

Application-Driven Register File Mapping for Rapid Task Preemption in Real-Time, Multi-Tasked Embedded Systems

Xiangrong Zhou
University of Maryland at College Park
ECE Department
xrzhou@umd.edu

Peter Petrov
University of Maryland at College Park
ECE Department
ppetrov@ece.umd.edu

ABSTRACT

Preemptive multi-tasking 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, not only the task responsiveness and throughput can be significantly deteriorated, but also the systems power consumption is increased. We address this problem by proposing a low-cost application-driven and OS-controlled register file mapping for a very fast context switch. By rapidly mapping a small fraction of the register file in a single clock cycle, a context switch is achieved that requires no memory transfers to preserve/restore the task register file. Compile-time analysis phase is introduced which collects register live information and renames the registers in such a way so as to constraint the small set of live registers for certain task regions to contiguous and mappable locations in the register file. 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. Our experimental results demonstrate that both the performance throughput and response time are significantly improved.

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

For cost, flexibility, and time-to-market reasons, multi-tasking environments have been widely adopted for real-time embedded applications. The set of independent or interacting tasks, comprising the embedded application, are implemented in software and share the available hardware platform. System software is responsible for the transparent switch or invocation of the tasks at run-time; each task safely assumes that it is the sole user of all the CPU resources. Tasks are invoked either by synchronous events such as periodic interrupts of the timer or by asynchronous events such as input interrupt generated by the application environment. For many real-time applications, such as in automotive and avionics control, the tasks are required to guarantee execution completion prior to a specified time deadline. Various recent research efforts have been focused on the design and implemen-

tion of scheduling algorithms with real-time guarantees. Switching between two tasks, however, has an inherent cost as it requires that a snapshot of the state of the active task be preserved, after which the saved state of the second task be brought from memory and loaded into the processor. The task state is referred to as a *context* and the switch between two tasks is typically referred to as a *context switch*. The performance and power costs of the context switch are a significant factor that needs to be taken into account by the scheduling algorithms and the *Worst-Case Execution Time (WCET)* analysis algorithms. The context switch cost directly determines the task responsiveness and the overall cost of supporting multitasking.

In this paper, we introduce an application-aware context switch methodology, which for groups of parallel tasks can completely eliminate the need to save and restore the register file during task switching with only a small increase in the size of the physical register file. This is achieved through the close co-operation of the compiler, the operating system 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 map 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* multi-tasking. In cooperative multi-tasking, 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 [1] 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 [2], 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 mul-

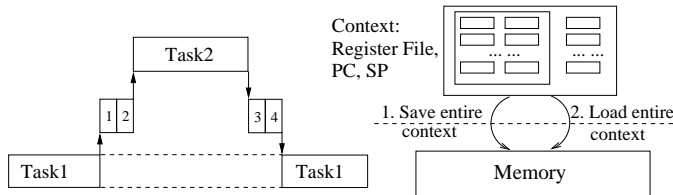


Figure 1: Context-switch mechanism in preemptive multitasking

titasking paradigm is further explored in [3], 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 [4]. Even though these approaches have the advantage of avoiding nondeterministic context switch overheads, an 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 multi-tasking 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*. As this approach has the distinctive advantage of better responsiveness and stability, most of real-time scheduling algorithms and OS multitasking support are based on it. 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.

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 [5] features a general-purpose register file of size from 64 to 256. It has been shown [6] that for some short tasks responsible to react to, and process data samples in sensor networks, the context switch overhead can be up to 30% of the total cycles.

Figure 1 illustrates the mechanism of general-purpose task switch in preemptive systems. Before the preempting task can start execution, there are two 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 lookup a priority queue which does not depend on the number of tasks in the system [7]. In this paper 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 turns 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 code and all user-level code operate on separate register files.

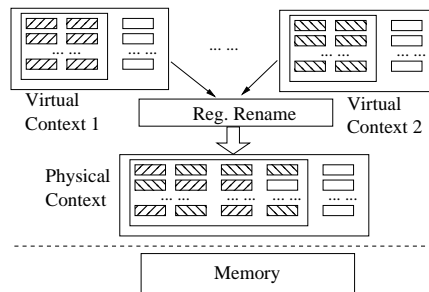


Figure 2: Hardware register renaming

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 run-time, each virtual register is renamed to a free physical register. This approach is very popular in superscalar processors and is mostly used to improve the program ILP by removing false and output dependencies. Context switches are fast as typically no or 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.

