Cross-Layer Customization for Rapid and Low-Cost Task Preemption in Multitasked Embedded Systems

XIANGRONG ZHOU and PETER PETROV
ECE, University of Maryland, College Park

Preemptive multitasking is widely used in many low-cost and real-time embedded applications for its superior hardware utilization. The frequent and asynchronous context switches, however, require the preservation and restoration of the task state, thus resulting in a large number of memory transfer instructions. As a consequence, task responsiveness and application throughput can be significantly deteriorated. To address this problem we propose a cross-layer customization framework which through the close cooperation of compiler, OS, and hardware architecture achieves rapid and low-cost task switch. Application information extracted during compile-time regarding state liveness is exploited in order to preserve a minimal amount of task state on task preemption. We introduce two complementary techniques to implement the application-aware state preservation. The first technique utilizes compiler-generated custom routines which preserve/restore an extremely small live context at judiciously selected points in the application code. The second technique requires more sophisticated hardware support. It employs an OS-controlled register file mapping to achieve a rapid context switch. By mapping a small fraction of the register file in a single clock cycle, a context switch is achieved requiring no memory transfers for the majority of cases to preserve/restore the live state. The effect of aggressively replicated register files, where each task is given its own replica, is achieved with the hardware cost of only adding from 25% to 50% extra physical registers. Through the utilization of these novel mechanisms, a significant improvement on task response time is achieved as the context-switch cost is minimized.

Categories and Subject Descriptors: C.1.2 [**Computer Systems Organization**]: Embedded Systems; D.4 [**Software**]: Operating Systems—*Process management*

General Terms: Algorithms, design, experimentation, performance

Additional Key Words and Phrases:

ACM Reference Format:

Zhou, X. and Petrov, P. 2009. Cross-layer customization for rapid and low-cost task preemption in multi-tasked embedded systems. ACM Trans. Embedd. Comput. Syst. 8, 2, Article 14 (January 2009), 28 pages. DOI = 10.1145/1457255.1457261 http://doi.acm.org/10.1145/1457255.1457261

14

Authors' address: X. Zhou and P. Petrov, ECE, University of Maryland, College Park; email: ppetrov@ece.umd.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1539-9087/2009/01-ART14 $5.00 DOI 10.1145/1457255.1457261 http://doi.acm.org/10.1145/1457255.1457261

1. INTRODUCTION

Many modern embedded applications, such as personal organizers, cell phones, and various hand-held devices, constitute complex computing systems where multiple execution tasks cooperate in implementing the product specification. Due to market demands, a large number of capabilities need to be supported, such as aggregated multimedia data processing (speech, audio, video), communication protocols (GSM/CDMA, VoIP, Bluetooth, CAN), security functions, user interfaces, and many others. The utilization of embedded processors for real-time and time-critical control applications have been growing rapidly. The modern automotive industry, for instance, has adopted the approach where tens to hundreds of such processors are used throughout a single automobile [Hansson et al. 1997]. They are used for traction control, antilock brake systems, engine control, and many other control and time-critical tasks. Many real-time data acquisition and processing systems such as sensor nodes and networks, impose strict real-time constraints and response time in order to capture, process, and identify rapidly appearing objects and physical phenomena. At the same time, all this processing power needs to be achieved with extremely energy-efficient and low-cost embedded processors.

The inherent multitasking nature of these applications has led to implementations where multiple software tasks are mapped for execution on a high-performance embedded processor such as the Intel XScale [Intel Corporation] and the ARM9 [ARM Ltd.], which offer multitasking support in the form of MMUs and hardware timers, and readily available operating systems (OS) which utilize this hardware to implement various forms of multitasking.

General-purpose OSs have been known to impose deficiencies in meeting the real-time constraints of many embedded applications. The main reasons for this are the lack of real-time scheduling and the high cost in terms of performance and delay of the context switch procedure. Real-time OS (RTOS) kernels [Sastry and Demirci 1995; WindRiver] have been introduced in order to achieve more deterministic scheduling of tasks where certain priorities need to be followed. The RTOS scheduler ensures that tasks are scheduled for execution according to their completion deadlines. However, the cost of saving and restoring the task state remains quite high, as this mechanism depends only on the size of the hardware state that needs to be preserved in order to transparently restore the preempted task back to execution. For instance, the process state that needs to be saved on context switch includes program counter, register files, status registers, address space mapping, and the like. Therefore, a significant number of memory access operations need to be performed in order to store the state of the preempted task and to load the state of the new task to be executed. To minimize the size of the state, light-weighted multithreading was introduced [Byrd and Holliday 1995]. In this approach, a number of threads share some of their state, most commonly their address space. To achieve a context switch, only the register files, and state registers needs to be saved and restored. Due to stringent power constraints the modern embedded processors follow the RISC and VLIW paradigms. In these architectures, and even more so in VLIW, the register file is usually large in order to enable the compiler or software developer

to exploit instruction level parallelism and maintain high instruction execution throughput; a typical modern VLIW architecture [Faraboschi et al. 2000] contains general-purpose register files of size from 64 to 256. Even though such register files are clustered, the registers from all the clusters need to be saved on context switch. This implies that it easily takes a few hundred cycles to save and load the general-purpose register file on context switch. Such a significant overhead has two major impacts on the system. First, the system performance is negatively affected, and second, the response time, which is the time between an event triggering a suspended task and the moment when the task resumes execution, is significantly degraded.

In this article, we propose a novel cross-layer customization methodology which significantly reduces the context-switch overhead and, thus, improves both the total system throughput and the response time for each task. The proposed low-cost context-switch mechanisms are achieved through the active cooperation of compiler, microarchitecture, and operating system (OS). A general-purpose OS or RTOS conservatively saves and restores the entire content of the register file. Such a course of action is needed since no application knowledge is available to the OS task scheduler regarding which registers are alive in that task and thus need to be saved. Similarly, when the task execution is resumed all the register values associated to that task are loaded into the register files. Such a conservative approach has been inherited from general-purpose computing systems, where no prior knowledge regarding the application structure is known, and both the microarchitecture and the OS kernel need to be designed with generality and worse-case assumptions in mind.

We present two complimentary techniques for low-cost and rapid-task preemption. Both of them follow the principle that through the close cooperation of compiler, OS, and architecture, very fast and low-cost task switch can be implemented where only a minimal amount of task state is swapped on task preemption. The techniques differ in the way this is achieved; a trade-off is explored between hardware support and the number of cycles needed to perform the context switch. A typical application usually spends most of its execution times in loops or functions, which are generally referred to as *phases* or *hotspots* [Merten et al. 2001; Sherwood et al. 2003]. Consequently, the code in each such hotspot is typically highly optimized. By applying the proposed methodology independently to each program hotspot all of the benefits from the proposed approach can be achieved with minimal additional hardware.

In the first approach, the compiler-driven context switch (CCS), the compiler identifies what is the minimal number of live registers that needs to be preserved and provides custom software routines to the RTOS kernel. These routines are synthesized by the compiler to save and restore only the live registers for a few switch points or basic blocks in the application inner loops and "hot-spot" regions. The switch points/blocks are being optimally identified by the compiler with the property that only a minimal number of general-purpose registers are alive at these points, hence drastically reducing on the overhead of the context-switch procedure. A minimal hardware support is introduced, which is programmed by software and captures the addresses of the switch points and blocks. These switch points/blocks, even though very few, are encountered very

often during the program execution as they are fixed by the compiler at positions which are frequently executed and inside the application innerloops. The preemption is subsequently deferred to such a switch point/block, where the OS kernel invokes the custom routine for that point/block to store the state and then invokes the custom switch routine for the task which execution is to be resumed.

The second technique introduced in this paper relies on an introduced pool of extra registers, which are judiciously used by the compiler and controlled by the OS to allocate the minimal set of live registers for the switch points/blocks. We refer to this technique as the register mapped context switch (RMCS). Similarly, this is achieved through the close cooperation of the compiler, the OS context switch mechanism, and a cost-efficient hardware support in the form of a limited virtualization of the register file address space. Compile-time register live analysis followed by a register renaming step actively "packs" the set of live registers into a set of contiguous registers. At context-switch time the OS exploits the limited mapping capabilities of the register file to remap the set of registers, which are alive during the time of preemption. The effect of the proposed technique is similar to the effect achieved by aggressively replicating the register file to each task and simply switching between the register file replicas during context switch. However, such fast context switch is achieved with a significantly smaller hardware overhead as compared to multiple replicas of the register files. We show that a pool of extra registers consisting of 25% to 50% of the register file is sufficient to provide a context switch with no saving and restoring of general-purpose registers for groups of parallel tasks. When the combined number of live registers for all the parallel tasks exceeds the pool of available physical registers, only then and only for the less critical tasks that exceed the pool of available physical registers, the corresponding parts (pages) of the register files containing live registers will be moved to memory.

