THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

APPLICATION CONFIGURABLE PROCESSORS

By

CHRISTOPHER J. ZIMMER

A Thesis submitted to the
Department of Computer Science
In partial fulfillment of the
Requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 2006

The members of the Committee approve the Thesis of Christopher Zimmer defended on November 20, 2006.

David Whalley
Professor Co-Directing Thesis

Gary Tyson
Professor Co-Directing Thesis

Robert van Engelen
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

This is dedicated to everyone who has ever helped me.

ACKNOWLEDGEMENTS

I would like to thank my co-advisors Dr. David Whalley and Dr. Gary Tyson for helping me through this process.
I acknowledge Antonia Emperato for her support and help during the writing of this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

As the complexity requirements for embedded applications increase, the performance demands of embedded compilers also increase. Compiler optimizations, such as software pipelining and recurrence elimination, 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. Compilers for embedded systems have difficulty exploiting these optimizations since they typically do not have enough registers on an embedded processor to be able to apply the transformations. In this paper, we evaluate a new *application configurable processor* utilizing several different register structures which can enable these optimizations without increasing the architecturally addressable register storage requirements. Using this approach can lead to an improved execution time through enabled optimizations and reduced register pressure for embedded architectures.

CHAPTER 1

Introduction

As embedded systems become more complex, design time for embedded processors is staying the same or decreasing while demand for these systems is rising. One method for reducing the design time in an embedded system is to use a general-purpose solution for the design of the processor. These conventional embedded processors come with many different designs and extensions to make them smaller and more efficient. The ARM processor is an example of such a processor; it is a 32-bit RISC processor that has 16 registers, only 12 of which are typically available for general use. There are several extensions for the ARM, such as the Thumb and Thumb2. The ARM processor is a highly used processor in embedded systems and can be found in devices including Palm Pilots, cell phones, and various other electronic devices. As the applications that these embedded systems support become more complex, performance also becomes an issue. It is in this area where we find a need for *application configurable processors*. Figure 1.1 shows a small example of ARM assembly code that has been compiled using available optimizations. This small example is using roughly 70% of the available registers in the ARM instruction set. It should be obvious from this example the difficulty a compiler may have applying optimizations for this type of machine.

Gaining performance through code optimizations in embedded systems is often exhausted very early in the optimization process due to a shortage of available registers and the inability to perform many optimizations that would require extra registers. Conventional embedded processors often have few registers to keep the encoding of the instruction set small and easy to decode. Due to the limited amount of registers in conventional embedded processors, even simple segments of code can use all of the available registers to perform a given task. There are several compiler level methods of exploiting the available registers. Spilling values from registers to memory can reduce the register requirements for a block of code; however, this method of freeing registers can have a significant negative impact on performance.

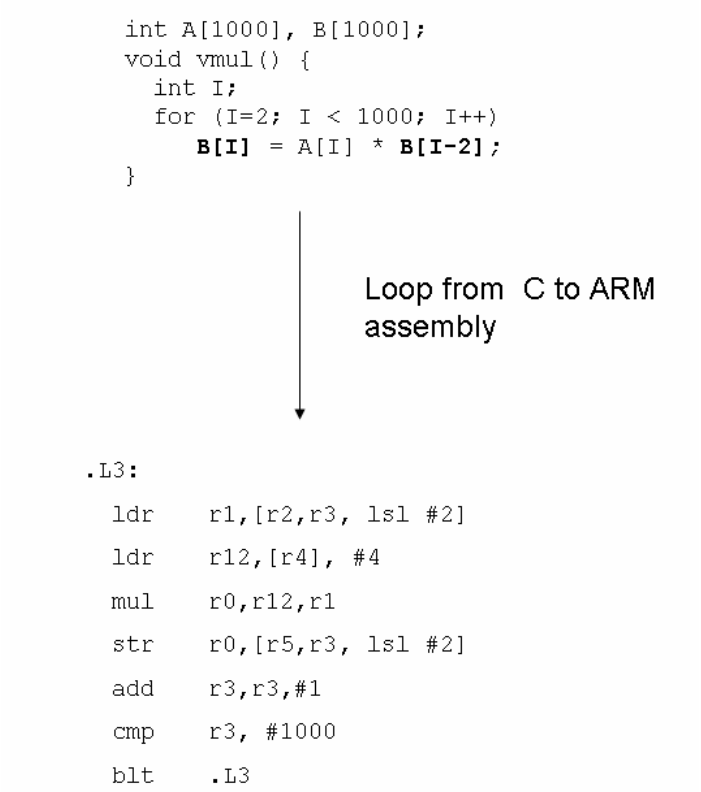


Figure 1.1 Compiler Generated ARM Code

Application configurable processors help to alleviate the shortage of registers by using small additional register structures to exploit register usage patterns found in code or produced by compiler transformations. By identifying the exploitable patterns, the *application configurable processor* could utilize its series of customized register structures to act as an enlarged register mechanism which is accessible through the architected register file. This would allow the compiler greater flexibility during software optimizations and greater control over the available registers during register allocation. The overall impact of this approach can lead to significantly improved performance and a reduction in application wide register pressure as well as enable more aggressive compiler optimizations to be performed in areas otherwise impossible.

CHAPTER 2
Experimental Framework

For this research compiler we used the Zephyr portable compiler. We utilized the Edison Design Group (EDG) front end to compile standard C code. The EDG front-end is a high level only compilation tool that is capable of parsing many of the different C standards as well as C++. We used the Very Portable Optimizer [1](VPO) backend configured to generate ARM code as our backend. The VPO backend is designed to be both portable and efficient. This is accomplished in VPO through the use of VPO's Register Transfer Lists (RTL). RTL's are VPO's intermediate language that provides flexibility through being machine independent. Many of the optimization phases of VPO are performed on RTL's in a machine independent manner. This approach minimizes the amount of code that requires machine dependent information thus enabling VPO to be a portable compiler. To complete the process we were able to utilize GCC-cross compiler tools in order to assemble and link our binaries.

We used the Simple-Scalar [2] simulation environment to perform the evaluation and simulation for our research. Our Simple-Scalar simulation environment was setup and configured to take a Simple-Scalar ARM binary output from VPO and simulate cycle accurate statistics and results. Our research utilized two of the available simulators in the Simple-Scalar suite. The Sim-Safe simulator is a functional only simulator that provides a minimum amount of statistics regarding the binary execution. The functional simulator was used in creating and debugging the system for enabling *application configurable processors*. The Simple-Scalar simulator is set up in a modular fashion so that architectural features may be created and modified in a very straight forward method. The final simulator that we used for our experiment was Sim-Outorder. The Sim-Outorder simulator is a cycle-accurate simulator that, in spite of its name, can be configured to run as either out-of-order or in-order processor. Sim-Outorder also provide a 2-level memory system and support multiple issue widths and speculative execution. The Sim-Outorder simulator was utilized to provide more detailed statistical information regarding the application of our research to our compiled binaries.

Digital Signal Processing (DSP) applications are a very common application now found in embedded systems. Common examples of applications that require DSP software are mp3 players. The decoding of the mpeg 3 codec requires several stages and filters to be run on the

data as it passes through the processor. Each of these filters or stages can be considered a DSP kernel at their smallest. The worst case performance in these situations is real time performance. This would not allow for any mistakes or the quality of output would not meet user standards. It is often a benefit in these types of applications to exceed real time performance so that a buffer of calculated data can be accrued. This benefits the processor by allowing extra time to provide quality of service.

Several DSP applications were chosen from 2 suites of benchmarks to evaluate the usefulness of this experiment. We chose DSP benchmarks because we felt they were representative of many of the tasks that modern embedded processors may be required to accomplish. DSP benchmarks are typically short in length: typically no longer than 50 – 100 lines of code. However, these benchmarks are frequently required to do a large amount of numerical processing as well as relatively complicated calculations in the body of a loop. It is becoming a common technique in embedded design to use an application specific processor to perform a very common task to what is being calculated in these DSP kernels. One of the goals of our research is to add another solution as an alternative to the growing trend of application specific processors. Our goal and reason for selecting these benchmarks is to show that our *application configurable processor* is a method that will be able to adapt to changes in performance needs and application complexity posed by newer embedded applications. We would like to achieve this goal as well as offer the compiler more flexibility to optimize the applications that run on an *application configurable processor*.

All of the DSP benchmarks that we used provided their own data sets; however, none of them produced output. To verify the correctness of our solution we temporarily added output statements to evaluate our results versus the base compiler with no support for customized register structures. When we were able to verify the result for the different machine configurations available to the compiler the output statements were removed. Our primary experiments were focused on enhancing the optimizations that are performed on loops. We found that many of the DSP benchmarks contained sufficient loops to evaluate the effectiveness of our method. We selected the benchmarks on the premise that all of the performance measurements were gleaned from loops that had achieved a steady state. To achieve the steady state the loops were run long enough to negate any cost that the start up cost of the loop might have required.

Table 2.1 DSP Benchmarks used in Testing Application Configurable Processors

Program	Description
DSP Stone Fir2Dim	Performs convolution on an input matrix
DSP Stone N Real Updates	Performs N Number of Updates on Data Arrays
DSP Stone Matrix	Performs a series of Matrix Multiplications
DSP Stone Dot Product	Applies a Dot Product Calculation Algorithm
DSP Stone FIR	Applies Finite Impulse Response Filter
Conv 45	Performs a convolution algorithm
Mac	Performs a multiple multiply accumulate operations

Our experimentation with *application configurable processors* spans a few different experiments to evaluate the applications which seemed feasible. The first experiment is set up to test the feasibility of applying register queues to an optimization in order to exploit a governing characteristic and improve the performance and ability to optimize. This experiment will be applied to improve the performance of loops by removing redundant loads created by recurrences. The second experiment performed using *application configurable processors* is retargeting register allocation in high register pressure situations to reduce register pressure and better allocate the architected register file. The final determination of this experiment is to show a positive reduction in the registers used in basic blocks with high register pressure. The final experiment was set up to test the ways in which *application configurable processors* can aid the application in terms of performance. In this experiment we enable software pipelining in a system where there is no room for optimizations due to register pressure. Our goal in this experiment is to show a positive increase in the overall performance of that application as well as the ability to actually perform this optimization with the constrained resources. In this experiment we will be measuring for cycle accurate performance improvement. All of these experiments are tested and run using a base case versus the compiler optimized equivalent code. All of these experiments are run through functional simulation to determine that the corresponding results are accurate.

CHAPTER 3
Application Configurable Architecture

We had several goals in the development of this system. We wanted our mechanism of storage to be fast and as easily accessed as the architected register file. We wanted our mechanism to act as an extension that would allow the instruction set of the architecture to remain the same. These design decisions come from the need to keep code size small and to retain performance of our system. To manage this, we keep the modifications to hardware and software as small and efficient as possible. Several different types of register structures are created to aid register allocation and optimizations by reducing register pressure. One of the main goals of the design of our application configurable processor is to be able to use the existing ISA of a conventional embedded design even with the significant modifications to the underlying hardware.

With the intentions of keeping our mechanism fast we designed our structures to be a minimally invasive change to the access of the current architected register file. This design decision would allow a minimally invasive hardware addition and allow us to only have to modify the architected register file itself in order to add the register file extensions. The other portions of the pipeline which must be modified to enable *application configurable processors* are the register dependence calculations performed during decode and writeback. With the additions that we add to our system the calculation of data hazards becomes slightly more difficult and will be discussed later in this paper.

