Rapid and Low-Cost Context-Switch through Embedded Processor Customization for Real-Time and Control Applications

Xiangrong Zhou
University of Maryland
College Park, USA
xrzhou@umd.edu

Peter Petrov
University of Maryland
College Park, USA
ppetrov@ece.umd.edu

ABSTRACT

In this paper, we present a methodology for low-cost and rapid context switch for multithreaded embedded processors with real-time guarantees. Context-switch, which involves saving and restoring the thread state, has constituted not only a large performance overhead for many multithreaded embedded systems, but also an obstacle creating a significant delay in the response time for many time-critical control applications. The proposed technique exploits application information extracted during compile time to make sure that only a minimal amount of thread state is saved and subsequently restored on preemption. The register liveness within the application inner loops is analyzed and a few points, referred to as switch *points, are identified where the program has minimal number of live registers. At run-time the preemption point is deferred to a switch point and the Real-Time Operating System (RTOS) kernel invokes a switch point specific code generated by the compiler to save and restore the thread state in a custom fashion. Through the utilization of these novel mechanisms, a drastic improvement on both performance and response time is achieved. The presented experimental results demonstrate the effectiveness of the proposed technique on a number of widely-used computational kernels and embedded applications.*

Categories and Subject Descriptors

C.1 [**Computer Systems Organization**]: Processor Architectures; D.4 [**Software**]: Operating Systems—*Process Management*

General Terms

Design, Experimentation, Performance

1. INTRODUCTION

The utilization of embedded processors for real-time and time-critical control applications have been growing rapidly in recent years. The modern automotive industry, for instance, has adopted this approach where tens to hundreds such processors are used throughout a single automobile. They are used for traction control, anti-lock break 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 phenomenon. At the same time, all this processing power needs to be achieved with extremely energy-efficient and low-cost embedded processors.

A typically System-On-Chip (SOC) implementation contains processor cores, I/O peripheral, and on-chip RAM/ROM. In order to improve the hardware utilization and to reduce the final product cost, multiple tasks are typically scheduled to run on a single processor. It has been shown that for many applications, where a data processing is combined with supporting a communication channel, sharing a processing core with DSP and control capabilities is much more efficient in terms of inter-process communication overhead as compared to having dedicated DSP and control processors. However, running multiple tasks on the same processing core requires some operating system (OS) support in order to perform efficient task scheduling.

General-purpose OSs are incapable of meeting the hard real-time requirements for such systems. 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. Real-time OS (RTOS) kernels [1] 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, etc. 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 multi-threading was introduced [2]. 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. However, due to stringent 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 very large in order to enable the compiler or software developer to exploit instruction level parallelism and maintain high instruction execu-

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

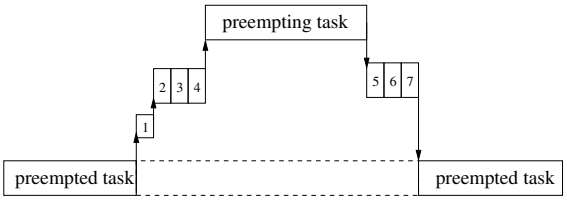

Figure 1: Context-switch in preemptive multitasking

tion throughput; a typical modern VLIW architecture [3] 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 total performance and throughput of the system 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 paper, we propose a novel technique which drastically minimizes this overhead and significantly improves both the total system throughput and the response time for each task.

The proposed low-cost context-switch mechanism is achieved through the active cooperation of compiler, microarchitecture, and operating system kernel. A general-purpose OS or RTOS conservatively saves and restores the entire content of the register file. This needs to be done 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.

In the proposed approach, the compiler identifies what is the minimal number of live registers that needs to be saved and provides custom software routines to the RTOS kernel. These routines have been synthesized by the compiler to save and restore only the live registers for a few *switch* points in the application inner loops and "hot-spot" regions. The switch points 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 special hardware support is introduces, which is programmed by software and captures the address of the switch points. These switch points, even though very few, are encountered extremely often during the program execution as they are fixed by the compiler at positions which are frequently executed and inside the application inner-loops. The preemption is subsequently deferred to such a switch point, where the OS kernel invokes the custom routine for the switch point to store the state and then invokes the custom switch routine for the task which execution is to be resumed.

