THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

EFFECTIVE EXPLOITATION OF A LARGE DATA REGISTER FILE

By

MARK C. SEARLES

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 Mark Searles defended on September 29, 2006.

David Whalley
Professor Co-Directing Thesis

Gary Tyson
Professor Co-Directing Thesis

Xin Yuan
Committee Member

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

For Tink and her pixie dust

ACKNOWLEDGEMENTS

I would like to thank my advisors – Dr. David Whalley and Dr. Gary Tyson – for their, well, advice, guidance, and direction. I also would like to thank the other members of the Compilers Group, in particular the senior members, who consistently offered their expertise on the research compiler/simulator infrastructure. Your insights were invaluable as they saved countless hours by allowing me to leverage the existing framework, to suit my needs, rather than reinventing it.

Lastly – and most importantly – I thank my wife, Jennifer, without whom none of this would have been possible. Your unfailing and unwavering support is invaluable; I love you with all my heart.

TABLE OF CONTENTS

List of Tables vii
List of Figures viii
Abstract ix

1. INTRODUCTION 1

1.1 Performance of the Memory Hierarchy 2

2. LDRF ARCHITECTURE 4

2.1 Architecture Support 4
2.1.1 Use of Register Windows in the LDRF 5
2.2 LDRF Access Time 6
2.3 Identification of Candidates for the LDRF 7
2.3.1 Characteristics of the Ideal Candidate Variable to be Promoted to the LDRF 7
2.3.2 Restrictions on Variables Placed in the LDRF 8

3. EXPERIMENTAL FRAMEWORK 10

3.1 Compiler 10
3.2 Simulator 10
3.3 Benchmarks 11
3.3.1 MiBench Embedded Applications Benchmark Suite 11
3.3.2 Digital Signal Processing Kernels 12
3.4 Experimental Test Plan 13

4. COMPILER ENHANCEMENTS 15

4.1 Frontend Modifications 15
4.1.1 Gregister and Gpointer Keywords 15
4.1.2 Syntax, Semantic, and Type Checking 17
4.1.3 Type Extension of LDRF Variables 18
4.2 Middleware Modifications 19
4.3 Backend Modifications 20
4.3.1 Modifications to Generate Naïve LDRF Code 20
4.3.2 Application-Wide Call Graph 20
4.3.3 Coalescing Optimization 21
4.4 Compilation Tools 31
4.4.1 LDRF Static Compilation Tool 32
4.4.2 LDRF Assembler Directive Collection Tool 32
4.5 Assembler Modifications 34
4.6 Linker Modifications 34
4.7 Compiler Analysis Tools 35

5. SIMULATOR ENHANCEMENTS 36
5.1 Sim-outorder 37
5.1.1 Analysis Tools 37
5.2 Sim-profile 39
5.3 Sim-wattch 39
6. EXPERIMENTAL TESTING 40
6.1 Experimental Results and Discussion 42
6.1.1 Execution Time Analysis 43
6.1.2 Memory Instruction Count 45
6.1.3 Data Cache Access Patterns 46
6.1.4 Power Analysis 46
6.1.5 Coalescing Optimization 47
7. RELATED WORK 50
8. FUTURE WORK 53
8.1 Automation of Variable Promotion to the LDRF 53
8.2 Permit Promotion of Dynamically Allocated Variables to the LDRF 54
8.3 Alternative LDRF Architecture to Support Non-Contiguous Accesses 55
8.4 Extension of the Coalescing Optimization 56
9. CONCLUSIONS 58
APPENDIX A: SIMPLESCALAR CONFIGURATION 60
REFERENCES 62
BIOGRAPHICAL SKETCH 64

LIST OF TABLES

Table 3.1 MiBench Benchmarks Used for Experiments 12

Table 3.2 DSP Kernels Used for Experiments 13

Table 4.1 Naive LDRF RTL Forms 19

Table 4.2 MIPS Register Names and Uses 28

Table 5.1 Sample Global Variable Behavior Data 38

Table 6.1 Selected Processor Specifications 41

Table 6.2 Benchmark Characteristics 43

LIST OF FIGURES

Figure 1.1 Performance Impact of Memory 2

Figure 1.2 Number of Bits to Contain Offset 3

Figure 2.1 LDRF and VRF Relationship 5

Figure 2.2 LDRF using Register Windows 5

Figure 4.1 LDRF Compilation Pathway 15

Figure 4.2 Variable Declaration with LDRF Storage Specifier 16

Figure 4.3 Variable Declaration with LDRF Pointer 17

Figure 4.4 Significant Optimization Stages 22

Figure 4.5 Sink Increments Algorithm 24

Figure 4.6 Coalescing Algorithm 25

Figure 4.7 Sequential LDRF References 27

Figure 4.8 Register Renaming Algorithm 29

Figure 4.9 Register Rename Example 30

Figure 4.10 Coalescing Example 31

Figure 4.11 Example LDRF References File 33

Figure 6.1 MiBench and DSP Performance Results 44

Figure 6.2 Reduction in Memory Instructions 45

Figure 6.3 Data Cache Contention 46

Figure 6.4 Percent Energy Reduction 47

Figure 6.5 Coalescing and Performance 48

Figure 6.6 Coalescing and Energy 49

Figure 8.1 Coalescing Across Blocks 56

ABSTRACT

As the gap between CPU speed and memory speed widens, it is appropriate to investigate alternative storage systems to alleviate the disruptions caused by increasing memory latencies. One approach is to use a large data register file. Registers, in general, offer several advantages when accessing data, including: faster access time, accessing multiple values in a single cycle, reduced power consumption, and small indices. However, registers traditionally only have been used to hold the values of scalar variables and temporaries; this necessarily excludes global structures and in particular arrays, which tend to exhibit high spatial locality. In addition, register files have been small and have been able to hold relatively few values, particular in comparison with the capacity of typical caches. Large data register files, on the other hand, offer the potential to accommodate many more values. This approach – in comparison to utilizing memory – allows access to data values earlier in the pipeline, removes many loads and stores, and decreases contention within the data cache.

Although large register files have been explored, prior studies did not resolve the complexities that limited their usefulness. This thesis presents a novel implementation of a large data register file (LDRF). It employs block movement of registers for efficient access and is able to support composite data structures, such as arrays and structs. For maximum flexibility, the implementation required extension of a robust research compiler, creation of several stand-alone tools to aid compilation, and modification of a simulator toolset to represent the architectural enhancement. Experimental testing was performed to establish the viability of the LDRF and results clearly show that the LDRF, as implemented, exceeds the threshold for it to be considered a useful design feature.

CHAPTER 1

INTRODUCTION

As the gap between processor speed and memory speed widens, it is appropriate to investigate alternative storage systems to minimize the use of high latency memory structures; one such alternative is a large data register file. Even if the data actually resides in a data cache rather than main memory, there are several, well-known advantages of accessing data from registers instead of memory. These advantages include: faster access time, accessing multiple values in a single cycle, alias free data addressing available early in the execution pipeline, reduced power consumption, and reduced bandwidth requirement for the first-level data cache. This thesis considers use of a Large Data Register File (LDRF) that acts as an alternative data storage location. Despite its increased size, the LDRF retains many of the advantages of a traditional register file as well as other additional advantages.

Customarily, registers have been constrained to hold the values of scalar variables and temporaries. Also, traditional register files have not supported inclusion of aliased data. In part because of these restrictions, the register file can be managed very effectively – there typically are only a small number of live scalar variables and temporaries at any given point in the execution of an application. Further, because of the limited number of live registers and the restrictions on the data that may be placed in the register file, a small register file was well suited to meet its needs. Hence, a large register file was unnecessary and even caused difficulties, in part due to longer access times and additional state to maintain context switches. The LDRF is not a replacement for the traditional register file, but rather works in concert with it. The LDRF is a large store of registers without the restrictions of a traditional register file and provides an effective storage alternative to the data cache. It relaxes many of the constraints inherent in the traditional register file. For instance, the LDRF supports the storage of composite data structures, including both local and global arrays and structs. In addition, aliased data may be stored in the LDRF. Architectural and compiler enhancements have been implemented to ensure that, despite the expanded capabilities of the LDRF, the data residing in the LDRF may be accessed as efficiently.

The remainder of this thesis is organized in the following manner. The remainder of this chapter is use to motivate this work by showing that memory operations comprise a significant portion of the overall instruction count and also to establish that eliminating these memory operations, in favor of register accesses, has the potential to greatly increase performance. Chapter 2 discusses the architectural modifications required to support the LDRF as well as the characteristics of data that is best suited for inclusion in the LDRF. In Chapter 3, the experimental framework is described to provide context for the remainder of the work. In Chapter 4, the compiler modifications necessary to support the LDRF are robustly detailed. The simulator modifications are discussed within Chapter 5. Chapter 6 presents the experimental testing and the results. Chapter 7 describes other work related to this research. In Chapter 8, thought is given as to future work that may be performed to improve the LDRF implementation. Lastly, concluding remarks are provided in Chapter 9.

1.1 Performance of the Memory Hierarchy

The performance of the memory hierarchy has long been a critical factor in the overall performance of a computer system. There are two primary reasons that this is so: (1) memory operations comprise a significant portion of the overall instruction count (see Figure 1.1, which shows 26%, on average, of instructions are memory operations for six representative MiBench [10] applications) and (2) memory speeds are significantly slower than processor speeds. Accordingly, many techniques [13][22][18] have been studied to hide the latency of main memory from the processor. However, it is not feasible to hide all such latencies. Figure 1.1 also shows that significant performance benefits would be realized if a memory operation were to complete in the same amount of time as a register operation. While it is not suggested that this

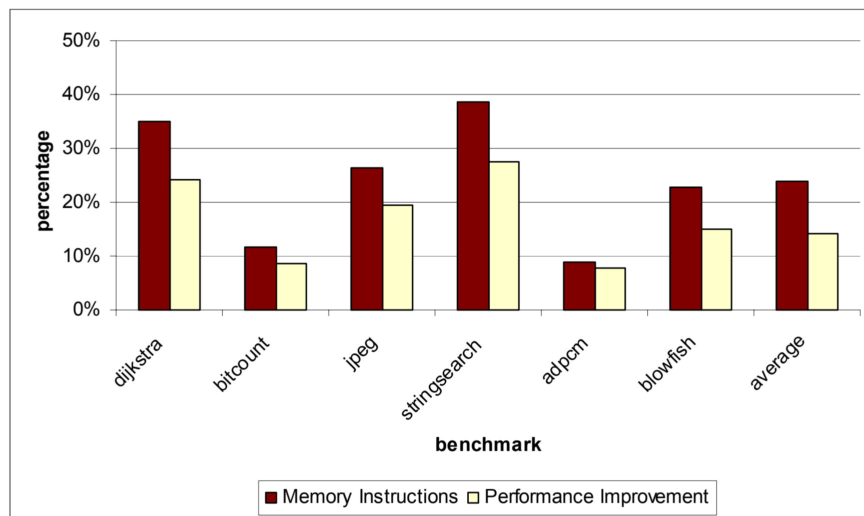

Figure 1.1 Performance Impact of Memory

goal is necessarily feasible, it will be shown that a large register file could significantly reduce the number of memory operations, in favor of register operations, which thereby can potentially realize performance gains.

Accessing memory – whether accessing a data cache or main memory – is slower than accessing a register file because memory structures are physically farther from the processor and therefore incur longer wire delays. In addition, they require a tag-lookup in order to determine if the desired data is available at the current memory hierarchy level. In addition, instructions to access memory, on most architectures, are inherently burdened by the base address plus offset calculation, which must be performed before a load/store may be attempted. In the simple, classical, five-stage pipeline, this calculation is performed in the execute stage. The load/store is then subsequently performed in the memory stage. For simplicity's sake, this base plus offset calculation is always performed even though it is often not necessary, i.e., when the offset is zero. Figure 1.2 shows the number of bits required to capture the offset associated with a memory instruction. Noteworthy is that the offset is zero for 86% and 64% of the memory instructions that access the static data and heap segments, respectively. Also, as few as four bits are needed to capture 97% and 99% of the offsets associated with memory instructions accessing the data and heap segments, respectively. On the other hand, the stack segment requires ten bits to account for 99% of its references. As will be seen, these signature characteristics – particularly of the static data segment – are of material importance to the viability of the LDRF.

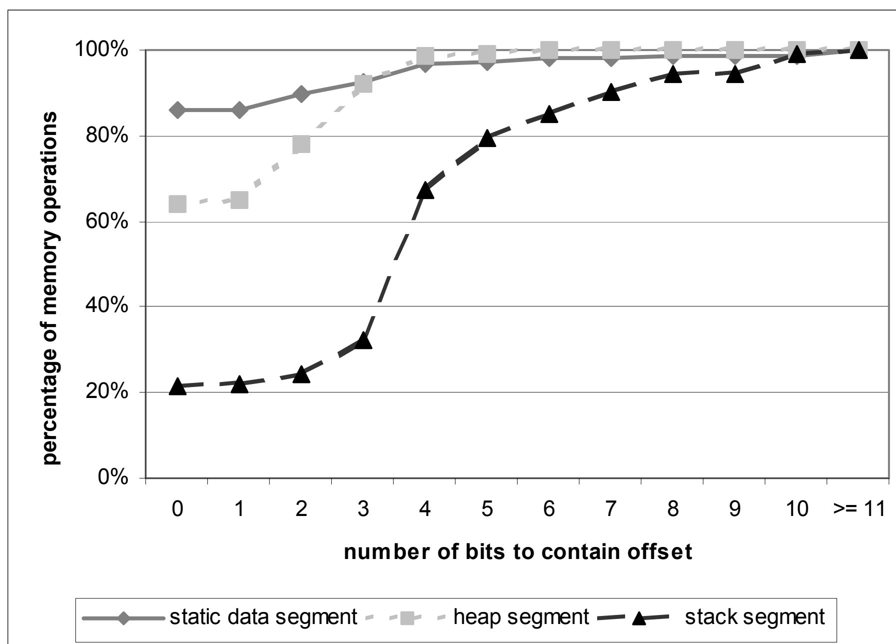

Figure 1.2 Number of Bits to Contain Offset

CHAPTER 2

LDRF ARCHITECTURE

2.1 Architecture Support

The LDRF is an architected register file that can efficiently support thousands of registers and acts as a storage system that provides data to the traditional register file. Traditionally, registers have been constrained to hold the values of local scalar variables, as well as temporaries, and have excluded composite structures. Although such a register file can be managed very effectively, there are typically only a small number of live scalar variables and temporaries at any given point in the execution of an application. In contrast, the LDRF supports storage of composite data structures, including both local and global arrays and structures. Inclusion of scalars, within the LDRF, is restricted to global scalars. In addition, aliased data may be stored in the LDRF. Architectural and compiler enhancements have been made to ensure that, despite the expanded capabilities of the LDRF, it may be accessed as efficiently as a traditional register file.

The primary differences between our approach and earlier register schemes lie in the ability to (1) promote the wider range of application data values to the LDRF and (2) perform block transfers of data to/from the LDRF. First, in contrast to earlier register schemes, global – as well as local composite variables within – may be promoted to the LDRF. Many global arrays, particularly in numerical applications on embedded processors, are well suited for the LDRF. Second, the LDRF supports block transfer of sequential registers, as depicted in Figure 2.1. After loading the registers from the LDRF to the VRF, subsequent accesses to the registers occur from the VRF in a conventional manner.

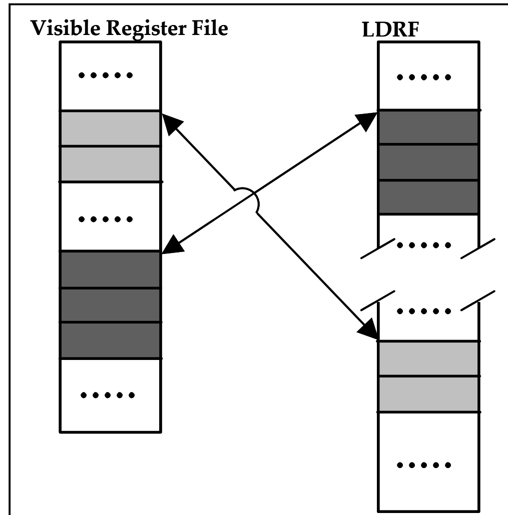

Figure 2.1 LDRF and VRF Relationship

2.1.1 Use of Register Windows in the LDRF

An initial implementation of the LDRF architecture used the similar concepts as those discussed in Chapter 2.1, but it is distinguished from the presented implementation by the use of register windows. This implementation is depicted within as depicted in Figure 2.2, where two register windows have been established. One is sized at two registers and the other is sized at three registers. The register windows were established via a register window pointer (RWP) and several RWPs were available such that several windows could be established at the same time. The RWP served to create a mapping between the VRF and the LDRF. Only those LDRF registers that were mapped to the VRF were available at any given time. A window, when unmapped such that a new mapping could be created, would cause the contents of the window

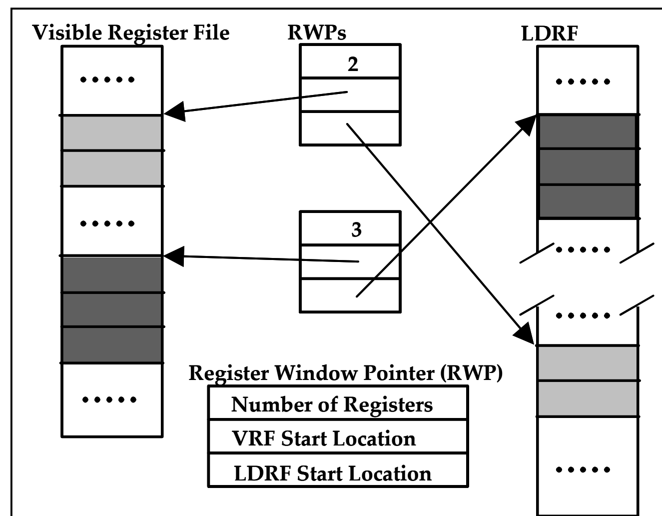

Figure 2.2 LDRF using Register Windows

to be written to the LDRF. This, in effect, is a store of the mapped portion of the VRF into the LDRF on remapping of a window. Store instructions, therefore, were unnecessary in many situations as the store was accomplished by the remapping. However, the additional encoding bits required to indicate which RWP was being accessed offset this aspect.