The first system designed to enable the *application configurable* extension was the value mapping table. The concept of the mapping table in our work is very similar to the map table that can be found in many out-of-order processor implementations. Using a similar concept to the ideas in the Register Connection approach [3], this table provides a register look up and corresponding data structure from which to attain the value. This structure is a very quick access map table and is the only redirection required to implement this system. This map table is modified and set as the first logic that must occur during every access to the architected register file. The size of the table in our set up was limited to the amount of registers available in the architected register file, but could even be made smaller and faster to access by removing special purpose registers.

The implementation of this map is such that the lookup is based on the address of the register itself. The table will then provide a value which corresponds from which register file the value must be retrieved. If there is no value set in the table, then the value will be retrieved from the architected register file. The alternate register files in our design are set up to be number ordered and grouped by type for easy compiler identification. This map table allows the system to use the specifier of an architected register file to access one of the customized register structures. By using the already existing register specifiers we are able to integrate this research into an already exisiting instruction set without having to completely rewrite it. The use of the map table is able to provide us with the full benefit of the already existing instruction set and our new customized register files.

The most important addition to the decode stage are the customized register files. Our customized register files can be defined as a fast access memory structure that is large enough to hold multiple values, but enforces a reading and writing semantic to all of the values in it. The reading and writing semantic is used to exploit identifiable reference patterns that can be found in code or caused by the process of optimizations. We have implemented several different customized register files which mimic these behaviors. Figure 3.1 shows the modifications to an access to the architected register file during either the decode stage or writeback during the pipeline. The map check occurs anytime there is a register file access. From this figure it can be determined that there is no mapping between R1 and any of the customized register structures. If the value contained in register R1 was needed in the next stage of the pipeline, the map table looks up the mapping and the processor accesses the value from the architected register file. However this table also shows us that a mapping instruction for a mapping between R6 and Q1 has occurred. This mapping means that when any set or use request comes into the decode stage of the pipeline for R6; it will be accessing the customized register structure Q1. Though the specifier is used for R6 the register queue enables a single specifier to store significantly more values than previously. The different register structures are accessed and passed differently but are all available in order to exploit a given reference behavior of a set of registers and map them into one register.

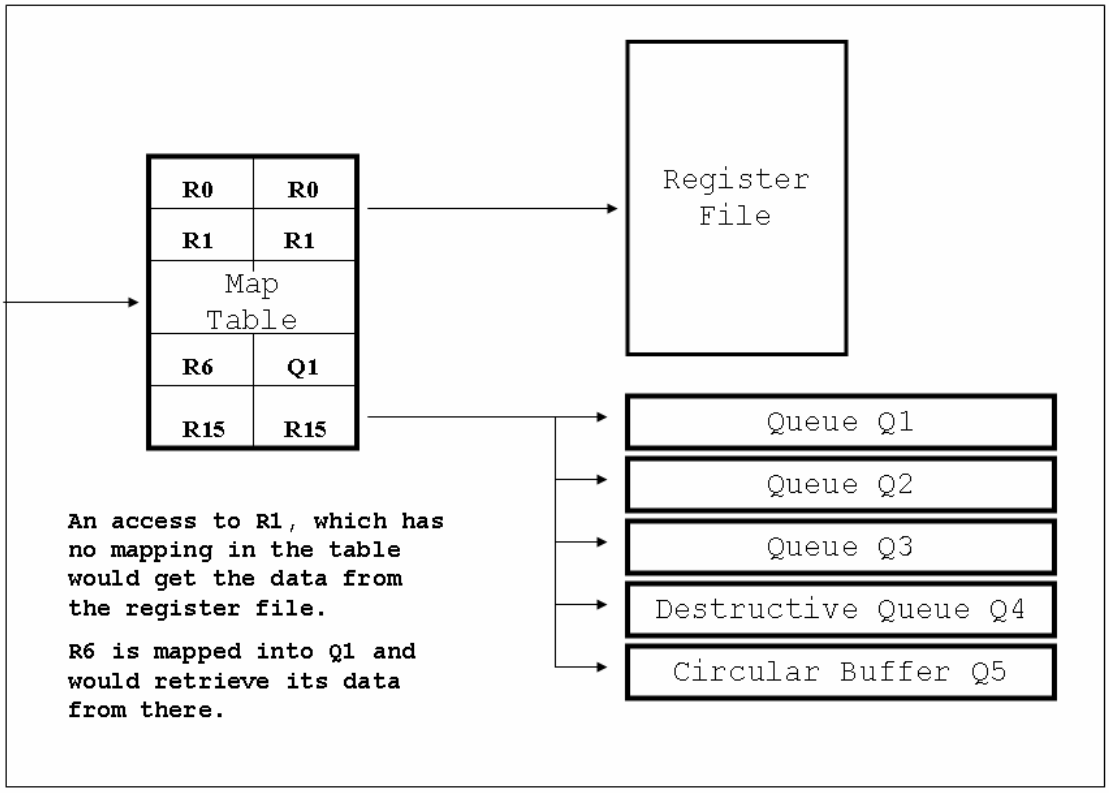


Figure 3.1 Customized Register File Access

In this section we will be discussing the architecture and behavior of the different customized register files that we have experimented with so far.

Register queues are the main type of the customized register file that we have used so far in our research. The FIFO reference behavior is an identifiable pattern found in many aspects of computer architecture and code generation. Register queues in our terms refer to a customized register file that can hold multiple data items at once using a single FIFO register and when the structure is referenced the oldest data item is the value that is returned. In our implementation of register queues we found it necessary to create different semantics for certain types of queues that allow us greater flexibility when using them in different scopes. We have so far experimented with destructive and non-destructive read queues. First, let us define the applications of our register queue files. The register queue files are preset to a certain size that is specified in an instruction. This preset size is the point at which all data will be retrieved from this instance of the register queue. In the non-destructive register queues [Figure 3.2] when data is read from the queue the values in the queue are not pushed forward. In a destructive read queue [Figure 3.3] the values are pushed forward whenever a read occurs.

These different queue types lend themselves well to two different types of reference behaviors that the compiler can exploit. In both types of register queues the values stored in them will be pushed forward when a set to the register file occurs. Queues so far have shown to be the most common type of customized register file to be needed by application configurable processors. Several optimizations create a FIFO reference behavior as well as being a naturally common reference behavior regardless of the optimizations which have been performed.

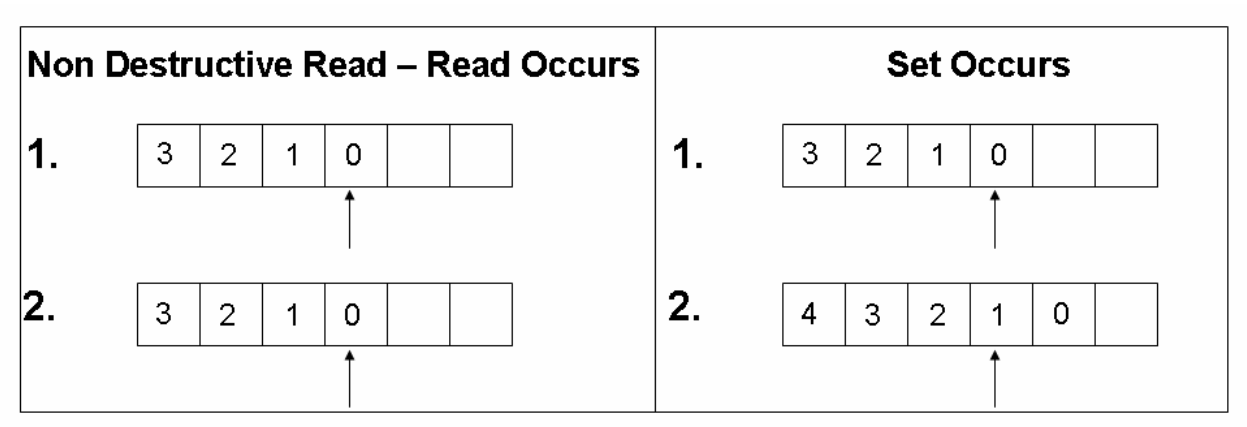


Figure 3.2 Non Destructive Read Queue Behavior

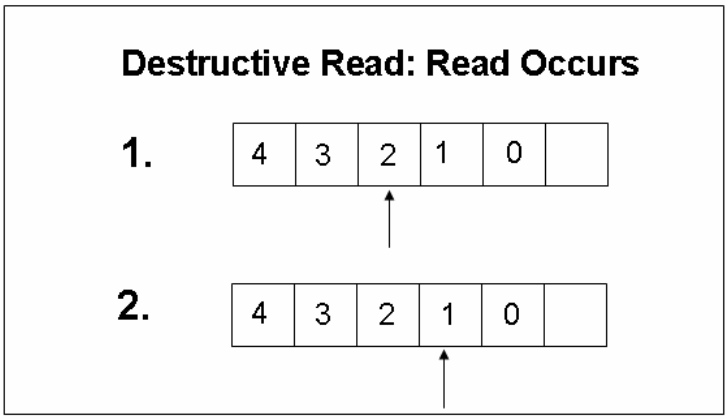


Figure 3.3 Destructive Read Queue Behavior

Circular buffers are another type of customized register structure used to hold a set of values reference in a repeated manner. In many loops that occur in embedded programs the compiler often finds it necessary to use registers to hold the address of values that are loaded or stored. This can often use several different registers to store these values. A circular buffer can be used to store these values and advance as the loop iterates. The circular buffer register file is just this, a storage mechanism very similar to the destructive read queue. When a read

occurs in this register file the value will be passed and then the read position will be incremented to read the next value. When the read position has reached the end it will loop back to the beginning of the structure and begin providing the value from that point. Circular buffer register files are a successful mechanism for storing all of the loop invariant values and providing correct data throughout the loop. Figure 3.4 shows the destructive read nature of the customized circular buffer.

