Register Reassignment for Mixed-width ISAs
is an NP-Complete Problem

Bor-Yeh Shen, Wei Chung Hsu, and Wuu Yang
Institute of Computer Science and Engineering,
National Chiao Tung University, Taiwan, R.O.C.
{byshen, hsu, wuuyang}@cs.nctu.edu.tw

ABSTRACT

Code size is an important issue in many embedded systems. In order to reduce code size, newer embedded RISC processors employ a mixed-width instruction set, where processor architectures support interleaved execution between normal (usually 32-bit) and narrow (usually 16-bit) instructions without explicit mode switch. However, because of the restriction of the encoding length, narrow instructions can only access a limited set of registers. Therefore, for a mixed-width instruction set, proper register allocation can reduce code size. One approach is to re-assign the registers after traditional register allocation. In this paper, we prove that this register reassignment problem is NP-complete by showing that the 0-1 knapsack problem is a special case of this problem. We also propose a method for register reassignment for a mixed-width instruction set with the main goal of code size reduction.

1 keywords: Mixed-width ISA, Code Size Reduction, Register Reassignment, Thumb-2, Knapsack Problem, NP-complete.

1. INTRODUCTION

Reducing code size is very important in embedded systems since embedded environments are highly constrained in memory size. Besides, improving code density can reduce not only network traffic but also embedded system costs such as memory requirements and power consumption [1, 2]. In order to improve code density, many processors are equipped with a narrow instruction set (usually 16-bit long) in addition to a normal instruction set (usually 32-bit long). One of the most popular mixed-width instruction set is Thumb-2 [3], which can freely mix 32-bit and 16-bit instructions. Other processors, such as ARCompact [4], microMIPS [5], and AndeStar [6], have similar features. However, since a narrow instruction has fewer bits than a normal instruction, a narrow instruction is usually restricted to use smaller opcode fields, implicit operands, a limited set of registers, or a smaller range of immediate values. For example, register fields are normally 3 bits long in 16-bit Thumb-2 instructions, whereas in 32-bit Thumb-2 instructions, register fields are normally 4 bits long. Smaller opcode fields mean a narrow instruction set can only support a subset of normal opcodes. Limited registers may cause higher register pressures in the narrow instruction set.

In this paper, we will consider an ARM-like processor architecture. ARM provides a mixed-width instruction set, Thumb-2, which includes both 16-bit and 32-bit instructions. Normally, a 16-bit instruction can access only registers r0 through r7 while a 32-bit instruction can access all registers (i.e., r0 through r15). An instruction may access 0, 1, 2, or 3 registers. For example, Figure 1 represents two encoding methods, 16-bit and 32-bit, of the adds instruction in Thumb-2. The encoding length of register Rm, Rn, and Rd are 3-bit and 4-bit in the 16-bit and 32-bit instruction respectively. Figure 2 shows an example code fragment for Thumb-2 mixed-width instruction set. For the same opcode, say adds, if registers r0 and r1 are used, it can be encoded as a 16-bit instruction. On the other hand, if register r8 is used instead, the instruction has to be encoded as a 32-bit instruction. Similar situations occur for the subs opcode. Of course, there are opcodes, say bl (branch with link), bfc (bit field clear), clz (count leading zeros), mls (multiply and subtract), rbit (reverse bits), and tbb (table branch byte) in Thumb-2, which have only the 32-bit format. We will consider only opcodes with both formats.

Our approach is to permute the registers in an assembly program so that as many instructions can be encoded in the 16-bit format as possible. It turns out that this register reassignment optimization problem is NP-complete because the 0-1 knapsack problem is a special case of this problem. We also propose an algorithm for the reassignment problem, whose time complexity is \(O\left(\frac{n^2}{m}\right)\) and space complexity is \(O(n^3)\).
The remainder of this paper is organized as follows: in Section 2 we will define the register reassignment problem formally and prove the NP-Completeness of this problem. Section 3 presents an algorithm, together with an example and time complexity analysis. Section 4 summarizes the related work and compares the differences between our work and previous research. Section 5 concludes this paper.

2. REGISTER REASSIGNMENT PROBLEM (RRP)

Mathematical Formulation

The register reassignment problem can be defined formally as follows:

**Instance:** Let $G$ be a particular input assembly program. Given an integer $n$, a register set for normal instructions $R = \{r_0, r_1, \ldots, r_{k-1}\}$, a register set for narrow instructions $Q = \{q_0, q_1, \ldots, q_{m-1}\}$, where $Q \subseteq R$, $k, m \in \mathbb{N}$, and $m \leq k$. $n$ is the maximal number of register fields in a normal instruction; $R$ is the set of all the registers; and $Q$ is the set of all the registers that can be used in narrow instructions. Let $S = \{P|P \subseteq R$ and $0 < |P| \leq n\}$. For each $P \in S$, define $cnt(P)$ as the number of instructions in $G$ that can be encoded as narrow instructions when all registers in $P$, but not any proper subset of $P$, are re-assigned to registers in $Q$.