The register window approach, although sound, proved to be unnecessarily complex. For example, the implicit stores only occurred on a new mapping or expiration of an existing mapping, i.e., due to function return. It was, therefore, necessary to determine if an explicit store was necessary. Also, for each LDRF register that was not contained within an existing RWP, a RWP needed to be established before it could be accessed. This created an overhead burden, which is amortized over the number of registers contained within the register. However, because the complexities did not offer a sufficient return, an alternative was sought.

2.2 LDRF Access Time

The time required to load from/store to a single element of the LDRF is less than the access time to comparable memory hierarchy structures, such as the data cache. It is important to note that the ability to transfer multiple elements, via parallel register moves, compounds the savings. The reduced access time is, primarily, from two sources: (1) earlier pipeline access and (2) lower latency when compared to a data cache. In fact – at the naïve level – LDRF and memory instructions have similar characteristics and similar encoding. For example, similar to a cache, the address of a LDRF variable is included within the instruction encoding and is necessarily used when accessing that variable within the LDRF. However, unlike a cache, LDRF instructions do not support an offset. This is a significant distinction to the LDRF instruction encoding. As was shown in Figure 1.2, only a small percentage of memory instructions require a base plus offset calculation; it follows that very few LDRF instructions require a base plus offset calculation. To support the calculation within that small percentage, the calculation is performed in an instruction prior to the one that accesses the LDRF. Considering that accessing the LDRF does not require an address calculation, the values from the LDRF will be available – in terms of a simplified five-stage pipeline – after the execute stage, whereas the values from the data cache are not available – again, in terms of a simplified five-stage pipeline – until after the memory stage. Earlier access to data is a clear advantage. The second primary source of reduced access time is that of lower latency. Access time to the LDRF is in line with that of scratchpad memory, which is typically one cycle. Data cache latencies, on the other hand, may be a single cycle but are commonly two or three cycles, particularly on general-purpose machines [11]. Lastly, the data within the LDRF is known to be present; there are no LDRF

“misses”. Despite the high hit rate of most caches, misses do occur and do incur a miss penalty. The LDRF also distinguishes itself in that it does not require a data TLB lookup nor does it require a data cache tag lookup. Thus, use of an LDRF reduces conflict misses within both the DTLB and the data cache.

2.3 Identification of Candidates for the LDRF

At this time, the responsibility to allocate variables to the LDRF resides with the programmer, who uses directives within the high-level source code to specify that a given variable should reside in the LDRF. As stated, the LDRF supports scalars as well as composite date structures. Because existing schemes effectively allocate local scalars and temporaries to the traditional register file, the LDRF is best utilized by allocating arrays and structs to it. Also as afore mentioned, the LDRF supports aliased data. Recalling that the LDRF provides data to the traditional (aka visible) register file, analogous to the role of the data cache, it is reasonable that the LDRF can support aliased data, with the analogous constraints as with the data cache.

2.3.1 Characteristics of the Ideal Candidate Variable to be Promoted to the LDRF

The ideal variable that is a candidate to be promoted to the LDRF will possess the following qualities:

- *Accesses to the data should exhibit high spatial locality* – Multiple consecutive locations, in the LDRF, can be simultaneously accessed. This feature is exploited by data with high spatial locality.
- *Data should be frequently accessed* – Since the data is accessed more efficiently via the LDRF, it is advantageous to place frequently accessed data in the LDRF to maximize the performance benefits.
- *High Access to size ratio* – Considering the limited size of the LDRF, the ratio of the number of accesses of a given variable to its size can be a more telling metric rather than simply the number of accesses.
- *Integers and single-precision floats are preferred* – As will be discussed in greater detail in Chapter 4.1.3, values, when transferred from the LDRF to the VRF, are neither sign nor zero extended. This is accomplished by storing values within the LDRF on a four-byte boundary, which is the size of a LDRF register. Integers and single-precision floating point data types naturally comply with this boundary and therefore make best use, from a space perspective, of the LDRF. Shorts and chars,

however, must be extended when first stored into the LDRF to be the same size as a LDRF register.

2.3.2 Restrictions on Variables Placed in the LDRF

There are restrictions on the variables that may be placed within the LDRF. They are as follows:

- *Size and address of the data must be statically known* – The compiler must ensure that the total number of bytes allocated to the LDRF does not exceed its size. In addition, by statically allocating variables to the LDRF, tag storage is not required, which saves space and provides faster access. The address of the data must be fixed. Thus, heap and run-time stack data are not candidates.

- *The entire variable must be placed within the LDRF* – In particular, this is germane to structs, which contain discrete fields. An individual field of a struct may not be placed in the LDRF; the entire struct is to reside in the LDRF or not at all.

- *An individual variable must be smaller than the remaining capacity of the LDRF* – it follows that, if the entire variable must reside in the LDRF, then any variable to be considered for LDRF promotion must be smaller than the size of the LDRF.

- *A variable, to be placed in the LDRF, may not be passed to most library routines* – Although LDRF variables may be passed to functions, the function parameter must either use pass-by-value or it must use pass-by-reference and the parameter must be declared as a gpointer. The *gpointer* keyword, which is discussed in greater detail within Chapter 4.1.1, is an added, reserved keyword to indicate that a pointer points to a LDRF variable. Because many library routines use pass-by reference parameters, and because it is not advisable – nor feasible in most instances – to modify the library routines, LDRF variables may not be passed to them. If a LDRF variable must be used with a library routine, then use of a temporary variable may suffice. The temporary is assigned the value of the LDRF variable, passed to the library routine, modified by the library routine, and the LDRF variable is then assigned the new value of the temporary variable.

- *Local variables within recursive functions may not be placed in the LDRF* – although local variables may reside in the LDRF, local variables within recursive functions may not. Because the number of recursive calls is not known at compile time, the number of instances of the local variable is unknown. Therefore, the amount of

space, within the LDRF, that will be required to accommodate all of the instantiations also is unknown. This violates the necessity to know the size, at compile time, of all elements that are being added to the LDRF.

- *Initialized character strings may not be placed in the LDRF* – As will be discussed in greater detail in Chapter 4.1.3, integral variables are extended to the size of integers within the frontend of the compiler. The extension causes difficulties when initializing strings. A string is stored as a character array; when declared to reside in the LDRF, the type is extended from that of a character to an integer. However, the initializer will fail in its attempt to initialize what is now an integer array to a string.

CHAPTER 3

EXPERIMENTAL FRAMEWORK

The experimental framework consists of four major areas: (1) compiler, (2) simulator, (3) benchmarks, and (4) experimental test plan.

3.1 Compiler

Our research compiler is a retargetable compiler for standard C. It uses a LCC frontend [9] and the Very Portable Optimizer (VPO) [3] as its backend. LCC, which can be implemented as a backend as well, is a robust frontend developed not only for production of efficient code, but also for speed of compilation. LCC produces stack code as its output. The code is used, by the middleware, to prepare input for VPO. VPO was designed to be portable as well as efficient. It utilizes Register Transfer Lists (RTLs) as an intermediate representation and produces machine-targeted assembly as its output. The RTLs themselves are machine-independent representations, which means that many of the code-improvement transformations may largely be written in a machine-independent manner, with a small portion of the code dedicated to machine-dependent specifications. VPO, therefore, has the advantage that it may easily be ported to a new architecture. The PISA port of VPO was used, which is a MIPS-like instruction set [16]. For ease of familiarity, the ISA will be referred to as the MIPS, which is a 32-bit architecture and is widely used on embedded systems.

3.2 Simulator

As the LDRF is an architectural enhancement, all testing was necessarily performed on a simulator. The simulator chosen was SimpleScalar, which is widely used for computer architecture research [5]. The SimpleScalar toolset provides several simulators, each of which may be modified to suit one's needs. Primarily, three simulators were used within this research:

1. *sim-outorder* – a detailed out-of-order issue superscalar processor with a two-level memory system and speculative execution support. This simulator is a performance

simulator and provides cycle accurate measures. It was used to collect the measurements that are directly discussed within Chapter 6.

2. *sim-profile* – acts as a functional simulator with profiling support. It was used, primarily, to collect information that led to the identification of candidate LDRF variables as well as identification of benchmarks that were amenable to the LDRF.
3. *sim-wattch* – similar in design to sim-outorder, except that it also provides power measurements.

3.3 Benchmarks

Two benchmark suites were used to gauge the effectiveness of the LDRF: (1) the MiBench Embedded Applications Benchmark (MiBench) Suite [14] and (2) a DSP kernel suite. Using two benchmark suites was useful for a variety of reasons. In short, use of two benchmark suites ensured a thorough testing and fair evaluation of the LDRF. The various data structures, algorithms, and program structures employed within the benchmarks touched upon all commonly encountered situations and served to ensure that the modifications to the compiler and simulator were sufficiently complete and sufficiently robust.

3.3.1 MiBench Embedded Applications Benchmark Suite

The applications within the MiBench Suite are a set of representative embedded programs. In all, there are thirty-five applications within six categories. For the experiments performed, one representative application was chosen from each category. The applications selected are listed within Table 3.1. The MiBench Suite was most useful to verify the breadth of the compiler and simulator modifications. Here breadth is refers to inclusion of a wide-variety of data types and data structures within the LDRF. This is of particular importance, since – as will be discussed in Chapter 4 – the programmer is currently responsible for assigning variables to the LDRF and the compiler must be able to accept any constructs that are supported by the programming language. Breadth also refers to a wide-variety of programming techniques to utilize these structures. Here, again, the compiler must be able to accept any legitimate programming constructs or fail gracefully.

Table 3.1 MiBench Benchmarks Used for Experiments

Program	Category	Description
Adpcm	Security	Compresses speech samples using variant of pulse code modulation
Bitcount	Automotive	Bit manipulation tests
Blowfish	Security	Symmetric block cipher
Dijkstra	Network	Dijkstra's shortest path algorithm
Jpeg	Consumer	Creates a jpeg image from a ppm
Stringsearch	Office	String pattern matcher

Most of the MiBench applications are bundled with sample input as well as corresponding output. With respect to the input, the small sample input was used when available. The size of the small input was deemed more than sufficient to evaluate the LDRF; cycle counts for the small input ranged from a few million to a few hundred million cycles. The output was used to verify the correctness of not only the LDRF code – a term used to refer to a benchmark that includes LDRF variables – but also the applications when using the original code.

3.3.2 Digital Signal Processing Kernels

The Digital Signal Processing (DSP) kernels that were selected are representative of the key building-block operations found in most signal processing applications. In comparison to the MiBench applications, these kernels are relatively simple. To broadly generalize, the code comprising the DSP kernels amount to no more than a hundred statements; however, these statements are perform number of mathematical computations. The mathematical focus is quite useful to evaluate the depth of the implemented compiler enhancements. Here depth refers to successful compilation of lengthy, convoluted statements.

Although the DSP Kernels do provide input sets, they do not produce any output. To verify the program correctness of the kernels, output statements were added to both the original and LDRFerized code. These output statements were compiled dependent upon the definition of a preprocessor macro. Thus, the kernels were compiled and run, initially, with the output statements to verify program correctness. The kernels were then recompiled and rerun without the output statements. It is possible – naturally – that the output generated by the executables created by our research compiler, both with and without the LDRF, may match one another yet may still be incorrect (both would, therefore, be incorrect in the same way). To alleviate this concern, the original code also was compiled with GCC and the resulting executable was run on a physical machine rather than a simulator. The three sets of output were compared against one

another; if the output of all three executables agrees with one another, it seems fair to conclude program correctness in all cases.

All measurements were collected when investigating the kernels without compilation of the output statements. This was done to minimize the number of changes to the benchmarks. Without output, the kernels do not lend themselves to a definitive test for program correctness. Correct compilation, when including the output statements, does suggest that the kernels also will compile correctly without the output statements. However, the assembly code that was produced also was inspected, by hand, to identify any anomalies. This process is actually quite effective, as the LDRF assembly is easily compared against assembly that does not use the LDRF. Lastly, we were mindful of the results and investigated any which did not seem reasonable. Since many of the DSP kernels are similar in structure, there is the expectation that they will behave in a similar fashion. If a kernel were to deviate from what was reasonable, it was investigated more thoroughly to determine the cause and appropriate actions were taken as necessary.

The kernels typically include one or two loops and much of the execution time is spent within these loops. However, the iteration count for some of the loops was sufficiently low so that the loops did not reach a steady state due to instruction cache misses and the cost of startup and termination code. Therefore, the results were skewed by the warm-up period. To alleviate this situation, an outer loop was added to cause the inner loop to iterate many more times. Doing so does not change the code within the inner loop but does cause the loop to reach a steady state, which is necessarily a fairer test.

Table 3.2 DSP Kernels Used for Experiments

Program	Description
Conv45	Performs convolution algorithm
Fir	Applies finite impulse response filter
IIR1	Applies infinite impulse response filter
Jpegdct	Jpeg discrete math transform
Mac	Multiple-accumulate operation
Vec_mpy	Performs simple vector multiplication

3.4 Experimental Test Plan

A test plan was devised to determine the efficacy of the LDRF. Potential areas of interest to determine the LDRF efficacy include: static code size, dynamic instruction count, energy

consumption, and performance. In addition, the size of the LDRF was investigated and, in particular, how the areas of efficacy are affected by the size of the LDRF.

Initially, the benchmarks are run using the original code. These results provide a base case against which the LDRF results may be compared. In addition, compiler and simulator analysis tools are invoked during compilation/simulation of the original code that aid in identification of candidate LDRF variables, which are variables that warrant promotion to the LDRF. Once the candidate LDRF variables have been identified, then the high-level source code are modified – as appropriate – to include the LDRF variables. Three LDRF sizes – “small”, “medium”, and “large” – are identified based in part on the size requirements of the benchmarks and in part by the constraints imposed the desired architectural characteristics of the LDRF. Candidate variables are added to the LDRF such that those variables that may be expected to provide the greatest positive impact are added first. Subsequent variables are added until either no further LDRF appropriate variables exist or the LDRF has reached capacity. The benchmarks, coded for the LDRF, are then be compiled and run within the simulator framework. As applicable, LDRFerized benchmark output is compared against the original code output to verify program correctness.

CHAPTER 4

COMPILER ENHANCEMENTS

In order to support the LDRF, modification to many of the stages of the compiler toolchain – the frontend, middleware, backend, and assembler – were required. Also, two stand-alone compilation tools were developed to ease compilation of program containing LDRF variables. The compiler environment, inclusive of the tools, is depicted in Figure 4.1.

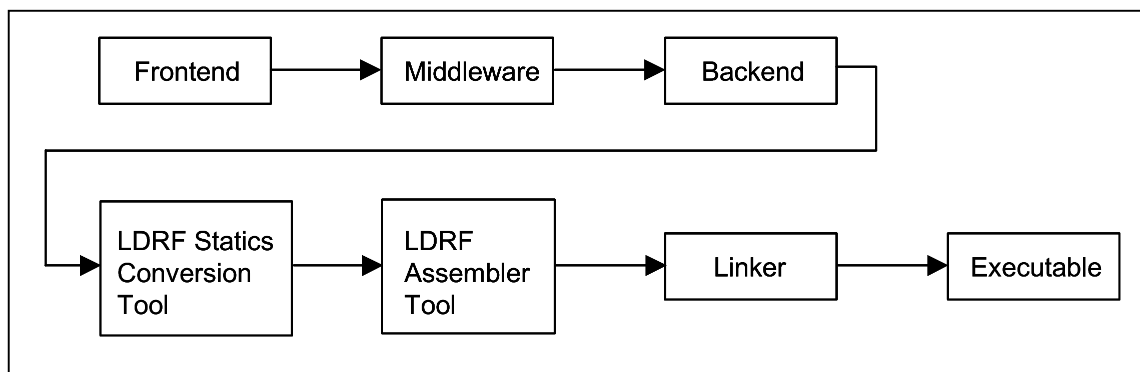

Figure 4.1 LDRF Compilation Pathway

4.1 Frontend Modifications

Currently, it is the programmer's responsibility to allocate data to reside in the LDRF by use of directives within the high-level source code. This provides the programmer, who ostensibly has the greatest understanding of their application, the opportunity to use the LDRF to its best advantage. The frontend, therefore, is charged with not only generating correct code to be passed to the middleware, but – more importantly – is charged with performing the myriad of syntax, semantic, and type checks necessary to ensure that the programmer has not erred in its use of LDRF variables at the source code level.

