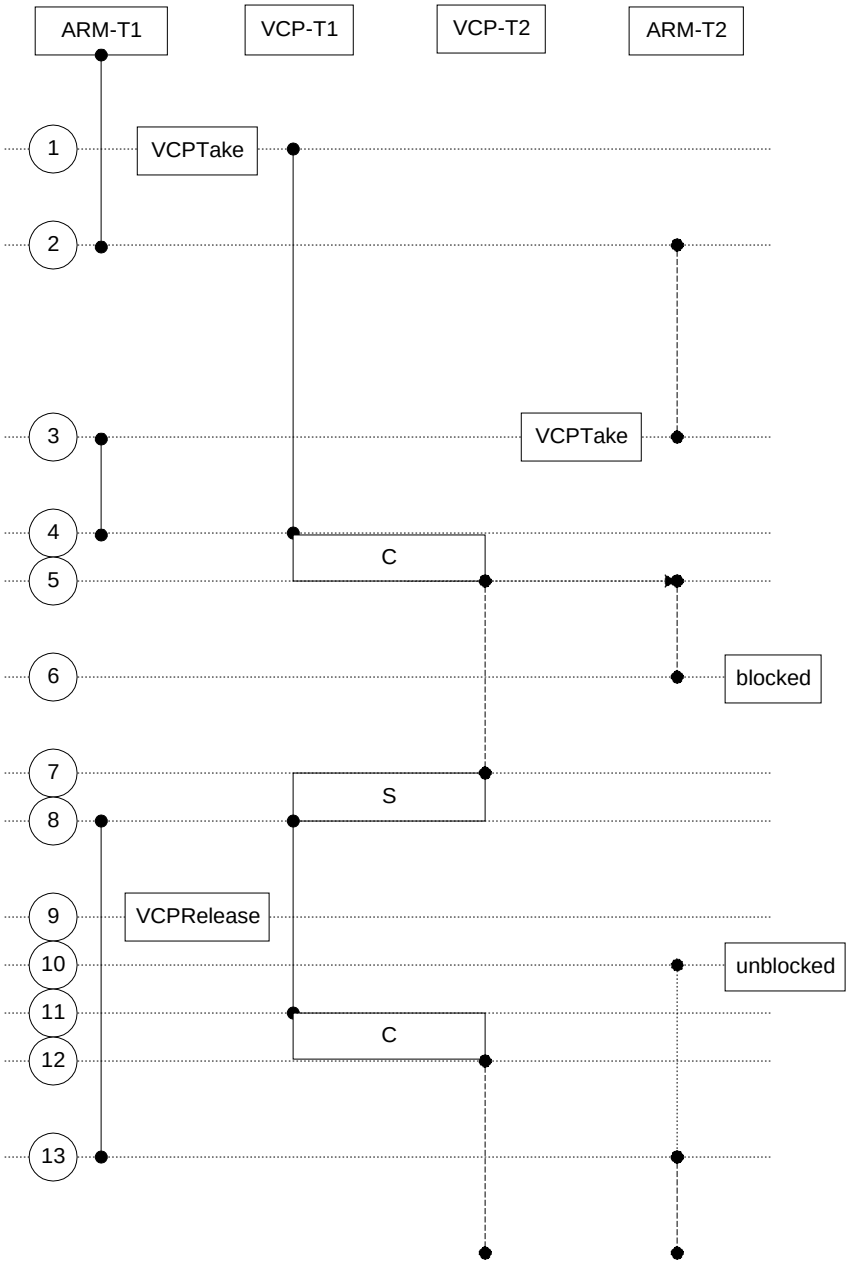

Figure 1: A Sample Scheduling Sequence

Alternative Approaches

On the ARM side, there are two compelling reasons to use preemptive scheduling. The register state is relatively small, 16 words which cost only one blocking cache-line refill because our caches are write-back, with 64 bytes line size. The expected load on the ARM is interrupt- and system-intensive, and latencies must be minimized. But there are a number of other ways in which we could have approached the VCP scheduling problem. In this section we briefly illustrate a few of the models we considered and some of the reasons why we rejected them.