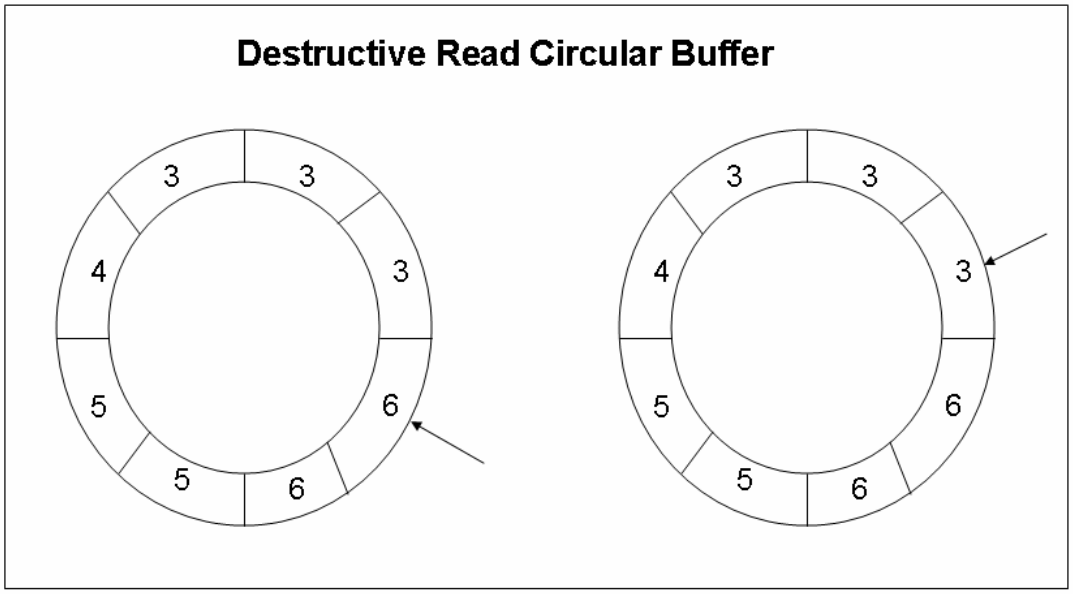


Figure 3.4 Circular Buffer Read Behavior

Stacks enable the application configurable processor to exploit any last-in first-out reference patterns that might occur in code. The customized register stack uses a destructive read method of modifying the read position into the array to mimic a stack. However, a set to the customized register stack also modifies the read position of the array to enable the last in first out semantic. Figure 3.5 shows a stack with a read position set after two reads from the stack had already occurred. Figure 3.5 shows the modified position of the read pointer when a new value is set to the top of the stack. This depicts the read position being incremented in the stack and the value being updated.

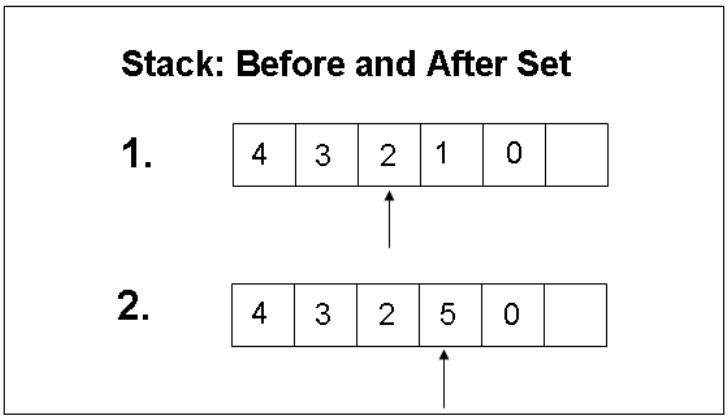


Figure 3.5 Modifying Stack Values

The modifications to the instruction set architecture were designed so that no changes to the registers or instructions themselves were made. *Application configurable processors* require only one addition to the existing ISA: an instruction which controls both the mapping and unmapping of a customized register structure. To accomplish this task, it is assumed that the semantics of the instruction can take on different meanings depending on what structure it is referencing. This instruction is the *qmap* instruction which we added to the ARM ISA. This instruction contains three pieces of information and performs all of its tasks in the decode stage of the pipeline. The *qmap* instruction is laid out as follows:

- *qmap* <register specifier> , <mapping semantic> , <register structure>
- <register specifier> The register specifier in this situation refers to the register from the architected register file which will point to a customized register structure.
- <mapping semantic> The mapping semantic refers to the set up information for the customized register structure. In the case of non-destructive read queues, this sets the position at which a value will run off of the queue.
- <Register Structure Specifier> The register structure specifier itself is numbered similarly to the numbers in the architected register file.

This instruction can have a different meaning for the customized register structure depending on whether or not it is mapped. Therefore the same instruction is used to map and unmap a customized register structure. This works by determining that if there is no mapping in the map table between a register from the architected register file and a customized register file,

it will insert a mapping, using the field from the instruction for setting up the structure. If there is a mapping that exists when this instruction is called, it will remove the mapping from the table.

Other customized register structures are available in the *application configurable processor* mainly for the purpose of reducing register pressure. They succeed in consuming many of the different reference patterns found in the loops, and though they provide no performance gain from their existence, they do reduce register pressure significantly in these register restricted environments. Figure 3.6 gives an example of a *qmap* instruction being used on a loop and mapping the values into a customized register stack. The *qmap* instruction in the example below creates a mapping between the register specifier r1 and creates a mapping to the customized register file q1. In this example the customized register file at q1 enforces a stack-based reference behavior for the code produced. By enabling these live ranges to be written into a stack, we have produced two additional free registers that can be used to enable more optimizations to reduce the numbers of spills to memory that could occur in similar basic blocks.

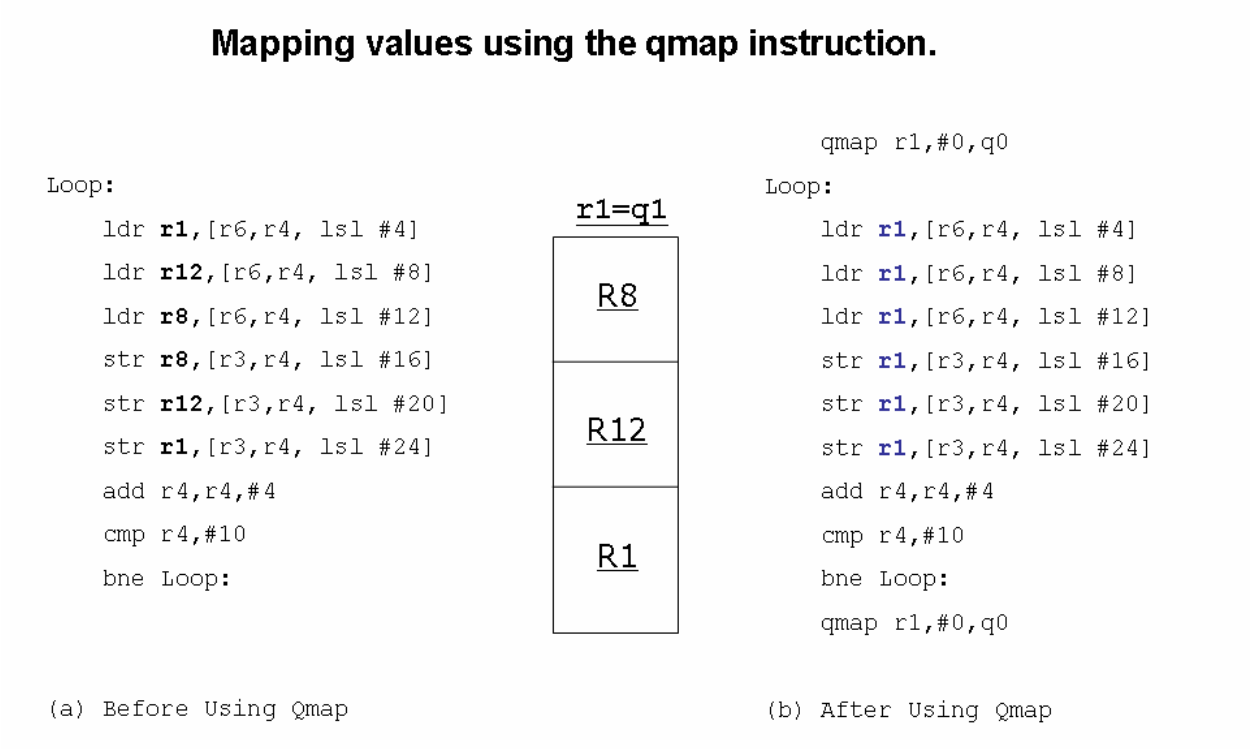


Figure 3.6 Qmapping into a Customized Register Structure

By combining these different resources together we are able to introduce an *application configurable processor*; a processor which can be easily tailored to perform machine specific

optimizations on an application to enable better performance. By utilizing the different register structures together and separately we are able to exploit patterns that occur naturally in code and generate a better solution than existed previously. The next three sections of this thesis present three extremely viable mechanisms of use for *application configurable processor* that will hopefully illustrate the ease of use and adaptability of this mechanism to new and better designs.

CHAPTER 4
Recurrence Elimination Using Register Queues

The first experiment was designed to test the feasibility of an *application configurable processor* using register queues and optimizations that have not been applied in previous work. This evaluation would enable us to determine the effect that the increased register pressure situations that occur in embedded processors have on an optimization in a embedded compiler. *Application configurable processors* were originally targeted at enabling more aggressive optimizations, but it became readily apparent that an increase in registers could help any optimizations, that exhibited a reference behavior post-optimization.

Recurrence elimination is one of the set of loop optimizations found in our experimental compiler VPO. Recurrence elimination is an optimization that can increase performance of a loop by reducing the cycles required to accomplish each iteration of the loop. A recurrence relation in math is an equation which defines a sequence recursively, where each term in the sequence is defined as a function of the prior terms. In computer science this can refer to a calculation that uses a value previously calculated and stored in a data structure that is being accessed in the current iteration of a loop. Common recurrences can be found in a variety of code involving arrays where an lower indexed element was set earlier in a loop but referenced in the current iteration. This value being referenced from a previous iteration often will occur as a load from an array in code, this load must occur on every iteration of the loop. Recurrence elimination identifies the recurrences and replaces the load in the loop with a series of moves to pass the value from iteration to iteration to save the cycles that the load would require. The series of moves are used because in theory a series of moves might require less cycles to perform than a load. Figure 4.1 shows an example of a vector multiply which has been compiled down to ARM assembly using VPO. The same code is then optimized using recurrence elimination and the load caused by the recurrence is removed from the loop in an attempt to reduce the cycles required during each iteration of the loop.