The software-based approaches reduce the number of load/stores to preserve the task contexts. In [8], a methodology is introduced where the context switch is performed when no scratch registers are alive thus saving cycles of not preserving them. In [9], compiler-generated context switch handlers are used by the OS. These handlers save and restore only the set of live registers for both tasks. Eventhough the number of cycles needed for context switch is reduced, the software approaches still require extra memory instruction to save/restore context.

The technique we propose combines the benefits of the hardware and the software approaches to achieve the fast context switch of replicated register files by using compile-time analysis and extraction 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. FUNCTIONAL OVERVIEW

In this paper we propose a system level customization methodology, which addresses the problem of context switch performance and task responsiveness overheads. Figure 3 depicts the design flow and the major steps in applying the proposed technique. 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. The minimal set of live registers for these regions 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 application 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 1 cycle) mapped into the ISA-visible

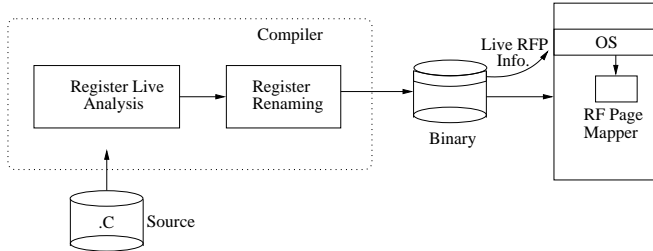


Figure 3: Methodology functional overview

register file address space. In order to further control and minimize the hardware cost for the re-mapping, 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 re-mapping event for all the cases where the small set of live registers is accommodated within these pages.

Figure 4 illustrates the organization of the proposed mapped register file. The first several RFPs are mappable through a simple hardware block, which would be described in details in Section 5. For our experiments we have considered RFPs of size 8 and have allowed for only the first 4 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 by the application code.

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.

At run-time when a preemption event occurs, a special hardware defers the preemption until a region from the CDFG 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. Thus, a very small sized pool of 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. In the subsequent two sections of the paper we detail the required compiler/OS support and the hardware architecture for implementing the register file with remapping capabilities. We conclude the paper with a comprehensive set of experimental results.

4. COMPILER AND OS SUPPORT

The purpose of the compile-time support, which our technique introduce, is to first identify the sets of live registers throughout the task CDFG. The proposed technique is applied independently on the application hot-spots, which represent heavily executed functions or loops. Often times almost all of the execution time is spend in these hot-spots, which can be easily identified through profiling.

Figure 5 shows an example of an instruction sequence and the corresponding live register information for each instruction in the sequence. A *live range* for a register starts when it is being defined (written to) by an instructions and ends at the last instruction that uses the value in that register. During the time before this register is written to again,

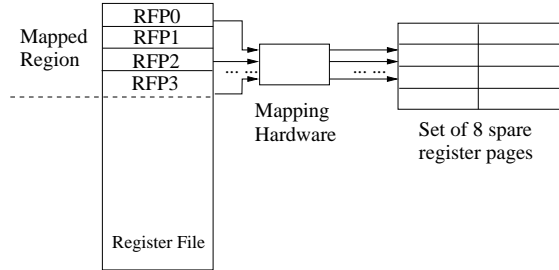


Figure 4: Mapped register file organization

the register is said to be dead as it does not carry useful data. Figure 5 shows an example of possible live ranges. Clearly, the number of live register changes as the instructions execute. It can be seen that during instruction $Inst_i$ all the registers from r_1 to r_{12} are alive, while during instruction $Inst_j$ only registers r_1 , r_2 , and r_3 are alive.