2. RELATED WORK AND MOTIVATION

The two widely adopted schemes for task switch control are the cooperative and the preemptive multitasking. In cooperative multitasking, the task voluntarily releases the control of the CPU to the OS at certain points of its execution. This release typically occurs when the task finishes execution or when the task computation load is low and is waiting for a lengthy I/O operation. Such an approach is followed in TinyOS [Levis et al. 2005] where tasks are executed in a manner of run-to-completion. In this approach, longer tasks need to be partitioned into shorter ones. As pointed out in [Bhatti et al. 2005], such run-to-completion scheme can cause problems with meeting real-time constraints, as it is not possible to partition many tasks, which can result into a situation where a single task occupies the CPU for a long time. The cooperative multitasking paradigm is further explored in [Shivshankar et al. 2005], where the authors have proposed to integrate multiple threads into a single thread statically during compile time. The benefits of cooperative multitasking for networking applications have been analyzed in [Albrecht et al. 2004]. Even though these approaches have the advantage of avoiding nondeterministic context switch overheads, an

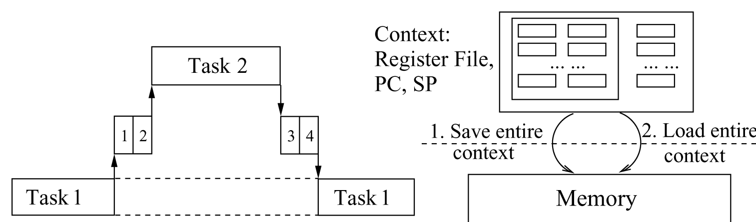

Fig. 1. Context-switch mechanism for preemptive multitasking.

extra limitation on the dynamic behavior is created as they require that all preemption points are known during compile time. The degraded responsiveness to asynchronous events has been the major disadvantage of cooperating multitasking.

In preemptive multitasking, the OS can pause a low-priority task and assign the CPU to a higher priority task—an OS controlled event referred to as preemption. Preemptive multitasking relies on a timer to generate interrupts at regular time intervals. When such an interrupt occurs, the execution control is transferred to a kernel routine that determines whether a task switch needs to be performed and, subsequently, to perform the context-switch. This process is depicted in Figure 1. When the preemption interrupt occurs, the OS kernel executes two basic procedures in order to perform the preemption. In Step 1 the kernel saves the state of Task 1, while in Step 2 it loads the state of the preempting Task 2. Switching back to the original task is performed in the same way. When Task 2 is preempted, identical pair of steps are executed, denoted as Step 3 and Step 4 in Figure 1.

As this approach has the distinctive advantage of better responsiveness and stability, most real-time scheduling algorithms and OS multitasking support are based on it [Nieh and Lam 2003; Chandra et al. 2000]. However, the frequent preemptions interrupt the normal task execution and bring extra performance and power overheads in the form of cycles needed to preserve and then restore the task context. The task context includes the entire register file and all the status registers, such as the program counter (PC) and the stack pointer (SP); its size is by far dominated by the register file. As task preemption exhibits asynchronous behavior, the OS kernel must be conservative and preserve/restore the entire state.

Due to cost and power constraints the majority of modern embedded processors follow the RISC and VLIW paradigms. In these architectures, and even more so in VLIW, the register file is traditionally very large in order to enable aggressive compiler optimizations targeting instruction parallelism and execution throughput. A typical modern VLIW architecture [Faraboschi et al. 2000] features a general-purpose register file of size from 64 to 256. It has been shown [Hill and Culler 2001] that for some short tasks responsible to react and process data samples in sensor networks, the context switch overhead can be up to 30% of the total execution cycle.

Figure 1 illustrates the mechanism of general-purpose task switch in preemptive systems. Before the preempting task can start execution, there are two

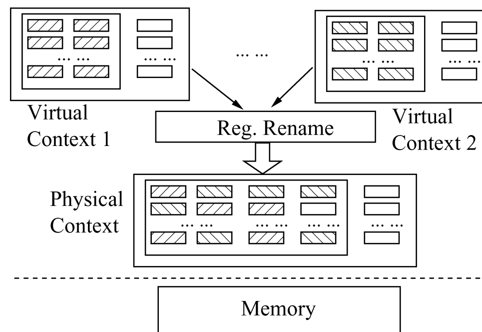

Fig. 2. Hardware register renaming.

steps as shown. First, the state of preempted $Task_1$ is saved and then the state of the preempting task $Task_2$ is to be loaded into the processor hardware. When $Task_2$ is brought back to execution, the identical steps but with reversed state are performed. The context switch overhead is the sum of the execution cycles for all these steps of saving and restoring the hardware state. In modern RTOS kernels, deciding which task is next takes only a few cycles in order to look up a priority queue which does not depend on the number of tasks in the system [Bovet and Cesati 2002]. In this article, we focus on the cost of the context switch only in terms of execution cycles needed to save and restore the states of the preempted and the preempting tasks.

Reducing the context switch overhead has been the focus of various research projects. The main goal is to reduce the number of load/store instructions needed to save and restore the task context. A simple hardware scheme assigns a separate register file to each task. During task switch, the preempting task immediately starts execution by using its own register file copy. The obvious drawback of this approach is the excessive hardware overhead in terms of an extra register file copy for each parallel task in the system. In practice, a restricted version of this approach is used where the kernel and user-level code operate on separate register files.

Instead of having a distinct physical register file for each task, another hardware solution is to have a relatively small ISA-visible set of registers, while implementing a significantly larger physical register file. This organization is illustrated in Figure 2. At runtime, each virtual register is renamed to a free physical register. This approach is very popular in superscalar processors, such as Intel Pentium 4 [Hinton et al. 2001] and Alpha 21264 [Kessler 1999]. Such hardware register renaming is mostly used to exploit the available ILP in the program. Context switches are fast as typically only a small part of the physical register file needs to be preserved in memory. However, due to its per-register granularity and the fact that the renaming hardware needs to be activated at every cycle, the approach suffers from excessive power consumption and as such is not applicable to embedded systems. With a similar objective, in [Oehmke et al. 2005] the physical register file is implemented as a cache that captures a large number of virtual registers. Fast access to subroutine and multithread contexts is achieved with a nontrivial power overhead.

The notion of fast context switch point has been first introduced in [Baker et al. 1995]. Each instruction is marked with a special bit to indicate whether a fast context switch is possible at that point. A fast context switch point is defined as an instruction where all scratch registers are dead. Scratch registers are a subset of all the registers which are caller-saved across function call boundaries; the context switch mechanism saves and restores all the remaining nonscratch registers. Consequently, this is a “all-or-nothing” approach targeting old architectures with rather small register files and no register windowing. VxWorks [WindRiver], on the other hand, provides a special hardware context for interrupt service code in order to avoid preserving the task context and, thus, improving responsiveness to various system generated events. In [Redstone et al. 2003], the authors have proposed a simultaneous multithreading platform with minithread execution. This approach, however, introduces a nontrivial hardware overhead. In Barthelmann [2002], the authors have proposed to reduce the task context in the static OS by finding the live set of each task and merge the set by using the preemption priority information. Albrecht et al. [2004] utilize and explore cooperative multi-threading instead of asynchronous preemption. Other research has shown that for some applications with known set of tasks and well-known runtime characteristics and interactions, an efficient cooperative multitasking system can be synthesized through software thread integration [Dean 2005, 2000]. In a more dynamic system, however, with preemptive multitasking, the active task may have to be suspended at arbitrary point so that another task is placed for execution. Even though the task switch overhead is reduced, the system responsiveness is limited as the compiler must statically decide on the way tasks are interleaved. In a way, the context switch points are explicitly defined by the compiler or the software developer. The tasks are effectively merged and various optimizations can be performed across tasks.

The task-switch customization methodology we propose achieves the determinism and efficiency of cooperative multitasking with the asynchronous and dynamic properties of preemptive multitasking. Application-specific information is utilized at context-switch time to preserve the live portion of the task state either through application-specific software routines or through register file remapping. In the first techniques compiler-generated software routines are used by the context switch mechanism to preserve only the minimal set of live registers during preemption in an application-specific manner. In the second methodology, the benefits of hardware and software approaches are combined to achieve the fast context switch of replicated register files by using compiler analysis of application-specific knowledge regarding live registers. The compiler renames the set of live registers into small groups of contiguous registers. The physical register file is extended with a small set of spare registers with limited mapping capabilities. At preemption time, the OS maps the small fraction of the register file containing live registers to a subset from the pool of registers assigned to capture the live registers of the preempting task at the moment when it has been previously suspended. The effect of separate register files per tasks is achieved with the hardware cost of slightly increased ($\leq$50%) register file.

3. STATE LIVENESS AND PREEMPTION DEFERRAL

The cost of task preemption is largely determined by Steps 1, 2, 3, and 4, as shown in Figure 1. Deciding which task to schedule for execution is the responsibility of the OS scheduler. In modern RTOS kernels, this part can be very fast since it takes only a few cycles to lookup a priority queue structure—an operation, which does not depend on the number of tasks in the system [Bovet and Cesati 2002]. The techniques outlined in this article are independent from the particularities of the task scheduler; the focus is on the mechanism of efficiently preserving and restoring the states of two tasks involved in the preemption operation.