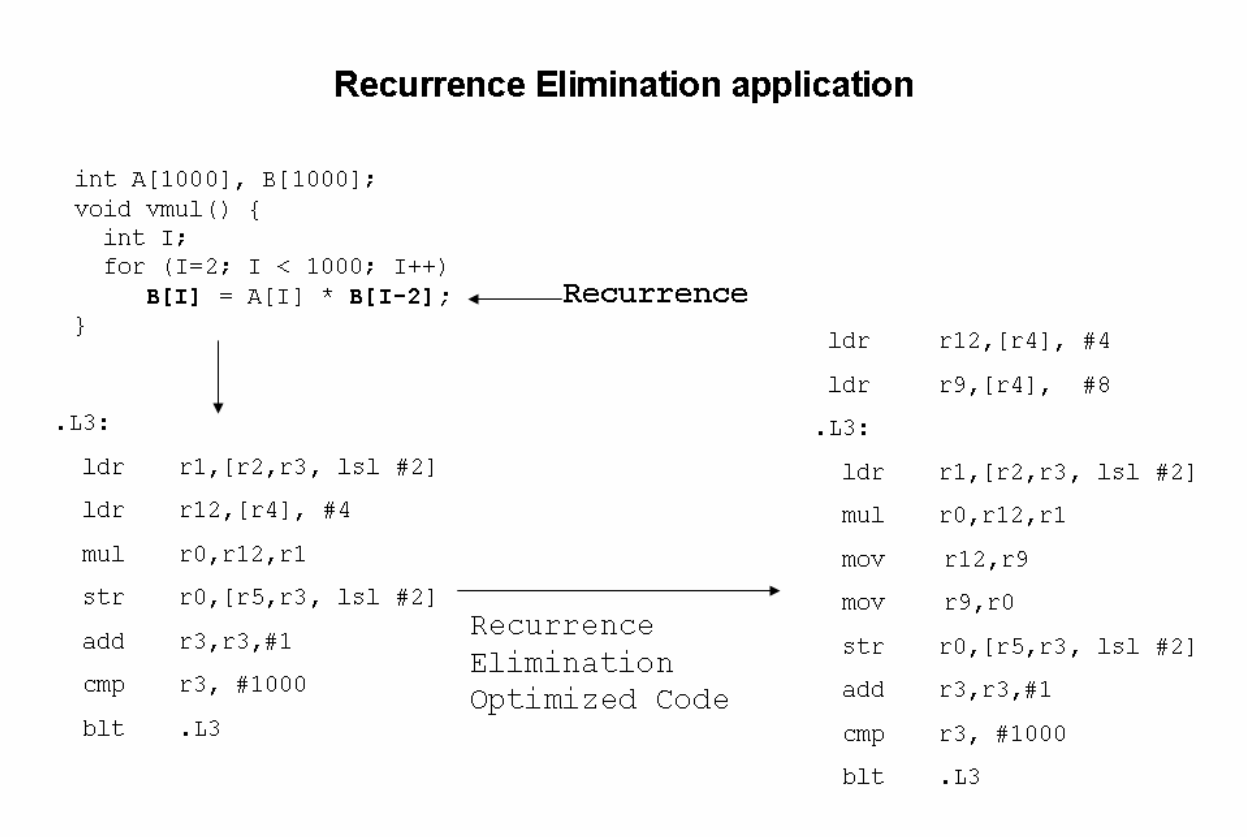


Figure 4.1 Recurrence Elimination Applied to a Vector Multiply

Recurrence elimination suffers from several drawbacks when the compiler is limited to using only an architected register file. The main drawback to using the architected register file during recurrence elimination is an increase in the storage requirements that occur when the loads are replaced with the series of moves. The extra storage requirements are a forced consequence of the needed registers to store a value from each iteration of the loop within the recurrence. As the size of the recurrence grows, so does the need for additional resources, which can significantly impact the size of the recurrences that can be removed as there may not be enough registers to perform this task. The next area of concern in recurrence elimination is the fact that the additional moves added are not at all free. At a certain point for each machine, the latency for the additional moves added will outweigh the latency for the load that was previously used. This extra latency significantly limits the size of recurrences to which this optimization may be applied.

Application configurable processors can aid recurrence elimination in a couple of ways. The first method addresses the first problem area that occurs due to using an architected

register to mask the recurrence. The access patterns of the recurrences to arrays are by design performed in a first-in first-out reference pattern. By determining the size of the recurrence being applied in the calculation prior to mapping, the values may be mapped into a customized register queue with a read position set accordingly. By performing this task the additional registers needed to store the values of the recurrence from previous iterations can be completely alleviated. The reduction of this register pressure can enable the optimization to be performed on recurrences of a greater size. Another additional side effect provided by this mapping is the complete removal of the original load and any additional moves that might have been caused by recurrence elimination. The removal of these instructions completely removes the latency provided by the original load instead of just a fraction of the latency. Which, give the recurrence elimination optimization more opportunities to improve the quality of the code. By performing this optimization using register queues, the optimized code has the potential to have greater all around savings including performance, code size, and register usage. Figure 4.2 shows the application of register queues to the example from Figure 4.1. This shows the complete removal of any needed moves without the need of any additional registers to enable this optimization. The registers r4, r2, and r5 are used as loop invariant registers and were all set prior to the execution of this loop.

```
 ldr     r12,[r4], #4
 ldr     r9,[r4],  #8
.L3:
  ldr    r1,[r2,r3, lsl #2]
  mul    r0,r12,r1
  mov    r12,r9
  mov    r9,r0
  str    r0,[r5,r3, lsl #2]
  add    r3,r3,#1
  cmp    r3, #1000
  blt    .L3
```

(a) Without using Register Queues

```
qmap     r12,#2,q0
qmap     r0,#1,q0
  ldr    r12,[r4], #4
  ldr    r12,[r4],  #8
.L3:
  ldr    r1,[r2,r3, lsl #2]
  mul    r0,r12,r1
  str    r0,[r5,r3, lsl #2]
  add    r3,r3,#1
  cmp    r3, #1000
  blt    .L3
qmap     r12,#0,q0
qmap     r0,#0,q0
```

(b) After Using Register Queues

Figure 4.2 Application of Register Queues to Recurrence Elimination

This experiment required the addition of a new optimization to VPO to propagate induction variables across the body of a loop. The goal of this induction variable substitution

phase was to actually help the existing optimization find more recurrences. This optimization required analyzing the different induction variables and mapping them back to their original sets. If we could determine that two different induction variables were originally set from the same data, then we could rename them to the same registers and modify the offset. By doing so, recurrence elimination would have an easier time finding the recurrences that occur. Some of the difficulties in applying this method consisted in simplifying the original set of the induction variable. The instruction references different registers we needed to follow multiple register lifetimes back to originating values to determine if they were the same.

In order for the optimization to remain the same as it was previously, the application of register queues to this optimization was performed after the optimization phase had already occurred. Our recurrence analysis function would determine the size of the recurrence based off of the moves present in the applied code. It would then analyze the registers used to apply the recurrence calculation and determine the best mapping pattern using the fewest registers to apply this calculation using queues. The next phase of the optimization would then remove the series of moves applied by the recurrence elimination algorithm and map the applied registers into a queue. This transformation often occurred as a mapping of two registers to one queue. As shown in Figure 4.2, by mapping a register to the first position in the queue, the compiler is able to set the value of the queue and read the value just set as well as using the second register to read the value set from two iterations ago.

To test the correctness of this experiment we applied a functional simulation to provide a validity check. The functional Simple-Scalar simulator was first modified with a table and a table lookup for each set or read from a general purpose registers. The *qmap* instructions were added to the ARM instruction set as described above and the simple scalar simulator was modified to be able to decode the new instruction. When successful table look-ups occur the result of the look up could then be routed out of the appropriate *customized register file*. For the purpose of recurrence elimination we found it necessary to implement non-destructive read register queues, since we can determine the value of the queue is going to be set upon every iteration of the loop. The destructive read register queues were added to the simulator.

The analysis of this work has demonstrated to us that there are opportunities in enabling compilers to better perform and consume less available register space. By applying register queues to the result of a function optimized using the recurrence elimination optimization, we

were able to increase the size of the recurrences and remove the moves required to perform this optimization. No performance metrics were collected regarding this improvement to the optimization, as it was designed to test the feasibility of applying customized register files to optimizations that had not been retargeted previously.

CHAPTER 5

Register Allocation for Application Configurable Processors

Register allocation in embedded systems can be difficult in areas of high register pressure. Compilers often must spill values to memory and use many more memory operations to successfully hold all of the values of data being utilized. Situations like this occur frequently enough where this pressure can affect optimization stages from very early in the compilation process. While designing the customized register structures to enhance the reference pattern of certain code optimizations, we identified that our customized register files could significantly reduce register pressure in general register allocation. Similar to the concept of over writing the uses of all the loop invariant registers with a circular buffer and a single register specifier, we began finding different patterns in basic blocks that would allow us to significantly reduce register pressure. The compiler could benefit from reduced register pressure to reduce spills to memory and to offer more flexibility to compiler optimizations.

Customized register structures can be utilized in register allocation to identify specific patterns of register use and replace several overlapping registers with a single register structure. When this method is applied to general block allocation then suitable live ranges can be mapped. This method can significantly reduce the allocable registers needed to perform the tasks within the basic block. Reduced register pressure in this situation, if provided early enough, could also increase the chances that an optimization might improve the quality of the code.

The stack register structures are commonly exploited patterns that are referenced frequently when using customized register structures. Register stacks are one of the common live range patterns identified. These are identified in many basic blocks as a large one-set, one-use live range, containing subsequently smaller one-set, one-use live ranges. These live ranges are identified during register allocation and their pattern can be over written with a single register specifier mapped into a register stack. The register stacks in the application configurable processor simply act as a last-in, first-out data structure that only requires a mapping. The mapping semantic for a register stack is not needed for this specific structure. The reference patterns of the stacks that we identified force all of the sets to occur in code before any uses occur in the instructions. To accomplish this we must enforce a destructive read policy for our register stacks. Similar to the circular register buffers, when a read occurs in

a register stack, the read position is decremented to the next earlier element. Figure 5.1 shows an example that might commonly be seen in applications where arrays are being filled with values for later use. In this example the three identified live ranges use three different register specifiers to set and use the values for the three live ranges. By mapping the replacing the three different registers specifier with a single specifier mapped into a queue, we are able to reduce the registers used by this series of instructions.

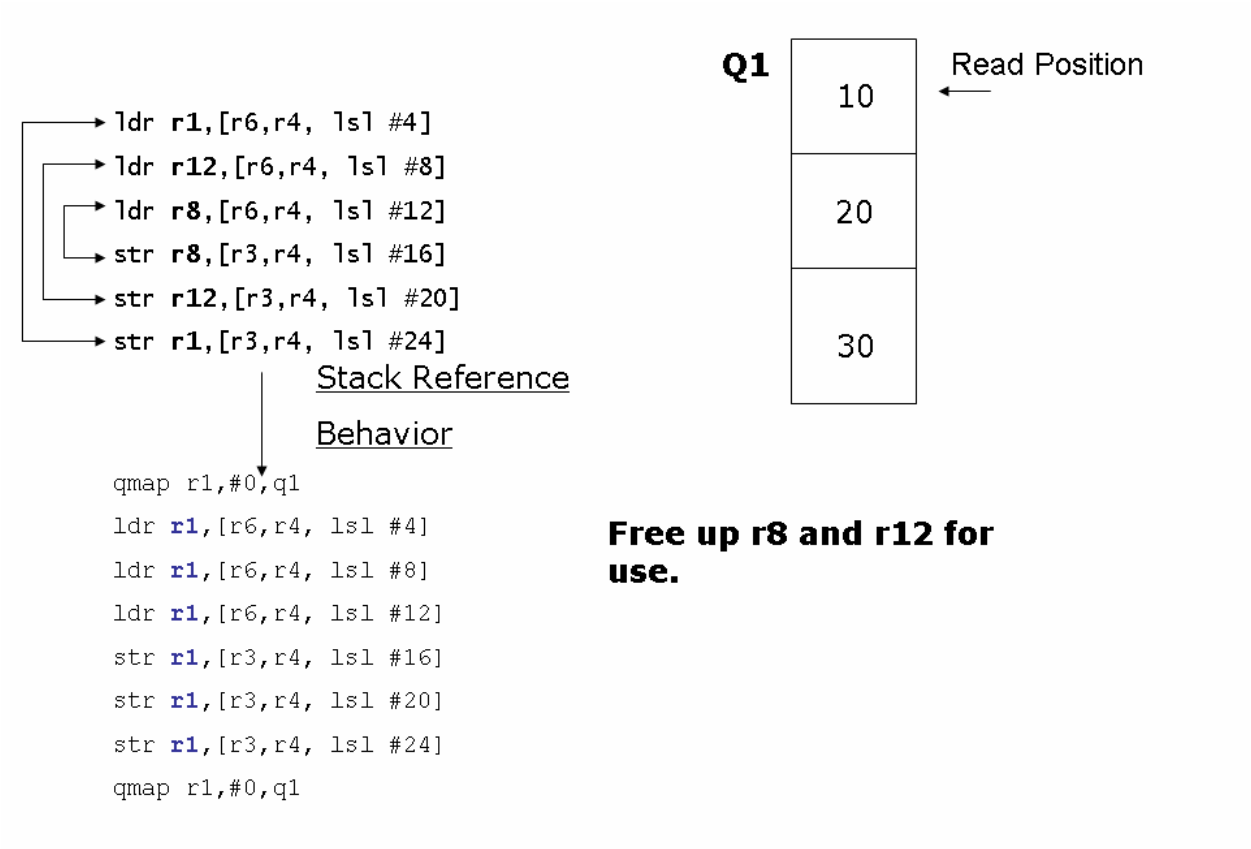


Figure 5.1 Example of Overwriting Live Ranges Using a Stack.

