Reducing Context Switch Overhead with Compiler-Assisted Threading

Pekka Jääskeläinen Pertti Kellomäki Jarmo Takala Heikki Kultala
Mikael Lepistö
Department of Computer Systems,
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
Email: firstname.lastname@tut.fi

Abstract

Multithreading is an important software modularization technique. However, it can incur substantial overheads, especially in processors where the amount of architecturally visible state is large.

We propose an implementation technique for co-operative multithreading, where context switches occur in places that minimize the amount of state that needs to be saved. The subset of processor state saved during each context switch is based on where the switch occurs.

We have validated the approach by an empirical study of resource usage in basic blocks, and by implementing the co-operative threading in our compiler. Performance figures are given for an MP3 player utilizing the threading implementation.

1 Introduction

Multithreading is an important implementation technique for computer-based systems. In some domains, such as embedded systems, the system needs to carry out concurrent activities or react to external events in a timely fashion. By isolating the activities from each other, multithreading helps to modularize the system. Multithreading is also commonly used for hiding the latency of time consuming operations like disk or memory access, or co-processor execution, improving the throughput of the system.

One of the main problems in the use of multithreading is the cost of *context switching*, i.e. suspending one thread (we refer to independent units of execution in this paper as *threads), broadly synonymous with "process", "task", and so on.* and resuming another one. The problem is especially pronounced in explicitly parallel "wide" processors such as VLIWs. Such processors typically have a large amount of processor state that needs to be saved and restored at context switches.

This paper concentrates on the context save/restore overhead. We propose an implementation technique for multithreading based on compiler support and simple timer hardware. The technique exploits the fact that resource utilization is not constant over a program. A compiler needs to track the resource usage of the program it is compiling, so it can place context switches where they are cheapest, i.e., requiring minimum amount of state to be saved. Since context switches occur at well defined places, the state saving code can be tailored according to the context used at the call position. An example application presented in this paper shows that the additional overhead from threading can be almost halved when compared to the case when all context is saved on every context switch.

The rest of the paper is organized as follows. Section 2 discusses implementations strategies for multithreading, and Section 3 describes our implementation. A case study using the implementation is described in Section 4. Related work is reviewed in Section 5, and Section 6 presents the conclusions and outlines future work.

2 Implementing multithreading

In order to be able to suspend execution of a thread and resume it later, a thread scheduler needs to save and restore the architecturally visible state of the processor. This is one of the main overheads of threading. Additional overhead is also produced by cache misses due to the sudden change of control flow to the new thread of execution [7].

Scheduling of threads can be classified as *preemptive* or *co-operative*. The difference between these methods is that in the former a scheduler can suspend a thread and start executing another one in arbitrary places, while in the latter a thread voluntarily suspends itself to give other threads execution time.

2.1 Architecturally visible state

The architecturally visible state of a processor consists mainly of registers. This includes general purpose registers, floating point registers, status registers such as carry flags, and so forth. The values of these need to be preserved by the scheduler over context switches, typically by saving them in a per-thread stack.

In some architectures, function unit pipelines (or rather their latencies) are also visible to the programmer, hence a context switch needs to save and restore the pipeline registers as well, so that code relying on knowledge of pipeline latencies (i.e. statically scheduled code) works correctly even over a context switch. Pipeline registers may not be directly accessible to software, in which case context switching needs to be disabled while there are operations whose results are waiting to be written to registers.

At any given point in the execution of a program, only a subset of the architecturally visible state may be relevant to the program. If a value in a register will not be used by the program anymore, there is no need to save and restore the contents at a context switch. This fact is exploited in our threading implementation.

2.2 Thread scheduling

In preemptive thread scheduling, hardware interrupts are used for implementing context switching. Perhaps the main attraction of preemptive scheduling is the strong modularity it provides. The individual threads can be arbitrary code, and the scheduler only needs to know very little about them. The modularity comes with a price, however. When a thread is suspended, the scheduler does not usually know anything about the actual processor resource usage of the thread, so all architecturally visible state needs to be saved. Many modern processors have large register banks, and the cost of saving the state can be substantial.

In co-operative scheduling, threads interleave by periodically voluntarily releasing control to other threads. This can be much more lightweight than preemptive scheduling, as it can be implemented using user-level subroutine calls instead of interrupts, which may be costly, especially if they require the processor to switch from user mode to supervisor mode and back. However, co-operative scheduling requires all threads to be well behaved, otherwise a single thread could hoard all the execution time.