4.1.1 Gregister and Gpointer Keywords

To allow the programmer to assign variables to the LDRF, the frontend of the compiler was enhanced to allow two new keywords to be specified in the source code: (1) *gregister* and (2) *gpointer*.

The *gregister* keyword is a storage class specifier and may be used in conjunction – as appropriate – with most other storage class specifiers. Figure 4.2 provides a declaration example using the *gregister* keyword. For example, a variable that has been declared to be static as well as a gregister maintains the characteristics of a static variable, one of file scope, with the additional qualification that the variable will reside in the LDRF. Although the *gregister* keyword works in concert with some storage class specifiers – i.e., *static or extern* – it is inappropriate to use it with others, such as *register* or *auto*. The *register* keyword indicates that the given variable should reside in a register, which is a sensible specification for oft-used, nonstatic, local variables. However, it is not permissible to use the *register* keyword in conjunction with the *gregister* keyword. To do so, in effect, instructs the compiler to allocate the given variable to both a register and the LDRF, which is not possible. The *auto* keyword, which is the default storage class specifier, allocates memory for variables from the run-time stack. The combination of the *gregister* keyword and the *auto* keyword poses a similar problem as when combining the *gregister* and *register* keywords, it instructs the compiler to allocate the given variable to two locations – the run-time stack and the LDRF.

```
gregister int a[1000];
```

Figure 4.2 Variable Declaration with LDRF Storage Specifier

The second added keyword – the *gpointer* keyword – is used to declare a pointer to a LDRF variable. A LDRF pointer must point to a LDRF variable and a LDRF variable – in turn – may only be pointed to by a LDRF pointer. The LDRF pointer itself does not reside in the LDRF; it will reside in a traditional register, which is the common allocation practice for pointers. Figure 4.3 provides a declaration example using the *gpointer* keyword. Use of pointers is a common programming practice and the utility of the LDRF is greatly enhanced by inclusion of a LDRF pointer. Many programmers are accustomed to pointer use and, in addition, many good programming practices dictate their use. In particular, use of pointers is of value when passing arguments to functions. It is a common programming practice to pass a pointer as an argument. This is particularly true for composite data structures, which are well suited for inclusion in the LDRF. Note that the existing pointer syntax was insufficient to ensure program correctness – it is necessary to qualify the pointer as one that points to a location within the LDRF. Without such a qualifier, the compiler is unable to distinguish between a traditional pointer and one which points to an object in the LDRF. This is of vital importance when performing type checking, which is discussed in greater detail within this chapter.

```
gpointer int *ptr;
```

Figure 4.3 Variable Declaration with LDRF Pointer

An undue burden would be placed on the programmer if the *gpointer* keyword were not available. This is especially so when modifying existing source code to use the LDRF. With the inclusion of the *gpointer* keyword, as well as the *gregister* keyword, the programmer can update existing source code to use the LDRF simply by including these keywords, as necessary, within declarations. No additional modifications are necessary.

4.1.2 Syntax, Semantic, and Type Checking

The additional syntax checks, as required by the addition of the *gregister* and *gpointer* keywords, are straightforward: the character strings “gregister” and “gpointer” must be recognized as reserved keywords. Verification of correct usage of these keywords is reserved for semantic and type checking.

The primary role of the semantic checks, with respect to the LDRF, when evaluating a declaration specifier is to ensure that the *gregister* and *gpointer* keywords are used solely – and correctly – within declarations. More specifically, the *gregister* and *gpointer* keywords are used within a declaration specifier, which is comprised of a storage class specifier, type specifier, and type qualifier. Unfortunately for the compiler, these keywords may occur in any order within a declaration. In addition, because of the permissible combinations of the *gregister* keyword with other storage class specifiers, the declaration specifier may contain two storage class specifiers if one of which is the *gregister* keyword.

The afore mentioned considerations are sufficient to semantically check scalar variables; however, composite data structures, notably C-style structs, require additional handling. Structs may be declared to reside within the LDRF, but the entire struct, including each of its fields, must necessarily reside in the LDRF. Therefore, when a struct is declared to reside in the LDRF, the compiler must ensure that it is permissible for each of its fields to reside in the LDRF and must extend the LDRF storage specification to each of the fields. Furthermore, if the compiler encounters a struct field that has been denoted as residing in the LDRF, it must confirm that the struct itself has been declared to reside within the LDRF.

The *gpointer* keyword is most accurately described as a pointer qualifier, as it qualifies a pointer to point to a variable that resides in the LDRF. From a semantic perspective, the

compiler need only verify that the *gpointer* keyword has been used in conjunction with a pointer, an example of which was provided in Figure 4.3. The *gpointer* keyword is commonly used to qualify pointer function parameters. This is obvious utility – it allows a pointer, to a LDRF variable, to be passed to a function. This aligns well with the tendency for arrays to be passed to a function via a pointer.

Type checking ensures that declarations and expressions adhere to the typing conventions that are dictated by the programming language. Many of the type checking responsibilities, of the compiler, are associated with the use of the gpointer keyword, and have been touched upon. The compiler, for example, must verify that a gpointer is assigned an LDRF address. This is accomplished by (1) appending a bit to the type of the pointer – its lvalue – to indicate that it is a gpointer and (2) by appending a bit to the underlying type of the pointer – its rvalue – to indicate that the value of the variable resides in the LDRF. The *gregister* keyword, in addition to its role as a storage class specifier, imparts type information as well. The type of a LDRF variable must be appended as a gregister type, in addition to the typical type information that maintained. Appending the gregister type information allows the compiler to verify that, if the address of a LDRF variable is assigned to a pointer, it will be a gpointer. For example, consider "x = &n;"; if x is declared as a gpointer, then n must be declared as a gregister (or vice versa). This is in addition to the standard type checking evaluations that are made during compilation.

4.1.3 Type Extension of LDRF Variables

The size of a traditional register is typically that of an integer; commonly 4 bytes or 32 bits. Chars and shorts often use a smaller space; chars commonly occupy 1 byte in memory and shorts commonly occupy 2 bytes. A sign – or unsign – extension occurs immediately prior to loading the value into a register. The LDRF supports variables of different types and sizes; however, it eliminates the need for a sign/unsign extension by storing each integral value as an integer. In addition, it simplifies alignment constraints as each LDRF integral value is aligned to the width of an integer, which also is the width of a register. The values, within the LDRF, are therefore "preloaded" into registers and are available for immediate use. To accomplish the extension, the compiler changes the type of all gregister integrals, i.e., chars and shorts, to that of an integer when examining the declaration. This extension is not necessary for floating point types – neither single nor double precision – as they occupy either 4 or 8 bytes. Double-precision floats occupy two LDRF registers in order to accommodate its space requirements.

Although type extension simplifies data movement from the LDRF into a traditional register, it does have some drawbacks. Notably, this approach is not as space efficient as memory,

which allocates the minimum space required to accommodate a data element. The LDRF, on the other hand, will allocate one LDRF register – or four bytes – to accommodate a 1-byte data element, such as a char. In addition, the extension causes difficulties when initializing strings. A string is stored as a character array; when declared to reside in the LDRF, the type is extended from that of a character to an integer. However, the initializer will fail in its attempt to initialize what is now an integer array to a string. This failure represents an acknowledged limitation of the current LDRF implementation; however, it is one that could be overcome, in the future, if warranted.

4.2 Middleware Modifications

The main role of the middleware is to generate naïve RTLs based on the stack code provided by the frontend. The middleware, therefore, has been modified to generate RTLs that store to/load from the LDRF; the structure of these RTLs is very similar to that of RTLs that store to/load from memory. To generate the RTLs that access the LDRF, three new memory characters were created, as depicted in Table 4.1.

Table 4.1 Naive LDRF RTL Forms

Memory Character	Description	Usage within a Store	Usage within a Load
G	Integer LDRF character	`G[ r[2] ]=r[3];`	`r[3]=G[ r[2] ];`
J	Single-point precision LDRF character	`J[ r[2] ]=f[3];`	`r[3]=J[ r2] ];`
N	Double-point precision LDRF character	`N[ r[2] ]=f[3];`	`f[3]=N[ r[2] ];`

The middleware also is charged with creating assembler directives such as variable declarations. The middleware was modified such that the LDRF variable declarations – inclusive of alignment, initialization, or space directives – are generated as commented assembler directives with a distinguishing initial character sequence so that: in terms of the assembler, these directives are ignored; however, in terms of the LDRF Assembler Directive Collection Tool, they are easily recognized.

4.3 Backend Modifications

The VPO backend is responsible for producing assembly for use by the assembler. In addition, it performs the classical backend optimizations, such as instruction selection, common subexpression elimination, strength reduction, and constant folding, to name but a few. In order to accommodate the LDRF, VPO was first modified to generate naïve code accessing the LDRF. Once naïve code was correctly generated, then optimizations to produce more efficient code were implemented. The focus of the optimizations is to create LDRF instructions that are likely to coalesce with one another, thereby forming a block access that moves multiple values to/from the LDRF in a single instruction.

4.3.1 Modifications to Generate Naïve LDRF Code

Relatively few modifications are necessary in order for the backend to generate correct assembly code that accesses the LDRF. The most significant change was modification of the machine description, which was modified to include representations for the LDRF instructions. Semantic checks also were put in-place to ensure that the LDRF instructions generated by the backend are valid.

An important distinction to the assembly instructions representing LDRF accesses are that they do not support offsets, as will be discussed in detail in Chapter 4.5. Within the RTL representation, offsets are supported. Doing so permits a variety of code modifications to be considered. Once the code modifications have completed, and the assembly is being generated, the compiler identifies any LDRF instructions that use an offset and issues it as two assembly instructions: (1) an addition instruction that adds the base to the offset and (2) a LDRF instruction that uses the base plus offset value, provided by the addition instruction, as the address to be accessed within the LDRF.

4.3.2 Application-Wide Call Graph

VPO comes equipped with the capability to generate a call graph, which is often used within interprocedural optimizations, for a given source file. This capability was extended to create an application-wide call graph, which is used within the coalescing optimization (see *Chapter 4.3.3 Coalescing Optimization*). It should be noted that the generation of the application-wide call graph necessarily mandates a two-pass compilation. In the first pass, the call graphs for each source file within an application are created; in the second pass, the application-wide call graph is cobbled together from the individual call graphs and code is then generated.

4.3.3 Coalescing Optimization

The objective of the coalescing optimization is to combine – or coalesce – two or LDRF instructions, e.g., two loads, that use sequential LDRF addresses into a single LDRF instruction. The single instruction acts as a block LDRF access, as it accesses more than one LDRF register at a time. This has the potential to increase performance as well as to counteract the code bloat endemic to loop unrolling, a necessary step within coalescing optimization.

The coalescing optimization is a multi-step optimization and depends on many stages to have the greatest likelihood of success. Although many of the stages are implemented to support the coalescing optimization, they are complete optimizations unto themselves. However, because of their support role, their discussion is tailored to highlight their functionality within the overarching coalescing optimization. The major stages of the optimization are listed below to provide context and – stages that result in code modification – also are illustrated within Figure 4.4; Figure 4.4(d) illustrates the desired end. Note that, within the figure, bold RTLs have been modified from their previous state. The bold is only for the ease of the reader. This style is used throughout the thesis' figures.

- *Loop unrolling* – Figure 4.4(a); a classic optimization, used to increase the likelihood of sequential LDRF accesses within a single basic block
- *Sink increments* – Figure 4.4(b); increases the likelihood that sequential LDRF accesses will be of a form that is amenable to coalescing
- *Identify sequential LDRF accesses* – Find the LDRF references, within a basic block, that are sequential and are therefore candidates for coalescing.
- *Rename registers* – Figure 4.4(c); renames registers so that the traditional registers, present within the LDRF instructions, are sequential
- *Coalesce accesses* – Figure 4.4(d); coalesce two or more properly formed LDRF instructions so that a block access from/to the LDRF is performed.

```
gregister e[100];

 for(i=0;i<100;i++)
    e[i] = i;

(a) unrolled 2x              (b) sink increments
      r[5]=0;                      r[5]=0;
   L2                           L2
      G[r[6]]=r[5];                G[r[6]]=r[5];
      r[6]=r[6]+4;                 r[5]=r[5]+1;
      r[5]=r[5]+1;                 G[r[6]+4]=r[5];
      G[r[6]]=r[5];                r[6]=r[6]+8;
      r[6]=r[6]+4;                 r[5]=r[5]+1;
      r[5]=r[5]+1;                 PC=r[5]<r[2],L2;
      PC=r[5]<r[2],L2;

(c) rename registers         (d) coalesce accesses
      r[4]=0;                      r[4]=0;
   L2                           L2
      G[r[6]]=r[4];                r[5]=r[4]+1;
      r[5]=r[4]+1;                 G[r[6]..r[6]+4]=r[4..5];
      G[(r[6]+4)]=r[5];            r[6]=r[6]+8;
      r[6]=r[6]+8;                 r[4]=r[5]+1;
      r[4]=r[5]+1;                 PC=r[4]<r[2],L2;
      PC=r[4]<r[2],L2;
```

Figure 4.4 Significant Optimization Stages

4.3.3.1 Loop Unrolling Optimization. Loop unrolling, a common loop transformation optimization, combines two or more iterations of an innermost loop into a single iteration. Doing so increases the static code size as two or more copies of the loop body are now contained within a single iteration, but it does reduce the number of overhead instructions, which serves to reduce the number of branch instructions that are executed. These effects provide a performance boon at the expense of increased static code size, but more importantly make the loop more amenable to coalescing LDRF references. The coalescing optimization, as will be discussed in greater detail in Chapter 4.3.3, coalesces two or more LDRF instructions into a single LDRF instruction. In order to do so, the addresses within the coalesced instructions must be sequential. Loop unrolling lends itself to creating such instructions. Consider a simplistic loop such as one that iterates through each element of a LDRF array; unrolling the loop four times will create a single loop body that now has four sequential LDRF accesses.

VPO is equipped with a loop unrolling optimization. However, portions of the optimization are machine-dependant. These portions were originally ported for the ARM, but not for MIPS. Therefore, the optimization was necessarily ported to the MIPS so that it may be used in support of the LDRF and any other research effort, targeted for the MIPS, that benefits from loop unrolling.

4.3.3.2 Sink Increments Optimization. Although the loop unrolling optimization is useful to create sequential LDRF accesses, those sequential accesses will not necessarily be in the form to be exploited by the coalescing optimization. To increase the likelihood that the coalescing optimization will be successful, the LDRF instructions – within the RTL representation – should use a base plus offset notation to reference the LDRF address as this greatly simplifies the process of identifying sequential LDRF addresses. The Sink Increments Optimization endeavors to put LDRF instructions into this format. To illustrate consider Figure 4.4; the RTLs within (a) represent the loop body when unrolled two times. The RTLs within (b) represent the RTLs after the Sink Increments Optimization has been performed. Note that the second LDRF location is now of the form base plus offset and also note the ease to which the two LDRF accesses might be identified as sequential. Although the primary objective of the optimization is to sink the increments to create LDRF accesses of a specific form, it has the potential for a secondary side-benefit: two instructions (the LDRF access and the increment) are combined into a single LDRF access, thereby saving an instruction. Although it is always true that the increment and the LDRF instruction are merged together, the increment is not necessarily eliminated. It may be necessary to move it after the LDRF instruction to retain program correctness. This distinction is clarified within the discussion of the optimization's algorithm. This optimization also may be used to sink increments into traditional memory locations. Although creation of sequential memory locations may not be beneficial in and unto itself, the reduction of instructions is directly beneficial.