The preempting process starts immediately after Step 2. Initially, the state of the preempted task is saved (Step 1). After that the scheduler spawns or resumes the new task by loading the context of the preempting task and eventually loads the PC with the program counter value for the new task. The switch back operation to $Task_1$ is similar but in reverse order. The context switch overhead is the sum of the instructions from Steps 1 through 4 where the processor resources are used for saving and restoring state instead of executing instructions from the application tasks. The context-switch response interval is the time between the beginning of Step 1, where the RTOS kernel starts saving the state, and the end of Step 2, where the first instruction of the preempting task is executed. The shorter this switch interval is, the more responsive the preempting task (and the system as a whole) is. This property is of extreme importance to many time-critical control application, where a suspended task is resumed due to an event from the environment and the processing of this event needs to start as soon as possible.

3.1 Register Liveness Analysis

The timer interrupt which triggers the preemption procedure occurs asynchronously from the application execution and thus can interrupt the task at arbitrary positions. This implies that the processor state actually utilized by the active task is unknown to the OS kernel. Such a knowledge is impossible to be conveyed to the OS scheduler from the compiler as it needs to be done for each instruction inside the application—a tremendous overhead which no real system can afford. Therefore, the OS kernel must be conservative in its assumption regarding the actual register utilization and, thus, save all the general-purpose registers. As mentioned earlier, in modern processor architectures, such as VLIW, this number could be easily near or above a hundred and result in a significant performance/power overhead during context-switch and deteriorated responsiveness.

If, however, the OS kernel is provided with extra intelligence regarding the actual usage of the processor state, only the live registers need to be saved as part of the task context. During compile-time, and especially after the register allocation phase, the compiler has a complete knowledge regarding register utilization. At each instruction position inside the application, the lists of assigned and free (dead) registers are available to the compiler as it is the register allocation of the compiler which allocates the physical registers to program variables.

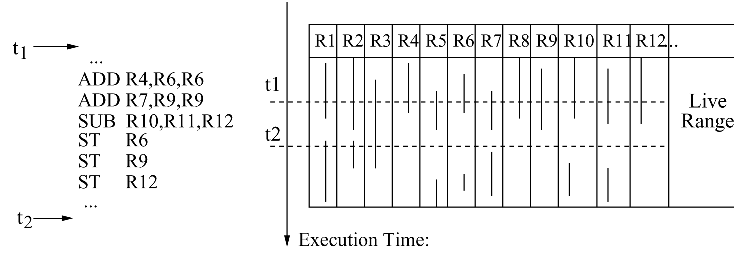

Fig. 3. Register live ranges and live state.

Figure 3 depicts an example of a code sequence including the register live intervals. The y-axis of the figure corresponds to the instruction sequence (cycles), while the x-axis represents all the general-purpose registers. The vertical lines for each register correspond to live intervals, which is defined as the time interval between two instructions during which this register is alive. The live interval starts with the register definition by the first instruction and ends with the last usage of that register before it is defined again by a subsequent instruction.

After its last usage and before its subsequent definition by another instruction this register is dead, and thus does not contribute to the task state during this interval. From the example assembly code in Figure 3, it can be seen that during the first *ADD* instruction register R_6 becomes alive (we assume that the destination register of the instruction is the third register). The first ST instruction is the last usage of register R_6 where it is saved to memory. Therefore, after this ST instruction R_6 is no longer alive and it need not be stored during context switch. Similarly, registers R_9 and R_{12} are no longer alive at time t_2. Consequently, for this example at point t_2 three fewer registers need to be saved as part of the task context. Consequently, it can be seen that for this example at time t_1 all registers from R_1 to R_{12} are alive. Therefore, if the timer interrupt happens at this moment and context-switch is required, all registers need to be saved. It is evident, however, that if a context-switch is to occur during t_2 instead, only the value of the three live registers, R_1, R_2, and R_3, need to be saved and subsequently restored when the task is resumed for execution later. The savings are quite significant as only 3 out of 32 or 64 registers need to be saved and later restored.

As the number of live and dead registers change at each instruction, it is practically impossible to provide the liveness information to the OS as it would require a tremendous amount of memory. However, if the live information for t_2 only is captured, and the preemption action which happened during t_1 is *postponed* to t_2, a very efficient context-switch can be performed. The time between t_1 and t_2 is spent in executing useful instructions from the preempted application task, which improves on the system throughput. Furthermore, since at time t_2 only a small fraction of the registers is to be saved/restored the response time compared to the general-purpose preemption approach is greatly improved. We present detailed evaluation results in Section 6.

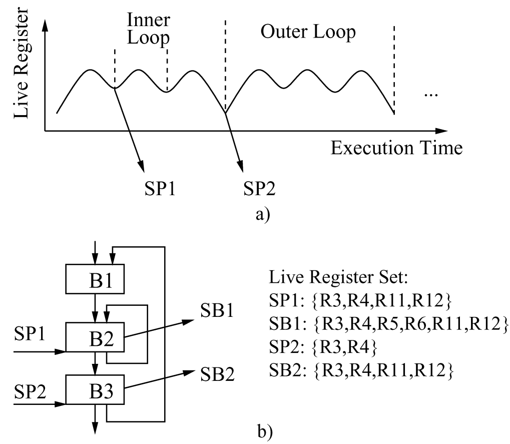

Fig. 4. Application "hotspot" with two switch points and blocks.

3.2 Switch-Points and Blocks

A major contribution of the proposed methodologies is the introduction of the concepts of the switch-point/switch-block and the mechanism of preemption deferral until such a point/block is reached in the program. The switch-points are points (instructions) in the program similar to t_2 from the example in Figure 3. The switch-points have the property that the set of live registers is minimal. It is noteworthy that these points correspond to particular program locations, which can be captured by a PC value and no extra instructions are introduced into the application code. The switch-blocks is a similar concept but instead of referring to a single instruction, it refers to an entire *basic block* from the application's control-data flow graph (CDFG). The set of live registers for each basic block, including the switch-blocks in particular, is defined as the union of the live registers for all the instructions within that basic block.

Figure 4 shows an example CDFG of an application hotspot consisting of a two-level loop-nest; the innermost loop consists of a single basic block, while the outermost features two basic blocks. As explained above, the number of live registers throughout the CDFG fluctuates and thus exhibit local minimum for some instructions. From Figure 4(a) it can be seen that one such minimum exists in the innermost loop and one in the outermost loop nest. These local minimums points correspond to two switch-points, denoted as SP_1 and SP_2 in the figure. The switch-blocks, denoted as SB_1 and SB_2, correspond to the basic blocks with minimal number of live registers. Typically these are the basic blocks in which the switch-points reside. An example set of live registers for these points and blocks is shown in Figure 4b.

Such a distribution of the switch-points and switch-blocks is representative for a typical loop. This is due to the fact that the scheduled loop, whether it is heavily unrolled or not, typically contains a number of load instructions that load the data to be computed from a number of arrays into a set of registers; it proceeds by the computation, and finishes up by storing the values in these registers back to memory. This typical sequence ensures that at the end of the

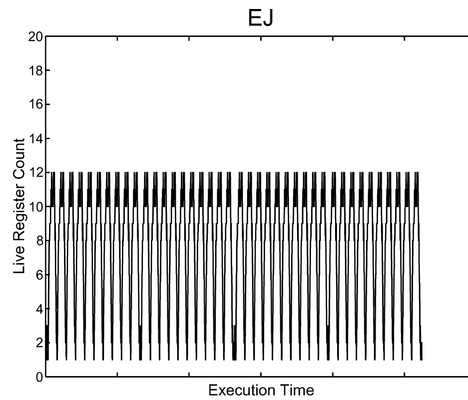

Fig. 5. Register liveness for EJ.

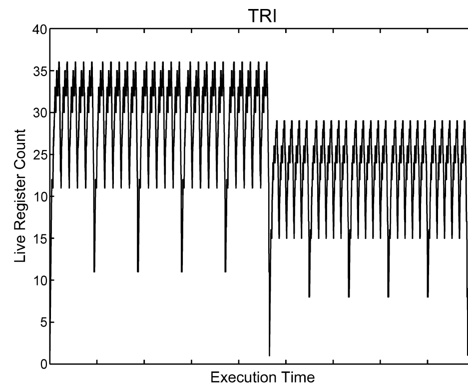

Fig. 6. Register liveness for TRI.

loop only a very few registers are alive and these are the registers carrying a few local variables, including the loop index registers. Consequently, efficient switch-points/blocks are easily identified at the end or at the very beginning of even tightly scheduled loop bodies. The graphs in Figures 5 and 6 confirm this supposition. These two graphs plot the number of live registers as a function of execution time for the numerical kernels *tri* and *ej* (tridiagonal matrix transformation, and extrapolated Jacoby transformation—two numerical kernels, which we use as part of our experimental setup). It is evident from this graph that very efficient switch points exist and are easily identifiable as the local minima in the number of live registers. A drop in the number of live registers towards the end of the loop body is quite common and is easily explained with the fact that by the end of the loop all computed values are stored in memory and their registers are no longer alive. These registers will be defined in the beginning of the next loop iteration where the next set of data will be loaded from memory.

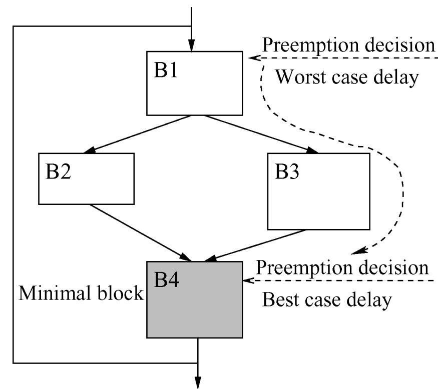

Fig. 7. Switch-points/blocks placement.