There have been several approaches to improve context switch performance. In [4], the tasks are executed in a manner of run-to-completion. No task switch is allowed, and longer tasks are partitioned manually into multiple shorter ones. As pointed out in [5], such run-to-completion scheme can cause problems with meeting hard real-time constraints, as it is impossible to partition many tasks, which will result into this task occupying the CPU for a long time. In [6], the authors have proposed to integrate multiple threads into a single thread statically during compile time. Consequently, all the run-time overhead of thread context switch is eliminated.

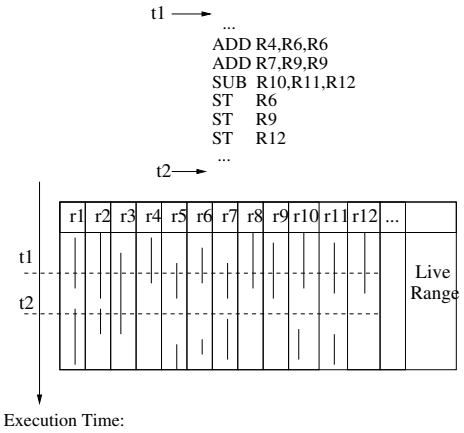

Figure 2: Register live ranges

Even though this approach has the advantage of avoiding nondeterministic context switch overheads, it creates an extra limitation on the dynamic behaviors of the embedded system as it requires that all preemption points are known during compile time. In [7], the authors have introduced a methodology, where the context switch is performed when no scratch registers are live thus saving time by not dealing with the few special scratch registers in the architecture. This approach, however, necessitates the utilization of register renaming hardware, a feature present only in superscalar processors due to its excessive hardware cost and power inefficiencies. The authors in [8] utilize and explore cooperative multi-threading instead of task preemption. Even though the switch overhead is reduced, the system responsiveness and dynamic features are highly limited as it must be relied on all the tasks to cooperate and be analyzed and compiled together.

2. FUNCTIONAL OVERVIEW

Preemptive multi-task systems rely on a timer to generate interrupts at regular time intervals. When such an interrupt occurs, the execution control is obtained by a kernel routine which purpose is to determine whether a task switch needs to be performed and, subsequently, to perform the context-switch. This process is depicted in Figure 1. At Step 1, which is invoked on a regular timer interrupt, the RTOS kernel decides whether a thread switch is needed. During Step 2 the state of the preempted task is saved, while Step 3 decides which new task should be switched to; during Step 4, the state of the preempting task is loaded. Switching back to the original task is performed in the same way.

Steps 1 and 3 depend on number of factors, such as task priorities, I/O availability, and task completion deadlines. In modern RTOS kernels, this part is very fast since it takes a very few cycles to lookup a priority queue structures which does not depend on the number of tasks in the system [9]. The issues concerned in this part of the routine are outside the scope of the paper. The preempting mechanism starts immediately after step 1, if it has been determined that context-switch is needed. Initially, the state of the preempted task is saved. After that the scheduler spawns the new task by loading the context of the preempting task and eventually load the PC register with the program counter value for the new task. The switch back process is similar but in reverse order. The context switch overhead is the sum of these instructions from steps 2 through 7 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 2 where the RTOS kernel starts saving

the state and the end of step 4 where the first instruction of the preempting task is executed. The shorter this switch interval, the more responsive the switch handler is. This property is of extreme importance to many time-critical control applications 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.

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 RTOS kernel. Such a knowledge is impossible to be conveyed to the 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 RTOS kernel must be conservative in its assumption regarding the actual register utilization and, thus, save all the general-purpose registers. As we mentioned before, in modern processor architectures, such as VLIW, this number could be easily above a hundred and lead to a significant performance/power overhead during context-switch.