Destructive register queues are another identifiable register structure that can aid in reducing register pressure during general register allocation. The reference pattern that is identified for these structures are one-set, one-use patterns that overlap with the set of the next live range contained in the previous live range, but the use must occur after the previous live range has ended. To exploit this reference pattern, we use a register structure which will only pop data off of the queue when it is read. All sets to the queue have no effect on which value will be read. This is so that the live range sizes do not have to be symmetric to exploit the reference behavior. These are a destructive read and non-destructive set register structure, the

read position is always set to the first value added and decrement only upon a use from this register structure. As with the register stacks, the mapping semantic for the destructive read register queues contain no mapping information.

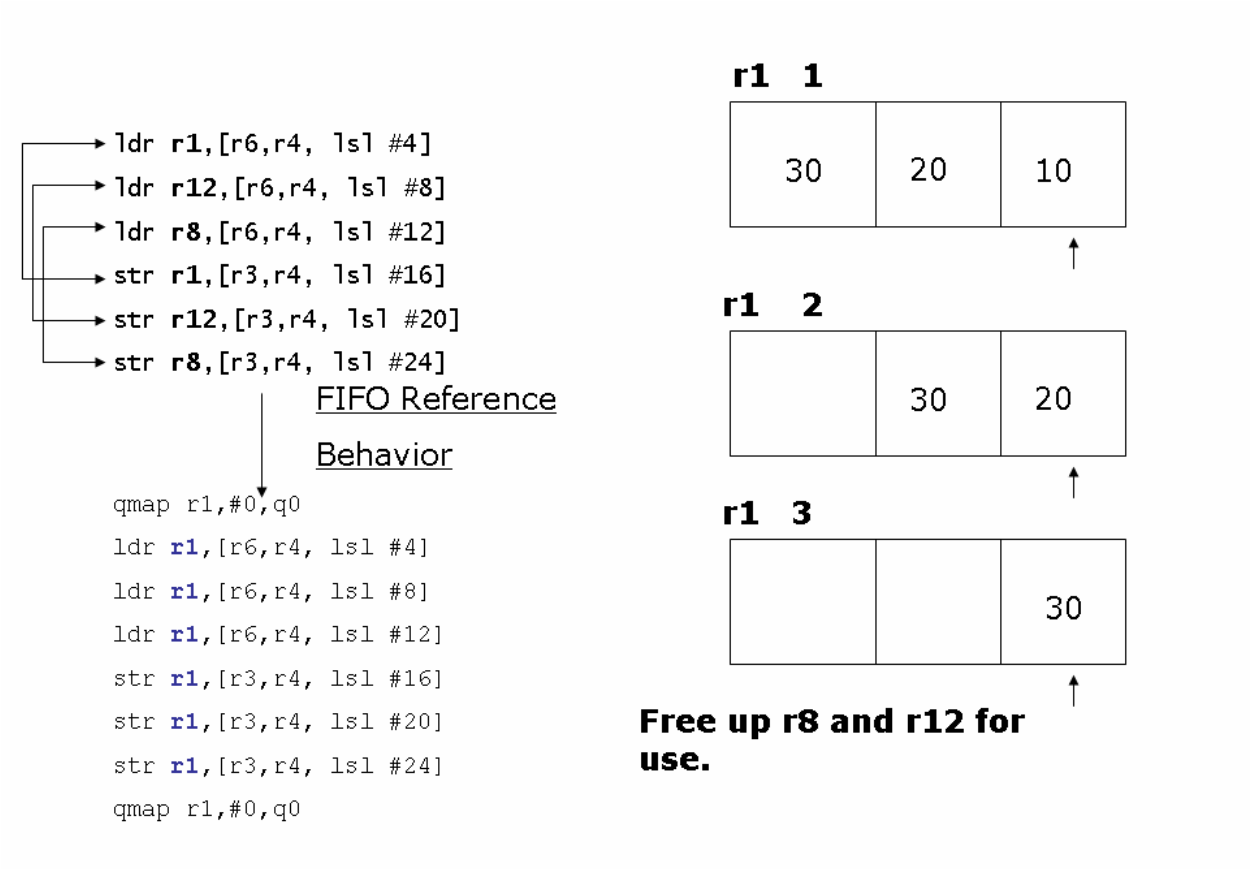


Figure 5.2 Destructive Read Queue Register Allocation

The modifications made to VPO to support this optimization was applied after original register allocation had already occurred. We added the analysis to enable register allocation using customized register files at this point so that we could provide more accurate live range analysis without having to completely rewrite register allocation in VPO. The analysis stage first detects only basic blocks that have a relatively large amount of register pressure. The blocks are then analyzed to determine if any of the live ranges contained portray any exploitable reference behavior. When these types of live ranges occur they can be over-written to the specified customized register file and a pattern list will be stored to the basic block structure itself. The registers used in the live ranges are then replaced with a customized register type. Once this has occurred VPO is then allowed to continue applying optimizations and code

generation continues with break points inserted after other optimization phases to determine whether the organization of the reference behavior that was over- written is still being exhibited. If it is found that the reference behavior has been changed by another optimization, we are able to modify the mapping to the new reference behavior or revert to code used prior to mapping and rerun the optimization phases. At the end of the optimization process before the assembly or object code is written out the custom register specifiers used are replaced by a standard register specifier and the mapping is created.

CHAPTER 6

Software pipelining

Software pipelining [4] is an aggressive loop optimization that can be applied to loops in an attempt to free each iteration of the loop from stalls incurred by dependencies involving high latency operations. Software pipelining performs this method by laying out the kernel of a loop iteration by iteration with stalls incurred. By laying out the kernel of this loop in such a manner and setting an iteration interval eventually a software pipelined kernel will be created and the preceding information will become the prologue of the software pipelined loop and the instructions following the loop will become the epilogue of the software pipelined loop. By applying this procedure, the compiler is able to remove a large portion of the latency incurred and fill it in with independent instructions. Figure 6.1 displays an original loop kernel with stalls included. These stalls flow through the pipeline upon every iteration of the loop, significantly increasing the cycles per instruction for the body of loop. Figure 6.2 shows the software pipelining method described above with the generated kernel code shown in bold in the figure.

```
Stalls Present when Loop Run
.L3:
  ldr    r1,[r2,r3, lsl #2]
  ldr    r12,[r4], #4
           stall
           stall
           stall
  mul    r0,r12,r1
           stall
           stall
           stall
           stall
           stall
           stall
           stall
  str    r0,[r5,r3, lsl #2]
  add    r3,r3,#1
  cmp    r3, #1000
  blt    .L3
```

Figure 6.1 Stalls inserted in loop.

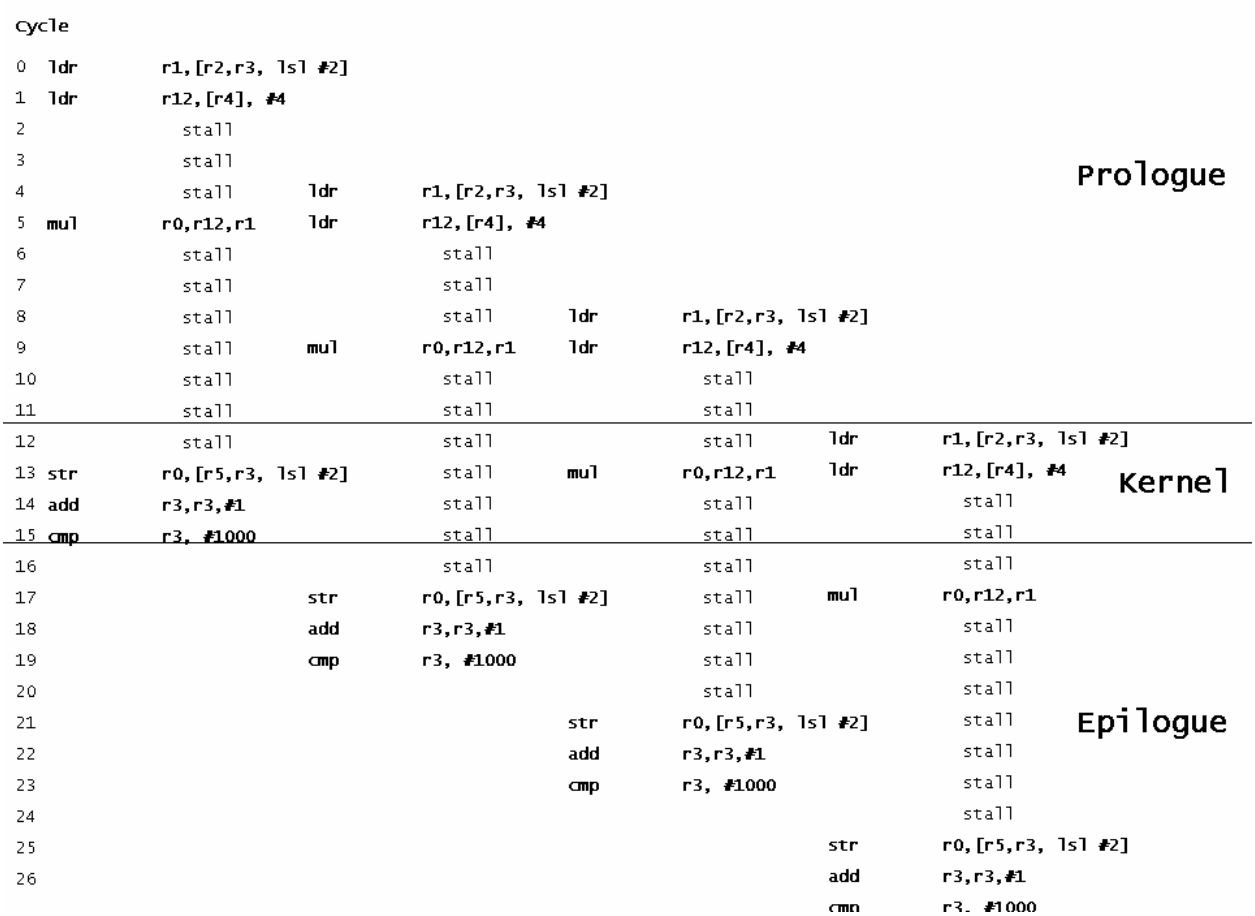


Figure 6.2 Generic Pipelining Example

Modulo scheduling [5] is an optimized method of software pipelining that enables the compiler to determine the pipelined loop body through simpler mechanisms. Software pipelining, as described previously, requires methods of storing the unrolled and iterated loop body several times over. Modulo scheduling provides a resource constrained method of pipelining the dependencies in a loop through a simpler scheduling mechanism. Modulo scheduling takes into account dependence and resource constraint and schedules the loop instruction by instruction. The prologue and epilogue code are generated after the kernel is created in modulo scheduling. Our method of software pipelining requires first creating dependence graphs between the instructions. By creating dependence graphs and analyzing them we are able to assign a priority to instructions. By using a highest priority first function we schedule each instruction one at a time. For each instruction to be scheduled we calculate an earliest start and a maximum time so that the instruction is not scheduled ahead of any instructions which it might be dependent on. The maximum time can be several iterations away