3.3 Live State Preservation

The process of context-switch is deferred and acted upon only at a switch-point or a switch-block. The decision of whether switch-points or switch-blocks are to be used is based on the trade-off between responsiveness and state amount that needs to be preserved. Clearly, the state amount is minimal at the switch-points and as such less than in the switch-blocks. However, switch-blocks are reachable slightly faster and in some cases can provide better responsiveness even though a slightly larger state needs to be preserved. Since the application hotspots are typically comprised of small amount of code executed iteratively, a very small number of switch points/blocks will be enough to service the context switch mechanism and achieve significant reductions in performance overhead and improve significantly the response interval for each task. As can be seen from the experimental results reported later in the article, for the majority of cases only the definition of a single switch point/blocks is enough to achieve these improvements.

Identifying the switch-points/blocks for the application hotspot CDFGs is the first phase of the proposed technique. These points/blocks are identified statically by the compiler subject that the amount of live registers is minimal. In order not to deteriorate the original response time, the distance between any instruction in the application hotspot to the nearest switch point/block in the dynamic execution flow must not exceed the overhead reduction of the context-switch. In other words, the number of cycles to the nearest switch point/block must be smaller than the number of cycles saved when performing the preemption at the switch point/block. In such a case, even though the preemption might be deferred with several cycles, the response time would be better compared to the general-purpose case. Of course, this is due to the fact that the proposed technique drastically reduces the number of registers that need to be saved and restored during context switch. Figure 7 illustrates this important point. Basic block B_4 has been determined to be the single switch-block for the application loop shown by its CDFG. The worst case in terms of preemption deferral corresponds to the situation where the preemption interrupt has occurred during the

first instruction of the loop. In this situation, the preemption is differed until the execution reaches the switch-block B_4 at which point a very fast task switch following one of the two techniques described in the following sections is utilized. Clearly, the best case is when the preemption interrupt occurs while the execution is inside the switch-block. In this case, the customized task switch is immediately executed for the switch-block B_4. The compiler identifies as many minimal points/blocks as needed in order to ensure that even with a worst-case preemption deferral, the net result is faster than general-purpose preemption. In our experiments we have observed that one minimal point/block per application hotspot is sufficient for the majority of cases.

Our experiments show that extremely efficient switch points/blocks exist even for applications with high register utilization due to large amount of ILP. Second, due to the limited number of switch points for each application hotspot, the information regarding live registers that need to be utilized by the RTOS kernel is minimal. The two proposed techniques, CCS and RMCS, efficiently solve this problem of transferring the application liveness information in to the OS and preserving/restoring the set of live registers with minimal cost. The CCS techniques uses custom-generates software switch routines for each switch-point/block. These routines are registered with the OS kernel when loading the application and invoked by the kernel. The RMCS techniques uses limited register file mapping to preserve the live state by simply re-mapping the few registers that carry the live state for the switch-point/block.

4. COMPILER-DRIVEN CONTEXT SWITCH

The compiler-driven context switch (CCS) technique relies on the compiler to synthesize a special pair of software routine for each switch-point/block. These routines preserve and restores exactly the minimal set of the live registers at that instruction or basic block. Since the register liveness information is available after the register allocation phase of the compiler, the switch points are determined at compile time. Finding the switch-points and switch-blocks does not introduce any overhead in the compiler, as it can be done simultaneously with the register allocation phase. As the registers are assigned to each instruction in the generated code, the switch points are dynamically identified as the points in the code with minimal number of live registers. The number of switch-points/blocks depends both on the size of the hotspots and the maximal number of switch-points/blocks which the hardware support allows. The size and structure of the hotspots will determine how many switch points are needed. This is driven by the constraint that a switch-point must be quickly reachable from any point within the hotspot code as explained in the previous section. As we have observed in our experiments, because of the typical small size of the application hotspots, one or two switch-points/blocks are usually enough for most of the applications to cover all the hotspots. In a subsequent section of this article, we will describe and analyze the required hardware support for the CCS technique; there we will show that the proposed approach can easily maintain tens of switch-points with minimal hardware and power cost, and no performance overhead.

4.1 Compiler and OS Support

Fundamentally, the information regarding the set of live register for each such switch-point needs to be conveyed to the RTOS kernel so that this information is utilized to minimize the task state that needs to be saved and restored. Traditionally, the RTOS kernel features a code which as a part of the context switch module saves the state of the preempted task, including all the general-purpose registers, and load the entire state of the preempting task. One possible approach would be to save a list of live register indices in some memory structure, such as the stack frame of the task, and have the RTOS kernel use these indices to save and restore only live registers. This approach can be implemented efficiently only if a special hardware, such as a simple DMA-like control unit is used to transfer the set of live registers to/from kernel memory. A software implementation would be quite inefficient as it has to read in the indices and decode them to actual register indices in a "switch"-like statement. The solution we propose through the CCS technique is software only, but instead of storing a set of live register indices and have the RTOS kernel use them, our method has the compiler generate the custom software code to be used to save and restore the set of live registers for each switch point. For instance, if only registers R_1 and R_5 are alive for a particular switch-point, the compiler will generate two special software routines. The first one will simply store R_1 and R_5 to an address inside the stack frame of the task, while the second would load these two registers from the task's stack frame. Prior to entering the hot-spot these two routines will be registered with the RTOS kernel and will be called back as a replacement of the general-purpose RTOS routines for storing and loading the entire register file.

Consequently, for each switch point and its corresponding live register set, one context saving and one context loading software routine is generated at compile time. They are implemented in a similar way as the functions inside the RTOS kernel; there is only store or load instructions in them—no caller save or callee save mechanisms are needed. The target registers are the live register for that particular switch-point/block. The entry addresses of these subroutines are associated with the program counter of the switch point and are registered with the RTOS kernel and the hardware support prior to entering the application hotspot. In order to avoid any security issue, these routines even though invoked from within the kernel code, where the context switch has been decided, will be executed at the privilege level of the preempted and the preempting tasks. Therefore, these routines can access only memory within the address space of the task and thus expose no security problems. The routines can be executed within the task address space since they use the memory of the task to store the live registers. Clearly, the code of these custom routines is added to the application code. However, these switch routines are very small as they consist of a single load/store instruction per live register. As shown in our experimental results, the number of live registers is extremely small (within 5–15 for most of the cases).

Prior to each hotspot, the compiler would insert a small setup code the purpose of which is to set the processor into a mode of low-cost task switch and

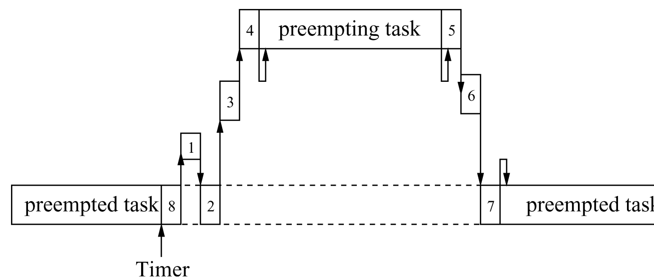

Fig. 8. Structure of the CCS live state preservation.

register the call-back functions for the switch points within the hotspot to the operation system kernel. During the preemption phase and the resume phase at runtime, the operation system will use these functions for each switch point to perform the saving and loading of the live registers.

With the introduced concepts of deferred task switch and switch-points/blocks, the preemption interrupt signal from the timer and the actual task switch become decoupled.[1] The structure and sequences of events of the proposed approach are depicted in Figure 8. When the timer interrupt for preemption occurs, it is registered with the interrupt controller but the execution of the preempted task is continued until the switch-point is encountered. Step 8 corresponds to the execution of the preempted application code before reaching a switch-point. As the switch-points are encountered dynamically quite often, this step is rather short; nonetheless it includes the execution of actual application code and is, thus, a useful computation related to the task at hand. While executing the several instructions from the preempted task leading to the switch-point, a special hardware mechanism is activated to identify the occurrence of the switch-point and trigger a signal when this happens. Step 1 is performed at the moment when the switch-point is encountered. The timer interrupt is now acted upon and the RTOS kernel is invoked. This step is identical to the one in the original general-purpose context switch mechanism as was shown in Figure 1. Here, the RTOS kernel decides whether a task switch is warranted. If the RTOS scheduler decides that a task switch is needed, the custom software routine generated by the compiler to save the live registers is executed; this corresponds to Step 2. Step 3 is the part of the RTOS scheduler code, which decided which task should be activated for execution. And finally, Step 4, which is the custom code for loading the state of the preempting task is executed. After this routine, the control is transferred to the saved PC of the preempting task. An identical sequence of steps is followed when the new task is preempted in turn from other task or from the original task. The benefits of the proposed approach stem from the fact that Steps 2 and 4 are much shorter

[1] It is possible, however, in order to minimize the overhead of taking the timer interrupt to program the interrupt controller to defer taking this interrupt only when a switch-point/block is hit. In this way, Step 1 will be executed just before Step 2 and any pipeline flushes associated with taking an interrupt, if present, will be eliminated.