There are various ways to implement co-operative scheduling, from explicit yielding of control at source code level to compiler-assisted threading which is not visible to the programmer. We discuss some of these techniques in Section 5, and compare them to our implementation.

3 Compiler assisted co-operative threading

We have implemented a toolset for designing application specific processors based on the Transport Triggered Architecture (TTA) [1]. The toolset, called TTA-based Codesign Environment (TCE) [5], includes a retargetable compiler, an instruction set simulator, and a processor generator.

The Transport Triggered Architecture is similar to VLIW, but it exposes even more architectural details of the processor to programs. Preemptive context switches saving all the architecturally visible state would be quite expensive, so we are interested in reducing the cost of context switching by exploiting the information on resource usage of the threads. The target domain of the toolset is embedded systems where assuming co-operation between the compiler and the scheduler is more acceptable than in general purpose systems.

3.1 Empirical analysis of basic blocks

In order to obtain savings by using co-operative threading, context switches should be performed at code sections where the amount of context to be saved is small. This implies that there should be variance in the amount of context in different parts of the programs. We studied the amount of context in a set of programs in the DENbench 1.0 [3] benchmark from the EEMBC benchmark suite. The benchmark contains applications from the multimedia and digital entertainment domains. The benchmark consists of about 150 000 lines of C code.

We compiled the benchmark using the LLVM compiler version 2.2 [6], and analyzed the number of live registers at each instruction in the resulting binaries. The register allocator used was the linear scan [9] allocator implementation provided with the LLVM. The target machine had a total of 16 registers available. The variation in the size of the register context is summarized in Fig. 1.

The number of live registers varies from zero to fourteen, the average being around three live registers. This reflects the fact that the binaries are complete executable programs, so the binaries contain start up code, libraries, and the like, where the amount of available instruction level parallelism is relatively low. We have not analyzed which basic blocks contribute to the larger register context sizes, but many of them are likely to come from the loop kernels.

Our working hypothesis was that the minimum size of context to save is usually at the beginning or at the end of basic blocks, and near function calls. Function calls result in many of the live variables to be saved to stack before the call. This way the live ranges of the caller saved registers end before the function call, so the registers need not be saved again in case a context switch is inserted at the function call position. The set of registers saved by the caller

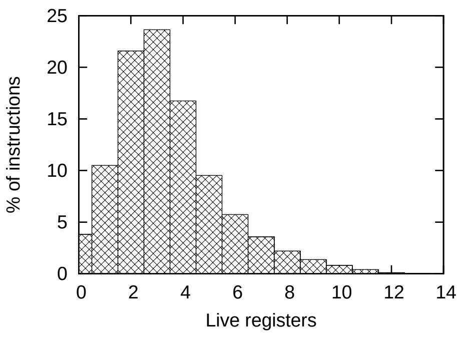

Figure 1. Distribution of register context size.

depends on the calling convention. In our case we used a calling convention which pushes all arguments on stack, and all live registers are caller saved. However, when function arguments are passed in registers, the registers add to the amount of context to save.

Table 1 shows statistics on the location and size of the minimum context within basic blocks in the benchmark. Each row in the table represents a subset of all the basic blocks. The first column indicates the maximum number of live registers for the blocks in the row, and the next three columns indicate the percentage of blocks where the minimum context size occurs in the beginning, only in the middle, or at the end, respectively. The fourth column shows the average minimum size of the context. The final column shows the average difference between the minimum number of live registers at either end of a basic block, and the minimum in the whole block.

For the columns “optimum in beginning” and “optimum at end”, a basic block is included in both percentages if the minimum context size occurs at both ends of the basic block. This explains why the sum of the percentages is greater than 100% in some rows. The column “optimum in middle only” indicates the percentage of blocks where the minimum context size only appears in the middle of a basic block, but not at either end.

The reason why we are interested in the beginnings and ends of basic blocks is that inserting a call to the co-operative thread scheduler in the middle of a basic block in an optimal way is considerably more costly than inserting one at either end of the block. Inserting in the middle requires scanning all instructions, and potentially rescheduling the instructions following the call. The last column in Table 1 suggests that we can optimize the majority of cases by only looking at the ends of basic blocks.