Figure 6 shows an example CDFG of an application hot-spot 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 minima for some instructions. From Figure 6a it can be seen that one such minimum exists in the innermost loop and one in the outermost loop nest. These points are referred to as *Minimal Points (MP)* and the two points from the example are marked as MP_1 and MP_2 . Similarly, the *Basic Block (BB)* with minimal number of live registers are referred to as *Minimal Block (MB)*. The two minimal blocks for our example are shown in Figure 6b. The set of live registers for each basic block, including the MBs in particular, is defined as the union of the live registers for all the instructions within that basic block. Such a distribution of minimal blocks, as shown in Figure 6b, is common for typical loops. That is because at the entry and exit of loop, only a few global register and loop-carried variable are live. It is also often the case that the minimal points reside within the minimal blocks. For all the benchmarks that we have used in our experimental study, the minimal points reside within the minimal blocks. The set of live registers for the two MPs and their corresponding MBs is shown in the figure.

The first phase of the compile-time part identifies the MPs and MBs for the application hotspot CDFGs. These points/blocks are identified statically by the compiler subject that the amount of live registers is minimal. Another constraint in identifying minimal points/blocks is to take into consideration the introduced preemption deferral as described above. 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.

If a preemption point occurs while the task is executing an MP instruction or is within a minimal block, the context switch can be handled by making sure that only the set of minimal registers for the MP or the MB are preserved. Such preservation can be effected by disconnecting their RFPs and at their place map to the register file address space the RFPs which have preserved the live registers of the preempting task.

Even though the live-context size has been reduced to minimum, an efficient way to convey this information to the OS is still needed. As the live register set can consists of arbitrary registers occupying multiple RFPs, possibly as many as the number of live registers, a “packing” step is needed that can rename the registers in the hot-spot in order to ensure that the set of live registers occupies a contiguous

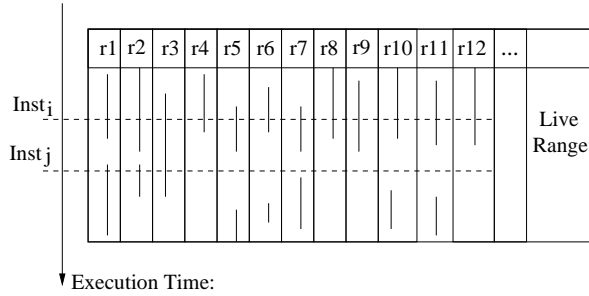


Figure 5: Register live ranges.

group of registers starting from r_0 . This register renaming step is performed during compile time and it only permutes the register names to achieve the aforementioned goal. The time-complexity is linear as it simply swaps register names with the objective of allocating the set of live registers for all the MPs and the MBs in the contiguous range from r_0 upwards. Figure 6b illustrates this phase. The live sets for both the MPs and their corresponding MBs are given. This register renaming step is performed during compile-time for all the application hot-spots. If there is more than one application hot-spot and some registers are kept alive across more than one hot-spot, transfer instructions may need to be executed when outside the hot-spots in order to “stitch” together the live ranges which have been assigned to different register names for the different hot-spots. Because these few transfer instructions, if existing at all, are executed outside the hot-spots and as such contribute to no performance overhead.

Clearly, the minimal points in the hot-spots would be the most beneficial positions to preempt the task as their live registers are minimal and after the renaming phase packed into one or two RFPs. Restricting preemption to only a few points within the hot-spot can be implemented with a simple hardware support where the preemption is deferred until the program counter of one of the MPs is reached. Nonetheless, the response is significantly faster than the one achieved by general-purpose context switch, as in our case only the preemption deferral delay would occur and no cycles will be spend saving and restoring the register file. Since the MPs are positioned within the inner-loops of the application hot-spots, this delay is minimal and typically in the order of several to ten cycles. Nonetheless, even this delay can be significantly reduced if instead of allowing preemption only at the MPs, preemption is allowed during the minimal basic block (MB) - usually it is the basic block of the MP. Because the RFPs consists of several registers (in our experiments 8), the register pages needed for the MB live registers are often times the same as the pages needed for the MP only. Consequently, allowing task switch during the MBs would not only further reduce the preemption deferral delay but will also come at either no extra cost or only a minimal increase in the RFPs that need to be mapped at context switch. Our experimental results evaluate the effect of MP-only preemption versus MB-based preemption in terms of number of pages needed to store the live registers and the impact on the deferral delay.

