Global Data Access Optimization Via Load/Store Instruction Extension

Xian-hua LIU*, Peng YUAN, Ji-yu ZHANG and Zheng GAO
Microprocessor Research & Development Center, Peking University, China
{liuxianhua, yuanpeng, zhangjiyu, gaozheng}@mprc.pku.edu.cn

Keywords: Global Data Access, Load/Store, Instruction Extension, Compiler Optimization

Abstract. The widening gap between processor speed and memory latency makes memory accesses become a major performance bottleneck for modern processor architectures. Eliminating the memory access and improving the program locality may help to improve the performance. This paper proposes the extension design of load and store instructions, describes the hardware/software cooperative optimization idea for global data access. The research work is based on the synthetic analysis of program data access behavior and compiler optimization ability observation, which has a great influence on the design decision. Compared with the popular GP-addressing technique, our method may not only save the dedicated registers, but also eliminate the overhead of calling functions in dynamic shared objects. Experiment results show an average reduction of 5.92% in dynamic memory access instructions and an average improvement by 2.16% in code size for SPEC programs.

Introduction

Global data often lie far from the memory access instructions in executable files and dynamic shared objects. In RISC architectures, memory access instructions cannot contain a full absolute memory address because of the limited encoding fields. Instead, all memory accesses must be done by using an address offset and a base address held in a register, or by using memory address value loaded into a general-purpose register. The former needs the determination of the base register value. Array elements and structure members are often accessed with this method and compiler cannot determine their address values. The latter needs two instructions “load address” and “load value”, which is usually used to access global data (including .data and .bss). In this case addresses of global data are placed in a constant pool which located near the function. Figure 1 shows the example. It leads to poor use of split data and instruction cache and TLB.

Many methods have been developed to eliminate the address entries in the constant pool and the “load address” instructions. The most popular method is GP-addressing which is widely used in many architectures such as Alpha [8], PowerPC[9], IA64[10] and MIPS [11]. It uses a dedicated GP (Global Pointer) register in ABI specification. Also it introduces GP register save and restore overhead while calling functions. Some architectures design special instructions to construct a complete 32-bit address and thus eliminate the address entries in the constant pool. Program behavior analysis and compiler optimization is the key to efficiently. For example, initial IBM 360 instruction design [3] relied on common instruction sequences. MIPS designers suggested that proposed instructions should be useful to the compiler [4] because most programs are written with high level languages.

![Figure 1. Addresses in constant pool.](image)

In this paper, we describe the design and optimization of extended load/store instructions, reducing the overhead of global data accesses including both address entries in the constant pool and “load address” instructions. These new instructions could be easily extended from the existing load/store
instructions with small change in ISA design, which balances the performance and instruction encoding. The synthetic consideration of data access behavior and compiler optimization supports the availability of the new instructions and exploits their potential. Our method reaches the same effect to GP-addressing without a dedicated register and corresponding GP save/restore overhead. Moreover, it can be implemented in various computer systems because it doesn’t affect ABI.

Related Work

We reduce the overhead of global data accesses in both architecture and compiler aspects. Fred Chow et al. [12] considered the global data access issue in terms of memory addressing mode. For global data they present GP-addressing method (Figure 2 (A)). Compiler dedicates a register called GP which globally points to a memory area. Data in this area can be accessed by one memory instruction with GP as the base address. Data out of the area still needs two instructions to access. Compiler places variables in the GP memory area according to their sizes and access frequency, trying to utilize GP-addressing better. An issue with GP addressing is the overhead of context saving and restoring while calling functions in dynamic shared objects. Program and its libraries have different GP values because of different data sections. Without inter-procedural analysis, The GP save/restore strategy is conservative for compilers. Compiler will save and restore GP in the call site. Once GP is assigned in a computer system it becomes to an ABI specification and all software must conform to it.