Max live (# regs)	Opt. in begin. (%)	Opt. in middle only (%)	Opt. at end (%)	Aver. min (#regs)	Aver. min(ends) - opt (#regs)
1	44.6	0.1	99.9	0.4	0.0
2	49.7	7.8	81.9	0.9	0.1
3	55.5	18.9	49.8	1.4	0.2
4	56.5	22.3	38.8	1.8	0.2
5	58.3	22.7	35.2	2.1	0.3
6	59.9	23.3	32.7	2.6	0.2
7	58.6	25.2	28.1	2.9	0.3
8	51.3	27.5	40.7	3.8	0.3
9	53.0	28.5	37.3	4.2	0.3
10	62.9	28.9	30.2	4.3	0.3
11	68.1	16.9	37.4	4.4	0.2
12	68.4	15.8	33.7	2.9	0.2
13	66.7	13.3	26.7	3.3	0.1
14	100.0	0.0	0.0	3.0	0.0

Table 1. Statistics of location and size of minimum context.

3.2 Thread yield call emitter

In our implementation of co-operative threading, the compiler first performs optimization and code generation, and as a final pass emits calls to the thread scheduler at the positions where the number of live registers is smallest. The compiler generates different variations of the thread scheduler function based on which registers need to be saved, and emits calls to the appropriate versions.

The current algorithm works with individual basic blocks and considers only the beginning and end of each basic block. As explained above, scanning the whole basic block for the optimal call location is not worth the increased compiler complexity.

The algorithm is best described with a simple example. Figure 2 shows a single thread that has been compiled to use co-operative threading. The compiler has emitted a call to “yield_r1()” in the first basic block of the thread. Not counting the branch instruction, there are two places where the call could be inserted. Emitting the call in the beginning would have required saving the two live-in registers *r2* and *r3*. The “add” instruction reads the registers and stores the result in *r1*, which is live at the end of the block. Since registers *r2* and *r3* are not used in the rest of the thread, the values in them become dead, thus the number of live registers at the end of the block is one.

No general purpose registers need to be saved in the thread switches of the other two basic blocks in the example thread. The value in *r2* becomes dead after it has been stored in memory in the second basic block. The last basic block does not have any live-in registers, because the values in registers *r1* and *r2* are not used in the block, and they are overwritten by the “ld” instructions. The two basic blocks can thus call the “yield_()” function which does not save any registers. In contrast, without the compiler-assisted thread switching all the yield calls would need to preserve all registers in the target. For example, suppose that the target processor has 8 general purpose registers. In the worst case, the execution path with three basic blocks would require saving and restoring 24 registers, which is quite large overhead when compared to preserving only a single register *r1* with the optimized version.

In the example, two calls to the thread scheduler occur right after each other if the execution goes through the middle basic block. It is very likely that the real time requirement of the application would allow omitting the thread yield call from the middle basic block altogether. However, the current algorithm does not yet look into the control flow graph as a whole, as analyzing the worst case runtime between two spots in the program is not in the general case as trivial as in this simple example.

Currently, we avoid the unnecessary calls by using predicated execution and a timer. In the benchmark platform, calls to the thread scheduler are guarded with the output of the timer. The timer is initialized to the desired thread switch latency, and when the timer reaches zero, the timer is reseted and its output switches to ’true’. Calls to the thread scheduler are predicated by the output of the timer, so they are converted to NOPs while the timer is counting. Once the time slice is over, the output goes to true and the scheduler is called. This produces minimal additional overhead to program execution.

The generated variations of the thread yield function are illustrated in Figure 2 inside the “threading runtime” box. They store the required registers to stack and call the thread scheduler function. The thread scheduler chooses one of the threads waiting for execution time, and switches execution to the selected thread by setting the stack pointer to point to the thread’s stack. This results in popping the return address of the thread from the stack and returning to the correct thread yield function. The thread yield restores the thread’s registers from the stack and returns, allowing the new thread to resume its execution until the next effective thread yield call.

4 MP3 case study

In order to analyze the reduction of context switch overhead in practice, a multithreaded example application was implemented. The application picked for the case study is the MP3 benchmark from the DENbench suite. The benchmark was modified to include a player thread that plays the samples decoded by another thread at fixed intervals, thus introducing a soft real time constraint.

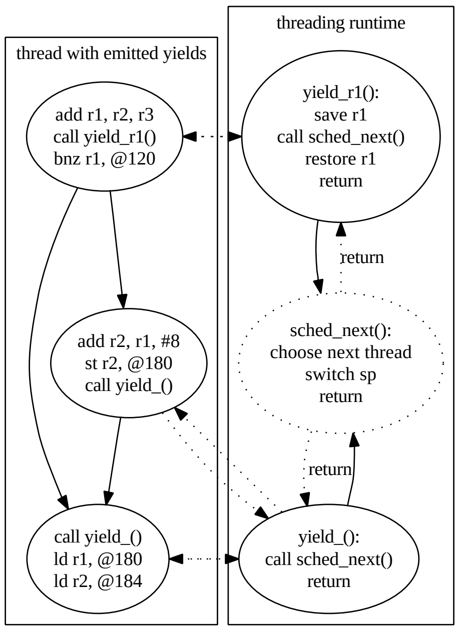

Figure 2. Threading with compiler emitted yield calls.