Figure 4.5 provides the pseudo-code necessary to implement the Sink Increments Optimization, which is performed on each single basic block within a function. First, the RTLs within the block are scanned to find a RTL that is an increment or a decrement (Line 3-4), which is of the form r[x] = r[x] $\pm$ n where *n* is a constant. Next, the remaining RTLs are scanned until another occurrence – either a set or a use – of the set register, within the RTL being sunk, is found. Once found, it is verified that an intervening function call has not occurred. If it has and the set register is not preserved across the call (Line 6), then the sink of the RTL must be aborted. The next occurrence of the set register is then examined (Lines 8-15). If it occurs as both a set and a use, then the right-hand side of the sink RTL is simply substituted into the use and the sink RTL may be removed (Lines 8-10). If it occurs solely as a use, then the right-hand side of the sink RTL is simply substituted into the use and the sink RTL must be moved immediately after the next occurrence. This is often the case with a memory or LDRF reference, where the use represents the memory or LDRF location within the reference. Lastly, if the next occurrence occurs solely as a set, then the sink must be abandoned.

```
1    foreach blk in function do
2       foreach rtl in blk do
3          if (rtl is not an increment or decrement) then
4             continue;
5          find the next occurrence of rtl->set
6          if (intervening function call and rtl->set is scratch) then
7             continue;
8          if (next occurrence sets and uses rtl->set) then
9             substitute right-hand side of rtl into next occurrence
10            remove rtl
11         elseif (next occurrence uses rtl->set) then
12            substitute right-hand side of rtl into next occurrence
13            move rtl immediately after next occurrence
14         elseif (next occurrence sets rtl->set) then
15           continue;
```

Figure 4.5 Sink Increments Algorithm

4.3.3.3 Coalescing Algorithm. The algorithm for the coalescing optimization is presented within Figure 4.6. The algorithm is complete as presented, but has a greater likelihood of success if loop unrolling and sink increments already have been performed. The first step is to perform a quick scan of the RTLs, within a block, to determine if it has LDRF references (Line 2-3). If it does not, then coalescing is clearly not possible. Once a LDRF reference has been found, then the remaining RTLs are scanned to identify as many sequential LDRF accesses as possible (Line 7-8). This step identifies a series of LDRF references, each of which is sequential to the previous access. The process by which sequential references are identified is discussed, in detail, in Chapter 4.3.3.4.

Once the series of RTLs with sequential LDRF references are identified, a check is performed to determine if two or more RTLs are contained within the series (lines 9-10; Figure 4.6). It quite obviously would be unnecessary to coalesce a single RTL. At this point, sequential registers, to be used in register renaming, must be identified (line 11; Figure 4.6). Ideally, *n* sequential registers will be available to coalesce the *n* RTLs. However, so long as there are two or more available sequential registers, then coalescing may proceed. If there are two or more, but less than *n* available sequential registers, then the first *n'* RTLs will be coalesced with one another, where *n'* represents the number of available sequential registers that were found. This situation is more likely to occur with longer series of RTLs, where it is more difficult to find a sufficiently long series of available sequential registers. In these cases, it is not uncommon for a series of, say, six RTLs to be coalesced as two blocks of three accesses rather than one block of six. Next, register renaming is performed (line 13; Figure 4.6) in accordance with the algorithm presented in Figure 4.7 and as discussed in Chapter 4.3.3.5. Once register renaming

is completed, then the LDRF accesses are of the correct form to be coalesced (lines 14-18; Figure 4.6); coalescing is discussed within Chapter 4.3.3.6.

```
1  foreach blk in function do
2     if (blk does not have LDRF references)
3           continue;
4     foreach inst in blk do
5        if (inst does not have LDRF references) then
6           continue;
7        foreach remaining inst in blk do
8           determine if inst is contiguous with last LDRF reference
9        if (numContiguous references < 2)
10          continue;
11       identify sequential, available registers for register renaming
12       if (numSequential renaming registers > 1)
13          rename registers so that loaded (or stored) registers are
            sequential
14       if (inst is a store)
15          coalesce numSequential contiguous references into last
            reference to form a single instruction
17       else
18          coalesce numSequential contiguous references into first
            reference to form a single instruction
```

Figure 4.6 Coalescing Algorithm

4.3.3.4 Identify sequential LDRF accesses. Identification of sequential LDRF accesses must be done carefully as there are many considerations that must be evaluated. The process is an exercise in memory location analysis, requires an awareness of function calls, sets, uses, and memory aliasing concerns. The optimization is structured conservatively, as dictated by good programming practices, to ensure program correctness is retained.

There are several circumstances that prevent a sequential RTL to be coalesced. Each of the scenarios, below, discusses a situation that would prevent coalescing; the scenarios are depicted within Figure 4.7. The terms *source RTL* and *sink RTL* are used within the discussion. The source RTL is the RTL for which a sequential RTL is sought; the sink RTL is a RTL whose location is sequential to the source RTL.

a. If there is an intervening, opposite LDRF access whose location is equal to the sink RTL, then no further coalescing with the source RTL may occur. Figure 4.7(a) represents this situation. Although the sink RTL is sequential with the source RTL, there is an intervening RTL that performs the opposite action and is sequential to the sink RTL. If the sink RTL where to be coalesced with the source RTL, the sink RTL would not receive the correct value from the LDRF. Note that the blocking RTL only prevents the

first sink RTL from being coalesced; it does not prevent coalescing of the second sink RTL.

b. If there is an intervening opposite LDRF access whose location is unknown, then no further coalescing may occur. The RTL representation of a LDRF reference, like a memory reference, is composed of two parts: base and offset. The offset is nothing more than a constant and is easily recognized. The base is represented by a value contained within a register. Without specific contrary knowledge, no assumptions may be made regarding the value within this register. Figure 4.7(b) represents this situation, the base address of the blocking RTL is not known and, therefore, no sink RTL may be coalesced with the source RTL. To establish the variable name that is associated with a LDRF reference, an analysis routine is performed prior to the coalescing optimization that incorporates the variable name into a RTL. Comparison of the variable names may be used to establish that the bases are different. Although the analysis routine robustly identifies local and global variable names, it is handcuffed when identifying variable names associated with function parameters. As the variable names that are passed to the function are not available until run-time, these names are not available to the compiler. Therefore, coalescing within a function may have a lower likelihood of success. This, to be sure, is dependent upon the function and the RTLs therein.

c. If the base location of the source RTL is set after its use, within the source RTL, then it is likely that no further coalescing may occur. Figure 4.7(c) represents this situation, where the base address (r[6]) of the source RTL is set after its use. Because it also is the base address of the source RTL, the intervening set must be considered when attempting to establish the two RTLs as sequential. In the example, the base address is set to a different variable and clearly coalescing with the source RTL is not permissible. Sets of the register containing the base address most often occur when a new memory location is needed and an offset is not used. In general, the Sink Increments Optimization eliminates these RTLs.

d. If there is an intervening function call that contains LDRF references or that calls a function that contains LDRF references, then no further coalescing the source RTL may occur. Figure 4.7(d) represents this situation. The application-wide call graph is used for to determine if a given called function, or one of its children, contain any LDRF references. The intervening function call has the potential to modify the LDRF location that we are attempting to coalesce.

```
(a)
   r[6]=x;
   r[5]=G[r[6]];    # source
         ...
   G[r[6]+4]=r[5];  # blocks sink 1
         ...
   r[5]=G[r[6]+4];  # sink 1
         ...
   r[8]=G[r[6]-4];  # sink 2
```

```
(b)
   r[6]=x;
   r[5]=G[r[6]];    # source
       ...
   G[r[7]]=r[5];    # blocks
                    # all sinks
```

```
(c)
   r[6]=x;
   r[5]=G[r[6]];    # source
         ...
   r[6]=z;          # blocks all sinks
         ...
   r[5]=G[r[6]+4];  # sink
```

```
(d)
   r[6]=x;
   r[5]=G[r[6]];    # source
       ...
   r[25]=ST;        # blocks all sinks
                    # if it, or children,
                    # contain LDRF
                    # references
```

Figure 4.7 Sequential LDRF References

4.3.3.5 Rename Registers Methodology. The purpose of the rename registers methodology is to rename the registers within a series of sequential LDRF accesses so that the registers are sequential. Renaming of registers is the final step prior to coalescing several RTLs with sequential LDRF accesses into a single RTL. It serves to create a block of sequential registers, within the traditional register file, which will align with a like-sized block within the LDRF. The register rename methodology may best be thought of as a step within the Coalescing Optimization – the renaming is done for the explicit purpose of coalescing and has limited use outside of that context. To be clear, this will necessarily rename the registers within their live range and not solely within the LDRF accesses.

Because the register rename methodology seeks to rename registers such that they are sequential, register pressure is of great concern. Not only must registers be available for renaming, but they must be sequential as well. Table 5.2 [20] provides an overview of the registers on the MIPS architecture and their use. Although the MIPS has thirty-two registers, not all registers are created equal in the eyes of the renaming methodology. Ideally, temporary registers will be used as they may be referenced without first saving their value and do not have an otherwise prescribed function. Those registers that do have a prescribed function, or those that must be saved and restored on subroutine exit, must be used with greater care. Because of these concerns, register renaming was restricted to registers within the inclusive range 4-25, yielding twenty-two possible rename registers.

Table 4.2 MIPS Register Names and Uses

Register number	Used for
0	Always returns 0
1	Reserved for use by assembler
2-3	Value returned by subroutine
4-7	First four subroutine parameters
8-15,24-25	Temporaries; may be used without saving
16-23	Subroutine register variables; must be saved and restored upon subroutine exit
26,27	Reserved for use by interrupt/trap handler
28	Global pointer
29	Stack pointer
30	Frame pointer
31	Return address for subroutine

The algorithm used to rename registers is presented in Figure 4.8. An example of the algorithm, in action, is provided within Figure 4.9. To put the algorithm in context of the overall coalescing optimization, the renaming occurs once the *n* sequential LDRF accesses, to be coalesced, have been identified. The first task is to identify *n* sequential, available registers where *n* is the number of sequential LDRF accesses that will be coalesced (Line 1). When seeking the sequential, available registers, the registers currently being used within the *n* LDRF accesses are evaluated to determine if they naturally happen to be sequential. If not, then it is attempted to identify *n* sequential, available registers. An available register is defined as either a register that occurs in one of the *n* LDRF accesses or one that is neither set nor used within the union of the live ranges of the LDRF accesses. The registers that occur within one of the *n* LDRF accesses may be considered available because they will be renamed. The union of the live ranges of the LDRF accesses is used as, once coalesced, each the LDRF accesses will have a live range equivalent to the union of each of the LDRF accesses.

```
1  identify n sequential, available registers
2  if (numSequential registers found < 2) then
3     break;
4  foreach register to be replaced do
5     determine if register is used before it is set
6  foreach register used before set do
7     insert predecessor block
8     add rtl to predecessor block to set new register to old register
9  if (replacing into stores) then
10    foreach register to be replaced do
11       if (no intervening set of register between successive stores) then
12          add rtl of the form r[x] = r[x]; immediately prior to the store
13 foreach rtl in live range of old register do
14    if (register in rtl->sets or register in rtl->uses) then
15       replace old register with new register
16 if (blk in loop) then
17    foreach rtl in blk do
18       if (rtl is loop invariant) then
19          move rtl to preheader
```

Figure 4.8 Register Renaming Algorithm

A check (Figure 4.8, lines 2-3; depicted in Figure 4.9(a)) is performed to verify that at least two sequential registers have been found. If not, then we break from the routine. Next, it is determined if any of the registers, to be replaced, have *not* been set – within this block – prior to their first use (lines 4-5). Each register that is used, before being set within the block, must necessarily have had its value set in a previous block. Rather than attempt to identify these sets, a predecessor block is inserted into the control flow. RTLs are added to this block that set the new register value to that of the old (Figure 4.8, lines 6-8; depicted in Figure 4.9(b)). At this point, it is determined if the LDRF accesses are a series of stores, which require special handling. As described in Figure 4.8 lines 9-12; depicted in Figure 4.9(c), it must be determined if each register to be replaced, within a store, has an intervening set between successive stores. If storing a constant value, there is unlikely to be an intervening set. If not, then a RTL must be added – immediately prior to the store – that is of the form r[x] = r[x]; where r[x] is the register to be replaced; during the renaming, the RTL will be modified to the form r[y] = r[x];. Once this has been accomplished, then renaming may be performed. For each register to be renamed, each RTL, within the live range of the old register(s), is examined and any occurrences of the old register(s) is replaced with the new register(s) (Figure 4.8, lines 13-15; depicted in Figure 4.9(d)). The last task (Figure 4.8, lines 16-19; depicted in Figure 4.9(e)) is to check if the register renaming is being performed within a loop and, if so, any loop invariant RTLs that may have been created by the register rename routine are moved to a preheader block.

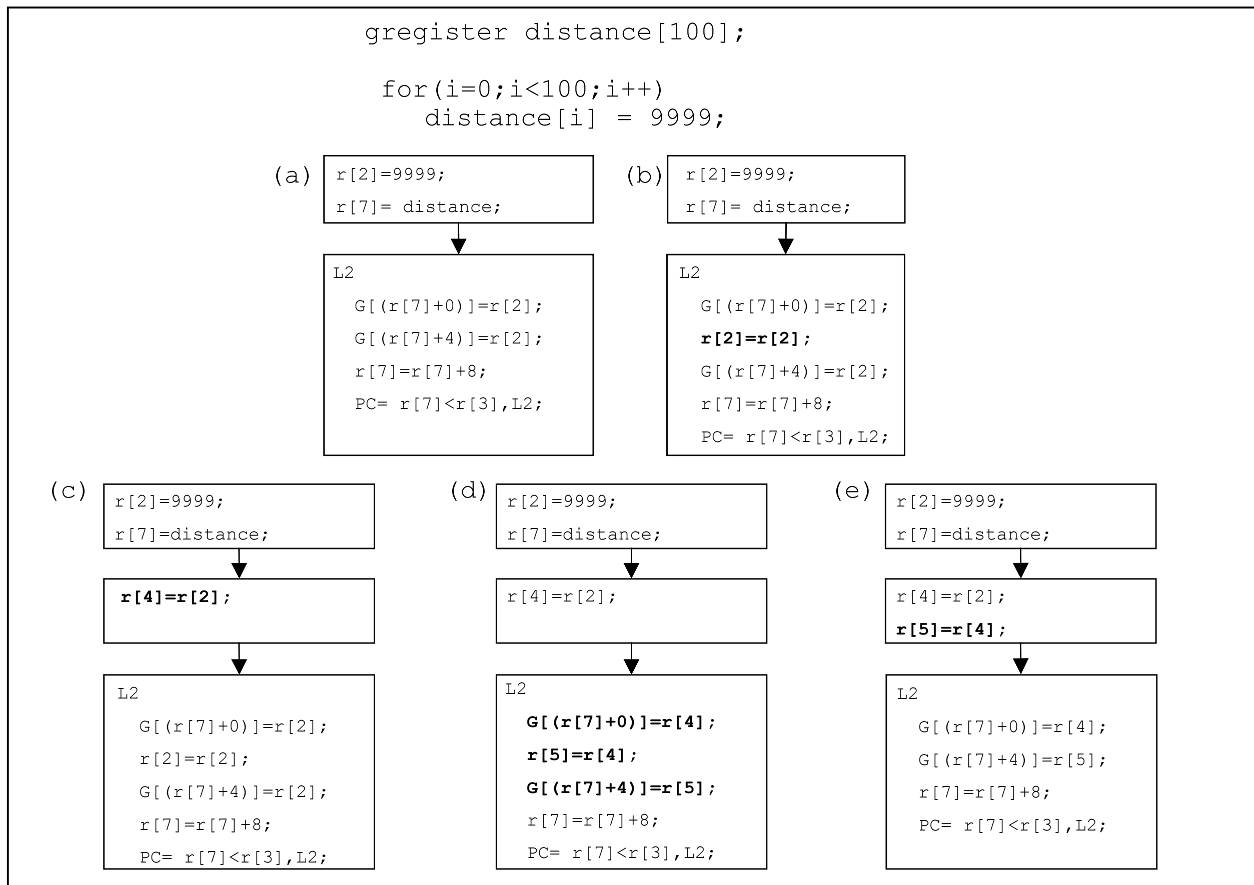

Figure 4.9 Register Rename Example

4.3.3.6 Coalescing of Sequential LDRF References. The actual coalescing of accesses is the most straightforward piece of the optimization. At this point, it is simply a matter of coalescing the multiple RTLs into a single RTL with multiple effects. The coalescing piece also serves to eliminate multiple RTLs that are now represented by the single coalesced RTL, thereby reducing the number of instructions.

As depicted within Figure 4.10, if coalescing stores, then the RTLs are coalesced into the last reference to form a single RTL with multiple effects. If coalescing loads, then the RTLs are coalesced into the first reference to form a single RTL with multiple effects. The number of RTLs that may be coalesced is dependent upon the hardware configuration of the LDRF; as the RTLs are coalesced, a check is performed to ensure that this limit is not exceeded.