Some architecture provides “load upper immediate” instructions such as LIS [9] in PowerPC and LUI [13] in MIPS, which load the upper half of a register with a 16-bit immediate value. These instructions facilitate the construction of a 32-bit address from two 16-bit immediate fields. Compiler without GP optimization can just generate two instructions to access global data without address entry. The method needs a 16-bit immediate field in the encoding field of both LUI and the “load value” instruction, and can cover all data in the 32-bit addressing space. Figure 2 (B) shows an example. Some architecture doesn’t dedicate a GP register because of limited registers. ARMCC compiler optimizes global data accesses by exposing common base addresses of different memory locations in a single function and then reduces the number of address entries in the constant pool as well as “load address” instructions. This method is also implemented in an early MIPS compiler for local variables accesses [12]. Without inter-procedural analysis compiler only limits this optimization method to variables in a single module. Figure 2 (C) demonstrates the method.

Figure 2. Different Data Access Methods.

The data size and GAT (global address table) sections are larger in 64-bit architecture. GP-addressing cannot guarantee a single instruction to access all global data in these situations. Srivastava et al. [5] exploited link-time optimization to reduce the “load address”. They found that many global variables nearby GAT can be accessed directly with GP. This method is more effective if the compiler segregates the small data into a single section. Haber et al. [6] introduces data reordering to eliminate “load address” operations and improve data cache performance. McIntosh et al. [7] used small variable packing to assist global data reordering optimization on Itanium processor.

The above work motivates our experiment in this paper. On one hand it is not necessary to explicitly construct or encode a 32-bit address in an instruction. On the other hand, GP-addressing
method needs a dedicated register to save/restore and is not easily adopted for other processors for history reason. These observations are especially obvious in embedded computer systems.

Proposed Design of Extended Memory Instructions

All the work in this paper is implemented and evaluated on the UniCore-2 RISC processor [14]. Simplescalar [2] toolset is used to investigate different performance data. GCC compiler [15] supports two-pass profile based optimization which can instrument the code to get the basic block execution counts. This can help to compute the data access frequency in the programs. The original load/store instructions in UniCore-2 processor support displacement addressing mode (Figure 3). It can be used in the GP-addressing method when Rn is GP register. This type of memory instructions is very common in RISC architecture such as MIPS, ARM and PowerPC.

![Figure 3. Encoding of Original Load/Store and Extended Instructions (the number shows the field length).](image)

The extended LDRL/STRL (load/store with long displacement) exploits the displacement addressing mode which uses PC as the default base, which is known as PC-relative addressing. It may save the 5-bit Rn field, expand the displacement field to 19 bits. Processor can distinguish the new and original instructions from the mode field. The original instructions can be easily extended to new ones in design phase. PC-relative addressing need not to save/restore context when calling functions.

Characteristics of Global Data Accesses

The displacement size and operand type are two key factors for displacement addressing mode [1]. The distance between the memory instruction and accessed data determines whether the displacement length is enough in the running program. The sum of text and data section sizes is the maximal distance value which must be covered by the displacement field. In fact, this distance is also affected by the linking model. In order to cover more distance, compiler can pack the global scalar data items together and place them beside the text section. Figure 4 gives the displacement field length requirement. “DP” line in the figure represents the requirement after global scalar data packing. This optimization reduces the length requirement of many programs by 1 to 2 bits compared with the length requirement of original programs ("original" line).

![Figure 4. Maximal displacement field length requirement in different cases.](image)
We evaluate the distribution of global scalar data accesses in different functions. Table 1 shows this result. Column 1 means the size percentage of functions which cover all the global scalar data accesses. Column 2 means the coverage of 99% of global scalar data accesses and column 3 means 95%. It is obvious that most global scalar data accesses are in a few percentages of functions. This motivates us to pack these hot functions together. From Figure 4 we can see that function packing reduces the displacement field length requirement obviously. A 20-bit displacement field can cover all the distances between the instruction and data in SPECint2000, and 19 bits are for 99% coverage and 18 bits for 95% coverage. After exploiting the optimizations of scalar data packing and hot function packing the proposed LDRL/STRL design can cover 99% of the global data accesses.