**Question:** Select a subset $R' \subseteq R$ such that

\[
\text{maximize } \sum_{P \subseteq R'} cnt(P)
\]

subject to $|R'| \leq |Q|$

Reassignment is a permutation of the registers. We do not allow two different registers to be re-assigned to the same register. In the above formulation, $n$, $R$, and $Q$ are determined from the underlying processor’s instruction set architecture. $cnt(P)$’s are determined from the particular input program $G$ as well as the underlying processor architecture.

NP-Completeness of the RRP

The register reassignment problem in Mixed-width ISAs is a combinatorial optimization problem. We can reduce the 0-1 knapsack problem to it by reformulating the above problem as follows:

**Question:** Select a subset $S' \subseteq S$ such that

\[
\text{maximize } \sum_{P \in S'} cnt(P)
\]

subject to $|\bigcup_{P \in S'} P| \leq |Q|

If $P_1, \ldots, P_{|S|} \in S$ in the above problem formulation are pairwise disjoint (i.e., for any $P_i, P_j \in S, i \neq j, P_i \cap P_j = \phi$), the problem can be re-stated as follows:

**Question:** Select a subset $S' \subseteq S$ such that

\[
\text{maximize } \sum_{P \in S'} cnt(P)
\]

subject to $\sum_{P \in S'} |P| \leq |Q|

For each $P \in S$ and $G$, $|P|$ and $cnt(P)$ are constants. Therefore, the 0-1 knapsack problem is a special case of this register reassignment problem. It has already been shown that the decision version of the 0-1 knapsack problem is NP-complete [8]. We may conclude...
1. begin main
2. selectedRegister := \{ \phi \};
3. tempStack := [ ]; /* an empty stack initially */
4. maxInstructionCount := 0;
5. selectNext(0, |\mathcal{R}| - 1, 0);
6. print selectedRegister, maxInstructionCount;
7. end main

Figure 3: The main procedure.

1. procedure selectNext(min, max, numberOfSelectedRegister)
2. local var instructionCount := 0;
3. if numberOfSelectedRegister = |\mathcal{Q}| then begin
4. for each \( P_i \in \mathcal{S} \) and \( P_i \subseteq \text{tempStack} \) do begin
5. instructionCount := instructionCount + cnt(\( P_i \));
6. end for
7. if maxInstructionCount < instructionCount then begin
8. maxInstructionCount := instructionCount;
9. selectedRegister := tempStack;
10. /* move elements of tempStack to selectedRegister. */
11. end if
12. pop tempStack;
13. return
14. end if
15. for each register \( r_i \in \{ r_{\text{min}}, r_{\text{min+1}}, ..., r_{\text{max}} \} \) do begin
16. push \( r_i \) to tempStack;
17. selectNext(i + 1, max, numberOfSelectedRegister + 1);
18. end for
19. pop tempStack;
20. end procedure selectNext

Figure 4: The selectNext procedure.

1. procedure cnt(registerSet)
2. local var instructionCount := 0;
3. for each instruction \( i \) in the input program do begin
4. if registerSet = registers used in instruction \( i \) then
5. instructionCount := instructionCount + 1;
6. end for
7. return instructionCount;
8. end procedure cnt

Figure 5: The cnt procedure.

| Table 1: An Example of the cnt Table and the selectNext Procedure. |
| --- | --- | --- | --- | --- |
| (a) The cnt Table. | (b) Step-by-step of the selectNext Procedure. |
| registerSet | cnt(registerSet) | selectNext | tempStack | selectedRegister | maxInstCount |
| \{r_0\} | 3 | (0, 3, 0) | [ ] | \{ \phi \} | 0 |
| \{r_1\} | 5 | (1, 3, 1) | \{r_0\} | \{ \phi \} | 0 |
| \{r_2\} | 7 | (2, 3, 2) | \{r_0, r_1\} | \{r_0, r_1\} | 12 |
| \{r_3\} | 0 | (3, 3, 2) | \{r_0, r_2\} | \{r_0, r_2\} | 22 |
| \{r_0, r_1\} | 4 | (4, 3, 2) | \{r_0, r_3\} | \{r_0, r_2\} | 22 |
| \{r_0, r_2\} | 12 | (2, 3, 1) | \{r_1\} | \{r_0, r_2\} | 22 |
| \{r_0, r_3\} | 10 | (3, 3, 1) | \{r_1, r_2\} | \{r_0, r_2\} | 22 |
| \{r_1, r_2\} | 24 | (4, 3, 2) | \{r_1, r_3\} | \{r_1, r_2\} | 22 |
| \{r_1, r_3\} | 15 | (3, 3, 1) | \{r_2\} | \{r_1, r_2\} | 22 |
| \{r_2, r_3\} | 8 | (4, 3, 2) | \{r_2, r_3\} | \{r_1, r_2\} | 22 |
that the register reassignment decision problem is also NP-complete. (It should be obvious that the decision version of the register reassignment problem belongs to the class of NP.)