Floating-point coprocessors provide a first, simple alternative model. In this model, the ARM and VCP are scheduled together, as a single entity. The floating-point register file is considered integral part of the thread state, and the instruction stream itself is common between processor and coprocessor. A context-switch requires the full context to be saved and restored. Register save/restore can be performed lazily by the OS kernel, provided it is possible to schedule a thread on the main processor while at the same time keeping the coprocessor disabled. If, and only if the thread then actually uses the coprocessor a trap is taken and registers are saved and restored. This works well if actual coprocessor usage is limited, or limited to just a few (one) processes at a time. Unlike the floating-point coprocessor case, the VCP is a resource that most threads want to use; therefore lazy-switching is an optimization of little value.
We found cooperative switching to be a much more appealing alternative. The coprocessor-disable-bit is still required in our design, but it only prevents the ARM processor from interfering with the VCP. It does not stop the VCP. It is used for protection against programming errors, and to provide better utilization of both processors. As long as they do not interact, we can schedule a new thread on the ARM and continue running the previous VCP task.
Compared to this approach, the virtual-VCP model that we have chosen does place the additional burden on the compiler/programmer of writing the context-switch code for the coprocessor. If the coprocessor state is limited, as is the case with Intel's MMX extensions [In96] , a blind context switch might be an acceptable cost that is offset by other considerations; for example, by not requiring any modification in existing commercial operating systems

A second model looks at the VCP not as a processor, but as a shared object that must be explicitly arbitrated for. A thread acquires exclusive ownership of the VCP, and prevents any other thread from using it (via the disable bit). The owner monitors for requests from other threads, and voluntarily releases the VCP after possibly saving some state. Library functions could make it simpler for the programmers to watch/dispatching for incoming requests.
This approach is attractive if the granularity of scheduling of the VCP is fairly coarse. There is more overhead in monitoring requests, and higher latency in obeying them. But if the ARM processor is otherwise totally idle these higher costs could be acceptable. Another advantage of this scheme is the reduced impact on the system scheduler, but at a higher cost on programmers and higher risk of programming errors. Recovering from errors is more difficult because control is distributed.

A refinement of the second model assigns a specialized thread to exclusively micro-manage the VCP. Other threads cannot access the VCP directly, they must forward their requests to the manager thread.
This model again assumes a fairly coarse scheduling of the VCP. It relieves programmers from having to write their own scheduler, since the manager thread would be provided as a library. It uses the VCP as a server of sorts, programmers only write the client side of a straightforward client-server application.

We chose the virtual-VCP model as the best of all worlds. It borrows the user-interface simplicity of the first model but keeps the door open for users to minimize overhead. Similarly to Scheduler Activations [An92] , we let compiler and/or programmers supply their own context-switching function. But we are not required to support arbitrary user-defined scheduling algorithms. Users that do not want to think of their programs as parallel programs can do so. A compiler can accept a few simple extensions to C/C++ to deal with user-defined vectors of complex data types, and execute in parallel an otherwise fully sequential program. Our model still identifies the VCP as an important shared resource, by requiring the programmers

to bracket their VCP uses. This provides additional feedback into the system scheduler, which might benefit from knowing which threads do/not need the VCP. Scheduling code is kernel-provided and has only limited risk exposure to user mistakes. We make programmers aware of the unavoidable VCP scheduling latency by requiring check points. But the only mistake possible is not to insert them frequently enough. We could also provide additional safeguards, at additional code cost.

Minimally Required Hardware Support

Our state saving optimizations, and the cooperative context-switching model are applicable to any other dual-processor system, and to single-processor systems as well. The scheduler needs to find out, one way or other, *when* it is appropriate to invoke *which* context-switching function. In our case, the VCCS instruction answers both questions. This could be reduced to atomically changing "the" context-switching function, say by mean of invoking the OS or setting some well-known memory location. The scheduler's timer ISR would just make sure that the function is invoked upon return. The "*when*" would be controlled by other means, such as priorities or real time constraints.

The TESTVC ARM instruction is not needed, at least in principle. Using memory or a shared register would also work. But synchronizing via memory and incoherent caches was too expensive, and the ARM and VCP when running do not see each other's general purpose registers. Therefore we needed a dedicated mechanism for barrier synchronization. The only other special instruction we use on the ARM side is STARTVC, which is needed to control the VCP's program counter. Alternatively, the VCP could jump to a fixed startup vector, and the code in there would transfer to the desired program counter.

Besides the VAE protection bit, there are three other control bits in the VIMSK register used by the ARM: CSE, VJE and VIE. Each one controls the behavior of a special VCP instruction: VCCS, VCJOIN, and VCINT respectively. These in turn match the three types of check points we defined: clean point, sync point, and exit point respectively.

Removing the VAE bit causes loss of protection, and exacts a performance price. If a thread is within a *VCPTake()/VCPRelease()* pair, the scheduler must assume that the thread is actively using the VCP, and initiate a VCP context-switch. This additional test is along the critical path of any ARM context switch. If the thread is not within the *VCPTake()/VCPRelease()* pair, it could still erroneously access the VCP and damage other threads.

All interrupts from the VCP to the ARM can be removed, and replaced by polling. This removes the VCINT instruction, and the VIE and VJE bits. Scheduling latencies suffer greatly. For instance, exit points are implemented via a halt instruction. The ARM only notices the VCP is halted when it polls. Either a special periodic interrupt is needed on the ARM side to guarantee a minimum polling frequency, or the application codes themselves must bear this responsibility.