The test input data used was the longest MP3 file in the DENbench suite (music128stereo), a 60 s sample with 44100 Hz sampling frequency. Thus, in order to fulfill the sample rate of 44100, the player thread had to be executed in an interval of about 22 μs.

The MP3 application was ported to a TTA processor designed with the TCE toolset. The relevant resources in the designed processor are listed in Table 2. The thread yield call emitter algorithm was implemented in the TCE compiler and the cycle counts were produced with the instruction set simulator. The simulator is cycle accurate, but does not include simulation of the memory hierarchy apart from a static RAM with a fixed access latency. Thus, the produced cycle counts are equal to the count of executed TTA instructions. The simulated memory setup is realistic for smaller embedded systems. However, in the future it would be interesting to analyze the peformance of the algorithm also in case of different cache configurations.

The simulated processor was assumed to run at a clock frequency of 350 MHz, thus calling the thread scheduler at about every 7 700 cycles. The clock frequency had to be set quite high due to the unoptimized MP3 implementation of DENbench.

According to Section 3, inserting thread scheduler calls in the beginning or at the end of basic blocks is usually very close to optimal with regards to the context to save. In our

Registers (32 bit)	16
Register file ports	2 x write, 4 x read
Function units	2 x MUL, 2 x ALU

Table 2. Resources in the simulated processor.

method	runtime (cycles)	overhead (%)	required frequency
No threading	17 051 815 737	-	284 MHz
Optimized	19 194 172 482	12.6%	318 MHz
Full save	21 116 286 373	23.8%	352 MHz

Table 3. Cycle counts with the different threading methods in the MP3 case.

case, inserting a call to the end produces another overhead due to the large number of branch delay slots in our architecture. Basic blocks often end in branch instructions, which need to be rescheduled in case a scheduler call is emitted at the end. This reduces possibilities for branch delay slot filling, thus leading to an additional overhead which is avoided when the call is always inserted in the beginning. Therefore, for this case study we implemented a version of the thread yield call emitter which always inserts calls in the beginning. The benchmark architecture has three branch delay slots. In architectures with little or no branch delay slots, this overhead would be avoided, making it equally costly to add the call in the beginning or at the end of the basic block.

The following variations of the threading implementation were compared:

"Full save" mimics the case when the threads are scheduled preemptively. The case is simulated by always calling the thread yield function which saves all the registers.

"Optimized" inserts scheduler calls in the beginning of basic blocks, but calls a thread yield function that saves only the registers live at that position.

"No threading" is a version without the player thread or any other overhead from threading. This is used to compute the additional overhead produced by threading.

The results are presented in Table 3. It shows the number of cycles passed when playing the 60s audio file, and the overhead of threading as a percentage. The last column indicates the minimum frequency at which the processor could be run if the MP3 player were the only task running. It is computed by dividing the cycle count with the length of the MP3 test sample.

Using the proposed method to reduce the context saving overhead seems to pay off. The overhead from threading is reduced from 23.8% to 12.6%, which indicates that the majority of the thread yield calls only require a small subset of the registers to be saved.

5 Related work

Preemptive scheduling is typically used in situations where the scheduler has minimal knowledge about the threads it executes. Hence, there is no information that the scheduler could exploit for choosing when to switch threads.

If there is some tolerance in the timing of context switching, the compiler and the operating system can co-operate to reduce the cost of context switches. Since the compiler has full knowledge of resource use, it can mark spots in a program where some resources are not in use.

In [10], the instruction encoding of the processor is augmented with a bit that indicates that a set of "scratch" registers is not in use at that particular point. The set of scratch registers is known both to the compiler and the scheduler. When an interrupt occurs, the processor does not perform a context switch immediately. Instead, it continues to execute instructions until either an instruction tagged with the bit is found, or a predetermined number of cycles has passed. In the former case the scheduler knows that it does not need to save the scratch registers, resulting in a cheaper context switch. In the latter case, the scheduler saves all registers.

The technique is quite close to ours. The major difference is that we propose a solution for implementing co-operative threading by exploiting the code generator of the compiler. In contrast, the solution in [10] assists a preemptive thread scheduler to preempt at positions with little context to save. Our technique allows more flexibility in choosing the subsets of state to save depending on the program position. No additional support from the architecture is required.