If, however, the RTOS 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. Figure 2 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 depict the time interval between two instruction during which this register is alive. 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 above the figure it can be seen that during the first ADD instruction register R6 becomes alive (we assume that the destination register of the instruction is the third register). The the first ST instruction is the last usage of register R6 where it is saved to memory. Therefore, after this ST instruction R6 is no longer alive and it does not need to be stored on the context switch. Similarly, registers R9 and R12 are no longer alive at time t2. Consequently, for this example at point t2 three fewer registers need to be saved as part of the task context. Consequently, it can be seen that for this example that during time *t1*, all registers from *r1* to r12 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 t2 instead, only the value of the three live registers, r1, r2, and r3, need to be saved and subsequently restored when the task is resumed for execution later. The saving are quite significant as only 3 out of possibly 64 or 128 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 them to the kernel as the information would require a tremendous amount of memory volume. However, if the live information for t2 only is captured, and the preemption action which happened during t1 is *postponed* to t2, a very efficient context-switch can be performed. The time between t1 and t2 is spent in executing useful instructions from the preempted application task, which improves on the system throughput. Furthermore, since at time t2 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.

The major contribution of the technique proposed in this paper is

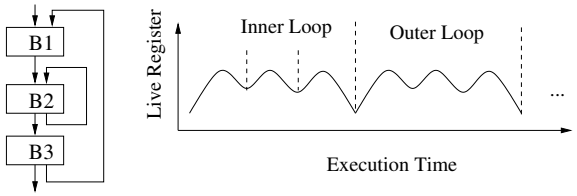

Figure 3: Application "hot-spot" with two *switch* points

the introduction of the concept of *switch* points in the application's hot-spots and the concept of *preemption deferral* until such a point is reached in the program. The switch points are points in the program similar to t2 in the example in Figure 2. These points have the property that the set of live registers is minimal. It is noteworthy that this 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 process of context-switch is deferred and acted upon only at a switch point. Since the application hot-spots, which typically correspond to inner-loops, are typically comprised of small amount of code executed iteratively, a very small number of switch points will be enough to service the context switch mechanism and achieve drastic 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 paper, in many cases, only the definition of a single switch point is enough to achieve these improvements.

The switch points are, consequently, defined as positions in the program hot-spots, where the number of live registers is minimal. In our experimental results we show that such extremely efficient switch points 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 hot-spot, the information regrading live registers that need to be utilized by the RTOS kernel is minimal. We propose a very efficient solution for this problem, which involves the synthesis by the compiler of special software routines associated to each switch point, which save and restore only the registers which are alive during these points. This is elaborated in details in the subsequent section of the paper.

In order not to deteriorate the original response time, the distance between any point in the application to the nearest switch point 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 points must be smaller than the number of cycles saved when performing the preemption at the switch point. 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.

A typical application usually spends most of its execution times in loops or functions, which are generally referred to as "hot-spots". In each such hot-spot, the instruction code is highly optimized. By applying the proposed technique to the hot-spot section of the code, all of the benefits from the proposed approach can be achieved with minimal additional hardware. Figure 3 depicts an example of such a hot-spot where the loop is nested and the innermost loop consists of a single basic block, while the outermost features two basic blocks. The figure on the right shows a possible record of the number of live registers for each instruction within the hot-spot. Two switch points have been identified, one at the end of basic block B2 in the inner loop and one in B3 in the outer loop. Such a distribution is representative for a typical loop, as we show the register liveness information for real application in Section 5 of the paper.

This is due to the fact that the scheduled loop, whether it is heavily unrolled or not, typically contains a number of load instructions which 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 loop only a very few registers are alive and these the registers carrying a few local variables, including the loop index registers. Consequently, efficient switch points are easily identified at the end or at the very beginning of even tightly scheduled loop bodies. Thus by selecting these application points as switch point, we can see that they will be encountered quite often and the number of live registers that need to be saved and later restored will be extremely reduced.