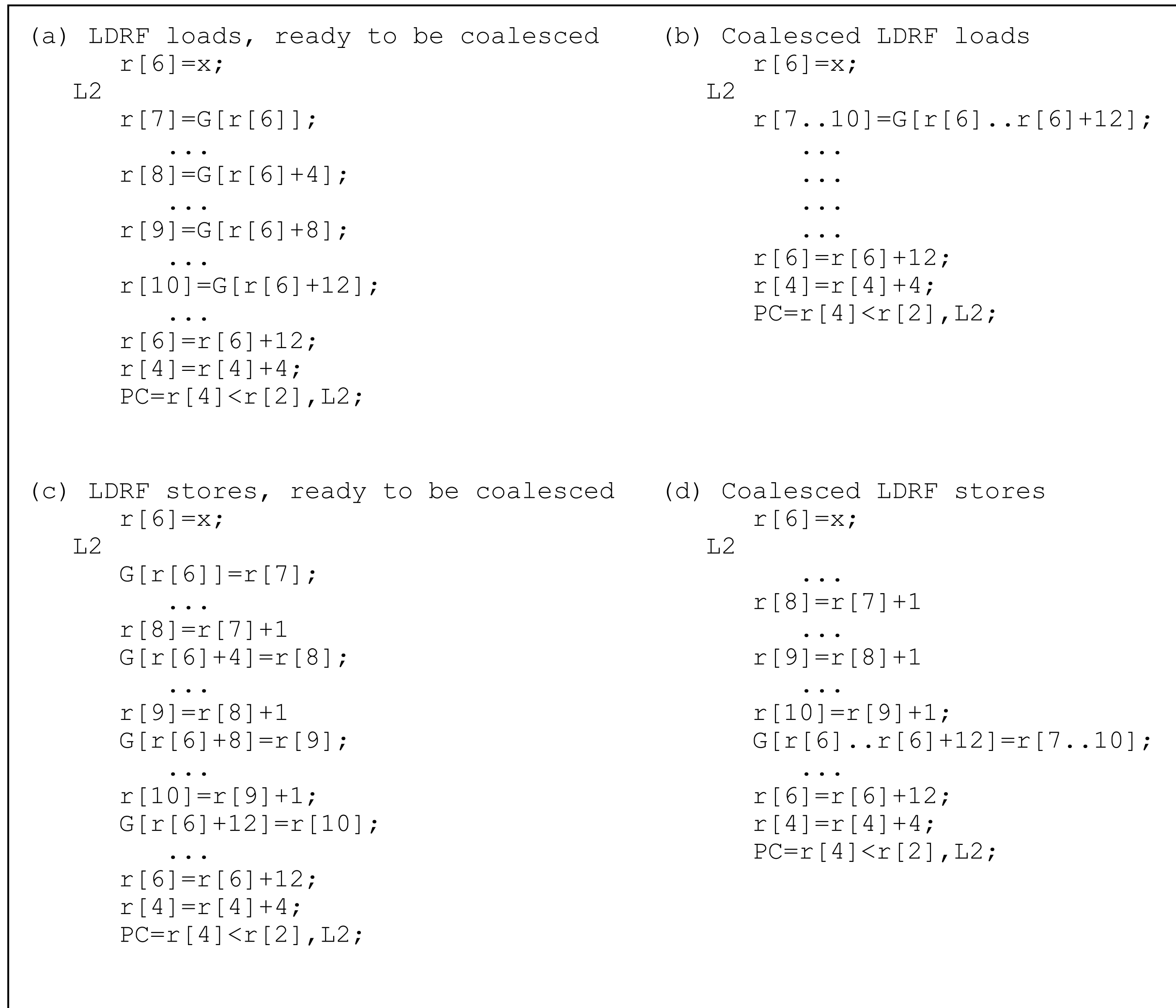

Figure 4.10 Coalescing Example

4.4 Compilation Tools

To simplify the compilation process, two compilation helper tools have been created. The first tool detects LDRF variables that have been declared as static and the second collects the LDRF variable declarations into a single assembly file. The overarching goal of these tools is to simplify the job of the linker. Conceptually, the LDRF is an architected large register file. In order to most accurately implement this representation within the compiler, two new assembler data sections would necessarily be created – analogous to the bss and data sections where global data (uninitialized and initialized, respectively) – to contain the initialized and uninitialized LDRF data. The linker would, therefore, have the added responsibility to manage the new data sections. The added complexity of not only adding two new data sections but also modifying the linker to manage them was deemed unnecessary. Rather, the approach taken was to move all LDRF assembler declarations – for both uninitialized and initialized LDRF declarations – into a

single assembly file to create a single, continuous block of LDRF declarations. In addition, the newly created assembly file would be the first assembly file to be linked, thereby causing the block of LDRF block of declarations to begin at a known location – the start of the data section. Lastly, a "filler" declaration is added to the assembly file, whose size is such that the cumulative space of LDRF declarations is equivalent to the overall size of the LDRF. Doing so creates an allocated block, by the linker, whose size is equal to the size of the LDRF and whose location, within the allocated memory space, is known. This block is interpreted, by the simulator, as the data that populates the LDRF during program load time.

4.4.1 LDRF Static Compilation Tool

Static variables, which have file scope rather than program scope, may be declared to reside within the LDRF. However, in order for the LDRF Assembly Directive Collection Tool (see Chapter 4.4.2) to be successful, all LDRF variables must have program scope. The LDRF Static Compilation Tool depicted in the compilation pathway of Figure 4.1, works by scanning a given assembly file, identifying any statically declared LDRF variables, and rewriting the declaration as a global declaration. To avoid collision with a global LDRF variable within any source file in the application, the name of the variable being declared is modified to include the base of the source filename. As a result, the assembly file must be scanned and any references to the static variable must be altered to use the new name.

4.4.2 LDRF Assembler Directive Collection Tool

The LDRF Assembler Directive Collection Tool (LADCT) is a compilation aid and, as depicted in Figure 4.1, is invoked immediately prior to the linker stage. The LADCT is responsible for:

- *Scanning each assembly file within a compilation* – a list of assembly files is maintained during compilation and it is these files that are scanned by the LADCT.
- *Recognizing LDRF variable directives* – these directives are distinguished by a signature character sequence that both identifies them as LDRF directives, but also signifies – to the assembler – that they are comments and should be ignored by the assembler.
- *Appending them, as global, initialized declarations, to a separate assembly file* – as a result of the LDRF Static Compilation Tool, all LDRF declaration directives are global. However, they will not necessarily be initialized. This is necessary so that

they naturally occupy a contiguous block. The LDACT will rewrite, as necessary, directives so that all variables are initialized.

- *Calculating the number of bytes allocated to the LDRF* – by calculating the number of bytes allocated to the LDRF, the LDACT provides a compile-time check to verify that the number of bytes allocated to the LDRF does not exceed the size of the LDRF.
- *Allocating appropriate space to a fill variable* – A fill variable is used to ensure that that the overall number of allocated bytes is equal to that of the LDRF.

The single assembly file, created by the LADCT, will contain all LDRF variable declarations. Figure 4.11 shows a sample LDRF references file. Note that there are four global declarations within this file: *LDRFADDR*, *ldrf_fill*, *dijkstra_L2*, and *c*. The first two are generated by the LADCT and serve specific purposes. *LDRFADDR* is the first declaration within every LDRF references file and acts as a label to identify the beginning of the LDRF region within the linked executable. *LDRFADDR* will not include any data definition directives; it will solely be used in its label capacity. The second, *ldrf_fill*, will have its data definition directives as the last set of such directives within the LDRF references file. It will contain two data definition directives – ".word 0" and ".space x", where *x* is the number of bytes that should be skipped in order for the overall number of bytes allocated to the LDRF to be equal to the size of the LDRF. The number of bytes, which should be skipped, is determined by subtracting the sum of the space occupied by the variables that have been allocated to the LDRF from the overall size of the LDRF. The third directive, ".globl dijkstra_L2", illustrates a static LDRF directive that has been converted, by the

```
.globl LDRFADDR
.globl ldrf_fill
.globl dijkstra_L2
.globl c
.data
.align 4
LDRFADDR:
dijkstra _L2:      # name of variable
.word 0            # allocate space
.align 4           # align on 4-byte boundary
c:
.word 0
.align 4
ldrf_fill:
.word 0
.space 131036
```

Figure 4.11 Example LDRF References File

LDRF Static Compilation Tool, into a global LDRF directive. The fourth directive, ".globl c", is an example of the common case – a global LDRF directive.

4.5 Assembler Modifications

The assembler is responsible for transforming assembly into machine language. To support the LDRF, two additions are made to the MIPS ISA. Namely, a load- and store-LDRF instruction:

- lg <number of registers>, <VRF start register>, <LDRF address>;
  i.e., lg 1, $2, b
- sg <number of registers>, <VRF start register>, <LDRF address>;
  i.e., sg 1, $2, a

The four fields of the instructions are, respectively:

1. Opcode (lg/sg; 6 bits) – load from LDRF or store to LDRF.
2. Number of registers to be accessed (3 bits) – the size of the block transfer to be undertaken. Note that three bits imposes an upper bound, of eight registers, on the number of registers that may be accessed. Considering typical register requirements, this upper bound seems sufficient.
3. Start register in the VRF (5 bits) – one of 32 registers.
4. Start address in the LDRF (18 bits) – one of, perhaps, 256K registers. Note that 256K represents the maximum number of LDRF registers that could be supported with the current encoding. In practice, the LDRF is considerably smaller.

The disassembler also was modified so that it correctly disassembles binary code, such as object or executable files, that contain sg or lg instructions. Successful disassembling is a critical step to aid debugging and to ensure that the assembler is generating correct machine code.

4.6 Linker Modifications

Because of the LDACT, no linker modifications were necessary. It is important to note, however, that the assembly file created by the LDACT is the first assembly file to be linked. Because it is the first assembly file to be linked, and because the starting address of the data section is known, the starting address of the LDRF also is known. More accurately, the starting

address of the LDRF may be identified by either the starting address of the data section or by the LDRF start address label that is introduced into the references file by the LDACT. Further, because the size of the LDRF is known, the ending address of the LDRF also is known. These considerations are of great importance within the simulator.

4.7 Compiler Analysis Tools

To ease the programmer's decision-making process when assigning variables to the LDRF, several analysis tools were developed. These tools collect information regarding the variables used and in what capacity they are used. Many of the compiler analysis tools work in concert with the simulator analysis tools to provide a comprehensive depiction of the variable.

Within the frontend, the name and data type of each variable is captured and output to a text file. The name has obvious utility – it provides a mechanism to identify which variable is being referenced. The data type provided is restricted to scalar, struct, or array and is used to help guide the LDRF promotion process. Although each of these data types may be promoted to the LDRF, arrays are often ideal candidates. This is because array elements are often accessed sequentially, particularly within a loop, and this is the desired usage to apply the coalescing optimization. Global scalars also are well suited for the LDRF; they are small and usually oft referenced. They, however, do not lend themselves to coalescing as they are but a single element. The third classification, structs, is the least amenable data type. Structs are often large, such that they occupy an appreciable portion of the LDRF, but neither their reference locality nor their temporal locality is assured. In addition, they do not lend themselves to the coalescing optimization. They do more so than scalars, but to a lesser extent than arrays.

CHAPTER 5

SIMULATOR ENHANCEMENTS

The SimpleScalar toolset was modified, as necessary, to accurately support the LDRF. The toolset is designed to allow architectural enhancements to be easily incorporated into its many simulators. Several of the notable modifications that affect all simulators include:

- Modification of the machine description, which is used by all simulators, to properly decode and execute.
- Modification of the machine description to include “free” memory instructions. For each memory instruction defined in the machine description, an analogous “free” instruction was defined. In order to determine the performance impact of memory, as depicted in Figure 1.1, the “free” memory instruction was substituted for the standard memory instruction at runtime. The “free” memory instructions have the same resource requirements and latency as the LDRF.
- Creation of a data structure to represent the LDRF as well as routines to interact with the LDRF. The data structure is populated, at program load, with the initial LDRF values. The routines perform such functions as reading from or writing to the LDRF. Sanity checks also are performed, i.e., verification that the block access is no larger than the maximum block size. Routines also were written to aid debugging, such as a routine that dumps the contents of the LDRF.
- Inclusion of a new fault type to indicate that either the LDRF was accessed by a non-LDRF instruction or to indicate that a LDRF instruction accessed something other than the LDRF. This fault proved quite valuable to help identify bad code generated by the compiler. In some situations, particular in the early development stages of the LDRF implementation, the compiler would generate incorrect code whereby a LDRF variable was not accessed with a LDRF instruction. By forcing the simulator to report a fault, the offending instruction is easily identified and investigated.

5.1 Sim-outorder

Sim-outorder required few specific changes to support the LDRF. This is a reflection of the extensibility of the simulator. The majority of changes were made to accommodate statistical needs or to accommodate analysis tools. With respect to statistics, many metrics were included, within the simulator, so that the efficacy of the LDRF could be measured.

5.1.1 Analysis Tools

Sim-outorder was modified to collect various metrics regarding variable usage. The purpose of the metrics is to provide a reasonable approximation of variable usage so that the programmer may more easily determine which variables to promote to the LDRF. Table 5.1 provides an excerpt of the data collected within sim-outorder. In it, five global variables are listed. Two of which, *errno* and *environ*, are system variables and are not candidates for the LDRF. It also shows *ldrf_fill*, whose purpose is to occupy the unused portion of the LDRF. It is useful as a check – it should never be accessed and its size, plus the cumulative sizes of LDRF variables, should equal the total size of the LDRF. The remaining two variables, *chair* and *table*, are user globals and should be considered for inclusion in the LDRF. The table provides the following information:

- *Base address* – base address of the variable within the memory space.
- *Type* – Type of the variable. If a user variable, then either array, struct, or scalar will be provided, which is supplied by the frontend. If it is not a user variable, then the character corresponding to the data section in which the variable resides will be provided. For example, 'D' refers to global initiated data; 'S' refers to global uninitiated data.
- *Name* – name of the variable.
- *Size* – size of the variable in bytes.
- *Number of uses* – Number of accesses for this variable. For both scalar and composite data structures, this is a cumulative count. It does not indicate which portions – with elements of an array or fields of a struct – were accessed. This information, though obtainable, is of limited usefulness, as the entire structure must be placed in the LDRF. The number of accesses is quite obviously an important metric when considering which variables to place within the LDRF.

- *Access density* – Number of accesses divided by number of bytes. The access density also is an important metric when considering which variables to include within the LDRF. Because the LDRF is of a fixed size, and because it may be that not all candidates will fit in the LDRF, is it desirable to first place those with high access density in the LDRF.
- *Average number of globals between accesses* – Provides the number of global references that occur between references to a given global. A smaller number of intervening references are indicative of a higher temporal locality, which is advantageous.
- *External distance between globals* – Refers to the average number of bytes between the global of interest and the global that was accessed immediately prior to it. This is a measure of spatial locality and those variables with high spatial locality are better suited for the LDRF.
- *Internal distance within the global* – Refers to the average number of bytes between accesses within a global variable; this metric is only applicable to composite variables. It also is a measure of spatial locality and its consideration is more important than the external distance. If a given global has a low internal distance, then it implies that it accesses locations near each other, i.e., sequential locations, are often referenced. A low internal distance, therefore, is suggestive of the likelihood that coalescing is possible.

Considering the values within Table 5.1, *chair* would appear to be a better candidate than *table*. Not only is it accessed more frequently, but it also is ten times smaller such that access density is twenty times that of *table*. It also has a low internal distance, which suggests that elements of the array are accessed sequentially.

Table 5.1 Sample Global Variable Behavior Data

Base Address	Type	Name	Size (bytes)	Num Uses	Access Density	Avg Num	External Distance	Internal Distance
0x100011c0	D	ldrf_fill	131072	0	0	0.00	0.00	0.00
0x100216f0	S	errno	4	5	1.25	1512.8	234.2	0.00
0x100216f4	S	environ	4	1	0.25	0.00	0.00	0.00
0x10021760	array	chair	80	4000	50.00	0.50	40.00	3.8
0x100217b0	array	table	800	2000	2.5	2.00	80.35	25.57

5.2 Sim-profile

Sim-profile was modified so that it had an awareness of the LDRF. More specifically, the LDRF was considered to be a region of memory. Doing so allowed the distribution of accesses with the various segments of memory: LDRF, stack, heap, and data sections to be more easily seen. This was useful when adding variables to the LDRF – the effect, from a memory access standpoint, could therefore easily be seen. Sim-profile also was modified so that two sets of statistics were maintained: one considering all instructions and one considering only VPO-compiled instructions. Because only user variables may be placed within the LDRF, all LDRF instructions will necessarily be VPO-compiled instructions and the benefits derived from using the LDRF are most fairly evaluated by comparison without consideration of system routines, which are not compiled by VPO. However, the VPO-code does not execute in a vacuum. To have a well-rounded understanding of the usefulness of the LDRF, it is necessary to consider how it impacts the performance of an application in its entirety. In other words, one must consider Amdahl's Law. For example, if the LDRF significantly speeds up the VPO-compiled portion of an application, but that portion represents a small portion of the application as a whole, then the usefulness of the LDRF – within the confines of that situation – must be questioned.