from a prior dependence for a long latency operation. If an instruction cannot be scheduled in the allotted time it is scheduled at the first available time and all instructions that have a dependence node with this scheduled instruction are invalidated and added back to the “to be scheduled” list. This method suffers from one drawback. It is unable to schedule cyclic loop carried dependences. Figure 6.3 (a) Shows the original vector multiply loop with the latencies of the dependent operation shown. By applying the modulo scheduling algorithm to this example we organize a loop shown in the scheduled kernel in Figure 6.3 (b) that eliminates a majority of the cycles wasted due to stalls. When we create a modulo scheduled loop we store an iteration value for each scheduled instruction. This iteration value can be later used to determine the instructions that need to be extracted to the prologue and the epilogue to create correct code. Figure 6.3 (b) shows the extracted prologue and epilogue and the modified kernel put together to enable this modulo scheduling.

```
.L3:
  ldr     r1,[r2,r3, lsl #2]
  ldr     r12,[r4], #4
            stall
            stall
            stall
  mul     r0,r12,r1
            stall
            stall
            stall
            stall
            stall
            stall
            stall
            stall
  str     r0,[r5,r3, lsl #2]
  add     r3,r3,#1
  cmp     r3, #1000
  blt     .L3
```

(a) Original Loop

```
Generated Prologue
qmap r0,#2,q1
 ldr r1,[r2,r3, lsl #2]
 ldr r12,[r4], #4
 mul r0,r12,r1
 ldr r1,[r2,r3, lsl #2]
 ldr r12,[r4], #4
 mul r0,r12,r1
Scheduled Kernel
0 ldr r1,[r2,r3, lsl #2]
1 ldr r12,[r4], #4
2 str r0,[r5,r3, lsl #2]
3 add r3,r3,#1
4 cmp r3, #1000
5 mul r0,r12,r1
6 bgt .L3
Generated Epilogue
str r0,[r5,r3, lsl #2]
add r3,r3,#1
str r0,[r5,r3, lsl #2]
add r3,r3,#1
qmap r0,#0,q1
```

(b) Modulo-Scheduled Loop with Queues

Figure 6.3 Applied Modulo Scheduling.

One of the primary reasons that software pipelining isn’t performed in many machines with a reduced register set is because software pipelining causes a significant increase in

register pressure to the optimized loop. Using modulo scheduling we are able to more accurately identify an optimal kernel which reduces typical software pipelining register pressure, but not enough to make it feasible. The reason for the increase in register pressure is because of the extracted loop iterations moved into the prologue and epilogue. When instructions are duplicated and moved to the prologue to enable an early start for the high latency instructions it requires register renaming to keep the values correct. In the small example above, we would have to rename approximately three registers and add moves to insure correct values across the extracted iterations. One method previously presented to target this specific problem with software pipelining was the rotating register file which adapted a special set of registers to be able to perform the renaming and passing of the values automatically [6]. Rotating registers require that each register used in the rotating register file be accessible in the instruction set. This method of handling the extra registers would require a significant addition to, not only the hardware, but the instruction set as well.

Application configurable processors and the customized register structures can enable software pipelining for architectures where it had been previously infeasible. The extra registers required for software pipelining are similar to storing the values from previous iterations as they occur in recurrence elimination. These loop carried values have a FIFO reference behavior which can be exploited and mapped into a register queue. By identifying the loop carried values forced by software pipelining during the scheduling process, we are able to create a mapping of live range into a queue that will provide us with the extra storage space to enable software pipelining. A requirement of this application of register queues is that the register specifier for the register queue is live across the prologue, epilogue, and body of the loop. The enforcement of this requirement can make it difficult to apply register queues to loops that already have extremely high register pressure. To provide our register queues with every opportunity to optimize, we are able to use other customized register structures to reduce register pressure within the loop. Using these structures can free up available registers to allocate across our increased loop body. One method of reducing inner loop register pressure is applying a register circular buffer to the loop to consume the invariant registers that are used within the loop. By mapping the pattern of values into a circular buffer, we may be able to free up values throughout the entire loop. This register savings allow us to perform software pipelining with register queues in higher register pressure situations. Figure 6.4 shows our previously modulo scheduled example utilizing register queues to contain the loop carried value. This small example when extracted would require a few extra registers to store values generated during the prologue of the newly formed loop. As the example shows, during the execution of the

prologue two values are written into the queue mapped to register r0. When register r0 is accessed in the body of the loop, it will get the value from the second position inside of the queue. Each subsequent iteration of the loop will push the values of the queue forward allowing the store to attain the correct value during each iteration. This small example can be scaled many times than what is shown and contain many more live ranges. This flexibility allows us to employ software pipelining in reduced register environments. By applying software pipelining in this example we were able to remove the 10 stalls shown in Figure 6.3 completely from the loop. When the comparison and increment are looked at we can see that this loop iterates 1000 times. By applying modulo scheduling to this small example we were able to roughly reduce the execution of this loop by 10,000 cycles.

```
qmap r0,#2,q0
ldr r1,[r2,r3, lsl #2]
ldr r12,[r4], #4
mul r0,r12,r1
ldr r1,[r2,r3, lsl #2]
ldr r12,[r4], #4
mul r0,r12,r1

L3:
ldr r1,[r2,r3, lsl #2]
ldr r12,[r4], #4
str r0,[r5,r3, lsl #2]
add r3,r3,#1
cmp r3, #1000
mul r0,r12,r1
bgt .L3

str r0,[r5,r3, lsl #2]
add r3,r3,#1
str r0,[r5,r3, lsl #2]
add r3,r3,#1
qmap r0,#0,q0
```

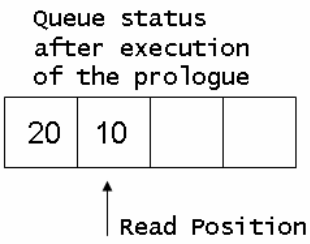


Figure 6.4 Register Queue Applied to Loop.

We used the VPO compiler to enable this research. The compiler at the time of the research was without any type of software pipelining implementation. We used the Modulo Scheduling algorithm to implement our software pipelining method. To enable modulo

scheduling we add dependence graphs to VPO, and the ability to calculate height based priority as explained in the Modulo Scheduling paper [5]. We added software pipelining to the group of loop optimizations. Adding this optimization here meant that aggressive register allocation had already occurred. This often meant to that the same registers specifiers could potentially be used to specify different live ranges within the same loop block to reduce register pressure. The reuse of these specifiers could create difficult mapping situations for the queues and to overcome these issues a new register type was added to the compiler. The new register type was a q specifier that would only be used during the compilation process.

The algorithm we applied first identifies suitable loops for software pipelining that are identified as matching specific criteria. Two renaming phases were used in determining which live ranges would be mapped into a register queue. The first renaming sequence was performed prior to modulo scheduling. A backup copy of the loop was created and any instructions that had latency greater than the iteration interval of the loop would have its live range renamed to our new register type. This new register type would disable the compilers register analysis optimizations from incorrectly identifying our modified RTL's as dead and remove them. The second renaming occurs after scheduling has been applied and the rest of the newly formed loop carried dependencies could be identified to be promoted to a queue. The next step uses iteration calculations that were determined during the scheduling of the loop to generate a prologue and epilogue for the newly formed kernel. Having previously replaced the loop carried live ranges with our own custom register type; we are then able to identify registers to map into queues to contain the reference behaviors. The pseudo code for applying registers queues to modulo scheduling is provided in Figure 6.5. One area of interest shown in the algorithm presented in Figure 6.5 are the issues of memory addresses. Aliasing and recurrences can make software pipelining infeasible, if the number of iterations extracted are greater than the size of the recurrences. Fortunately we can use the analysis and optimization performed in recurrence elimination to enable us to detect these types of cases and avoid generating incorrect code. When we detect an alias or or overlapping memory address might occur that could create incorrect code, the software pipelining is abandoned and the original loop is restored.

```
Foreach loop:
    if inner most loop
        create priority graph
        identify obvious queue canidates
        modulo schedule based on priority
        identify generate queue canidates
        generate prologue code
        generate epilogue code
        find available registers for mapping
        foreach mapping needing register
            while conflicts
                identify and resolve possible register conflicts
                associate available register with mapping
        resolve memory offset conflicts
        insert mapping code into prologue and epilogue
    if successful
        write generated pipelined code over the old loop
    else
        clear generated code
        continue to next loop
```

Figure 6.5 Modulo Scheduling and Queue Mapping Algorithm

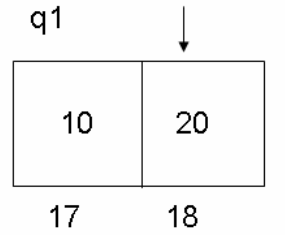
By applying a new register type to the VPO compilation environment, we had to overcome many challenges to insure proper compilation. Live and dead variable analysis are functions in VPO which determine the live ranges of variables and determine which RTL's can be removed. By allocating the same register specifier to point to either the architected register file or a customized register structure, we confused VPO's concept of a register. A register located in the architected register file cannot be set twice in two consecutive instructions and still contain both values. This type of situation would cause assignment elimination to remove the first set of the two consecutive sets. When the same register specifiers are mapped into queues for software pipelining, this situation becomes quite common. We had to get around this problem when the mappings were final by designating a special RTL template that would stop dead variable elimination from removing the mapped registers specifiers live range.

Another issue when employing software pipelining when using register queues is in the dependence calculation inside of the simulator itself. When a dependence is calculated in the simulator it will check the register specifier to determine if the current instruction is dependent on an instruction already being passed through the pipeline. By employing a method of obfuscating the register specifier we are no longer able to determine if a dependence occurs with determining whether the register specifier is mapped into a customized register file. The method we used to solve this is a tiered dependence check. The only addition to dependence

checks for the architected register file are a map table check to provide whether or not the specifier is actually in the architected register file. If the specifier is not mapped to the architected register file, then each unique position within the customized register files resolve to a unique address which can then be used to determine a dependence in the customized register structures. This situation does not arise very frequently, in that the compiler typically will not schedule customized register files in situations that these dependences can occur. Figure 6.6 shows the code from our previous example and where a dependence appears to have occurred. There is actually no dependence between these two instructions because the specifier read in by the store in a queue resolves to specifier 18 while the set in the previous multiply is actually setting the specifier at 17.

```
Generated Prologue
qmap r0,#2,q1
 ldr r1,[r2,r3, lsl #2]
 ldr r12,[r4], #4
 mul r0,r12,r1
 ldr r1,[r2,r3, lsl #2]
 ldr r12,[r4], #4
 mul r0,r12,r1
 Scheduled Kernel
 0 ldr r1,[r2,r3, lsl #2]
 1 ldr r12,[r4], #4
 2 str r0,[r5,r3, lsl #2]
 3 add r3,r3,#1
 4 cmp r3, #1000
 5 mul r0,r12,r1
 6 bgt .L3
 Generated Epilogue
 str r0,[r5,r3, lsl #2]
 add r3,r3,#1
 str r0,[r5,r3, lsl #2]
 add r3,r3,#1
 qmap r0,#0,q1

(A) False Register Dependence
```



(b) Unique identifier

Figure 6.6 Customized Register Structure Unique Naming

CHAPTER 7

Results

Using the simple scalar simulation environment we were able to conduct our experiments using several different processor setups with varying latencies. The results were collected using the simple scalar cycle accurate simulator sim-outorder. For our software pipelining experiment, sim-outorder was run in-order with varying functional unit latency for the different runs of the benchmarks. This experiment was run using the VPO [1] compiler backend ported for the ARM ISA. The compiler had to be modified to be able to support the use of different register types.