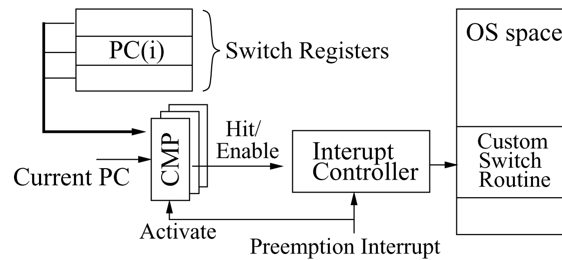

Fig. 9. CCS hardware support.

and faster than their corresponding steps in the general-purpose task switch mechanism, where the entire register file is saved and restored.

4.2 Hardware Support

The purpose of the introduced hardware support is to capture and dynamically identify the switch-points/blocks. This is the only hardware support required for the CCS technique. Since switch-points/blocks are defined within the application hotspots, inserting instructions to enable/disable preemption interrupts (in order to completely eliminate the hardware support) would result in nontrivial performance overhead. These instructions would have to be executed at each loop iteration, since preemption interrupts (either timer interrupt or external events) are asynchronous with respect to the program code and cannot statically predict at what time during loop execution will occur. The scheme that we propose, instead, does not introduce any new instructions within the application hotspots. Furthermore, the proposed methodology supports any interrupts that result in task preemption, such as timer, special external interrupts triggering specific tasks, etc. Any such interruption would trigger the detection hardware and be deferred until the next switch-point/block.

Clearly, the switch-points are uniquely identified with the address of the corresponding instruction in the application hotspot, while the switch-blocks are defined by their start and end address. Here, we outline the hardware support for identifying switch-points. The hardware mechanism required for switch-blocks is almost identical—toward the end of this section, we explain the difference. The addresses of the switch-points defined for the particular application hotspot are stored in a set of special register, which we refer to as *switch registers*. Once a preemption interrupt/request occurs, the current PC must be compared to the each of the switch registers at each clock cycle. When there is a match, a signal triggering a switch-point hit which enables the preemption procedure and initiates the low-cost context-switch mechanism at that point.

The structure of the introduced hardware is shown in Figure 9. The addresses of the switch points are stored into a set of switch registers. This is performed by a code inserted by the compiler prior to entering the application hotspot. These few instructions are executed only once prior to entering the hotspot. As we have explained earlier and will show in the next section, one or at most

two switch-points are typically enough per application hotspot for the majority of the applications. Therefore, the area overhead of the introduced hardware support is extremely minimal, as it contains only several switch registers, each 32-bit wide, and a set of comparators for the parallel check with the set of active switch registers. It is worth noting that these registers become part of the task state and need to be saved and restored on context switch. We account for these extra cycles in our experimental results. In steady state, when the program executes and no preemption interrupt has been observed, this hardware is disabled. At the moment when a preemption interrupt (timer or other such interrupt) occurs, the circuit is activated and the comparison between the PC and the switch registers is performed from this cycle until a match is found. The preemption is deferred since the interrupt controller is disabled at that moment and the program execution continues. This is the time period that coincides with Step 8 in Figure 8. Consequently, the power overhead of this activity is negligently small as this comparison is performed only for several cycles until the switch point is reached. The index of the switch register that matches is provided to the RTOS kernel in order to inform it which switch point has been reached so that the appropriate custom routines for saving the task state are executed.

To support switch-blocks, an additional range comparator is required. A range (interval) check is performed only once when the timer interrupt occurs in order to check whether the PC is currently within the starting and ending address of the switch-block. If a match is detected, the signal triggering the switch-block hit is set. Otherwise, the task execution continues and the detection of the switch-block becomes completely identical to the detection of a switch-point by using the starting address of the switch-block, which is loaded into a switch register.

5. REGISTER-MAPPED CONTEXT SWITCH

The CCS approach still requires several execution cycles in order to preserve the set of live registers for the switch-point/block. The register-mapped context switch (RMCS) technique, outlined in this section, aims at eliminating even these extra cycles by using an additional hardware support. This techniques requires limited mapping capabilities of the register file so that a small part of the register file address space (the first several registers) is virtualized and can be remapped at context switch. Compile-time register live analysis followed by a register renaming step actively “packs” the set of live registers into a set of contiguous registers within the mappable parts of the register file. At context-switch time, the OS exploits the mapping capabilities of the register file to map the set of registers, which are alive during the time of preemption.

Clearly, the RMCS technique requires more hardware support in the form of extra physical registers and mapping capabilities to a part of the register file. The hardware support required to detect the occurrence of a switch-point or a switch-block, as described in Section 4.2, is required as well, since the actual process of preemption is deferred to such a point or a block. Fundamentally, the only difference between the CCS and the RMCS is in the way they preserve the

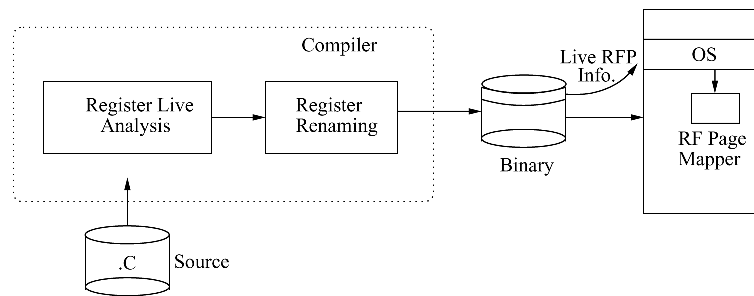

Fig. 10. RMCS methodology functional overview.

minimal live set at the switch-points/blocks. While the CCS technique explicitly saves and restores this set of registers, the RMCS approach keeps the live set in the register file by quickly remapping the address space of the register file where the live registers reside. In effect the live register of the preempted task are being hidden, while the live registers of the preempting task are mapped back (mounted) to the register file space which is ISA-visible. As this remapping can be achieved in a single clock cycle, the task switch procedure is almost instantaneous. The only small delay incurred is the delay of deferring the preemption until a switch-point/block is reached.

5.1 Compiler and OS Support

At compile-time the control and data flow graph (CDFG) is being analyzed with respect to register liveness information at basic-block and single instruction levels when the switch-points/blocks are identified. An important compiler phase that needs to be executed for the RMCS technique is that the minimal set of live registers for the switch-points or blocks is subsequently renamed (by the compiler) into consecutive registers residing in the low-addresses of the register file. This compile-time register renaming phase has no impact on the performance as it does not introduce any extra spill/fill code. The register file is extended to contain a pool of extra (spare) set of registers, which can be efficiently and rapidly (in one cycle) mapped into the ISA-visible register file address space. In order to further control and minimize the hardware cost for the remapping, the register file address space is partitioned into small register file pages (RFP). By mapping the first several RFPs only into a pool of spare register pages, the context switch procedure is reduced to a single cycle remapping event for all the cases where the small set of live registers is accommodated within these pages. Figure 10 depicts the design flow and the major steps in applying the RMCS technique.

Figure 11 illustrates the organization of the proposed mapped register file. The first several RFPs are mappable through a simple hardware block, which is described in details in the next subsection. For our experiments we have considered RFPs of size 8 and have allowed for only the first four RFPs from the register file address space to be mapped. In this way, different physical register pages can be mapped to the ISA-visible register address space. The unmapped pages are not visible to the application code.

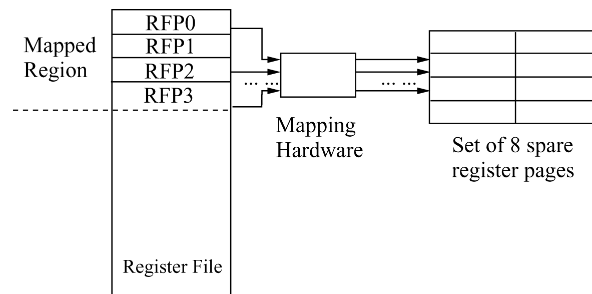

Fig. 11. Mapped register file organization.

The size in registers of the RFPs does not impact the physical organization of the pool of spare registers. As our mapping hardware simply replaces the most significant bits from the register address with an offset within the spare register pool, the size of the RFPs can be easily changed without any modifications to the table of extra registers. This hardware implementation also provides the ability to map any of the spare RFPs to any page from the first four in the register address space.

At runtime when a preemption event occurs, the special hardware defers the preemption until a switch-point or a block is reached for which the OS is aware about its live register RFPs. Subsequently only these few pages are mapped by executing a single instruction which controls the special register that defines the mapping. As the switch blocks or instructions have a minimal number of live registers, the number of RFPs that need to be mapped is minimal and, as shown in our experimental results, can be achieved for the majority of cases when running several tasks in parallel. It is demonstrated that a small sized pool of extra register pages is enough to simultaneously map the set of live registers for several parallel tasks. The effect of the proposed technique is identical to the effect achieved by dedicating a separate register file to each task and simply switching between these register files during context switch.