When the task is loaded by the OS for execution, the set of RFPs holding the live registers for all the hot-spots is provided to the OS from the program executable or from a special setup code executed just prior to entering the hot-spot. When the hardware support activates a preemption during an MB, it provides an identification to the OS context switch handler during which MB is the preemption to be executed. This information would allow the OS to preserve only the RFPs with live registers for that basic block by re-mapping them and, thus, disconnecting them from the ISA-visible register file ad-

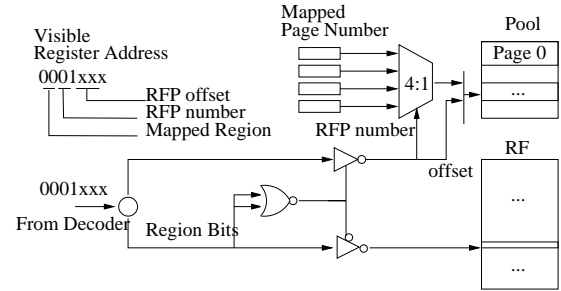


Figure 7: Mapped register file architecture

dress space. 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. If not, depending on the task priority, the non-critical tasks live pages can be preserved in memory during preemption and not occupy register pages from the pool. In such a scenario where too many parallel tasks co-exist, the OS would assign the most critical ones to non-swappable register pages in the pool - these tasks would still take full advantage of the proposed technique.

5. HARDWARE SUPPORT

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

Figure 7 presents the 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 register addresses corresponding to these 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. The mapping is implemented by replacing the four most significant bits from the ISA-visible register address 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. Each one of them defines the mapping for the first four RFPs from ISA-visible register space. 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

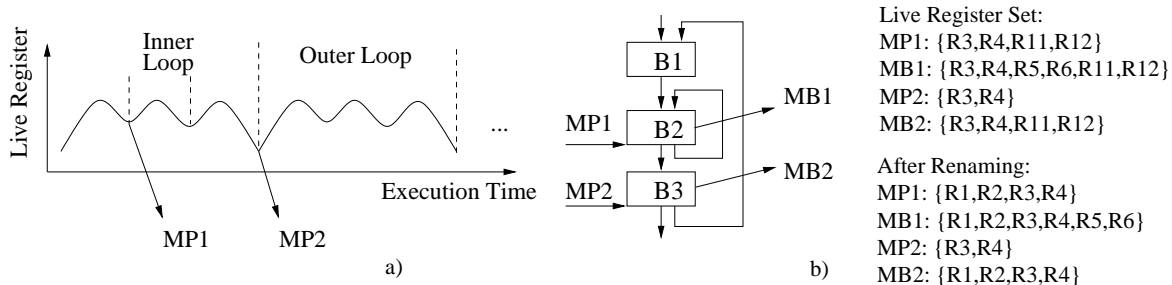


Figure 6: Application hot-spot with two minimal blocks

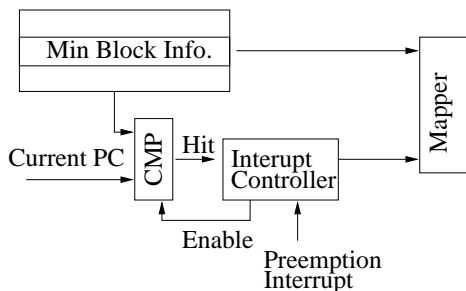


Figure 8: Hardware for detecting MP or MB

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 noteworthy that at this step the tasks with highest demands for responsiveness can be placed in the extra register pool and utilize the proposed zero-cost context-switch, while the live pages for the less demanding task assigned to the base register file and (only they) preserved at context-switch.

When a preemption interrupt occurs, the task-switch must be deferred to a minimal block. In order to do this, information regarding the MBs is needed. Each MB is clearly defined by its start and end addresses. During the preemption request, the hardware must first check whether the execution is within a minimal block and if so, invoke immediately the context-switch procedure. This check is performed by a range comparator. In the case of preemption request outside a minimal block, the context-switch procedure is deferred and the current task continues execution until the beginning of live block is reached. Reaching an MB is detected by comparing the current PC with the start address of the MB. Note that only the first comparison, which checks whether the current instruction resides within an MB, is a range comparison. The subsequent comparisons for detecting the beginning of an MP, if any, are simple value comparisons; if preemption is deferred to a minimal point, then only value comparisons are needed.