5.3 Sim-wattch

Sim-wattch was modified so that it had an awareness of the LDRF. Sim-wattch is an implementation of sim-outorder that also provides power measurements based on the Wattch power model [4]. Initially, sim-wattch was modified so that LDRF instructions could be handled properly, much in the same way that sim-outorder was modified. Next, it was modified so that it had a power awareness of the LDRF. The power model, for the LDRF, was modeled after that of the traditional architected register file. The notable differences pertain to the ability LDRF instructions to access blocks of registers. The power requirements of a LDRF access is presented in Equation 5.1, which on face value is the same computation performed to calculate the power consumption for a traditional register access. However, the power consumption for the decoder occurs *once per LDRF access*. Power consumption from the other contributors occurs *once per LDRF register*. The block LDRF accesses, therefore, save decoder power when the block size is greater than one register.

```
power = decoder + wordline + bitline + senseamp;
```

Equation 5.1 LDRF Power Consumption per Access

CHAPTER 6

EXPERIMENTAL TESTING

The experimental testing was performed within the experimental framework, which was presented in Chapter 3. The purpose of the testing was to (1) establish the viability of the LDRF, (2) better quantify the behavior of a system that employs a LDRF, and (3) identify existing areas that would benefit from additional work and to identify areas of new work. Some of the key questions include:

- *What is the optimal size of the LDRF?* A larger LDRF, quite obviously, provides increased storage capacity. However, it also consumes more power, more die area, and – at some point – will increase the latency of the LDRF. The balance of the larger size versus the costs associated with the larger size needs to be explored.
- *How many candidate variables, for LDRF promotion, are there within a typical application?* Although the compiler framework was enhanced to allow most variables to be promoted to the LDRF, restrictions remain. If there are few variables, per application, which may be promoted to the LDRF, then its usefulness is diminished. This question also serves to guide the question of LDRF size; a large LDRF is of no additional consequence if a smaller one is large enough to support all candidate LDRF variables.
- *What is the impact of naively generated LDRF instructions?* The LDRF is expected to improve several areas of performance. It is instructive to quantify these improvements.
- *What is the impact of the optimized LDRF code?* The optimized code should serve to extend the performance benefits of the naïve code and create additional improvements.

Select configuration parameters of the processor – without the LDRF – is provided in Table 6.1; the complete processor configuration is provided in Appendix A. Once the LDRF was added, there were two adjustments to the processor configuration: (1) the LDRF was added; although this seems trivial, it is relevant to the power measurements as the LDRF – just as any

other architectural feature – consumes power even when not being accessed and (2) the data level cache was reduced, by half, in size. This was done to provide a fairer playing field when comparing a system with the LDRF to one without it. If the size of the data cache was not reduced, one could argue that the results – for example, performance improvement – were as much from having more data storage space available as opposed to the benefit of the LDRF storage space, per se.

Table 6.1 Selected Processor Specifications

Parameter	Value
Machine Width	1
Load/Store Queue	8
Register Update Units	16
DL1 Cache Size	8K 4-way 2 cycle hit
IL1 Cache Size	8K 4-way 1 cycle hit
L2 Cache Size	1M 8-way 7 cycle hit
Memory Latency	150 cycles

With respect to the experimental setup, the following should be noted:

- The benchmarks were modified, by hand within the high-level source code, to declare specific variables to reside in the LDRF.
- The benchmarks were further modified to retain semantic correctness. These modifications were limited to the inclusion of the *gpointer* keyword within pointer declarations.
- Other than the afore mentioned modifications, the benchmarks were not modified in any way. In particular, the compiler was solely used to compile the benchmarks.
- The rolled, optimized code was used as the base case. Where appropriate, these results are explicitly provided. In those cases where it is not provided, the results given are with respect to the base case.
- Unless otherwise noted, the coalescing optimization was *not* applied.

6.1 Experimental Results and Discussion

To measure the baseline efficacy of the LDRF, a representative benchmark from each of the six categories of the MiBench suite was selected; six DSP kernels also were selected (see Tables 3.1 and 3.2). Table 7.2 shows the category, associated benchmark selected from within that category, as well as information pertaining to the variables eligible for promotion to the LDRF. In all cases, the variables in question meet the criteria set forth for a variable to be a good candidate for the LDRF. Furthermore, it should be noted that these variables would otherwise, at best, reside in the data cache since they are either scalar globals or composite data structures, neither of which is traditionally promoted to a register.

It is instructive to consider both the number of variables that are eligible as well as their total size; the total size is reported as the number of bytes after the variables have been extended, if necessary, to comply with the LDRF architectural conventions. Of the forty-seven variables that were eligible for LDRF promotion, nine required type extension. This represents twenty percent of the variables that were eligible and, on average, equates to a twenty-seven percent increase in the minimum LDRF size to accommodate all variables within a given application.

With respect to the LDRF size, 4,971 bytes were needed – on average and on a per benchmark basis – to accommodate all eligible variables. However, the average decreases to 1,638 bytes if a large array (40 Kbytes) with *Dijkstra* is excluded from consideration. This suggests that a LDRF of 1,024 registers – or 4,096 bytes – will provide sufficient capacity to contain nearly all eligible variables within a given benchmark. Also, note that although some applications offered few variables that could be promoted, they did constitute an appreciable size and, more importantly, constituted an appreciable reference count. For example, *Stringsearch* had but two variables eligible for LDRF promotion. However, they occupied 2,048 bytes, which would consume a considerable portion of a typical data cache.

Table 6.2 Benchmark Characteristics

Category	Application	Variables Eligible for LDRF Promotion	
		Number	Total Extended Size (bytes)
Automotive	Bitcount	2	2,048
Consumer	Jpeg	13	2,048
Network	Dijkstra	6	40,816
Office	Stringsearch	2	1,028
Security	Blowfish	6	400
Telecomm	Adpcm	3	2,420
DSP Kernel	Conv45	2	496
DSP Kernel	Fir	3	2,400
DSP Kernel	Iir1	3	2,400
DSP Kernel	Jpegdct	2	1,600
DSP Kernel	Mac	3	2,400
DSP Kernel	Vec_mpy	2	1,600

6.1.1 Execution Time Analysis

Figure 6.1 shows the performance improvement within the MiBench benchmarks as well as the DSP kernels as a function of LDRF size. The improvements are primarily from more efficient access, within the pipeline, to data. Recall that the data retrieved from the LDRF is available in the execute stage, rather than the memory stage, because the LDRF instructions do not require an offset calculation. For the MiBench benchmarks, there is an average performance gain – in terms of the reduction of cycles – of 6.4%, 9.63%, and 10.11% when using a 128-register, 512-register, and 1024-register LDRF, respectively. The results within a given benchmark are reflective of the data signature for that benchmark. For example, *Stringsearch*, which has the largest performance gain (12.45% in the worst case; 24.74% in the best case), utilizes – to near exclusivity – globals and more over global arrays (76% of memory references are to the static data segment). *Jpeg*, on the other hand, relies heavily on variables that occupy the stack and heap segments (93% of memory references are to the stack or heap segments). This is not to suggest that the LDRF is solely appropriate for applications with high use of globals/data segment. Rather, the LDRF is most appropriate for applications with high global/array/struct accesses. *Blowfish* is a good example of this – it has a 2.53% performance gain despite 0.01%

data segment references. The savings are derived from local arrays in non-recursive functions, which are promoted to the LDRF.

For the DSP kernels, there is an average performance gain – in terms of the reduction of cycles – of 11.8%, 17.75%, and 21.11% when using a 128-register, 512-register, and 1024-register LDRF, respectively. The loop- and array-oriented nature of DSP kernels is easily exploited by the LDRF. In general, the benchmarks show the expected behavior whereby performance gains increase as the size of the LDRF increases. *Conv45* and *Jpegdct* are apparent exceptions, as the performance improvement remains constant despite the larger LDRF size. In the case of *Conv45*, it cannot exploit a LDRF larger than 128 registers. In the case of *Jpegdct*, there are gains as the LDRF size is increased, but they too slight to be easily discerned within the figure. The modest gains are due to low reference counts of the variables that were added as the LDRF size increased. This does, however, underscore the importance of variable-promotion selection – if the variables with low reference counts were promoted first, there would be little performance gain and inaccurately suggest that the smaller LDRF was of little benefit. *IIR1* also is an apparent exception, as there is no apparent gain from a 512-register LDRF when compared to a 256-register LDRF. In this case, the variables promoted to the LDRF are sized such that they are not able to take advantage of the 512-register LDRF.

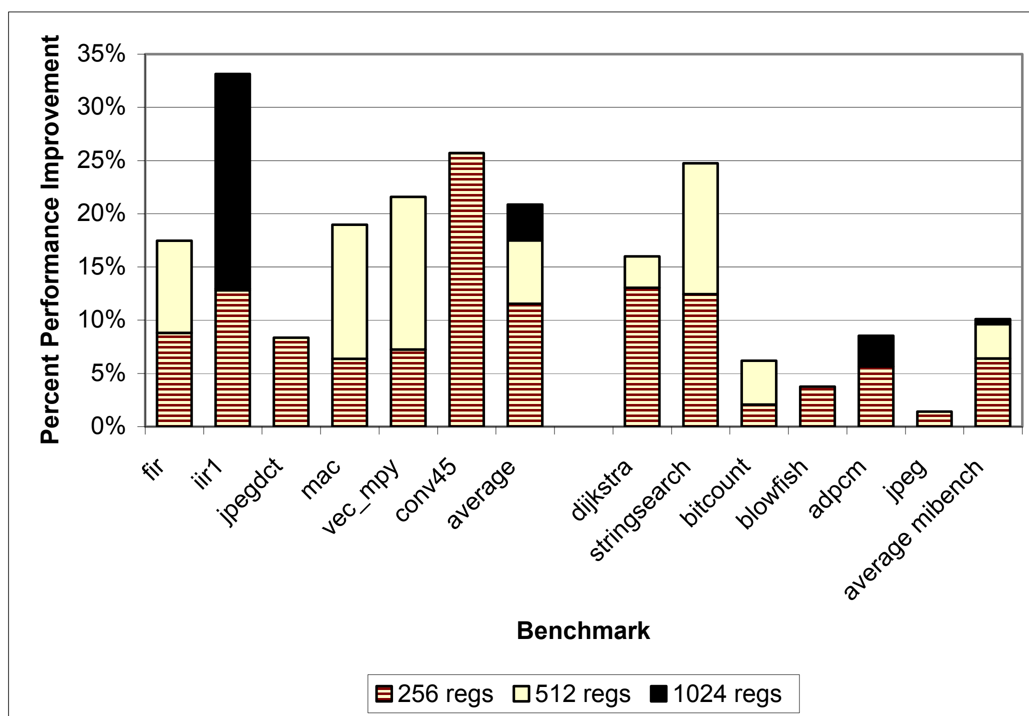

Figure 6.1 MiBench and DSP Performance Results

6.1.2 Memory Instruction Count

Figure 6.2 shows the reduction in memory instructions. On average, the MiBench benchmarks experienced a 35.55% reduction of memory instructions and the DSP Kernels experienced an average reduction of 99.97%. Even in the case of the MiBench benchmarks, the reduction is considerable. Note that – without coalescing – one LDRF instruction typically replaces one memory instruction such that the total number of instructions will be nearly the same. However, it is useful to consider the reduction of memory instructions, as these instructions are higher cost than the LDRF. In fact, the pattern of performance gains, particularly for the MiBench applications, closely mirrors the pattern of memory instruction reduction. For example, *Stringsearch* and *Jpeg* have the highest, and lowest, reduction of memory instructions, respectively. They also have the highest, and lowest, performance increase, respectively. The LDRF instructions, to reiterate previous comments, provide data at a later stage in the traditional pipeline and have a higher latency. In particular, if the memory instruction requests data that is not on-chip, i.e., in either the data level one cache or the data level two cache, then the latency will be interminable. When considering coalescing, the overall number of instructions will decrease, as one LDRF instruction will replace two or more memory instructions.

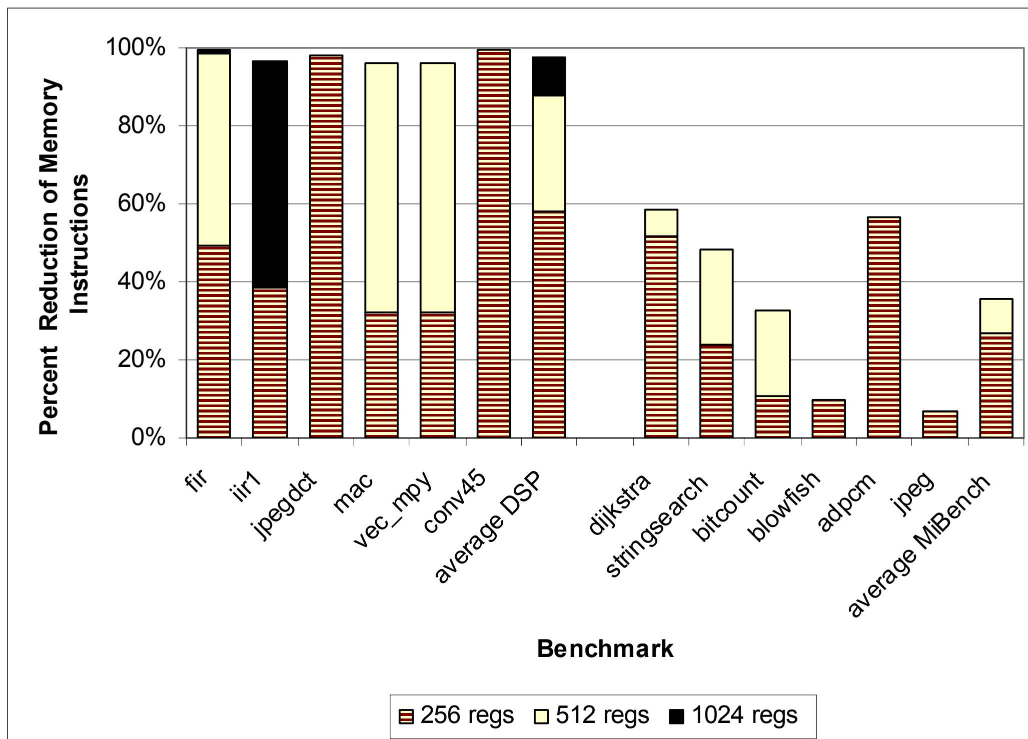

Figure 6.2 Reduction in Memory Instructions

6.1.3 Data Cache Access Patterns

Figure 6.3 shows the data cache access patterns for the MiBench applications. As expected, the total number of accesses to the memory hierarchy – comprised of the level 1 data cache, level 2 data cache, and memory accesses – decrease as data are moved from the memory hierarchy into the LDRF. It is interesting to note, however, the number of misses within the level 1 data cache *increases* when using the LDRF. This is due to the smaller sized cache that is used in conjunction with the LDRF. However, despite the increased accesses to the level 2 data cache, these benchmarks still realize an overall performance gain due to the gains associated with LDRF access.

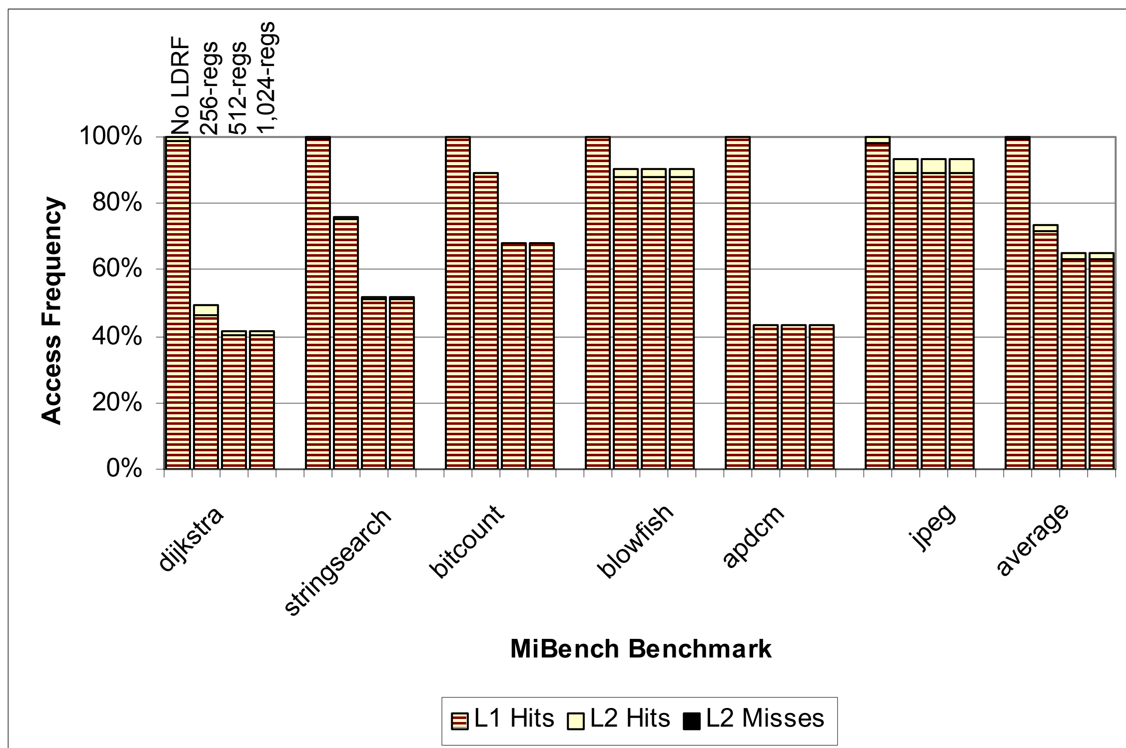

Figure 6.3 Data Cache Contention

6.1.4 Power Analysis

A system that employs the LDRF can reasonably be expected to consume less power than one that does not, even while also incurring a performance benefit. The performance benefit has been discussed and established for a system that uses the baseline LDRF. Power, often a concern not only for embedded systems but for general-purpose systems as well, can be as important a metric as performance. Figure 6.4 presents the total processor energy reduction when using the LDRF. When using the 1,024-register LDRF, an average savings of 8.04% is

realized for the MiBench applications, and an average of 20.14% is realized for the DSP kernels. As with the performance gains, there is a direct correlation between the increase in the number of LDRF instructions and the increase in the total energy savings. The LDRF instructions, as previously discussed, are less expensive than a corresponding memory instruction. It also should be noted that the energy reported is that of the processor; due to constraints of the Wattch model, it does not include the energy required to retrieve data from off-chip. Therefore, the energy savings presented in the figure may reasonably be considered conservative. The LDRF will likely reduce the number of main memory accesses and thereby realize even greater energy savings.