Table 1. Distribution of global scalar data accesses in functions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>cov-100%</th>
<th>cov-99%</th>
<th>cov-95%</th>
<th>Benchmark</th>
<th>cov-100%</th>
<th>cov-99%</th>
<th>cov-95%</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>44.33%</td>
<td>23.78%</td>
<td>15.98%</td>
<td>vortex</td>
<td>34.45%</td>
<td>7.03%</td>
<td>2.45%</td>
</tr>
<tr>
<td>gzip</td>
<td>4.01%</td>
<td>0.75%</td>
<td>0.53%</td>
<td>bzzip2</td>
<td>3.59%</td>
<td>1.73%</td>
<td>0.89%</td>
</tr>
<tr>
<td>mcf</td>
<td>0.16%</td>
<td>0.16%</td>
<td>0.16%</td>
<td>crafty</td>
<td>14.52%</td>
<td>4.13%</td>
<td>3.86%</td>
</tr>
<tr>
<td>parser</td>
<td>7.39%</td>
<td>0.63%</td>
<td>0.23%</td>
<td>perlbench</td>
<td>24.49%</td>
<td>7.60%</td>
<td>3.51%</td>
</tr>
<tr>
<td>twolf</td>
<td>21.13%</td>
<td>2.42%</td>
<td>1.68%</td>
<td>gap</td>
<td>9.62%</td>
<td>3.35%</td>
<td>2.19%</td>
</tr>
<tr>
<td>vpr</td>
<td>5.80%</td>
<td>1.03%</td>
<td>0.86%</td>
<td>eon</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

For integer programs in our experiment the common data types include 8-bit character, 16-bit half word and 32-bit word. Figure 5 shows the dynamic distribution of the sizes of global scalar data referenced from memory. This result helps to decide what types are most important. We find that word type data accounts for most global scalar data accesses. Since the word type data accesses are word aligned, we can further extend the representation range of the displacement field to 21 bits. We notice that the new instructions may cover all the SPECint2000 programs. Especially the scalar data packing, hot function packing and left-shifting are keys to the implementation.

Compiler Optimization of LDRL/STRL Instructions

Compiler bridges architecture and application, its optimization ability has great impact on instruction efficiency. The base rule for compiler optimization is to make the common cases fast and the rare case correct. In this section we describe the compiler optimization used to maximize the efficiency of LDRL/STRL instructions. These optimization methods guarantee not only the compilation correctness but the utmost utilization of the new instructions.

The most important issue for LDRL/STRL generation is that compiler must make sure the generated LDRL/STRL instructions can cover the accessed data. Otherwise the linker cannot fix the displacement field in the instructions. In fact, if the sum of text and data section sizes doesn’t exceed the instruction displacement range it is safe to generate LDRL/STRL instruction for all functions. But compiler still needs a careful work for robustness because it doesn’t know if the LDRL/STRL can cover all the data accesses in advance during compiling.

We solve it by profile based optimization. In the first phase we instrument the program to get the basic block execution counts. Compiler also gets the information about the text and data section sizes and each function size (Fsize) which determines the distance coverage ability and how to do hot function packing. In the second phase we compute data access frequency to assist the hot function.
candidate selection and generate LDRL/STRL instructions safely. Linker will finish the function packing and instruction displacement field fixing.

Function packing is only opened when the compiler knows the displacement range cannot cover all the distance between memory instructions and accessed data. In this case data access frequency is an important guide line when deciding which data benefits most from LDRL/STRL instructions.

Compiler visits each basic block in the function control flow graph to count the data access frequency and sum them for this function. Thus each function is associated with a total data access frequency \( (F_{\text{fa}}) \). We should try to make the sum of data access frequency maximal for functions in the displacement range. That is, the objective is to maximize \( \sum(F_{\text{fa}}) \) subject to \( \sum(F_{\text{size}}) < \text{displacement-range} \). This problem is similar to 0-1 knapsack. It is a NP problem and can be solved with dynamic programming algorithm. In the implementation compiler assigns each function candidate a separate ELF section and linker can pack them with the help of linker script.

Function packing decision is made in low level pass of the compiler. This is because that most memory instructions are exposed due to the finish of register allocation and other related optimizations such as PRE. Thus data access frequency are close to that in the final executable file.

