Application Configurable Processors

Chris Zimmer, Gary Tyson, David Whalley

Florida State University, Tallahassee FL 32306-4530, USA

ABSTRACT

Compiler optimizations such as recurrence elimination and software pipelining can significantly reduce execution time for applications, but these transformations require the use of additional registers to hold data values across one or more loop iterations. Embedded systems have difficulty exploiting these optimizations since they typically do not have enough registers to be able to apply the transformations. In this paper, we evaluate a new application configurable processor utilizing register queues which enables these optimizations without increasing the architecturally addressable register storage requirements. This leads to improved execution time for embedded architectures that normally cannot support these code optimizations.

KEYWORDS: Embedded Systems, Software Pipelining, Compiler Optimization

1. Application Configurable Processors

Our proposed application configurable processor is an extension to a conventional embedded processor that enables the configuration of the register file to support the reference patterns of an application. The is achieved by integrating multiple register file designs into the microarchitecture, and then mapping each architected register specifier to the register file that best matches the reference characteristics presented by the application. Register queues[1][3] are a register file structure that can take advantage of FIFO behavior within an application. This is done by enabling multiple outstanding writes to the same register, while retaining all the values written before the reads occur. By doing this a single register specifier can hold many live values. Therefore, a single register queue can replace the sets and uses of many different registers with a single register mapped into one of these structures. The addition of these structures to the instruction decode stage of the pipeline would allow the compiler to run optimizations, such as recurrence elimination and software pipelining, that would have otherwise been unable to be applied due to the lack of available registers.

Recurrence elimination is an optimization which removes the redundant loads of a recurrence from the body of the loop code and replaces it with a series of cheaper register moves. In a system such as the ARM, this optimization would quickly become infeasible depending on the size of the recurrence. Mapping a register onto a queue could eliminate all of the moves and decrease register pressure. This would allow the performance enhancement that is typically provided by this optimization without the associated register cost of performing register moves. Software pipelining [2] is the technique of scheduling instructions across successive iterations of a loop. Software pipelining is typically employed with the use of modulo variable expansion [2] or a rotating register

files[4]. Modulo variable expansion (a compiler technique) and rotating registers (a hardware technique) both need the availability of extra registers to accomplish the renaming required by the optimization. The extra registers needed are not readily available in embedded systems and to add more registers would not only require the addition to hardware but would also require a reconstruction of the ISA to allow for the extra encoding space needed to use these two methods to perform software pipelining.

Register allocation can also benefit from the addition of behavior-specific register files; register allocation is the code optimization that puts a large number of program variables into a small number of architected registers. Generally, register allocation attempts to maximize performance by keeping as many operands as possible in registers. Register allocation could be modified to allow it to identify general patterns within basic blocks and remove the sets and uses for many different instructions replacing them with a customized register structure that would be able to mimic the set/use pattern thereby reducing register pressure by enabling a single register specifier, mapped to a queue, to hold multiply live values at once (thereby increasing the effective size of the register file).

Application configurable processors require very few modifications to the already existing structure of the machine and the addition of a single instruction to the ISA. The proposed customized register structures are composed of a small table, similar in construction (not function) to the register rename table found in many out-of-order processor implementations, which maps a register into a customized register structure. The structures are composed of different patterns depending on the structure to which they are mapped. This small table is indexed by the register number and is accessed during the instruction decode phase. The table lookup would specify if the register data was found inside of a customized register structure or if it could be found inside the architected register file. This table would also specify where reads occur if the register is mapped to a queue. The customized register structures themselves would be added to the instruction decode stage as a series of buffers. The different buffers would have different behavioral patterns, that would be made known so that the compiler can exploit them appropriately. The different buffers are necessary to be able to exploit common patterns that may appear in code. For instance, register queues can have either destructive reads or non-destructive reads. A destructive read would push the queue forward on reads and writes. A non-destructive read queue would only be pushed forward on writes to the queue. While we have yet to explore other register file organizations, it is feasible to incorporate register stacks, content addressable registers, etc. into the collection of supported register file types.

The only change to the existing instruction set is the addition of a new instruction to map or unmap a register into the customized register structure. This new instruction would use two register operands and an immediate to provide a read position into the register structure (necessary to support non-destructive queue reads). The first register operand is the register from the architected register that is being mapped. The second register operand is the customized register structure to which it is being mapped. The immediate field is the read position for the register structure if it is needed. These instructions would be inserted before the determined pattern usage to map a register into a register structure and after the pattern usage to remove the mapping. Each set and use with that register after being mapped will refer to the customized register structure. This approach leaves the existing ISA intact with the addition of only a single instruction.

The final addition is to add support to the existing compiler to be able to use these new register structures. This requires changes to the optimizations themselves to set up the ordering of the instructions in an identifiable pattern. It also requires changing many of the instruction selection phases in an attempt to order the instructions in an exploitable pattern.

2. Software Pipelining with Register Queues

Software pipelining is an optimization that would typically be very difficult to perform using the registers available in a typical embedded processor. The extraction of several iterations from the loop requires some sort of register renaming, especially if loop unrolling is employed to obtain an optimal pipelined kernel. Some of the different methods employed to allow this to be done on a typical processor are register renaming, modulo variable expansion, or accessing a rotating register file. The rotating register file functions as a circular buffer, but at any point, any of the registers in the circular buffer may be accessed. To implement rotating registers, extra registers must be added, changing the encoding of the ISA to access these registers. This often turns out to be an efficient way to handle the register renaming required in software pipelining, however it is a very invasive method for an embedded system. Figure 1(a) depicts the kernel of a loop containing some high latency operations that could be addressed software pipelining.