3. COMPILER AND OS SUPPORT

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 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 depends both on the size of the hot-spots and the maximal number of switch points which the hardware support allows. The size and structure of the hot-spots 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 hot-spot code. As we have observed in our experiments, because of the typical small size of the application hot-spots, one or two switch points have been enough for most of the applications to cover all the hot-spots. In the subsequent section of the paper we will describe and analyze the required hardware support; there we will show that the proposed approach can easily maintain tens of switch points with minimal hardware cost, and no performance and power overheads.

Additionally, the information regarding the set of live registers 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 code which purpose as a part of the context switch module is to save the state of the preempted task, including all the general-purpose registers, and load the entire state of the preempting task. One possible approach will 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 in a switch-statement to register indices. The solution we propose 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 R1 and R2 are live for a particular switch point, the compiler will generate two special software routines. The first one will simply store R1 and R2 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.

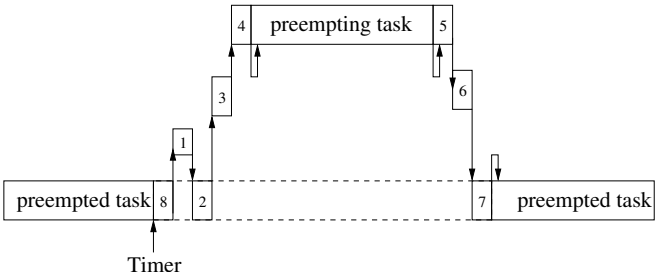

Figure 4: Structure of the proposed low-cost context-switch

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 similar way as the functions inside the RTOS kernel; there is only store or load instructions in them. No caller save or callee save are needed, and the functions can be called directly from within the kernel. The target registers are the live register for that particular switch points. The entry addresses of these subroutines are associated with the program counter of the switch point and are registered with the RTOS kernel prior to entering the application hot-spot. 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.

Prior to each hot-spot, 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 register the call-back functions for the switch points within the hot-spot 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, the preemption interrupt signal from the timer and the actual task switch become decoupled. The structure and sequences of events of the proposed approach are depicted in Figure 4. 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, which lead to the switch point, a special hardware is activated, which purpose is 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

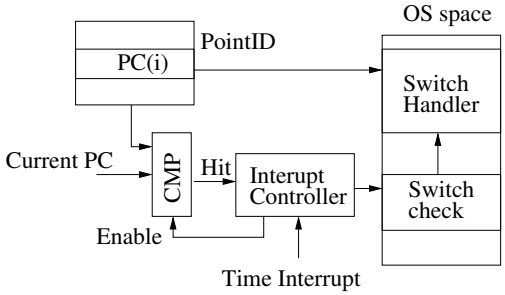

Figure 5: Hardware architecture

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

4. HARDWARE SUPPORT

The purpose of the introduced hardware support is to capture and dynamically identify the switch points. Clearly, the switch points are uniquely identified with the address of the corresponding instruction in the application hot-spot. This address is stored in a special register. Once a timer interrupt occurs, the PC must be compared to the address stored in the switch registered at each clock cycle. When there is a match, a signal triggers a hit which initiates the low-cost context switch mechanism at that switch point.

The structure of the introduced hardware is shown in Figure 5. The addresses of the switch points are stored into switch registers. This is performed by a code inserted by the compiler prior to entering the application hot-spot. These few instructions are executed only once before the hot-spot execution is started. As we have explained earlier and will show in the next section, one or at most two switch points are enough per application hot-spot 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 comparator. It is noteworthy, 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.

The comparison between the PC and the switch registers is performed only after the timer interrupt occurs. This is the time period which coincides with Step 8 in Figure 4. 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 which 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.