During task load-time, the OS can estimate whether the set of live register pages for all the current tasks can be accommodated within the pool of mappable RFPs. If not, depending on the task priority, the noncritical tasks live pages can be preserved in memory during preemption and not occupy register pages from the pool. For these tasks, the OS saves (and later restores) only the RFPs containing the live registers of the tasks. It is clear that for this no application-specific routines are needed as in CCS case—the OS only needs to know which RFPs to save/restore for each hotspot and switch-point/block for that task. In such a scenario where too many parallel tasks coexist, the OS would assign the most critical ones to nonswappable register pages in the pool and still take a full advantage of the proposed technique for the time critical tasks.

5.2 Hardware Support

Hardware support is required for two components of the proposed methodology. First, an RFP mapping hardware is needed, which would enable the

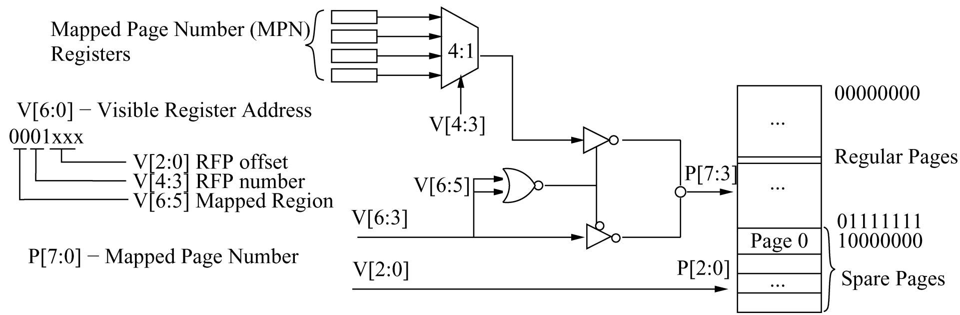

Fig. 12. Mapped register file architecture.

mapping of the first several RFPs in the register address space to be mapped to a pool of extra register pages. The second important hardware block is the module that detects the request for preemption in terms of timer interrupt or external asynchronous interrupt and subsequently defers the preemption point until a minimal block or a minimal point is reached. This hardware module was already described in Section 4.2, since it is required for the CCS technique. In this section of the paper we will focus on the mapped register file organization.

Figure 12 presents the hardware architecture of the mapped register file. The presented organization assumes 128-entry base register file (ISA-visible) with RFPs of size 8; the first four RFPs are mapped to a pool of extra register pages. The size of the mapped pages as well as their number can be easily changed—such a change can also be performed at runtime with only a very small modification to the hardware we outline below. The ISA-visible register indices corresponding to the first four register pages can be mapped to either the baseline register file (their normal location) or mapped to the pool of extra register pages, which is a small additional register file. The mapping is implemented by replacing the four most significant bits from the ISA-visible register index with a new 4-bit value (or fewer bits, depending on the size of the pool) that selects a register page from the extra pool. The pool of spare registers can be easily implemented as an additional small register file. As shown in our experimental results, for all practical purposes the size of this additional set of register is within 50% of the basic register file. The 4:1 multiplexer in the figure is responsible for replacing the 4-bit register page number with the 4-bit physical page number. The four 4-bit mapped page number (MPN) registers are written by the OS and define the mapping of the current task. Fundamentally, this set of four MPN registers defines the current mapping for the first four register pages from the ISA-visible register space. The values stored in the four MPNs are analogous to the physical frame numbers used in traditional virtual memory to represent the address of the physical memory page. In our experiments, we have evaluated both 64-entry and 128-entry register file, which are typical for modern VLIW processors.

The size of the extra register pool determines the number of parallel tasks that can benefit from the proposed technique. If the set of all live registers

from the most demanding minimal block in each task can be accommodated within the extra pool, then no saving/restoring of register is needed for context switch for that group of tasks. The only action required by the OS context-switch handler is to replace the values of the four MPN registers. This can be accomplished with a single instructions as the four MPN registers consist of 16 bits total. When the extra pool of registers cannot accommodate all the live pages from the tasks, some of the tasks would be assigned to four pages from the pool, which would be preserved during context-switch. It is worth noting that at this step the tasks with highest demands for responsiveness can be placed in the extra register pool and utilize the proposed RMCS technique, while the live pages for the less demanding task assigned to the base register file and (only they) preserved at context-switch.

Overhead Analysis. The area overhead of the introduced hardware consists of the pool of extra registers, the mapping logic, and the preemption deferral logic. In our experiments, we show that 25% or 50% extra registers is sufficient for achieving zero-cost context switch for several parallel tasks. The mapping logic is fairly small as it constitutes of a 4-to-1 multiplexer, four MPN registers with a total volume of 16 bits and a few gates. The preemption deferral logic consists of a range comparator and a value comparator per minimal block. The silicon area needed for this is minimal compared to the area of modern pipelined embedded processor with instruction and data caches. In terms of power overhead, the preemption deferral hardware is active only during the few cycles between an interrupt and context switch. Only the 4-to-1 multiplexer is activated during a regular register file access; its power, however, is order of magnitudes smaller than the power needed by a baseline register file only. The minimal delay of the 4-to-1 multiplexer is introduced in the register access path, even though it can be mostly overlapped with the address decoder logic.

6. EXPERIMENTAL RESULTS

In evaluating the proposed technique, we have performed a quantitative analysis and comparison of baseline general context switch scheme and the proposed application-specific context switch mechanism. The evaluation is performed with the VLIW Example–VEX package [Fisher et al. 2005], which is developed and provided by HP research labs. It includes a state-of-the-art optimizing VLIW compiler and a simulator tool-chain. The VLIW processor core can be configured into various architectures including multiple clusters, register files, and functional units. Each cluster is configured to have two register files, four integer ALUs, two 16×32-bit multiply units, and a data cache port. The cluster can issue up to four operations per instruction. The register set for each cluster consists of 64 general-purpose 32-bit registers. The simulator that comes with VEX is a compiled simulator. For the purpose of simulation, a C code for a custom VEX simulator is generated for the application program. Each VLIW instruction is executed as a call to a function that implements the functionality of the operations within the simulated instruction packet. The number of executed cycles including pipeline stalls and cache misses is maintained captured and reported.

We have evaluated two baseline register files, one consisting of 64 registers and one of 128 registers. We have assumed that the first four pages (each of eight registers) in the register file are mappable for the purpose of the RMCS technique. To consider the effect of aggressive VLIW compiler optimizations on the proposed methodology, we have included two compiler setups: one where the applications are compiled with heavy loop unrolling and trace scheduling—we refer to this option as an aggressive optimization; the second optimization setup includes all scalar optimizations but no loop unrolling—we refer to this as a scalar-only optimization.

By instrumenting the compiled simulator for each benchmark, we mark the switch points and model the behavior of the hardware and the RTOS kernel when a switch point is reached. The timer interrupt is modeled by introducing a counter which keeps track of the executed cycles in a way similar to a hardware timer module. When a certain number of cycles is reached, we simulate the occurrence of a context switch. Since, in our study, we are interested in the responsiveness of the context switch procedure, we measure the delay between the timer interrupt signaling the preemption and the execution of the first instruction from the preempting task. This delay is determined by the preemption deferral interval and the actual cost in terms of cycle for preserving and loading the state of the two participating tasks. These are the two factors that we take into account in our study.

Prior to instrumenting the compiled simulator, we compile the benchmark application, and the assembly code for the application hotspots is parsed by a separate script which identifies the register live ranges. The traditional algorithm for this analysis, as described in Aho et al. [1986], is used. In a subsequent step, we identify the switch-points and the switch-blocks as the positions/basic-blocks in the application hotspots, where the number of live registers is minimal. We report on the delay and task switch cost reductions for both switch-points and switch-blocks in order to analyze the advantages and disadvantages of both approaches. The baseline architectures constitute a single-cluster and dual-cluster cores. In the single-cluster case, each preemption is assumed to involve 64 store instructions and each resumption has the same amount of load instructions. For the dual-cluster architecture, the registers saved are doubled which results to a total of two groups of 128 instructions. For the CCS approach, we have accounted for the cycles needed by the custom switch routines to execute the context switch. Similarly to the baseline organization, we have assumed a single load/store unit per cluster. Consequently, based on the number of live registers for each switch-point/block, we have accounted for the corresponding execution cycles introduced by the custom switch routines.