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 mod-

ern 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 orders 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 order to quantitatively evaluate the proposed methodology we have developed an experimental environment that includes a compile-time analysis tool for register live analysis and run-time simulation. As a baseline processor we have used the Vex package [10] provided by HP labs, which consists of state-of-the-art optimizing VLIW processor model, industry-strength compiler support, and compiled simulation toolset. The VLIW processor core can be configured into various architectures including multiple clusters, scalable register files, and functional units. Each cluster is configured to have a general-purpose register file, 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 32-bit registers and 8 1-bit branch registers.

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 8 registers) in the register file are mappable. 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 -H2 -O3 optimization flags, which includes heavy loop unrolling and trace scheduling; and in the second setup optimization options are -H0 -O0 which include all scalar optimizations but no loop unrolling.

The benchmarks are initially profiled and the hot-spots identified. An off-line tool that we have developed reads in the assembly files and performs the register live analysis and the register rename phases. For each hot-spot, the minimal points and their minimal blocks are identified. To simulate the run-time behavior of preemption, we have instrumented the assembly files to include an extra “dummy” instruction in each inner/outer loops of the hot-spots, which sole purpose is to model the behavior of a timer interrupt for the purpose of task preemption. We have provided a call-back function for this special instruction, which the compiled simulator invokes regularly.

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*, and the *TRI triangular matrix conversion*; these kernels manipulate large matrices and are extensively used in many algorithms. The application group includes the *2D-DCT* - discrete cosine transform

	EJ	LU	TRI	MMUL	2D-DCT	ADPCM	SHA	SUSAN
# of hot-spots	1	2	2	1	2	1	5	1
# of MP	1	1,1	1,1	1	1,1	1	1,1,1,1,1	1
# of Live Reg.	2/8	20/22,20/27	21/24,15/17	11/17	14/23,12/24	11/11	5/20,11/18,11/14,11/18,10/13	20/20
# of Live Pages	1/1	3/3,3/4	3/3,2/3	2/3	2/3,2/3	2/2	1/3,2/3,2/2,2/3,2/2	3/3
Delay(avg)	16/10	24/23,28/23	33/31,30/29	18/15	15/4,19/8	21/21	22/13,13/2,15/12,13/2,15/12	24/24
Delay(Worst)	32/25	47/46,54/49	65/63,60/58	35/30	28/14,37/24	42/41	42/33,25/10,29/26,25/10,29/26	48/47
Delay Red.(%)	88/92	80/82	75/77	86/90	87/98	84/84	88/94	81/81

Table 1: Characteristics and response reductions with aggressive compiler optimizations for 64-entry register file

	EJ	LU	TRI	MMUL	2D-DCT	ADPCM	SHA	SUSAN
# of hot-spots	1	2	2	1	2	1	5	1
# of MP	1	1,1	1,1	1	1,1	1	1,1,1,1,1	1
# of Live Reg.	3/16	27/28,20/38	29/33,20/24	15/22	20/42,18/40	15/15	6/16,11/20,11/24,11/18,10/23	21/26
# of Live Pages	1/2	3/4,3/5	4/5,3/4	3/3	3/6,3/5	2/2	1/2,2/3,2/3,2/3,2/3	3/4
Delay(avg)	15/3	28/27,28/24	40/36,38/36	22/18	16/8,20/3	24/24	29/23,28/9,30/11,28/11,30/6	25/25
Delay(Worst)	28/11	55/53,55/51	78/74,75/73	42/39	30/22,38/15	48/47	56/51,54/31,59/26,54/35,59/26	49/44
Delay Red.(%)	94/99	89/90	84/86	91/93	93/98	91/91	89/96	90/90

Table 2: Characteristics and response reductions with aggressive compiler optimizations for 128-entry register file

that are widely used in many image and video processing applications, the *ADPCM* speech coder MediaBench[11], the *SHA* hash algorithm, and the *SUSAN* the image recognition from MiBench [12].