We modified the machine design in (Simple-Scalar) to be able to support the different designs that we identified as being useful. These were the destructive and non-destructive read queues, stacks and circular buffers for our work so far. Each of these structures was implemented as a circular buffer that would behave differently when a set or use occurred and also performed different tasks depending on the instruction mapping semantic explained in the ISA support description previously. Our preliminary tests were performed on a simple in-order ARM processor with added feature support for our customized register structures and the added instruction to map and unmap these structures.

We took our first group of results using several DSP benchmark kernels. We measured the steady state performance of the loops. Figure 6 depicts the percent difference in performance from software pipelining with register queues vs. the base line loop which could not be pipelined without queues because of register pressure. Our preliminary results showed us in Figure 7.1 that as the latency grows for multiplies, we are able to successfully pipeline the benchmarked loops and realize up to a 60% improvement in performance. The increase in the multiply latency is a realizable factor in low power embedded design. Many low power multipliers trade off extra cycles in order to obtain considerable savings in power. These improvements do come at the cost of increased code size of the loop of up to roughly 300% in some cases, but this is due to the prologue and epilogue code needed by software pipelining to align the loop iterations. Figure 7.2 shows the performance savings as the load latencies rise. These loops often provide more high latency instructions to schedule out of the loop. In many of the lower latency tests, iterative modulo scheduling was able to generate a loop that did not need a prologue or epilogue. In many of our benchmarks we found that by applying software

pipelining with register queues, we are able to circumvent increasing register pressure in many simple cases by as much as 50% for the ARM. This means that software pipelining would require 50% of the usable registers for the ARM in order to even be applied. The performance loss in fir in Figure 7.1 is due to the scheduling of a loop carried inter-instruction dependency. This is the main drawback of using a modulo scheduling algorithm. When these dependencies occur, the algorithm is unable to schedule other dependent instruction in a manner to consume the latency. In this example the best effort was made to consume the cost of the multiply by scheduling the loads, however the jump in latency from 16 to 32 was unable to completely remove the cost of that specific multiply. Table 7.1 show an relationship between the original number of registers found in a few of the loops which we software pipelined and the number of registers needed to pipeline the loops using our customized register structures. The final column in the table shows the number of registers which the customized register structures consumed. This table should make it fairly obvious that the number of registers consumed by the customized register structures is able to take a great deal of pressure off of the architected register file. One special consideration displayed in the table is in regards to the Mac benchmark. The original loop used 10 architected registers for the storage requirements for the loop. When this loop is scheduled we are able to apply our circular buffers to this loop and consume some of the loop invariant registers. This shows that even after we've applied modulo scheduling to the loop we are able to reduce register pressure within the loop.

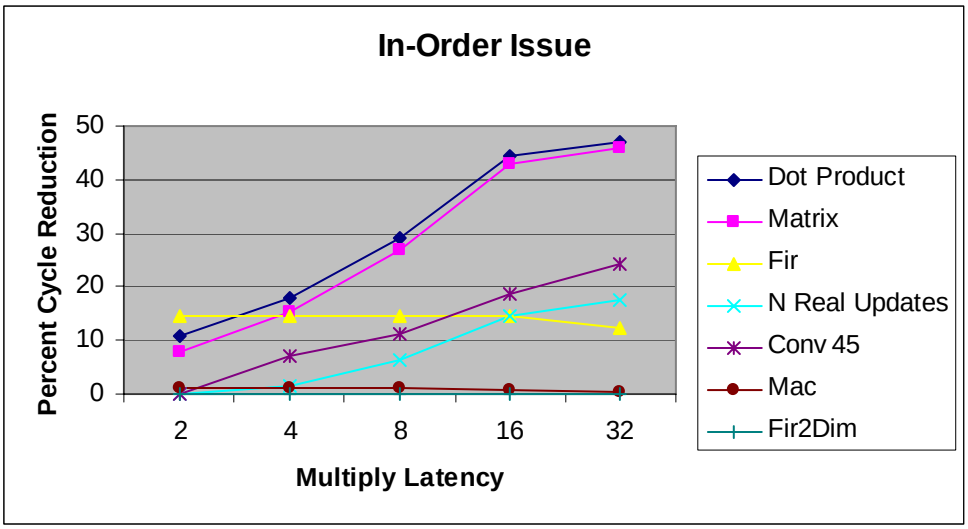


Figure 7.1 Scaling Multiply Latency Results

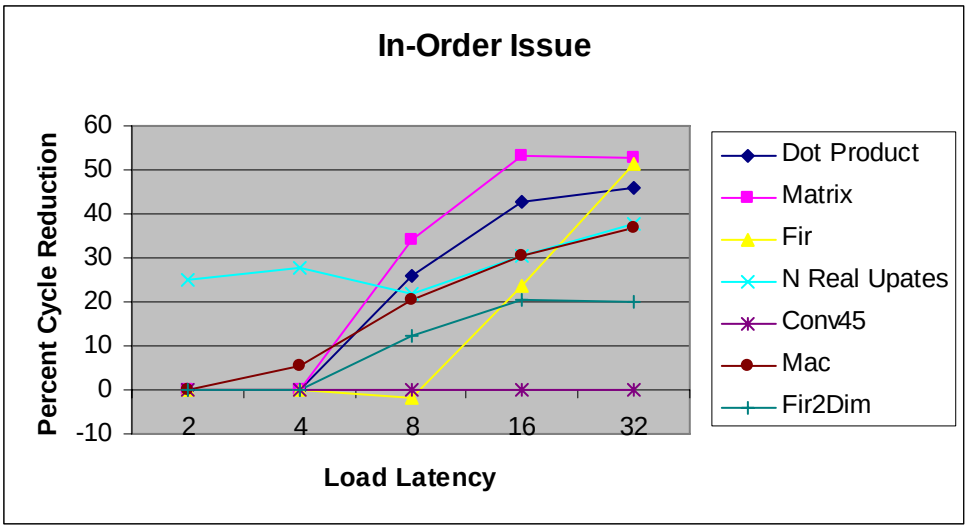


Figure 7.2 Scaling Load Latency Results

Table 7.1: Scaling Load Latency: Register Requirement Increase

Benchmark	AR in Original Loop	AR needed to Pipeline	AR contained in customized structures
	Loads 8x4 Register Savings Using Register Structures		
N Real Updates	10	10	6
Dot Product	9	9	4
Matrix Multiply	9	9	4
Fir	6	6	4
Mac	10	8	10
Fir2Dim 3 Similar	10	10	4
	Loads 16x4 Register Savings Using Register Structures		
N Real Updates	10	10	6
Dot Product	9	9	4
Matrix Multiply	9	9	4
Fir	6	6	4
Mac	10	8	12
Fir2Dim	10	10	4
	Loads 32x4 Register Savings Using Register Structures		
N Real Updates	10	10	9
Dot Product	9	9	8
Matrix Multiply	9	9	8
Fir	6	6	12
Mac	10	8	18
Fir2Dim	10	10	8

Our results are very consistent with the general results of software pipelining. This optimization provides significant cycle savings due to the reduction of stalls per loop. One of the important issues that are not shown in our figures is that as our cycle latency for our operations grow, software pipelining will find it necessary to extract more iterations of the loop to the prologue in order to absorb the large cycle latency. For example the DSP Matrix kernels that we used as a benchmark primarily absorbs the cycles caused by the large multiply latency. When the latency is four the optimization extracts two iterations and detects two instances of loop carried dependencies that need to be mapped into a register queue. In a conventional ARM this would increase the needed registers by 25%. As the cycle latency for these instructions rise, the numbers of registers needed to absorb the extracted iterations increase to well over 300% of the available registers for the machine.

These series of results show how a superscalar implementation of our experiment could provide even greater performance benefit. Superscalar issue is a possible method of increasing instruction level parallelism in modern computers. Applying superscalar implementation to reduced register architectures can effectively increase the latency of dependent instructions though the increase in ILP can often improve performance. By applying software pipelining to a situation which might potentially increase dependencies we are able to reduce the increase in latency significantly. Figure 7.3 and 7.4 shows a scaling multiply latency for both superscalar issue 2 and issue 4. The savings in cycle reduction increase as the issue levels increase. Figure 7.5 and 7.6 show scaling load on a superscalar implementation with issues of 2 and 4. These results shows us that in many situations it is preferable to focus modulo scheduling on loads to provide greater savings. Loads may often have more opportunities to be scheduled because in our experiments the load instructions were far more frequent than the higher latency multiplies. As with Fir from the in-order single issue results, N Real Updates also has a point in it at which modulo scheduling is unable to completely remove the latency from the loop. In this situation modulo scheduling simply schedules the best within its ability given the modulo scheduling budget constraints. The examples below show results that are very consistent with the results we obtained from a single issue in-order pipeline.