5. 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 [10], which is developed and provided by HP research labs. It includes a state-of-the-art optimizing VLIW compiler and simulator tool chain. The VLIW processor core can be configured into various architectures including multiple clusters,

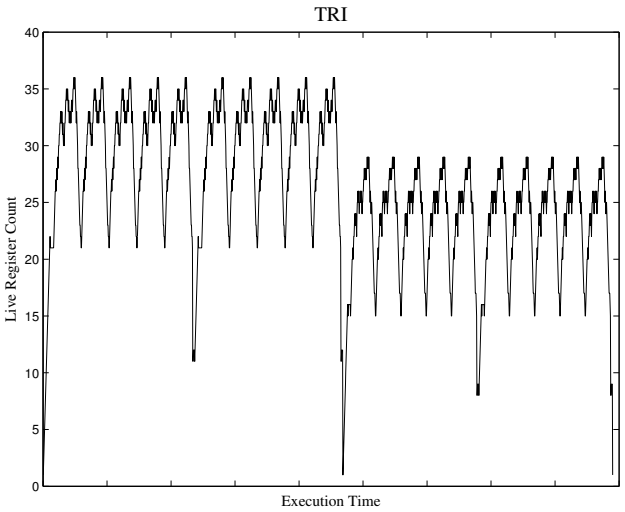

Figure 6: Register liveness for TRI

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 instructions. The register set for each cluster consists of 64 general-purpose 32-bit registers and 8 1-bit branch registers. The simulator that comes with VEX is a compiled simulator. For the compiler program, it generates a C program, which implements a custom simulator of the VEX execution for that particular program. Each VLIW instruction is executed as a call to function that implements the functionality of the operations within the simulated instruction. The number of executed cycles and pipeline stalls is maintained in global counters.

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 context switch.

Prior to instrumenting the compiled simulator, we compile the benchmark application (using optimization level O3), and the assembly code for the application hot-spots is parsed by a separate script which identifies the register live ranges. The traditional algorithm for this analysis, as described in [11], is used. In a subsequent step, we identify the switch points as the positions in the application hot-spots where the number of live registers is minimal. The baseline architectures constitute a one-cluster and two cluster cores. In the single-cluster case, each preemption involves 64+8=72 store instructions and each resumption has the same amount of load instructions. For the two-cluster architecture, the registers saved are doubled which results to a total of two groups of 144 instructions.

In our experimental study, we have utilized the following benchmarks. Extrapolated Jacoby (EJ) method, the LU decomposition, and the TRI (converting a matrix in a triangular form) are numerical kernels manipulating large matrices and are extensively used in many algorithms; the 2D-DCT is the discrete cosine transform, widely used in many image and video processing applications, while ADPCM and G.721 are speech processing benchmarks from Mediabench set [12].

Figure 6 shows the register liveness for the TRI kernel. It is evident from this graph that very efficient switch points exist and are easily identifiable as the local minimums in the number of live

	EJ	LU	TRI	2D-DCT	ADPCM	G721
hot-spots	1	2	2	3	1	1
Freq(%)	100	49,51	54,46	28,29,43	100	100
Switch points	1	1,1	1,1	1,1,1	1	5
Live Regs	2	19,18	21,15	7,3,3	10	10,10,10,10,7
Reduction(%)	97	74,75	71,76	90,96,96	86	86,86,86,86,90
Delay(avg)	20	30	33	21	24	36
Delay(Worst)	36	46	54	43	37	49
Reduction(%)	72	58	54	71	67	50

Figure 7: Run-time characteristics and reductions for a single-clustered baseline architecture

	EJ	LU	TRI	2D-DCT	ADPCM	G721
hot-spots	1	2	2	3	1	1
Freq(%)	100	49,51	54,46	28,29,43	100	100
Switch points	1	1,1	1,1	1,1,1	1	5
Live Regs	2	32,30	31,23	8,6,6	14	17,17,17,17,8
Reduction(%)	98	78,79	78,84	94,96,96	90	88,88,88,88,94
Delay(avg)	19	42	48	35	39	49
Delay(Worst)	35	64	74	92	64	63
Reduction(%)	75	65	52	56	56	43

Figure 8: Run-time characteristics and reductions for a dual-clustered baseline architecture

registers. A drop in the number of live registers at the end of the loop body is quite common and it can be 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.