Co-operative threading can be visible at the source code level in the form of *co-routines*, which go back to at least Simula 67 [2]. Co-routines can be provided either as a built-in feature of the language, or as a library.

Especially in library based co-routine implementations, the programmer often needs to explicitly yield control to other threads, which can be both cumbersome and error prone. A source level technique that relieves the programmer from this burden is to transform a threaded program into a behaviorally equivalent sequential program. The Phantom serializing compiler [8] accepts a C program that employs a subset of POSIX threading primitives, and outputs a standard C program that is equivalent to some interleaving of the threads in the original program. One of the main benefits of this approach is that it does not require any special hardware or runtime software support, so it is applicable even to low end microcontrollers.

Our technique shares some of the benefits of the Phantom approach: threading support is implemented at compile time and no additional hardware support is required. However, as described in Section 3, we can use simple timer hardware and predicated execution to make even quite fine grained scheduling efficient.

At the machine code level, the natural unit for scheduling is the basic block. The *quantum scheduler* [4] is a real-time scheduler based on co-operative scheduling of basic blocks. The application is compiled to (or given the decade, possibly written in) assembly code, and processed by the *quantum assembler*, which transforms all control transfer instructions into calls to the scheduler. The scheduler uses a trampoline strategy to pass control between basic blocks: it decides which basic block to run next, manipulates the stack suitably, and executes a return instruction to continue at the next basic block.

The quantum scheduler is able to provide hard real time guarantees on suitable hardware. If the execution time of each instruction is known, the execution time of each basic block can be calculated. The quantum assembler annotates the basic blocks with this information, and the quantum scheduler maintains a real-time clock based on the information.

Our implementation is conceptually similar to the quantum scheduler in that there is a call to the scheduler in each basic block. However, instead of maintaining time information programmatically, we exploit a timer to minimize the unnecessary thread scheduler calls. Our technique also works without a timer with the expense of the overhead of the extra thread scheduler calls.

6 Conclusions and future work

The benefits of compiler assisted co-operative thread scheduling come from reducing the context switch overhead by saving only a subset of processor context at each thread switch. Calls to the scheduler are emitted by the compiler which is able to choose the locations with the least context to save.

The presented MP3 case study on an example processor showed reduction in threading overhead from 23.8% to 12.6% in comparison to an unoptimized threading implementation which saves all registers at every context switch.

In addition, the proposed technique is interesting because it makes the distinction between preemptive scheduling and cooperative scheduling less meaningful. To the programmer, compiler assisted co-operative thread scheduling looks exactly like preemptive scheduling, because the programmer is freed from manually calling the thread scheduler in her programs. On the other hand, in the point of view of the operating system or the runtime environment, threading is still co-operative.

In the future, we plan to implement an additional optimization that removes unneeded thread scheduler calls based on worst case runtime analysis.

Acknowledgment

This work has been supported in part by the Finnish Funding Agency for Technology and Innovation under research funding decision 40163/07.

References

[1] H. Corporaal. *Microprocessor Architectures: from VLIW to TTA.* John Wiley & Sons, Chichester, UK, 1997.

[2] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Some features of the SIMULA 67 language. In *Proceedings of the second conference on Applications of simulations*, pages 29–31. Winter Simulation Conference, 1968.

[3] EEMBC. Denbench 1.0 software benchmark databook. PDF. `http://www.eembc.org/TechLit/Datasheets/denbench_db.pdf`.

[4] S. B. Guthery. Self-timing programs and the quantum scheduler. *Commun. ACM*, 31(6):696–702, 1988.

[5] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala. Codesign toolset for application-specific instruction-set processors. In *Proc. Multimedia on Mobile Devices 2007*, pages 65070X–1 – 65070X–11, 2007. `http://tce.cs.tut.fi/`.

[6] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symp. Code Generation and Optimization*, page 75, Palo Alto, CA, March 20–24 2004.

[7] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *ASPLOS-IV: Proc. 4th int. conf. on Architectural support for programming languages and operating systems*, volume 26, pages 75–84, New York, NY, USA, April 1991. ACM Press.

[8] A. C. Nacul and T. Givargis. Lightweight multitasking support for embedded systems using the Phantom serializing compiler. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 742–747, Washington, DC, USA, 2005. IEEE Computer Society.

[9] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[10] J. S. Snyder, D. B. Whalley, and T. P. Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, pages 35–42, 1995.