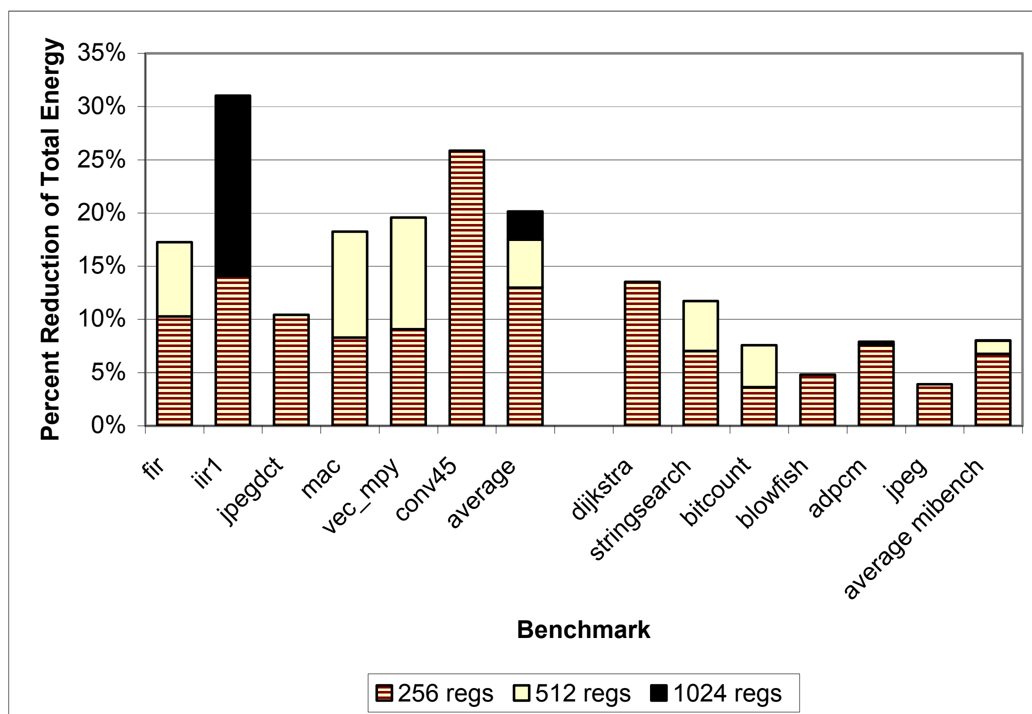

Figure 6.4 Percent Energy Reduction

6.1.5 Coalescing Optimization

Although the benefits of the LDRF have been demonstrated, these benefits are derived from unoptimized – with respect to the LDRF – code. More specifically, the code has been generated without the benefit of the coalescing optimization, which has the potential to amplify the benefits of the unoptimized code. Figures 6.5 shows the effect of coalescing when applied to the DSP kernels. The figure shows the performance of four processor configurations – no LDRF, 256-register LDRF, 512-regsiter LDRF, and 1,024 register LDRF – running the DSP kernels where

the loop unroll factor (UR) has been increased by a power of two, such that it varies from a UR = 1 (no unrolling) to a UR = 8. At the highest unroll factor, the average performance gains are 31%, 34%, and 34.5% for the three sizes of the LDRF; recall that the gains for the unoptimized were 11.8%, 17.75%, 21.11% for the respective sizes. Note that loop unrolling itself provides some performance benefit. This benefit can be most easily seen in the case where no LDRF is used; the coalescing is responsible for the additional benefits.

Although coalescing opportunities do exist without loop unrolling, they are far more likely to occur when loop unrolling has been applied: of the six DSP kernels, loop unrolling creates sequential LDRF references that may be coalesced for all but *Jpegdct*. Although the unrolled innermost loops of *Jpegdct* do create sequential LDRF references, loads and stores to the same LDRF address are interspersed with one another such that coalescing would not retain program correctness. Because there is little coalescing within Jpegdct, the loops within Jpegdct have low iteration counts, and it cannot exploit a LDRF larger than 512-registers, its performance increases are relatively constant with increasing LDRF size. The behavior of *IIR1* is interesting as well. In contrast to the smaller sized LDRFs, the 1,024-register LDRF does not appear to realize a performance gain when unrolling. *IIR1* naturally has sequential LDRF accesses; these accesses coalesce when the unroll factor equals one (or no unrolling). In the case of the 1,024-register LDRF, the natural coalescing that occurs consumes the available registers within the traditional register file. Unrolling, in this case, is not helpful. In fact, although

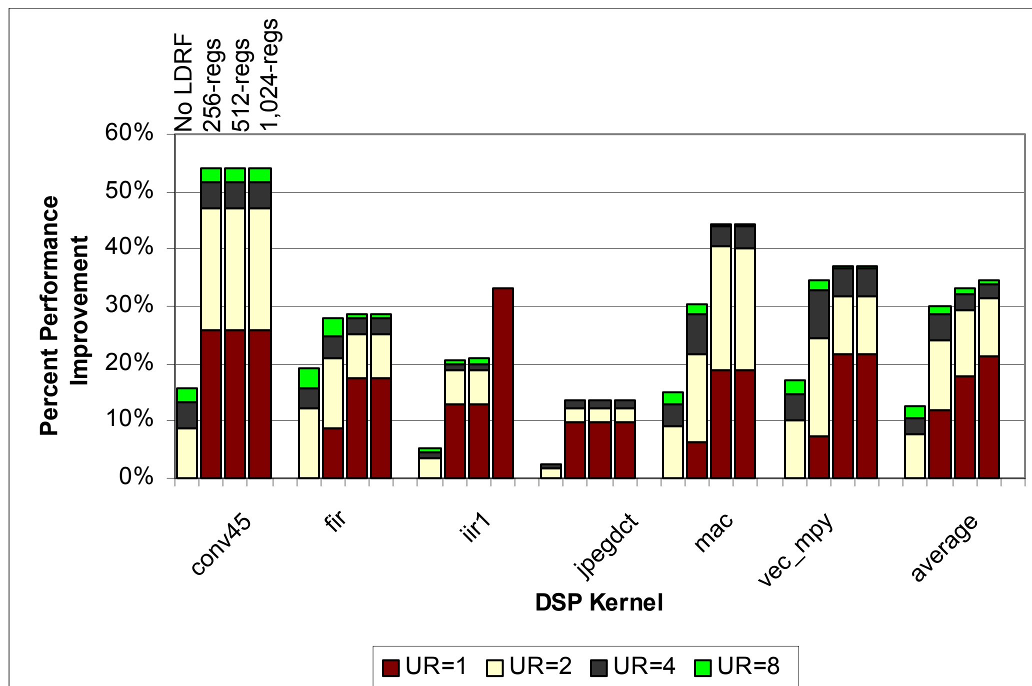

Figure 6.5 Coalescing and Performance

the unrolling reduces the number of branch instructions thereby providing a performance boon, it also adversely affects performance. Those LDRF instructions that are not coalesced, and would have an offset, are necessarily represented as two instructions as discussed in Chapter 4.5. The negative impact of the extra instruction, to calculate the effective address, is small and remains faster than a traditional memory instruction by virtue of the lesser latency of the LDRF.

The energy benefits, as related to the coalescing optimization, are provided in Figure 6.6. In general there is a decrease in the total energy consumed as the unroll factor increases. This is to be expected as the number of cycles required to complete the benchmark is decreasing. However, it also shows that a configuration with a LDRF – even the 1,024-register LDRF – and a smaller cache is far more energy conscious than the configuration without the LDRF and the larger cache. This is due in part to the performance improvement of a configuration with a LDRF, but also due to the decreased energy requirements of the LDRF and half-cache when compared to the full cache. The effects of increasing the LDRF size are most easily seen in *Conv45*. Because it can only exploit a 128-register LDRF, the decrease in the energy reduction, as the LDRF size increases, is an indicator of the additional energy required to power the increasing LDRF size. For *Conv45*, there is less than one percent differential between the energy consumed by the smallest, and largest, LDRFs.

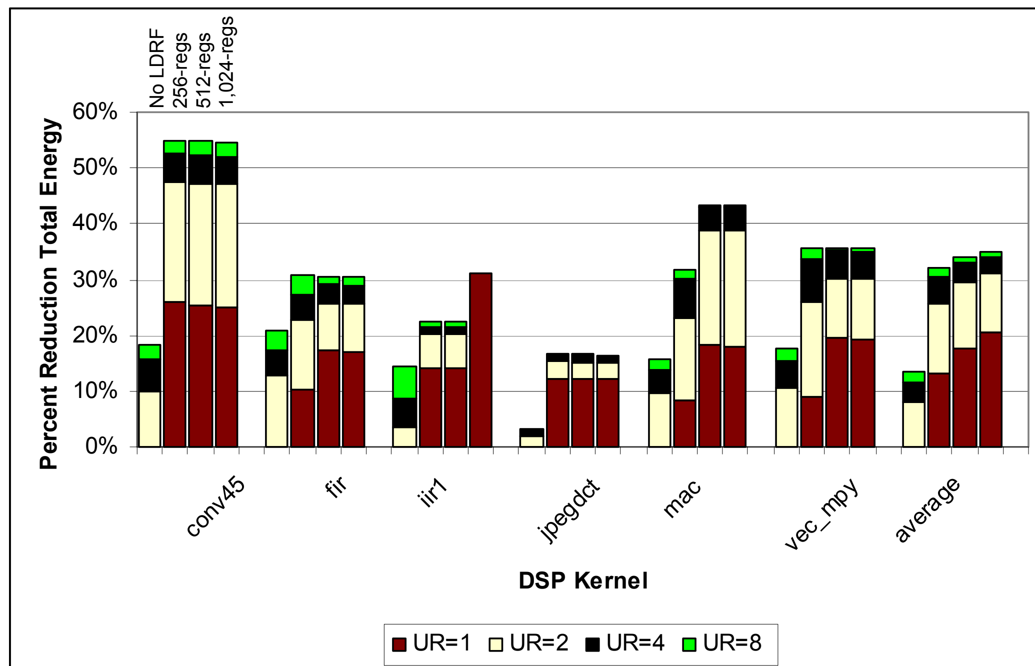

Figure 6.6 Coalescing and Energy

CHAPTER 7

RELATED WORK

Several approaches have been investigated to make use of a larger number of registers. Unlike our method, these approaches enforce severe restrictions on how the registers are accessed and/or can only be applied in a limited manner.

The register connection [14] method tolerates a high demand for architected registers by adding a set of extended registers to the core register set. It also incorporates a set of instructions to remap architected register specifiers into the extended set of physical registers. The remapping between the architected register file and the extended register file is performed one register at a time. This method is acceptable to avoid spill code for scalar registers that do not fit in the architected register file, but requires an excessive overhead penalty for remapping larger data, such as arrays.

The register queues [19] approach is an extension of the register connection method, in which each architected register could be mapped to a set of extended registers with a FIFO ordered queuing access pattern. The FIFO ordering of register queue accesses significantly reduces the need to remap architected register space to access multiple extended registers, but it also restricts the type of code than can automatically exploit this feature.

Rotating registers [7] have been used to facilitate software pipelining. After software pipelining a loop, often a value that is produced in one loop iteration is used in a subsequent iteration. Rotating registers facilitate access to these values without requiring registers to be copied from one register to another. The rotating register approach restricts the manner in which the extra registers are accessed and still has a one-to-one encoding of registers in the instruction set architecture, limiting the number of additional registers that can be added without a significant increase in instruction size.

Register windows have been used in processors, such as the SPARC architecture, to hold values associated with a function's activation [23]. Each time a function is invoked, a register window pointer (RWP) is advanced to the next window. When a return from a function is encountered, the RWP will be updated to point to the previous register window. Thus, windows

are accessed in a LIFO manner, while registers within a window are accessed in an arbitrary order. The main advantage of using multiple windows in this fashion has been to avoid saves and restores of registers that normally occur at the entry and exit of a function, respectively. The register stack engine (RSE) of the Itanium-64 employs a similar concept, except that the number of registers allocated by each activation of a function is not constant [12]. Like conventional register sets, register windows are used to hold scalar values and thus the total number of registers that can be effectively exploited is relatively small.

The Stack Value File (SVF) [15] is implemented as a very large register file, is used to store run-time stack data, and is implemented as a large circular buffer. The registers in the SVF are indirectly accessed through conventional loads and store instructions. Instructions that reference the run-time stack using a displacement of the stack pointer are morphed into register-to-register moves after being fetched and decoded. Thus, most accesses to the SVF occur earlier than accesses to the L1 cache, which reduces access latency. However, accesses to the SVF still require conventional load/store instructions. In contrast, our approach removes load/store instructions, avoids memory access to data not on the run-time stack, and allows direct access to LDRF registers.

Scratchpad memories are small compiler managed storage structures that can be overlapped or independent of memory addressing. Scratchpad memory is accessed in the same manner as main memory, but since the allowable address space is much smaller, the entire space can be placed in fast on-chip storage. This enables scratchpad memories to have access latency similar to L1 cache hit times, while guaranteeing the data will reside in the memory. The compiler generally decides which data is placed into scratchpad memories. There have been a number of studies examining allocation strategies [2][21][1]; most perform static allocation of the most heavily referenced data, while a few examine dynamic promotion of data from main memory into the scratchpad. New instructions to access the scratchpad storage are typically not required since the address space is shared with main memory. Advantages of using scratchpads is that they consume less power than a conventional L1 data cache and they are useful for ensuring that frequently referenced data can always be accessed in a single cycle. However, access to data in a scratchpad memory is not as efficient as the LDRF since access occurs later in the pipeline and only a single value can be accessed at a time.

The Smart Register File [17] modifies traditional register file access semantics to include an indirect access mode, which provides support for aliased data items in registers. As a result, the compiler can allocate data from a larger pool of candidates – such as composite data structures – than in a conventional system. This modification lends expansion of the traditional register file

into a larger structure because the compiler is now more likely to be able to take advantage of the additional registers. CRegs [8] is another technique that solves the aliasing problem. However, access to data in both the Smart Register File and CRegs is less efficient when using the LDRF architecture, as it does not support the access of multiple values.

Memory access coalescing [6] is a compiler optimization that coalesces multiple loads or multiple stores into a single instruction. By combining multiple loads into a single coalesced load, the movement of data from the cache can be handled more efficiently in the microarchitecture. While coalescing reduces the number of load and store instructions, it does not change the amount of data transferred from/to memory. Also, unlike our approach, memory access coalescing requires that these larger memory references must be aligned on an address that is an integer multiple of the data reference size. This alignment requirement is due to accessing data from a cache, can restrict the number of opportunities for coalescing memory references, and can require additional instructions be generated that performs checks to ensure that addresses are aligned. In contrast, variables referenced in the LDRF are guaranteed to be resident and sequential sets of these registers can be accessed without alignment restrictions.

CHAPTER 8

FUTURE WORK

The work to date clearly establishes the viability of the LDRF. However, there are several areas that merit additional work. Some future work is best viewed as a refinement of work already undertaken and some is simply new work. All of the potential areas for additional work serve to make the LDRF implementation more robust, more flexible, and more likely to be strongly embraced by the research community. Several of the potential areas for future work are discussed in the subsections that follow.

8.1 Automation of Variable Promotion to the LDRF

It is envisioned that allocation of variables to the LDRF will become an automated process that is controlled by the compiler. Akin with the *register* keyword, the *gregister* keyword will remain available to the programmer, but will become a suggestion to the compiler. Thus, the compiler will be the final decision-maker and also will promote variables to the LDRF on its own accord. This is seen as a natural progression of the research. One of the compiler's responsibilities – or, more generally, the computer for that matter – is to ease the burden of the programmer. Although it is logical for programmers to be familiar with their programs, to the extent that they may make educated variable promotions to the LDRF, that ultimate decision should reside with the compiler. This is particularly true in a large application with many source files, where it would be onerous for the programmer to be the sole decision-maker.