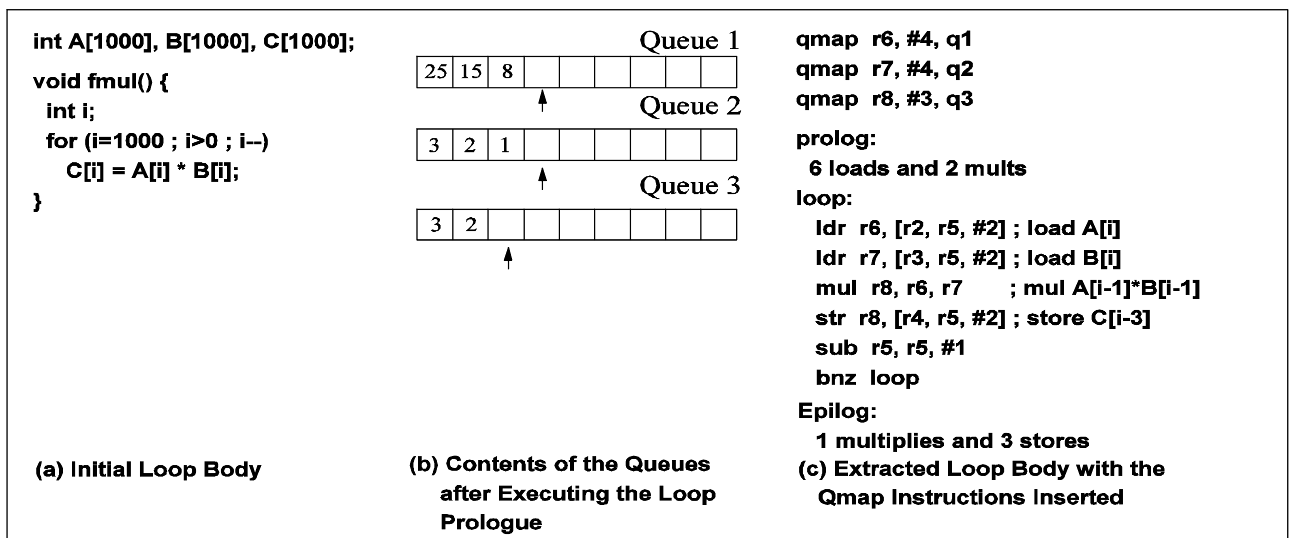

Figure 1: Example of Exploiting Queues with Software Pipelining.

Our implementation of software pipelining uses the method of iterative modulo scheduling to calculate the pipelined loop kernel and then create the prologue and epilogue code based on that scheduled loop. Iterative modulo scheduling employs a height-based priority scheduling to arrange the instruction chains in descending order of their priority. In this example, iterative modulo scheduling was able to extract three iterations from the loop. To complete this software pipelining correctly, three different queues must be used to map three different registers from the extracted iterations. Figure 1(b) depicts the state of the queues after having executed the prologue of the loop in which the queues are filled up. The read position of the queues are set so that when a use of the

register which is mapped into the queue is referenced, the decode stage will pass the value from that read position into the next stage. The prolog contains loads for the first three loop iterations and multiplies for the first two. This enables the loop body to be scheduled without stalls. Figure 1(c) shows the final code from the compiler with the mapping directives added into the pipelined loop at the very beginning of the prologue. When the loop terminates, the epilogue completes the last multiply and three stores then un-maps the register. If the latencies change or you execute on a superscalar pipeline the kernel of the loop remain unchanged but the qmap instructions are modified to account for the new latencies.

For our experiments we used the Very Portable Optimizer (VPO) compiler backend ported for the ARM ISA along with the Simple Scalar simulation environment. We added support for queues to the simulation environment and added iterative modulo scheduling to the compiler with support for register queues. We also added support for register queues in the recurrence elimination optimization. Our preliminary results show that using register queues enables software pipelining and recurrence elimination to be successfully applied for loop kernels with extremely high register pressure, even given the small register specifiers used in the ARM architecture. Note that these loops could not be improved without queues because they would run out of registers. Execution time reduction ranges from 25% to over 80% for these loops, while code size is increased between 10% to 300% due to the addition of prolog and epilogue code when software pipelining. The code size of the loop kernel is unchanged for the original loop. The architected register pressure is only slightly increased from the original loop, even though the number of live values held in the register file (including the register queues) is more than doubled on average.

3. Conclusion

Our preliminary work has shown that using an application configurable processor with customized register structure can allow the embedded designer to perform software pipelining and recurrence elimination in an embedded system with only minor hardware modification and virtually no significant changes to the ISA. This method can significantly reduce register pressure in many loops and facilitate optimizations that increase register pressure to improve the code. Using these types of optimizations would allow the designer to obtain the performance increases in the easiest places to glean performance. Our future work includes changing register allocation to be able to exploit register queues and eventually several usage patterns for other types of customizable register structures.

[1] K. Kiyohara, S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, S. Anik, and W. W. Hwu. Register Connection: A new approach to adding registers into instruction set architectures. Annual International Symposium on Computer Architecture, pages 247-256, May 1993.

[2] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. Conference on programming language Design and Implementation, pages 318-327, 1988.

[3] G. S. Tyson, M. Smelyanskiy, E. S. Davidson. Evaluating the Use of Register Queues in Software Pipelined Loops. IEEE Transactions on Computer, Vol. 50, No 8 pages 769 – 783, August 2001.

[4] B. R.. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. ACM SIGPLAN'92 Conference on Programming Languages Design and Implementation, Pages 283-299, June 1992.