Exploitation of a Large Data Register File

Mark Searles, David Whalley, Gary Tyson

Florida State University, Tallahassee, FL, 32306, USA

ABSTRACT

As the gap between CPU speed and memory speed widens, it is appropriate to investigate alternative storage systems. Our approach is to use a large data register file that is able to hold arrays and other global structures. Windows of registers are utilized for efficient access. The performance benefits realized from this approach include faster access, fewer instructions executed, and decreased contention within the data cache.

1 Introduction

Because the increases in processor speed have outpaced the increases in memory speed, the difference between the processor speed and memory speed has continually widened over the years. Despite aggressive instruction scheduling, processors are often unable to be fully utilized, as they must endure long latencies while waiting for data to be fetched from memory. Accordingly, it is appropriate to investigate alternative storage systems. Our approach is to use a large data register file (LDRF). There are several, well-known advantages of accessing data from registers instead of memory (even if the data actually resides in a data cache rather than main memory). These advantages include: faster access time, accessing multiple values in a single cycle, reduced power consumption, and reduced bandwidth requirement for the first-level data cache. The LDRF, despite its increased size, retains these advantages.

2 The Memory Bottleneck

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*, which shows 26%, on average, of instructions are memory operations) and (2) memory speeds are significantly slower than processor speeds. Accordingly, many techniques have been studied to hide the latency of main memory from the processor. However, it is not feasible to hide all such latencies. As seen in *Figure 1*, significant performance benefits would be realized if all memory operations were to complete in the same amount of time

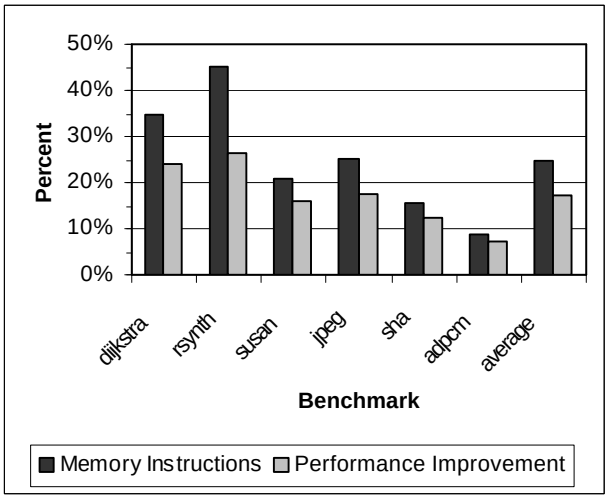

Figure 1: Performance Impact of Memory

as a register operation. While it is not suggested that this 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 realizes performance gains.

3 Architecture and Compiler Enhancements

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 C structs. 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 *in the same amount of time* as a traditional register file. The LDRF is able to achieve this by:

- *Use of register windows,* which restrict the available portion of the LDRF to a more manageable size. This, in turn, reduces encoding size and helps to retain fast access.
- *Block transfer of registers,* which allows for fast access to multiple, sequential LDRF registers. This access allows multiple values to be transferred to/from the LDRF within a single cycle.

The primary difference between our approach and earlier register windowing[1][2][3] schemes lies in the generality of the window manipulation capabilities and the ability to promote the wider range of application data values to the LDRF, i.e., global as well as static local composite variables. Many global arrays, particularly in numerical applications on embedded processors, are well suited for the LDRF. *Figure 2* depicts the interaction between the traditional (visible) register file (VRF) and the LDRF. Note that register window pointers (RWPs) are used to associate a contiguous set of VRF registers with an equal sized set of contiguous LDRF registers. RWP assignments cause the LDRF window of registers to be copied, via parallel register moves, to the VRF. After loading the registers from the LDRF window to the VRF window, subsequent accesses to the registers in the window occur from the VRF in a conventional manner.

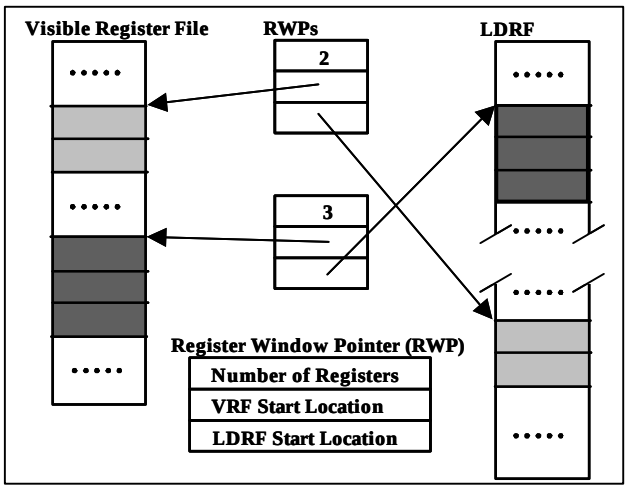

Figure 2. VRF / LDRF Relationship

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

- *Accesses to the data should exhibit high spatial locality* – Since accessing each window within the LDRF requires an assignment to a RWP, referencing several register within a single window amortizes the cost of such assignments.
- *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.
- *Size 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.
- *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 is a more useful metric rather than simply the number of accesses.