The initial infrastructure to support automated variable promotion is in-place. Recall the analysis tools, which provide rudimentary recommendations to the programmer as to which variables might be best suited for inclusion in the LDRF. If the initial infrastructure were retained, then a multi-pass compilation/simulation would be required. This is necessarily so as the simulator – not the compiler – provides the hit density, which is one of the most important metrics to consider when allocating variables to the LDRF. In this paradigm, the application would be compiled and executed in order to generate metrics, to be read in by the compiler, that allow the compiler to determine which variable to promote to the LDRF. This approach has the advantage that it directly builds on the existing infrastructure and, in addition, it uses dynamic

counts on which it bases its decisions. The disadvantage, which likely outweighs the afore mentioned positives, is that it requires not only a complete compilation but a complete execution before it may allocate variables to the LDRF. Considering that an application may take several minutes to compile, and several hours to execute, the time required may be prohibitive. As an alternative, the compiler would be responsible for all analysis necessary for it to evaluate and promote variables to the LDRF. A reasonable approach to achieve this end would be to retain the compilation process as a multi-pass endeavor. The first pass would generate static count estimates of the program variables which, when considered with their size, provides an estimate of their hit density. This information, as well as the other criteria that may be collected by the compiler, may then be used within the second pass to allocate variables to the LDRF. It would be a matter of research as to how best implement the second pass; likely, it would begin with the frontend. This would ensure that correct stack code would be produced for consumption by the middleware and down the compiler toolchain. It would, however, increase the compile time as the second pass would repeat the compiler sequence in its entirety. If it were deemed desirable to constrain the second pass to the backend, it would necessarily have the added complexity of not only extending the variables, placed into the LDRF, to integers but it also would need to adjust the code to reflect the extension to integers. Further, it would be responsible for actions that are normally performed by the frontend, such as type checking. As an example, consider an array that is declared – by the compiler – to be a local static gregister. It is subsequently passed, as a pointer argument, to a function. The backend must perform the type check, normally performed by the frontend, to verify that the function parameter is declared to be a gpointer.

8.2 Permit Promotion of Dynamically Allocated Variables to the LDRF

Currently, only variables whose sizes are known at compile time may be allocated to the LDRF. A reasonable approach, to be sure, as it eliminates significant complexity by disallowing dynamically allocated variables to be promoted to the LDRF. However, it also dramatically – for some applications – reduces the number of opportunities for LDRF promotion.

In order to support promotion of dynamically allocated variables to the LDRF, allocation routines would be evaluated and implemented. There are several well-established algorithms – including first fit, best fit, buddy – for allocating dynamic memory. These algorithms would be evaluated, in terms of use with the LDRF, to determine which is most appropriate. In addition, a LDRF-compatible malloc() and free() would need to be created. These routines could be made

available to the programmer, or they could overload the existing malloc() and free() routines so that the LDRF versions are called as appropriate.

It also would be a matter of research as to what portion of the LDRF is available for dynamic allocation. A simplistic approach would be to allocate some fixed portion of the LDRF to statics and allow the remainder of the LDRF to be used for dynamic allocation. This approach would be straightforward to implement and to establish the viability of dynamic allocation, but it is not appropriate for a robust solution. Its failing lies in its simplicity – because it establishes the static region as well as the dynamic region without first establishing the needs of the program, the allocation may be less than ideal for the given program. It would a far better use of space to allocate the statics, to the LDRF, and allow any remaining space to be used for dynamic allocation. This ensures that no space will be wasted, per se, but it does not necessarily make the best use of that space. For example, the statics that are allocated to the LDRF may have a low access count, particularly when compared to the access count of a dynamically allocated variable.

It is possible than dynamic allocation will exceed the available LDRF resources. Thought will necessarily need to be given as to the best course of action when this occurs. Either a run-time error, i.e., "Out of LDRF resources", could occur or a portion of the LDR could be moved to a different storage location. The former is simpler to implement but less flexible. The latter greatly increases the flexibility, but also greatly increases the complexity. It may be appropriate to consider dynamic allocation techniques that are employed in scratchpad memory systems [21], which faces similar challenges. The difficulty lies in that the preferred method is not access count, but access density, which necessarily cannot be determined at compile time for a dynamically allocated variable. These considerations, and others, make promotion of dynamically allocated variables challenging.

8.3 Alternative LDRF Architecture to Support Non-Contiguous Accesses

The LDRF architecture, as designed, permits block accesses where the blocks represent a continuous range to be transferred to/from the LDRF from/to a continuous range within the traditional register file. The coalescing optimization serves to coalesce the accesses into a single instruction thereby creating the block access. However, in order for the optimization to be successful, it must find an appropriately sized block of sequential registers. As the unroll factor increases, the block that might be created typically increases. For example, an unroll factor of eight lends itself to creation of a block of eight references. However, large sequential ranges of registers are often not available. One alternative would be to perform extensive renaming,

where even those registers that are identified as live within the register rename range would be made available for renaming. This would add considerable complexity as the live registers, if used for renaming, would necessarily require additional instructions to retain program correctness. The value of the coalescing a greater number of instructions would then need to be considered in context of the additional instructions. Likely, so long as the coalesced instruction was executed in particular when compared to the added instructions, there would be an overall performance improvement. However, this would be a matter of additional research. The second alternative is to relax the constraint that the registers must be sequential. This would simplify the renaming process as any *n* available registers could be used. However, it would increase the complexity of the interaction between the LDRF and the VRF. The added complexity would need to be evaluated from both a power and performance perspective. In particular, it would be necessary to establish that the redesigned LDRF retains its low latency.

8.4 Extension of the Coalescing Optimization

The Coalescing Optimization is currently restricted to a single basic block. Although this is sufficient to capture the majority of coalescing opportunities, it would be preferable to relax this constraint. As a motivating example, consider Figure 8.1. The for-loop simply loops over a

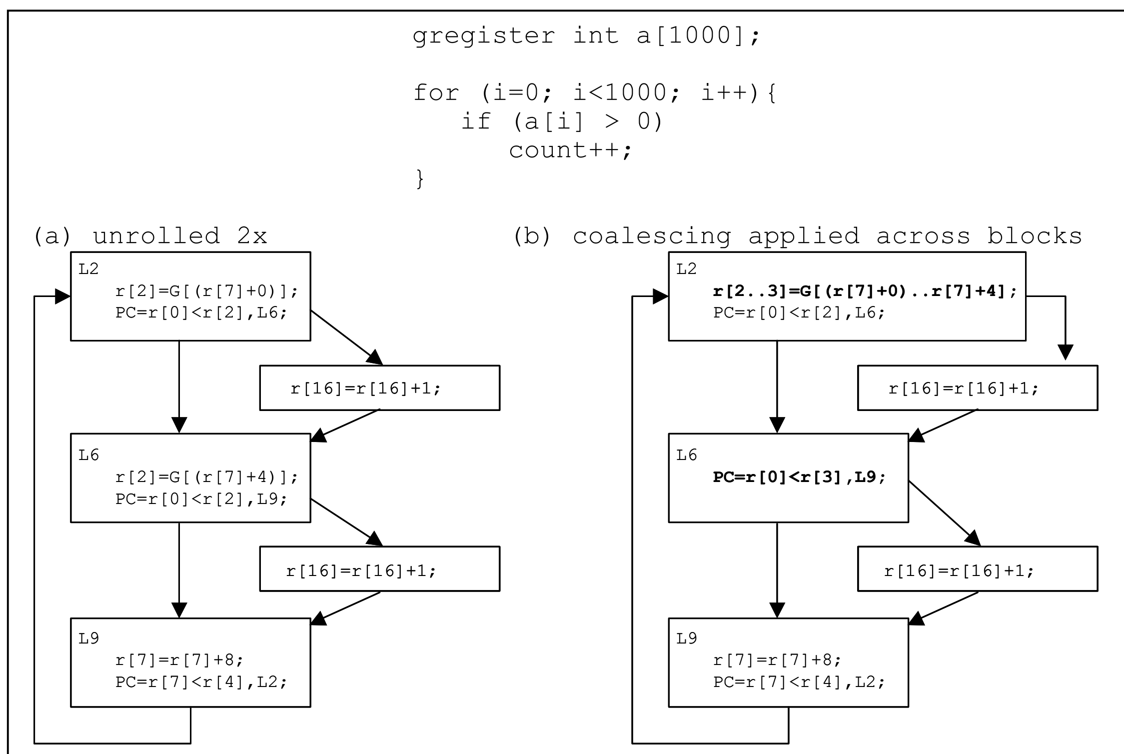

Figure 8.1 Coalescing Across Blocks

gregister array and tallies the number of positive values within the array. The RTLs within Figure 8.1(a) are those of the loop, which has been unrolled two times; those within Figure 8.1(b) show the RTLs as they might appear after coalescing has occurred. In both cases, the control flow is provided as well as the block boundaries. Because the LDRF access within the block labeled L6 is sequential with that of the LDRF reference labeled block L2, it is a candidate for coalescing across blocks. However, there are a number of other considerations that must be evaluated before the coalescing would be considered safe, including: (1) both LDRF accesses must be executed on every iteration of the loop and (2) the basic block hierarchy must be evaluated – both in terms of taken and not-taken branches – to verify that the coalescing is safe. The latter aspect will require each block across which the access will be coalesced must be evaluated for the myriad of conditions that invalidates a coalescing opportunity.

CHAPTER 9

CONCLUSIONS

Because the gap between processor speed and memory speed has continually widened over the years, the performance of the memory hierarchy has had an increasingly detrimental effect on performance. Accordingly, alternative storage locations need to be identified; particularly one which reduces the deleterious impact of the memory hierarchy. This thesis has presented such an alternative, a large data register file. The LDRF acts as an alternative to the data cache and retains many of the advantages of a small register file. It also provides additional added value. In particular, the LDRF supports aliased data as well as composite data structures, which typically are relegated to the first-level data cache, which provides slower access times. In comparison to a data cache, data will be made available – within a traditional 5-stage pipeline – one stage earlier when placed in the LDRF. In addition, the size and structure of the LDRF allow it to have a lower access time than a typical data cache.

Promotion of variables, to the LDRF, is currently the programmer's responsibility. However, several analysis tools are provided to ease the identification of candidate variables. In addition, inclusion of but two reserved keywords to accommodate the LDRF, at the high-level source code, makes promotion of variables to the LDRF rather simple. Robust and thorough compiler enhancements were implemented to verify that the LDRF variables are being used properly, and to generate both naïve and optimized code. Many of the optimizations that were put in-place to support the coalescing of LDRF instructions are applicable to a wide variety of situations, and are not solely applicable to the LDRF research.

Results show that the LDRF behaves as expected and confirms the thought that use of the LDRF is advantageous. Even without use of the coalescing optimization, there is a significant reduction of memory traffic, as oft-referenced variables are now located in the LDRF. The reduction in memory traffic equates to a reduction in power, a reduction in contention for the data cache – which can lead to fewer data cache misses, a reduction in the number of data TLB accesses, and a reduction in the number of cycles to complete an application. The coalescing optimization builds upon these results to further reduce the number of cycles to complete an application, to further reduce the reduction in power, and to reduce the dynamic instruction

count. The additive benefits of the coalescing optimization are a natural product of coalescing several instructions into one. Considering that many applications – particularly numerically intensive applications – spend the majority of time within tight loops, coalescing as few as two LDRF instructions – within a tight loop – into one instruction can have an appreciable impact on the performance of an application.

Without a fundamental shift in the methodology by which computing is performed, i.e., a complete reimplementation of the memory hierarchy, it is likely that the gap between processor speeds and memory speeds will continue to increase. Solutions to this pressing problem, such as the LDRF, will therefore increase in value.

APPENDIX A: SIMPLESCALAR CONFIGURATION

These values provided, below, represent the SimpleScalar configuration used for the experimental testing.

Parameter	Value
instruction fetch queue size (in insts)	4
extra branch misprediction latency	3
speed of frontend of machine relative to execution core	1
branch predictor type	bimodal
bimodal predictor configuration	2048
return address stack size	8
BTB configuration	512 sets; 4-way
instruction decode, issue, and commit bandwidth (insts/cycle)	1
run pipeline with in-order issue	False
issue instructions down wrong execution paths	True
register update unit (RUU) size	16
load/store queue (LSQ) size	8
ldrf hit latency (in cycles)	1
l1 data cache configuration	64 lines; 32 byte blocks; 4-way; LRU replacement policy
l1 data cache hit latency (in cycles)	2
l2 data cache configuration	1024 lines; 128 byte blocks; 4-way; LRU replacement policy
l2 data cache hit latency (in cycles)	7
l1 inst cache configuration	64 lines; 32 byte blocks; 4-way; LRU replacement policy
l1 instruction cache hit latency (in cycles)	1
l2 instruction cache configuration	Unified
flush caches on system calls	False
convert 64-bit inst addresses to 32-bit inst equivalents	True
memory access latency (in cycles))	150 first chunk; 4 inter chunks
memory access bus width (in bytes)	8

instruction TLB configuration	16 lines; 4K blocks; 4-way; LRU replacement policy
data TLB configuration	32 lines; 4K blocks; 4-way; LRU replacement policy
inst/data TLB miss latency (in cycles)	30
total number of integer ALUs	4
total number of integer multiplier/dividers available	1
total number of memory system ports available (to CPU)	2
total number of floating point ALU's available	4
total number of floating point multiplier/dividers available	1

REFERENCES

[1] F. Angiolini, L. Benini, and A. Caprara, “Polynomial-time algorithm for on-chip scratchpad memory partitioning”, *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, October 30-November 01, 2003, San Jose, California, USA.

[2] O. Avissar, R. Barua, and D. Stewart, “An optimal memory allocation scheme for scratch-pad-based embedded systems”, ACM Transactions on Embedded Computing Systems (TECS), v.1 n.1, p.6-26, November 2002.

[3] 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.

[4] D. Brooks, V. Tiwari, and M. Martonosi. “Wattch: A framework for Architectural-level power analysis and optimizations”, *Proceedings of the International Symposium on Computer Architecture*, pg 83-94, June 2000.

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

[6] J. W. Davidson and S. Jinturkar, “Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses,“ *Proceedings of the SIGPLAN ’94 Symposium on Programming Language Design and Implementation*, pg. 186-195, June 1994.

[7] J. Dehnert, P.Hsu, and J. Bratt, “Overlapped Loop Support in the Cydra 5,” *Proceedings for the Third International Conference on Architectural Support for Programming Language and Operating Systems*, pg. 26-38, April 1989.

[8] H. Dietz and C.-H. Chi. “CRegs: A New Kind of Memory for Referencing Arrays and Pointers”, In *Proceedings of Supercomputing '88: November 14-18, 1988*, Orlando, FL, pp. 360-367, January 1988.

[9] C. W. Fraser and D. R. Hanson, A Retargetable C Compiler: Design and Implementation, Addision-Wesley, 1995.

[10] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown, “MiBench: A free, Commercially Representative Embedded Benchmark Suite”, IEEE 4th Annual Workshop on Workload Characterization, Austin, TX, December 2001.

[11] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, Q1 2001.

[12] Intel Literature Centers, IA-64 Application Developer’s Architecture Guide, Intel, Inc. 1999.

[13] N. P. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers“, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pg. 364-373, 1990.

[14] K. Kiyohara, S. Mahlke, W. Chen, R. Bringmann, R. Hank, S. Anik, and W. Hwu, “Register Connection: A New Approach to Adding Register in Instruction Set Architectures.” *Proceedings of the International Symposium on Computer Architecture*, pg. 247-256, May 1993.

[15] H. Lee, M. Smelyanski, C. Newburn, and G. Tyson, “Stack Value File: Custom Microarchitecture for the Stack”. *Proceedings of the International Symposium on High Performance Computer Architecture*”, pages 5-14. January 2001.

[16] MIPS Technologies, “MIPS32® Architecture for Programmers Volume II: The MIPS32® Instruction Set”, Version 2.5, July 1, 2005.

[17] M. Postiff and T. Mudge, “Smart Register Files for High-Performance Microprocessors”, CSE-TR-403-99, University of Michigan, June 28, 1999.

[18] E. Rotenberg, J. Smith, S. Bennett, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," *Proceedings for the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29)*, pg. 24, 1996.

[19] M. Smelyanskiy, G. Tyson, and E. Davidson, “Register Queues: A New Hardware/Software Approach to Efficient Software Pipelining,“ *International Conference on Parallel Architectures and Compilation Techniques (PACT 2000)*, October 2000.

[20] D. Sweetman, See MIPS Run, Morgan Kaufmann Publishers, 1999.

[21] S. Udayakumaran and R. Barua, “Compiler-decided dynamic memory allocation for scratch-pad based embedded systems”, *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, October 30-November 01, 2003, San Jose, California, USA.

[22] S. P. Vanderwiel and D. J. Lilja, “Data prefetch mechanisms”, ACM Computing Surveys (CSUR), Volume 32, Issue 2, pg. 174-199, June 2000.

[23] D. Weaver and T. Germond, “The SPARC Architecture Manual”, SPARC International, Inc., Menlo Park, CA. 1994.

BIOGRAPHICAL SKETCH

Mark C. Searles

Mark Searles received a Bachelors of Arts Degree, in chemistry, from Cornell University. After several years in the working world, he realized his affinity for computing and elected to pursue a graduate degree. In the Fall of 2006, he graduated from The Florida State University with a Master of Science Degree in Computer Science. Upon graduation, he returned to the working world, albeit to a career more to his liking. His main area of interest is compilers.