3. OUR ALGORITHM

A Straightforward RRP Algorithm

Our algorithm is shown in Figures 3 and 4. During each invocation of the selectNext procedure, we select a register and add it to tempStack until the number of registers in tempStack (i.e., the current selection) is equal to |Q|. Then we invoke the cnt procedure, shown in Figure 5, for each \( P_i \subseteq \text{tempStack} \) to compute the total instruction count \( \text{instructionCount} \). (The cnt procedure can be implemented as a table that is computed from \( G \) in advance.) After we compute \( \text{instructionCount} \), \( \text{maxInstructionCount} \) (the current maximal instruction count) and selectedRegister (the set of the current selection) will be updated if necessary.

Example

Table 1 (a) shows an example of the cnt table computed from an input program. Suppose that \( R \) and \( Q \) are equal to \( \{r_0, r_1, r_2, r_3\} \) and \( \{r_0, r_1\} \) respectively. The main procedure will invoke selectNext(0, 3, 0) to find which registers need to be re-assigned to the registers in \( Q \). Table 1 (b) shows a step-by-step example of the selectNext procedure. The final result shows that we can encode 36 instructions as narrow instructions if we re-assign \( \{r_1, r_2\} \) to \( \{r_0, r_1\} \).

Register Permutation

After the selected registers are obtained from our RRP algorithm, a simple register permutation will be applied to the original input program. Figure 6 shows an example of the register permutation. Suppose that \( R = \{R_0, \ldots, R_7\} \), \( Q = \{R_0, \ldots, R_3\} \), and we obtain \( \{R_3, R_4, R_5, R_6\} \) (i.e., \( R' \)) from the RRP algorithm. We can easily construct a permutation table of \( R \) by combining two bijective functions: \( \{R_3, R_4, R_5, R_6\} \rightarrow \{R_0, R_1, R_2, R_3\} \) and \( \{R_0, R_1, R_2, R_7\} \rightarrow \{R_4, R_5, R_6, R_7\} \). Finally, our assembly rewriter will re-assign registers in original input program based on the permutation table.

Complexity Analysis

The depth of the recursive calls of the selectNext procedure is at most \(|Q|\). Let \( n = |Q| \). We assume \(|R| = 2n\). Therefore, the total number of invocations of the selectNext procedure is at most \( nC_n^{|R|} \), which is \( nC_n^{2n} = n！Π_{k=1}^{2n} = 2n-k+1 \). We assume that the maximal number of register fields in instructions is equal to 3. Thus, the number of iterations of the first for loop in the selectNext procedure is at most \( \sum_{k=1}^{3} C_k^{|R|} \), which is \( O(n^3) \). The time complexity of our RRP algorithm is \((n + n^3)Π_{k=1}^{2n-k+1} = O(n^3)\). The space complexity of our RRP algorithm is at most \(|Q| + \sum_{k=1}^{|R|} C_k^{|R|} \), which is \( O(n^3) \), since we need to maintain a selected register set and an instruction count table (for storing the results of the cnt procedure) for each \( P_i \in S \). After we obtain the result from the RRP algorithm, the register permutation and reassignment of the input program can be done in linear time.

4. RELATED WORK

Petrov and Orailoglu [9, 10] used a compiler-driven register name adjustment algorithm to reduce the power consumption of instruction fetch and register file access. They proved the register name adjustment (RNA) problem is NP-complete by mapping the travelling salesman problem (TSP) to an instance of the RNA problem. However, the RNA problem and the RRP problem are totally different since they focus on different issues. The difference between the RNA problem and our RRP problem is that the RNA looks for the most frequent register pairs in adjacent instructions and re-assigns their names to minimize the Hamming distance between them, whereas our RRP problem re-assigns register names to maximize the numbers of instructions that can be encoded as narrow instructions.

Ros and Sutton [11, 12] described a post-compilation reassignment technique based on Hamming distance to improve code compression. Their research also looks for register-pair frequencies. Instead of reducing the Hamming distance of registers in adjacent instructions, they focused on reducing the Hamming distance of registers in isomorphic instructions (i.e., non-register fields of these instructions are identical) and used a dictionary based encoding scheme to improve code density. In their research, an additional hardware decoder is required to decode the compressed instructions on-the-fly and feed them to the CPU.
5. CONCLUSIONS

Since code size reduction is important in embedded systems, it is useful to establish a lower bound of the code size for a mixed-width instruction set. This paper formalized the register reassignment problem and proposed an algorithm for it. Our algorithm is based on zero knowledge of the compiler, which generates the assembly program, and performs a post-pass register reassignment to improve code density. A straightforward extension of our algorithm may handle pre-assigned registers. A register is pre-assigned before the reassignment if it must be assigned to a certain physical register due to an architectural reason or restrictions of application binary interfaces (ABI), such as calling conventions. Since the time complexity of our algorithm is quite high and there will be more constraints on real applications, it would be interesting to investigate efficient approximation algorithms for the register reassignment problem in the future.

References