Barrier synchronization is clearly required but can be implemented in a number of ways; for instance, via memory. This still requires some form of atomic memory instruction, on both processors. VCJOIN is replaced by a halt. Again there is extra latency when the VCP is halted because it reached the barrier first, and the ARM does not notice.

This leaves us with VCCS, the CSE bit, and a somewhat inefficient system. But we can make it even worse: the VCP might have to check if a call to the context switch function is needed by testing a location in main memory. This requires either coherent caches or uncached memory-access instructions.

Status

The system codes described in this paper have been implemented and tested on a hardware simulator of the Talisman reference board. This simulator is not an RTL simulator. It is a functional simulator that provides accurate behavior of the MSP, the memory subsystem, and other on-board devices including video and audio logic. The main goal of this simulator is to allow expedient development of software, in a human-controlled and human-friendly environment. This simulation runs at about 150 kHz on a 90 MHz Pentium PC. It is complete enough to interactively run the Doom2 video game, although the term "interactive" might be used here with a bit of a stretch.
The codes were then transported to a real board, controlled by an RTL simulator of the MSP chip (QuickTurn box). This simulation runs at about 1 MHz, and controls the actual pins of the MSP chip's socket. It is intended to provide early debugging for both the board design(s) and the MSP chip itself.

The system can run a meaningful application within 20 KB of RAM, and no ROM. This includes all of the kernel+application code and data requirements. The kernel requires dynamic memory allocations only for new threads (50 words plus stack) and new code images (15 words plus image). As indicated in [To96] , the rest of the 4 MB system RAM is devoted to the 3D graphics system, and to audio data.

The timing aspects of the system are entirely predicated on the cache-refill and memory system performance. This is especially important considering that the Talisman board plans to use RAMBUS components, which can stream data at very high speeds but at a high latency cost. The MSP chip has limited on-chip caches, which cannot hide much of this latency. In our simulations we have found that system code execution with a RAMBUS memory system is about three times slower, on average, than a fast DRAM system of the same size.

The scheduler does not arm the timer if only one thread is runnable, and there are no threads in the sleep queue. This produces exactly zero scheduler overhead for this extreme but not infrequent case. In case multiple threads are runnable, the scheduler will generate overhead proportional to the number of context switches per second. Since the run queues are ordered, the number of runnable threads has only secondary effects on the overheads. The average context switch overhead is on the order of 6 microseconds (4 of which spent in the scheduler proper). This number is also used to estimate the latency with which a thread can be awaked by an ISR. Entering the ISR itself takes at most 0.8 microseconds from the time the interrupt pin is active, assuming the processor was not already inside an ISR (which is non-interruptible).
This extremely limited overhead allows applications to specify time constraints that are both accurate and fine-grained. We conservatively advise programmers not to stress the system below 50-100 microseconds, so that other effects can remain invisible. For instance, cache warm-up after a context switch, and occasional bursts of interrupts.

The longest sequence with interrupts disabled within the kernel is equivalent to one memory store operation, which is retired by the cache subsystem in a small number of cycles. This occurs in two instances. The first is in the implementation of the *AtomicCompareAndSwap()* primitive. All other synchronization primitives are implemented on top of this one. The second is in the condition signaling primitive, where an enqueue operation is inlined with interrupts disabled. This costs one hot-cache read, and possibly two stores (one hot-cache, one cold-cache).

The kernel source code is split into machine-independent and machine-dependent parts. The machine-independent code is common to the MSP and the other versions of the kernel (x86, Philips TriMedia).

References

[An92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, Henry M. Levy.
Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.
In *Transactions on Computer Systems*, pagg. 53-79, Vol. 10-1 February 1992.

[In96] Intel Corporation.
Intel Architecture MMX Technology: Programmer's Reference Manual.
Order no. 243007-002, March 1996.

[Jo96] Michael B. Jones, Joseph S. Barrera, III, Richard P. Draves, Alessandro Forin, Paul J. Leach, Gilad Odinak.
An Overview of the Rialto Real Time Architecture.
In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pagg. 249-256, September 1996.

[Pa97] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyarakis, Randi Thomas, Katherine Yelik.
A Case for Intelligent RAM: IRAM.
In *IEEE Micro*, April 1997.

[Le96] Le Nguyen et al.
Multi-media Signal Processor (MSP).
In *Proceedings of the HotChips Symposium*, August 1996.

[To96] Jay Torborg and Jim Kajiya.
Talisman: Commodity Real Time 3d Graphics for the PC.
In *Proceedings of SIGGRAPH 96*, August 1996.