In our experimental study, we have utilized two groups of benchmarks. The computation kernel group includes matrix multiplication (mmul), extrapolated jacoby (ej) method, the LU matrix decomposition (lu), and the TRI triangular matrix conversion (tri); these kernels manipulate large matrices and are extensively used in many algorithms. The application group includes the *2D-DCT*–discrete cosine transform that are widely used in many image and video processing applications, the *adpcm* and the *g721* speech coders from

Table I. Benchmarks Characteristics

	ej	lu	tri	mmul	2d-dct	adp	g721	sha	sus
# of h-s	1	2	2	1	2	1	1	5	1
Freq.(%)	100	49,51	54,46	100	49,51	100	100	19,21,20,20,20	100
# SP/SB	1	1,1	1,1	1	1,1	1	5	1,1,1,1,1	1

Table II. Characteristics and Response Reductions with Aggressive Compiler Optimizations for 64-Entry Register File

	ej	lu	tri	mmul	2d-dct	adp	g721	sha	sus
Live regs. SP/SB	2/ 8	20/22, 20/27	21/24, 15/17	11/17	14/23, 12/24	11/ 11	{3/5,7/7 8/8,9/9, 12/14}	5/20,11/18, 11/14, 11/18,10/13	20/ 20
CCS state red.%	97/ 89	72/69, 72/63	71/67, 79/76	85/76	81/68, 83/67	85/ 85	{96/93, 90/90,89/89, 88/88,83/81}	93/72, 85/75,85/81, 85/75,86/82	72/ 72
Avg. def. (cycl)	16/ 10	24/23, 28/23	33/31, 30/29	18/15	15/4, 19/8	21/ 21	36/32	22/13, 13/2,15/12, 13/2,15/12	24/ 24
Worst def. (cycl)	32/ 25	47/46, 54/49	65/63, 60/58	35/30	28/14, 37/24	42/ 41	49/45	42/33, 25/10,29/26, 25/10,29/26	48/ 47
RMCS resp. red.%	88/ 92	81/82, 78/82	74/76, 77/77	86/90	88/97, 85/94	84/ 84	72/75	83/90,90/98, 88/91, 90/98,88/91	81/ 81
CCS resp. red.%	84/ 78	50/48, 47/38	41/40, 53/52	69/63	66/57, 66/53	66/ 66	59/61	75/55,73/68, 71/68, 73/68,73/70	50/ 50

MediaBench [Lee et al. 1997], the *SHA* hash algorithm, and the *susan* image recognition program from MiBench [Guthaus et al. 2001].

Table I reports the structure of each application benchmark. The first row shows the number of hotspots, while the second row reports the execution frequency of each hotspot. The last row of the table shows the number of switch-points and switch-blocks for each hotspot. In the register live analysis for each application, we have found out that the switch-block always contains the switch-point. It is also always the case that the switch-block is always the basic block at the bottom of the hotspot CDFG. As explained earlier, this can be easily explained (and anticipated by us) by the fact that the registers carrying temporary variables as well as data items loaded and stored in memory are dead towards the end of the CDFG and only the registers carrying live variables across the loop iterations are alive.

Tables II, III, IV, and V show the achieved results. The four baseline architectures correspond to 64 and 128 entry register files, each with aggressive and scalar-only compiler optimizations. The aggressive optimizations include very heavy loop unrolling coupled with instruction scheduling. The scalar-only optimizations include no loop unrolling but all the basic optimizations including scheduling. The second row reports the number of live registers at the switch-points and the switch-blocks for all the benchmarks. This data is represented as a pair of number, the first corresponding to a switch-point and the second

Table III. Characteristics and Response Reductions with Aggressive Compiler Optimizations for 128-Entry Register File

	ej	lu	tri	mmul	2d-dct	adp	g721	sha	sus
Live regs. SP/SB	3/ 16	27/28, 20/38	29/33, 20/24	15/22	20/42, 18/40	15/ 15	{3/5,7/7 8/8,9/9, 12/14}	6/16,11/20, 24/24, 26 11/18,10/13	21/
CCS state red.%	98/ 89	81/81, 86/74	80/77, 86/83	90/85	86/71, 88/72	90/ 90	{98/97, 95/95,94/94, 94/94,92/90}	96/89, 92/86,83/83, 92/88,93/91	85/ 82
Avg. def. (cycl)	15/ 3	28/27, 28/24	40/36, 38/36	22/18	16/8, 20/3	24/ 24	49/45	29/23, 28/9,30/11, 28/11,30/6	24/ 24
Worst def. (cycl)	28/ 11	55/53, 55/51	78/74, 75/73	42/39	30/22, 38/15	48/ 47	63/59	56/51, 54/31,59/26, 54/35,59/26	49/ 44
RMCS resp. red.%	88/ 98	79/79, 78/81	69/72, 70/72	83/86	88/94, 84/98	81/ 81	62/65	77/82,78/93, 77/95, 78/91,77/95	80/ 80
CCS resp. red%	86/ 83	57/56, 63/51	46/53, 55/53	71/67	72/59, 70/63	70/ 70	56/59	73/67,70/72, 58/71, 70/72,69/72	64/ 60

Table IV. Characteristics and Response Reductions with Scalar-Only Compiler Optimizations; 64-Entry Register File

	ej	lu	tri	mmul	2d-dct	adp	g721	sha	sus
Live regs. SP/SB	3/ 14	18/20, 18/19	13/24, 10/19	9/13	13/15, 11/13	10/ 10	{3/5,7/7 8/8,9/9, 12/14}	6/6,5/9, 10/14, 10/15,9/13	20/ 20
CCS state red.%	96/ 81	75/72, 75/74	82/67, 86/74	88/82	82/79, 85/82	86/ 86	{96/93, 90/90,89/89, 88/88,83/81}	92/92, 93/88,86/81, 86/79,88/82	72/ 72
Avg. def. (cycl)	18/ 1	3/1, 3/1	8/1, 8/1	4/2	3/1, 3/1	14/ 14	40/36	2/2, 5/2,4/2, 4/2,4/2	14/ 13
Worst def. (cycl)	34/ 7	5/3, 5/3	15/5, 15/5	7/4	5/3, 5/3	28/ 27	57/53	4/3, 9/6,7/4, 7/4,7/4	27/ 25
RMCS resp. red.%	86/ 99	98/99, 98/99	94/99, 94/99	97/98	98/99, 98/99	89/ 89	69/72	98/98,96/98, 97/98, 97/98,97/98	89/ 90
CCS resp. red.%	81/ 75	70/67, 70/69	73/60, 78/68	83/78	77/75, 80/78	73/ 73	56/58	89/89,88/84, 81/77, 81/75,83/78	58/ 59

to a switch-block. It is noteworthy that this number directly corresponds to the number of execution cycles taken by the custom switch routines. For the single-cluster machine of 64 registers we have assumed one load/store unit, while for the dual-cluster machine with 128 registers two load/store units are assumed. Therefore, for the single-cluster organization, the number of cycles taken by the switch routines is identical to the number of live registers, while for the double-cluster, the number of cycles is equal to the half of the number of live registers. The next row reports the reduction in state (in percentage) that is achieved by

Table V. Characteristics and Response Reductions with Scalar-Only Compiler Optimizations; 128-Entry Register File

	ej	lu	tri	mmul	2d-dct	adp	g721	sha	sus
Live regs. SP/SB	4/ 17	18/20, 18/19	13/24, 10/20	9/13	13/15, 11/13	11/ 11	{3/5,7/7 8/8,9/9, 12/14}	8/8,5/9, 10/15, 10/14,9/13	20/ 20
CCS state red.%	97/ 88	88/86, 88/87	91/83, 93/86	94/91	91/90, 92/91	92/ 92	{98/97, 95/95,94/94, 94/94,92/90}	94/94, 97/94,93/90, 93/89,94/91	86/ 86
Avg. def. (cycl)	17/ 1	3/1, 3/1	8/1, 8/1	4/2	3/1, 3/1	14/ 14	54/49	2/1, 5/1,5/1, 5/1,5/1	11/ 11
Worst def. (cycl)	33/ 7	5/3, 5/3	15/5, 15/5	7/4	5/3, 5/3	28/ 27	71/67	3/2, 10/3,10/4, 9/4,9/4	21/ 20
RMCS resp. red%	87/ 99	98/99, 98/99	94/99, 94/99	97/98	98/99, 98/99	89/ 89	58/62	98/99,96/99, 96/99, 96/99,96/99	91/ 91
CCS resp. red.%	84/ 84	84/83, 84/84	84/79, 86/82	90/88	88/87, 89/88	80/ 80	52/55	92/92,92/91, 88/88, 88/86,89/88	76/ 76

storing and restoring the live registers only by the CCS methodology. Similarly, the data for switch-points and switch-blocks is shown for all the benchmarks and their application hotspots. The next pair of rows reports the average and worst preemption deferrals (in cycles) for both switch-points and switch-blocks. These numbers represent the number of cycles from the moment a preemption interrupt occurs to the moment of reaching the switch-point/block. The deferral delay is independent from the type of technique used to preserve the context (CCS or RMCS). Clearly, the deferral is smaller for switch-blocks as they are more quickly reachable on average than switch-points. The last two rows in the tables report the reductions (in percentage) achieved by the RMCS and the CCS methodologies. The baseline architecture is the general-purpose mechanism for storing the entire register file of the preempted task and loading the register file associated with the preempting task. For the CCS technique, this reduction takes into account the execution cycles needed by the custom switch routines, which, as explained above, are directly proportional to the number of live register for that switch-point/block. For the RMCS technique, a single-cycle switch to remap the appropriate RFPs is assumed.

It is worth noting that not only are the two preemption mechanisms significantly faster, but they are also useful instructions are being executed from the preempted task during the deferral interval. This is in contrast with the general-purpose mechanisms where it is not only significantly slower, but is also the case that the delay is pure overhead as only system software instructions are being executed.