Table 1 shows the achieved results for aggressively optimized benchmarks (-H2 -O3) for a 64-entry register file. The first row in the table contains the benchmark name. The next row shows the number of hot-spots identified for each benchmark. The third row shows the number of minimal points (and, thus, minimal blocks which in all the cases are the basic blocks where the minimal point resides) as well for each hot-spot. The next pair of rows show the number of live registers and the corresponding number of RFPs after renaming the live sets. Each entry is represented as a pair of numbers - the first number shows the result for the minimal points (MP) only, while the second number corresponds to minimal blocks (MBs). The next two rows have identical organization and report on the delay (in instructions) from the time the preemption request occurs and the moment when the task switch happens. These two rows show the average and the worst delays for both MPs and MBs. These numbers correspond to the responsiveness of the tasks when an interrupt for preemption occurs, as they represent how many cycles does it take to resume the execution of a suspended task. The last row demonstrates the significant reductions in response time, which the proposed methodology achieves compared to a general-purpose context-switch mechanism. For the majority of the applications, the reductions are well above 70%. Table 2 reports the result for a 128-entry register file. It has identical structure as Table 1. Table 3 also has an identical organization but reports the results for the applications compiled with the scalar-optimizations only for a 64-entry register file, while Table 4 reports the data for 128-entry register file. For this cases the reductions in task responsiveness are above 90% for most of the benchmarks. This follows from the fact (that can also be observed in these tables) that for the larger register file and especially with aggressive optimization, the number of live registers for both MPs and MBs is higher, in general.

In order to evaluate the impact of the size of the pool of extra registers, we have formed sets of multiple tasks for parallel execution. Tables 5 and 6 show the relation of page pool size and the supported multiple task set for worst case scenarios - when for all the tasks the preemption occurs within the MP/MB with largest number of live register

	A2	B2	A3	B3	A4	B4
25%	0/0	0/0	1/2	0/2	3/5	3/5
	0/0	0/0	0/4	0/3	3/7	2/7
50%	0/0	0/0	0/0	0/0	1/3	1/3
	0/0	0/0	0/0	0/0	0/3	0/3
75%	0/0	0/0	0/0	0/0	0/1	0/1
	0/0	0/0	0/0	0/0	0/0	0/0

Table 5: Live pages exceeding page pool; Aggressive optimizations

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 above, only the non-time 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. In Tables 5 and 6 we show for each task set the number of live register pages, which exceed the pool. 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; $A_2=\{EJ,LU\}$, $A_3=\{EJ, LU, TRI\}$, $A_4=\{EJ, LU, TRI, MMUL\}$. Sets B_i similarly cover the group of application benchmarks: $B_2=\{2D-DCT, ADPCM\}$, $B_3=\{2D-DCT, ADPCM, SHA\}$, $B_4=\{2D-DCT, ADPCM, SHA, SUSAN\}$. The structure of the entries is identical to the previous two tables: first sub-rows report for 64/128-entry register files, while horizontal pairs correspond to minimal point/minimal blocks preemption moments. It can be seen from this data that a 25% extra register pool completely covers all the 2-task sets and some of the 3-task sets, while 50% pool completely covers all the 2-, 3-, and some of the 4-task sets.

7. CONCLUSION

We have introduced an application-aware task preemption methodology which implements a zero-cost context switch with no need for register file preservation in memory. The proposed methodology achieves the zero-cost task preemption of replicated register files with the cost

	EJ	LU	TRI	MMUL	2D-DCT	ADPCM	SHA	SUSAN
# of hot-spots	1	2	2	1	2	1	5	1
# of MP	1	1,1	1,1	1	1,1	1	1,1,1,1,1	1
# of Live Reg.	3/14	18/20,18/19	13/24,10/19	9/13	13/15,11/13	10/10	6/6,5/9,10/14,10/15,9/13	20/20
# of Live Pages	1/2	3/3,3/3	2/3,2/3	2/2	2/2,2/2	2/2	1/1,1/2,2/2,2/2,2/2	3/3
Delay(avg)	18/1	3/1,3/1	8/1,8/1	4/2	3/1,3/1	14/14	2/2,5/2,4/2,4/2,4/2	14/13
Delay(Worst)	34/7	5/3,5/3	15/5,15/5	7/4	5/3,5/3	28/27	4/3,9/6,7/4,7/4,7/4	27/25
Delay Red.(%)	86/99	98/99	94/99	97/98	98/99	89/89	97/98	89/90