After compiler knows which functions are placed in the displacement range it can generate LDRL/STRL instructions for global data accesses in these functions. Assembler associates each LDRL/STRL instruction with a relocation type and related symbol. And linker will resolve the relocation and fix the displacement field according the address of corresponding symbol.

**Experimental Evaluation**

SPECint2000 programs with train input are the experimental benchmark. We choose only integer programs because their scale is close to application on the UniCore-2 CPU and generally floating point co-processor is optional for middle-end and embedded processors. Their program behavior data guides the design and compiler optimization of the extended memory instructions in our work.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code size</th>
<th>DMIC</th>
<th>Benchmark</th>
<th>Code size</th>
<th>DMIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>3.24%</td>
<td>2.83%</td>
<td>vortex</td>
<td>3.39%</td>
<td>11.39%</td>
</tr>
<tr>
<td>gzip</td>
<td>1.93%</td>
<td>8.26%</td>
<td>bzip2</td>
<td>1.65%</td>
<td>3.01%</td>
</tr>
<tr>
<td>mcf</td>
<td>1.37%</td>
<td>0.03%</td>
<td>crafty</td>
<td>1.92%</td>
<td>0.89%</td>
</tr>
<tr>
<td>parser</td>
<td>1.62%</td>
<td>3.59%</td>
<td>perlbmk</td>
<td>3.47%</td>
<td>13.05%</td>
</tr>
<tr>
<td>twolf</td>
<td>3.31%</td>
<td>10.76%</td>
<td>gap</td>
<td>1.51%</td>
<td>4.06%</td>
</tr>
<tr>
<td>vpr</td>
<td>2.07%</td>
<td>12.89%</td>
<td>eon</td>
<td>0.42%</td>
<td>0.28%</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>2.16%</td>
<td>5.92%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* DMIC is the reduction of dynamic memory instruction counts.

In this section we present experimental results for the use of LDRL/STRL instructions with SimpleScalar toolset. ABI of UniCore-2 processor doesn’t assign a GP register. We evaluate the performance gain from two aspects. Table 2 shows the results. Column 2 gives the code size reduction due to the elimination of addresses in the const pool and corresponding "load address" instructions. The average reduction is 2.16%. Eon gets little reduction because there are few global scalar data accesses. Column 3 gives the reduction of dynamic memory access instruction counts due to the elimination of “load address” instructions. Perlbmk gets 13% reduction. Some frequently called functions such as regmatch() and Perl_pp_nextstate() access global scalar data heavily. Twolf, vpr and vortex also have more than 10% of memory instruction count reduction. Mcf is a structure intensive program where most data members are accessed with the register holding the structure beginning of structure as the base address. So the LDRL/STRL instructions contribute little.

For the benchmark programs the method eliminates all the address entries in the constant pool and the “load address” instructions, the same effect to GP-addressing. Our method takes PC as the base address in the data accesses and saves an available register for allocation. Thus there is no the overhead of GP save and restore when executing function calls.
Conclusions

In this work we focused on the design of extended load/store instructions LDRL/STRL for global data accesses from the view of program behavior characteristics and compiler optimization ability. We first analyze the global data accesses in the programs and load/store instructions in the processor, which motivates us to extend the current instructions to new ones. Scalar data packing and hot function packing further support the availability of the design and the correctness of compilation.

The displacement length requirement analysis based on these optimizations shows that even 19-bit length can cover 99% global scalar data accesses and thus a 17-bit field is enough. This analysis also helps to optimize code in the processor with GP. For load/store instructions with more displacement field bits, compiler can utilize this result to optimize code within the displacement range and doesn’t generate GP-addressing code, reducing the overhead of saving/restoring GP.

Our method can also enhance the interaction with other compiler optimizations. The scalar data packing can utilize layout optimization to reduce data cache misses and page faults. And function packing can use reordering optimization to improve instruction cache behavior [16-17]. These optimization interactions may further increase the program performance.

Acknowledgement

We would like to thank the anonymous reviewers. This research was financially supported by the National Science Foundation of China, under grant No.6130004.

References