As is evident from the tables, when more issue bandwidth is available with the dual-cluster machine, the compiler is more aggressive in unrolling the loops in order to uncover and schedule parallel instructions. The loop body increases due to the larger unrolling factor, which in turn leads to an increase in the

Table VI. Live Pages Exceeding Page Pool; Aggressive Optimizations

		A2	B2	A3	B3	A4	B4
25%	one-cluster	0/0	0/0	1/2	0/2	3/5	3/5
	two-clusters	0/0	0/0	0/4	0/3	3/7	2/7
50%	one-cluster	0/0	0/0	0/0	0/0	1/3	1/3
	two-clusters	0/0	0/0	0/0	0/0	0/3	0/3
75%	one-cluster	0/0	0/0	0/0	0/0	0/1	0/1
	two-clusters	0/0	0/0	0/0	0/0	0/0	0/0

Table VII. Live Pages Exceeding Page Pool, Scalar-Only Optimizations

		A2	B2	A3	B3	A4	B4
25%	one-cluster	0/0	0/0	0/2	0/0	2/4	3/3
	two-clusters	0/0	0/0	0/1	0/0	0/4	1/1
50%	one-cluster	0/0	0/0	0/0	0/0	0/2	1/1
	two-clusters	0/0	0/0	0/0	0/0	0/0	0/0
75%	one-cluster	0/0	0/0	0/0	0/0	0/0	0/0
	two-clusters	0/0	0/0	0/0	0/0	0/0	0/0

number of live registers throughout the loop body. However, for the minimal points/blocks, the number of live registers is only slightly increased and not doubled, due to their typical occurrence towards the beginning or the end of the loop body. Nonetheless, this increase is limited by the available ILP. Furthermore, due to the more load/store units available for this architecture, the actual reductions achieved by the CCS technique are higher compared to the single-cluster architecture; the slight increase in live registers at the switch-points/blocks is compensated by the ability to save and load pairs of registers simultaneously.

In order to evaluate the impact of the size of the pool of extra registers required by the RMCS technique, we have formed sets of multiple tasks for parallel execution. Tables VI and VII show the relation of spare registers pool size and the supported multiple task set for worst case scenarios—when for all the tasks the preemption occurs within the switch-points/blocks with largest number of live register pages. If all these live pages cannot be accommodated within the extra pool of register pages, the number of pages which exceed the pool must be saved and restored at context switch. As explained in the paper, only the nontime critical tasks can be handled in this way, while the rest of the tasks can use the pool of register pages and benefit from the proposed methodology. Tables VI and VII report the number of live register pages that exceed the pool for each set of tasks; the tables report for aggressive and scalar-only compiler optimizations, respectively. We have evaluated three cases for the size of the pool of extra registers: 25%, 50%, and 75% of the size of the baseline register file.

The first row in the tables shows the set name. Set A_i consists of the first i tasks from the computational kernels group; A2 = {EJ,LU}, A3 = {EJ, LU, TRI}, A4 = {EJ, LU, TRI, MMUL}. Sets B_i similarly cover the group of application benchmarks: B2 = {2D-DCT, ADPCM}, B3 = {2D-DCT, ADPCM, SHA}, B4 = {2D-DCT, ADPCM, SHA, SUSAN}. The three main rows contain the data for the cases of 25%, 50%, and 75% extra register in the pool of spare register pages in addition to the baseline register file. For both single- and dual-cluster machines we report the number of live register pages that need to

be saved/restored to memory because of insufficient registers in the extra register pool. The two reported numbers correspond to the cases of using switch-points and switch-blocks, respectively. It can be seen from this data that a 25% extra register pool completely covers all the two-task sets and some of the three-task sets, while 50% pool completely covers all the two-, three-, and some of the four-task sets. Even in the cases where not all the live pages can be covered by the RMCS technique, the critical tasks can be assigned to the spare pages and utilize the RMCS methodology, while for the remaining noncritical tasks, the context-switch time can still be reduced as only the registers in the live pages would be preserved and restored during task preemption.

7. CONCLUSIONS

We have presented a novel mechanism for rapid and low-cost context switch for real-time and control-dominated embedded systems. The general-purpose preemption process used in modern RTOS kernels is replaced with an application customizable one, where only the minimal state needed to resume the task is preserved either in memory or in a pool of extra physical registers. This is achieved through the active cooperation of compiler, microarchitecture, and RTOS kernel. During compile-time, special switch-points/blocks in the application code are identified where the number of live registers is minimal. The information regarding these points is captured by the hardware support and used to defer the preemption interrupt slightly, in order to minimize the large number of overhead cycle associated with the preemption code in the kernel. First, we have outlined the mostly software-based CCS technique for state preservation and restoration, where compiler-generated custom routines are registered with the kernel for each switch-point/block and are used as a dynamic custom replacement of the high-overhead general-purpose context save and restore routines. Additionally, we have proposed the hardware-based RMCS approach. The register file is efficiently virtualized and a subset of it mapped to an extra pool of physical registers; the context-switch procedure is reduced to a rapid remapping of the live register pages of the preempting task. The proposed methodology constitutes a cross-layer system customization which involves the compiler, the microarchitecture, and the RTOS kernel. It has been demonstrated by extensive experiments that significant reductions can be achieved in both the task switch overhead and the system response time.

REFERENCES

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Boston, MA.

ALBRECHT, C., HAGENAU, R., AND DORING, A. 2004. Cooperative software multithreading to enhance utilization of embedded processors for network applications. In *Proceedings of the 12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP'04)*, IEEE, Los Alamitos, CA, 300–307.

ARM LTD. ARM920T technical reference manual. ARM Ltd.

BAKER, T., SNYDER, J., AND WHALLEY, D. 1995. Fast context switches: Compiler and architectural support for preemptive scheduling. In *Microprocessors and Microsystems,* 35–42.

BARTHELMANN, V. 2002. Inter-task register-allocation for static operating systems. In *Proceedings of the Joint Conference of Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems (LCTES-SCOPES)*. ACM, New York, 149–154.

Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. 2005. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.* (Special Issue on Wireless Sensor Networks) *10*, 4, 563–579.

Bovet, D. and Cesati, M. 2002. *Understanding the Linux Kernel* 2nd Ed. O'Reilly, Sebastopol, CA.

Byrd, G. and Holliday, M. 1995. Multithreaded processor architectures. *IEEE Spectrum.*

Chandra, A., Adler, M., Goyal, P., and Shenoy, P. 2000. Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *Proceedings of the Symposium on Operating System Design and Implementation*. 45–58.

Dean, A. G. 2000. Software thread integration for hardware to software migration. Ph.D. thesis, Carnege Mellon University.

Dean, A. G. 2005. Software thread integration and synthesis for real-time applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE).* IEEE, Los Alamitos, CA, 68–69.

Faraboschi, P., Brown, G., Fisher, J., Desoli, G., and Homewood, F. 2000. Lx: A technology platform for customizable view embedded processing. In *Proceedings of the International Symposium on Computer Architecture (ISCA).* ACM, New York, 203–213.

Fisher, J., Faraboschi, P., and Young, C. 2005. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufman, New York, NY.

Guthaus, M., Ringenberg, J. S., Ernst, D., Austin, T., Mudge, T., and Brown, R. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual Workshop on Workload Characterization (WWC-4).* IEEE, Los Alamitos, CA, 3–14.

Hansson, H., Lawson, L., Bridal, O., Eriksson, C., Larsson, S., Lon, H., and Stromberg, M. 1997. Basement: An architecture and methodology for distributed automotive real-time systems. *IEEE Trans. on Comput. 46*, 9, 1016–1027.

Hill, J. and Culler, D. 2001. A wireless embedded sensor architecture for system-level optimization. Tech. rep. University of California, Berkeley.

Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., and Roussel, P. 2001. The microarchitecture of the pentium 4 processor. *Intel Tech. J*.

Intel Corporation. Intel XScale Microarchitecture. Intel Corporation.

Kessler, R. 1999. The alpha 21264 microprocessor. *IEEE Micro 19*, 1, 24–36.

Lee, C., Potkonjak, M., and Mangione-Smith, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO'30).* IEEE, Los Alamitos, CA, 330–335.

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. 2005. Tinyos: An operating system for wireless sensor networks. *Ambient Intelligence*, Springer-Verlag, Berlin, Germany.

Merten, M., Trick, A., and Barnes, R. 2001. An architectural framework for runtime optimization. *IEEE Trans. Comput. 50*, 6, 567–589.

Nieh, J. and Lam, M. S. 2003. A smart scheduler for multimedia applications. *ACM Trans. Comput. Syst. 21*, 2, 117–163.

Oehmke, D., Binkert, N., Mudge, T., and Reinhardt, S. 2005. How to fake 1000 registers. In *Proceedings of the 38th Annual International Symposium on Microarchitecture (MICRO'38),* IEEE, Los Alamitos, CA, 7–18.

Redstone, J., Eggers, S., and Levy, H. 2003. Mini-threads: Increasing tlp on small-scale smt processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*, IEEE, Los Alamitos, CA, 19–30.

Sastry, D. C. and Demirci, M. 1995. The qnx operating system. *Computer 28*, 11, 75–77.

Sherwood, T., Perelman, E., Sair, G. H. S., and Calder, B. 2003. Discovering and exploiting program phases. *IEEE Micro 23*, 6, 84–93.

Shivshankar, S., Vangara, S., and Dean, A. 2005. Balancing register pressure and context-switching delays in asti systems. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ACM, New York, 286–294.

WINDRIVER. VxWorks, http://www.windriver.com.

Received October 2007; revised April 2008; accepted July 2008