Table 3: Characteristics and response reductions with scalar-only compiler optimizations; 64-entry register file

	EJ	LU	TRI	MMUL	2D-DCT	ADPCM	SHA	SUSAN
# of hot-spots	1	2	2	1	2	1	5	1
# of MP	1	1,1	1,1	1	1,1	1	1,1,1,1,1	1
# of Live Reg.	4/17	18/20,18/19	13/24,10/20	9/13	13/15,11/13	11/11	8/8,5/9,10/14,10/16,9/13	20/20
# of Live Pages	1/3	3/3,3/3	2/3,2/3	2/2	2/2,2/2	2/2	1/1,1/2,2/2,2/2,2/2	3/3
Delay(avg)	17/1	3/1,3/1	8/1,8/1	4/2	3/1,3/1	14/14	2/1,5/1,5/1,5/1,5/1	11/11
Delay(Worst)	33/7	5/3,5/3	15/5,15/5	7/4	5/3,5/3	28/27	3/2,10/3,9/4,9/4,9/4	21/20
Delay Red.(%)	93/99	99/99	97/99	98/99	99/99	95/95	98/99	96/96

Table 4: Characteristics and response reductions with scalar-only compiler optimizations; 128-entry register file

	A2	B2	A3	B3	A4	B4
25%	0/0	0/0	0/2	0/0	2/4	3/3
	0/0	0/0	0/1	0/0	0/4	1/1
50%	0/0	0/0	0/0	0/0	0/2	1/1
	0/0	0/0	0/0	0/0	0/0	0/0
75%	0/0	0/0	0/0	0/0	0/0	0/0
	0/0	0/0	0/0	0/0	0/0	0/0

Table 6: Live pages exceeding page pool, Scalar optimizations

of introduced pool of spare registers, which are mapped into the ISA-visible register address space. The proposed technique significantly improves the task responsiveness as compared to traditional context switch mechanisms and comes very closely to the instantaneous responsiveness of replicated register files. The introduced methodology requires minimal OS and hardware modifications and can benefit a large number of real-time, low-cost, and energy-efficient embedded applications.

8. REFERENCES

- [1] P. Levis et al., "TinyOS: An operating system for wireless sensor networks", *Ambient Intelligence*, Springer-Verlag, 2005.
- [2] 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, August 2005.
- [3] S. Shivshankar, S. Vangara and A. Dean, "Balancing Register Pressure and Context-Switching Delays in ASTI Systems", in *CASES*, pp. 286–294, September 2005.
- [4] C. Albrecht, R. Hagenau, and A. Doring, "Cooperative software multithreading to enhance utilization of embedded processors for network applications", in *Euromicro-PDP*, pp. 300–307, 2004.
- [5] P. Faraboschi, G. Brown, J. Fisher, G. Desoli and F. Home-wood, "Lx: A Technology Platform for Customizable VLIW

Embedded Processing", in *ISCA'00: The 27th Annual International Symposium on Computer architecture*, June 2000.

- [6] J. Hill and D. Culler, "A wireless embedded sensor architecture for system-level optimization", Technical report, U.C. Berkeley, 2001.
- [7] D. Bovet and M. Cesati, *Understanding the Linux Kernel (2nd Edition)*, O'Reilly, 2002.
- [8] T. Baker, J. Snyder and D. Whalley, "Fast Context Switches: Compiler and Architectural Support for Preemptive Scheduling", in *Microprocessors and Microsystems*, pp. 35–42, September 1995.
- [9] X. Zhou and P. Petrov, "Rapid and Low-Cost Context-Switch through Embedded Processor Customization for Real-Time and Control Applications", in *DAC*, pp. 352 – 357, July 2006.
- [10] J. Fisher, P. Faraboschi and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*, Morgan Kaufmann, 2005.
- [11] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "Media-Bench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", in *30th MICRO*, pp. 330–335, December 1997.
- [12] M.R Guthaus, J. S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", in *WVC*, pp. 3–14, Dec 2001.