The compiler and simulator collect metrics and perform subsequent analysis on these metrics to provide recommendations, to the programmer, as to which variables are best suited to the LDRF. SimpleScalar was used to collect performance statistics.

In our current research, the compiler used is a port of VPO (Very Portable Optimizer) for the MIPS architecture. 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. To allow the programmer to assign variables to the LDRF, two new source-level keywords are introduced:

- *gregister* – a new storage class specifier that dictates to the compiler that the variable is to be placed in the LDRF.
- *gpointer* – a new type qualifier, which is used when creating a pointer to a variable that has been declared to reside in the LDRF.

To ensure program correctness, semantic checks have been implemented in the compiler; i.e., a pointer declared as a gpointer may only point to a variable that resides in the LDRF. Similarly, a variable that resides in the LDRF may only be pointed to, or aliased by, a pointer declared as a gpointer. To ensure that the programmer has not over-allocated the LDRF, the linker verifies that the size of the data allocated to the LDRF, across all files that comprise a program, does not exceed the size of the LDRF.

Several compiler optimizations have been implemented to improve the code that references LDRF registers. Initially, the compiler has one RWP assignment for each LDRF access. Two areas for improvement are to reduce the number of RWP assignments and to increase the number of registers that makeup a RWP. Code duplication transformations, such as loop unrolling, are one such means to increase the number of registers accessed within a window. More specifically, a loop that sequentially access one or more LDRF arrays within it will have accesses to sequential LDRF registers when unrolled. The

sequential LDRF access instructions may be coalesced, into a single instruction, to increase the number of registers that make up the RWP as well to reduce the code bloat endemic to loop unrolling. *Figure 3* demonstrates the intermediate (Register Transfer List [RTL]) code that is generated for a loop that adds one global array to another. Note that the loop has been unrolled two times. The global arrays, *a* and *b,* have been declared to reside in the LDRF. To represent a register in the LDRF, a G[*address*] notation is introduced; to represent a register window, a *w* register is introduced. Data movement to/from the LDRF is represented by a single RTL with two effects: the first indicates the starting position in the VRF and the position to which it is mapped in the LDRF (i.e., r[5]=G[r[11]]); the second indicates which RWP is under consideration and is assigned the number of registers in the window (i.e., w[0]=1). In *Figure 3*, the data movement is made more efficient by coalescing sequential LDRF accesses into a window spanning two registers.

```
   r[11]=a;
   r[13]=b;
L2:
   r[5..6]=G[r[11]..r[11]+4]];w[0]=2;
   r[7..8]=G[r[13]..r[13]+4];w[1]=2;
   r[5]=r[5]+r[7];
   r[6]=r[6]+r[8];
   G[r[11]..r[11]+4]=r[5];w[0]=2;
   r[11]=r[11]+8;
   r[13]=r[13]+8;
   r[1]=r[13]<r[4];
   PC=r[1]!r[0],L2;
```

Figure 3. RTL Code for a Loop that Adds Elements of Two Arrays

Preliminary results show that the LDRF is a promising area of research. Two metrics that can be used as an indicator of the usefulness of the LDRF – number of cycles and number of memory operations – show marked reductions within select benchmarks of the Mibench benchmark suite. Specifically, using cycle accurate simulation via SimpleScalar's sim-outorder, the number of cycles was reduced by an average of 10.11% and the number of memory operations was reduced by an average of 43.56%. A third metric – the number of bytes allocated to the LDRF – is useful to gain perspective on the amount of data that has been placed within the LDRF to achieve the associated benefits. With respect to the previously mentioned results, from 420 bytes to 40KB were allocated to the LDRF to achieve these reductions, with the majority requiring less than 2KB of LDRF storage. While these results are preliminary, they demonstrate the viability of a large data register file. Future work will identify and exploit opportunities to utilize windows spanning several registers; investigate the size of the LDRF needed to capture the majority of eligible variables; and broaden the types of variables that are eligible for promotion to the LDRF. Each of these aspects is expected to increase the benefits realized by the LDRF.

4 Conclusions

The large data register file (LDRF) is a viable alternative storage area for frequently referenced data. In particular, it provides a mechanism to store composite data structures, such as arrays and structs, in a storage area that provides register-like access times. Preliminary results suggest that the LDRF yields appreciable performance benefits. Without such a storage area, these data structures are relegated to the first-level data cache, which access via loads and stores and incur longer latencies even in the case of cache hits.

[1] D. Weaver and T. Germond, "The SPARC Architecture Manual", SPARC International, Inc., Menlo Park, CA. 1994

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

[3] R. Ravindran, R. Senger, E. Marsman, G. Dasika, M. Guthaus, S. Mahlke, and R. Brown. "Increasing the Number of Effective Registers in a Low-power Processor Using a Windowed Register File". In *Proceedings of the 2003 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, October 2003.