Figure 7 shows the achieved results for a single-cluster platform. The first row in the table contains the benchmark name. The next row shows the number of hot-spots identified for each benchmark; the execution frequency for each hot-spot in percentage is presented in the third row. The fourth row shows the number of switch points identified and used for each hot-spot. The next pair of rows show the number of live registers at each switch point and corresponding reductions in percentage compared to the context save/restore overhead of the baseline architecture. The last two rows show characteristics of the delay from the time the preemption request occurs and the moment when the first instruction of the preempting task is executed. It includes the context save procedure, the scheduler overhead, and the context load code. These numbers correspond to the responsiveness of the system as they represent how many cycles does it take to resume the execution of a suspended task. The first row contains the average number of response cycles, while the second row contains the worst case.

The results for the dual-cluster baseline architecture are shown in Figure 8. As can be observed, when more issue bandwidth is available, the compiler is more aggressive in loop unrolling and parallel scheduling of instructions. The loop cycle increases due to the larger unrolling factor, which in turn leads to increase in the number of live registers. However, this increase is limited to the available ILP, and because of the more load/store units, the actual reductions that we achieve are higher compared to the single-cluster architecture, because the slight increase in live registers is compensated by the ability to save and load a pair of registers simultaneously.

Since the total overhead of context switch could be quite high, especially for systems where response time and real-time guarantees are of extreme importance, the significant time and response reductions achieved by the proposed technique are of extreme benefit to a large number of control and real-time embedded systems. An additional benefit of the proposed technique is that because of the drastic reductions in the number of cycles needed for context switch, the preemption period may be shortened, which will result in much better system responsiveness with the same context switch overhead.

6. 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 stored. This is achieved through the active cooperation of compiler, microarchitecture, and RTOS kernel. During compile-time, special switch points in the application code are identified where the number of live registers is minimal. The information regarding these points is captures by the hardware support and used to defer the preemption interrupt slightly, in order to save a large number of overhead cycle associated with the preemption code in the kernel. Compiler-generated custom software routines are registered with the kernel for each switch point and are used as a dynamic custom replacement of the high-overhead general-purpose context save and restore routines. This customization technique which involves the compiler, the microarchitecture, and the RTOS kernel has been shown to achieve significant reductions in both task switch overhead and system response time.

7. REFERENCES

[1] K. Ramamritham and J. Stankovic, "Scheduling algorithms and operating systems support for real-time systems", 1994.

[2] G. Byrd and M. Holliday, "Multithreaded processor architectures", *IEEE Spectrum*, August 1995.

[3] P. Faraboschi, G. Brown, J. Fisher, G. Desoli and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing", in *ISCA 00*, June 2000.

[4] P. Levis et al., "TinyOS: An operating system for wireless sensor networks", *Ambient Intelligence, Springer-Verlag*, 2005.

[5] S. Bhatti et al., "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms", *MONET, Special Issue on Wireless Sensor Networks*, vol. 10, n. 4, pp. 563–579, Auguest 2005.

[6] S. Shivshankar, S. Vangara and A. Dean, "Balancing Register Pressure and Context-Switching Delays in ASTI Systems", in *CASES 05*, pp. 286–294, September 2005.

[7] T. Baker, J. Snyder and D. Whalley, "Fast Context Switches: Compiler and Architectural Support for Preemptive Scheduling", in *Microprocessors and Microsystems*, pp. 35–42, 1995.

[8] C. Albrecht, R. Hagenau, and A. Doring, "Cooperative software multithreading to enhance utilization of embedded processors for network applications", in *PDP 2004*, pp. 300–307, 2004.

[9] D. Bovet and M. Cesati, *Understanding the Linux Kernel (2nd Edition)*, OReilly, 2002.

[10] J. Fisher, P. Faraboschi and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*, Morgan Kaufmann, 2005.

[11] A. Aho, R. Sethi and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[12] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", in *30th MICRO*, pp. 330–335, December 1997.