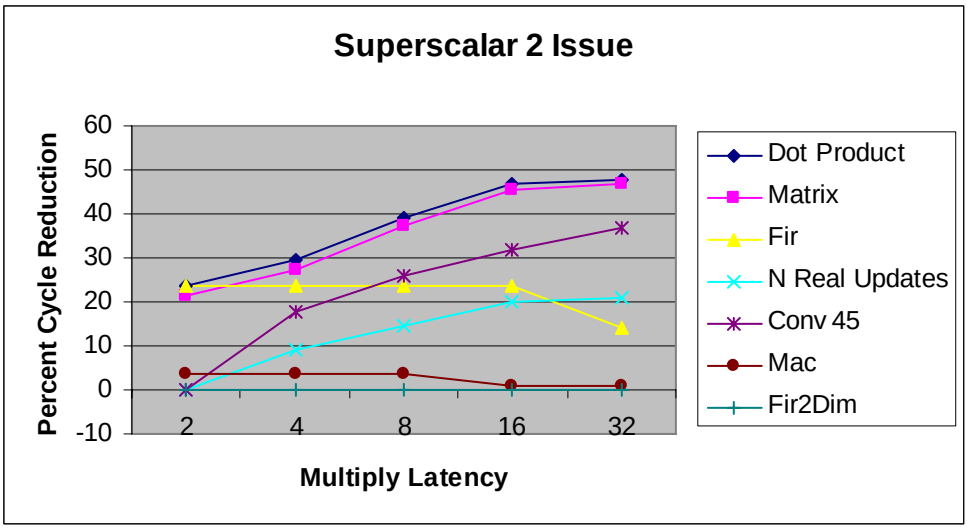


Figure 7.3 Superscalar 2 Issue Scaling Multiplies

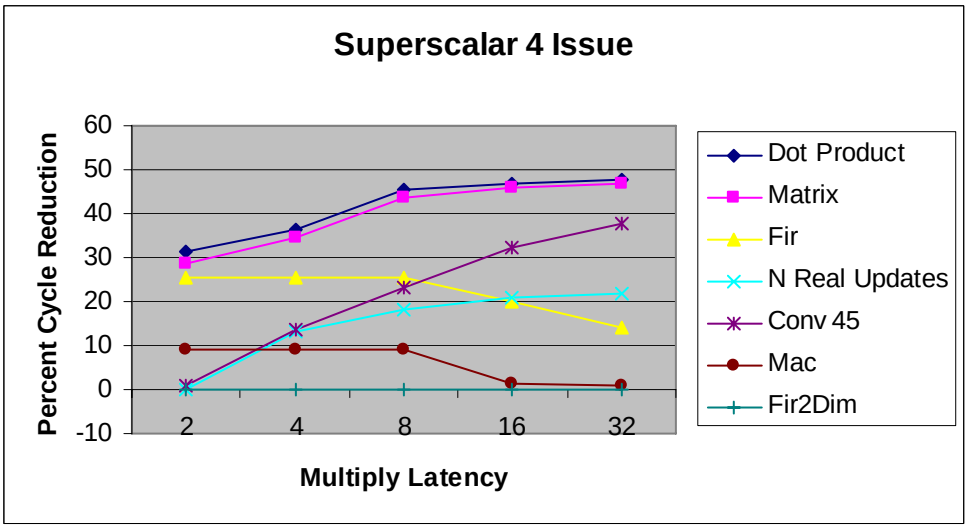


Figure 7.4 Superscalar 4 Issue Scaling Multiplies

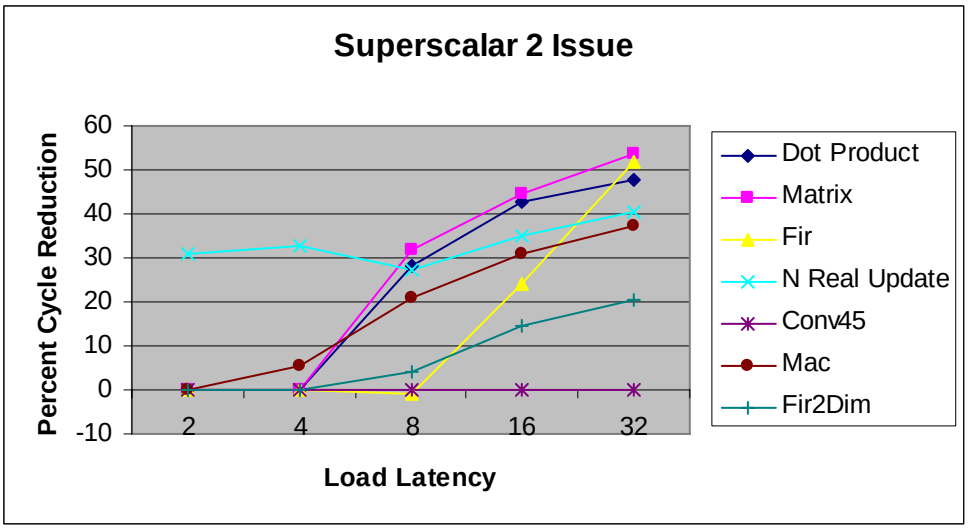


Figure 7.5 Superscalar 2 Issue Scaling Load Latency

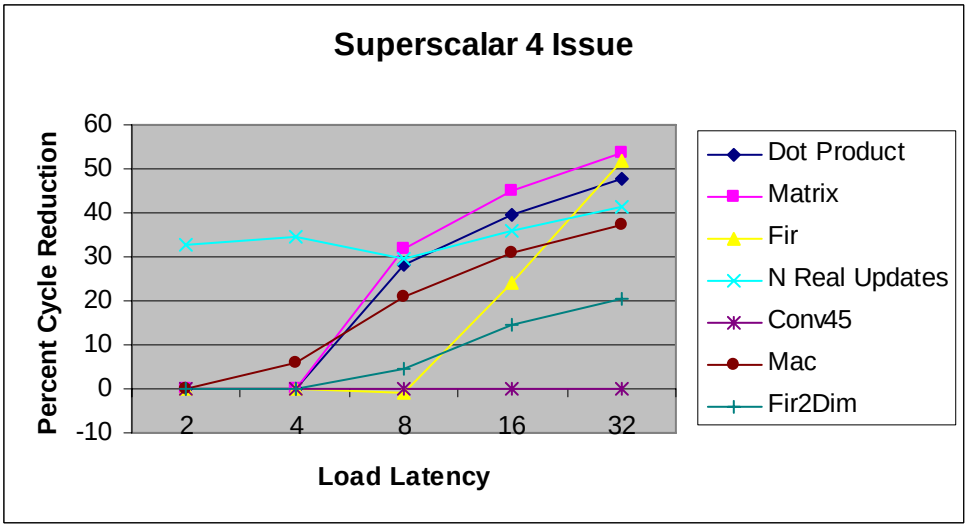


Figure 7.6 Superscalar 4 Issue Scaling Load Latency

The results show a reduction in performance and also a reduction in the number of registers needed by an optimization that significantly increases register pressure. The results shown are for loops that contain high latency operations are still relatively simple. One of the problems that currently restrict this research is that to effectively gain the savings in software pipelining using register queues, the mapping must exist over the entire range of the prologue loop and epilogue code. In loops of extremely high register pressure, where many of the registers are used in very complicated live range sets, finding an available register to map across the entire scope of this software pipelined loop can often be a limiting factor. However, it is often the innermost loops that are doing the most complicated and time consuming work of

the application. This means that even though we are limited to which loops we can apply this technique, the cycle savings are significant enough to warrant these extensions in many cases.

CHAPTER 8
Related Work

There have been several approaches for reducing the register pressure caused by software pipelining. These methods for reducing register pressure work under similar constraints, as our register queues; however, register queues offer much more flexibility without the cost of significantly modifying the ISA.

The WM machine [8] is a completely different concept of the traditional machine that utilizes FIFO queues that operate independently and asynchronously to manage the many different aspects of the traditional pipeline. This system is designed as a series of connected queues that manage the different functions of the pipeline. This paper introduced the concept of using queues in place of registers as a quick storage mechanism.

The register connection [3] approach introduced the idea of adding an extended set of registers to an ISA as a method of reducing register pressure in machines with a limited number of registers. This method employed a mapping table to associate a register in the register file with one of the extended registers. Each change in the mapping required the execution of an instruction. The map table used a one to one mapping for each register in the extended set. The register connection approach worked well for remapping scalars and various other data, but the overhead of mapping became expensive when using arrays and other large data structures.

Register queues [7] is the approach that is most similar to ours. Using register queues to exploit reference behaviors found in software pipelining showed that this method is effective in aiding the application of these optimizations. Exploiting the FIFO reference behavior that is caused by software pipelining, register queues was an effective means of holding the extra values across iterations and this significantly reduced the need to rename registers. However, this method limits the types of loops that can be effectively software pipelined because of constraints set by the reference behavior of the queues themselves. Our method described in this paper is an automation of this system with the addition of several tools which aid us in employing register queues to software pipelined loops.

Rotating registers [6] is an architectural approach for more effectively using registers to hold loop-carried values than simple register renaming. A single register specified can

represent a bank of registers which will act as the rotating register base. Use of rotating registers is similar to the renaming that would typically occur in software, but instead is all accomplished in the hardware. This method requires that each of the registers in the rotating bank be an accessible register, which in a conventional embedded architecture would require a larger specifier for a register that may not be possible in the given ISA. Application configurable processors provide much of the flexibility of the rotating register file, with only a small added cost for each access.

CHAPTER 9

Future Work

Using customize register structures to enhance optimization phases in a compiler has so far proven to be a successful mechanism in gaining several types of benefits. Some possible future work would be to analyze the different optimizations that are common throughout compilers and figure out the structures and ways that they could be enabled to perform better using an *application configurable processor*. VPO contains many different optimizations that could potentially be analyzed and matched with some sort of customized register structure to enable a level of optimization. Future work would also include adding from scratch more aggressive optimizations similar to the work we did for software pipelining. By applying many more aggressive optimizations and exploiting them using customized register files, it could be possible to see even greater improvements in application performance on embedded processors.

We have currently only experimented with a few customized register structures and have found success in the few we have tested. However, the applications for what other customized register structures could do is also a very fascinating area of research. One possible structure is similar to content addressable memory and utilizing a structure that could have multiple mappings and employ its own data management similar to cache, but allowing for much faster access times. By employing this as another customized register file we could use our ability to map several register specifiers into the same customized register and increasing the value storage ability for the general architected register file. A simple way this could work is by having the multiple specifiers mapped into a customized register file, we could employ a method of storing a cam tag in the architected register file itself and use that tag to provide a lookup for a value that we might find in a customized register cam. Since these customized register files are self regulated they could be designed with their own replacement algorithm and effectively create a cache-like system that could be accessed extremely quickly.

Chapter 10

Conclusion

Our work has shown that using an *Application Configurable Processor* can greatly reduce the register restrictions that inhibit many compiler optimizations in embedded systems. By reducing this pressure in our work with software pipelining we've seen some very good performance increases for our benchmarks. We have shown that these modifications allow the performance enhancing optimizations to occur and require very little change to the ISA itself and only minor hardware modification. Our research has demonstrated that it is possible to modify compiler optimizations to automate the allocation and modification of these customized register structures to make existing optimizations more effective. Our future work with *application configurable processors* will be using the different customized register files to exploit other identifiable reference behaviors caused by code optimizations and to aid the compiler in identifying these situations for the optimizations which are available.

References

[1] M. E. Benitez and J. W. Davidson. “A Portable Global Optimizer and Linker”. *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation*, pages 329-338. ACM Press, 1988.

[2] D. Burger and T. Austin, “The Simplescalar Tool Set, Version 2.0,” Technical Report 1342, Department of Computer Science, University of Wisconsin-Madison,1997.

[3] 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

[4] M. Lam, “Software pipelining: an effective scheduling technique for VLIW machines”, In *Proceedings of the ACM SIGPLAN ’88 Conference on programming language Design and Implementation*, pages 318-327, 1988

[5] B. R. Rau , “Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops”, *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63-74, Nov. 1994.

[6] 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 Implenetation*, Pages 283-299, June 1992

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

[8] W. A. Wulf, “Evaluation of the WM Architecture”, In *Proceedings of the 19th Annual International Symposium on Computer Architecture,* pages 382-390, May 1992

Biographical Sketch

Christopher J. Zimmer

Christopher Zimmer was born in Worthington, MN on May 28th, 1982. He attended Florida State University and completed an undergraduate degree in Computer Science in Spring of 2004. In Fall 2006, he received his Masters of Science Degree in Computer Science from Florida State University. Following the completion of this degree he began pursuing schools to continue his education in the area of compilers and computer architecture.