Investigation on the Optimization for Storage Space in Register-Spilling

Guohui Li, Yonghua Hu(✉), Yaqiong Qiu, and Wenti Huang

School of Computer Science and Engineering,
Hunan University of Science and Technology, Xiangtan 411201, China
hyhyt@126.com

Abstract. In order to make full use of the memory resources of computers, especially embedded systems, the multiplexing of storage space in register spilling is investigated and the corresponding method is presented in this paper. This method is based on the graph coloring register allocation method and on the basic principle of greedy algorithm. In this method, the register allocation candidates to be spilled, which do not conflict with each other, will be spilled to the same memory unit. Thus, in register spilling, less memory is needed and more load/store instructions using immediate values can be used. The effectiveness of the method is verified. Besides, the method is suitable for architectures with both scalar and vector operands.

Keywords: Register allocation · Spilling · Storage space · Compiling optimization

1 Introduction

In compiling, register allocation is an important optimization technology. The main conventional methods of register allocation can be classified into two main classes: linear scan register allocation [1] and graph-coloring [2]. There are also some other register allocation methods such as studying optimal spilling in the light of SSA [3], heuristic allocation method [4], allocation of repair strategies [5], layered allocation [6], etc. The graph-coloring method, which is a highly effective global register allocation method [7, 8], was proposed by Chaitin in 1981. The coloring process must ensure that the adjacent nodes have different colors. The number of colors that an interference graph needs in coloring is called as its register pressure, and the treatment modifying code in order to make the graph colorable is called "reducing register pressure".

Spilling symbolic registers [9–12] is a method for reducing register pressure. Its basic idea is to split the lifetime of a register allocation candidate into two or more lifetimes, i.e., dividing the live range of a symbolic register into two or more parts. Physical registers will get the gap time between live ranges of divided candidates. This process is helpful to reduce register pressure [13, 14]. The basic method of register-spilling is to spill candidates to memory. In conventional register-spilling algorithm [15, 16], each pending spilling candidate needs a storage unit, indicating that the storage space needed is directly proportional to the number of candidates to be spilled. This paper is devoted to the study of reducing the storage space for spilling in register allocation through storage units multiplexing. An algorithm that is based on the graph coloring register allocation method will be presented to realize this optimization. Not only will this algorithm not impact the register spilling result, but also it can greatly reduce the storage units needed in register spilling and thus improve the utilization of memory.

2 Optimization Algorithm

In order to optimize the storing of register allocation candidates (called webs in what follows) to be spilled, we define the following quantities to analyze the code:

(1) *NDimMtx* is an N-dimensional matrix template class, where *N* can be 1 or 2 for the problem considered in this paper.
(2) *AdjacentListRecord* is a data structure containing the needed information of an adjacent list record.
(3) *ApInstn* is code instruction object.
(4) *def* and *use* are definition object and use object, respectively.
(5) *DU* and *UD* are definition-use-chain and use-definition-chain, respectively.
(6) *spillWebs* is a set of webs to spilled (a web is made up of DUs), and *spillWebList* is the corresponding list form of *spillWebs*.
(7) *spillWebAdjList* is the adjacency list for the webs to be spilled.

The top-level process of our optimization method has the following main steps:

(1) First *Judge_NeedRegistersSpilling()* determines whether the register allocation needs spilling. If yes, the following steps should be executed.
(2) Next, *Find_SpillWebs()* function finds out all the webs to be spilled, and generates *spillWebs* and *spillWebList*.
(3) Next, *Get_SpillWebsConflictShip()* function analyzes the active ranges of the webs in *spillWebs*, judges whether they are overlapping.
(4) Then, *Assign_StorageSpaceToSpillWebs()* assigns storage units for each elements of *spillWebs*: it assigns the same storage spaces to the webs that don't conflict with each other, and assigns different storage spaces to those conflict with each other.
(5) At last, *Gen_SpillCode()* generates spilling and restore codes.

At the beginning of the optimization, the algorithm set the initial offset of the available storage space for spilling to *baseOffset*, a global static variable. Then, for each following register spilling process, the value of *baseOffset* will be equal to the address of the maximum offset determined by its previous register spilling process. The algorithm of the top-level structure of our method is as follows:

```
   Procedure Optimise_SpillStorageSpace(spillWebs,
spillWebList, baseOffset, offset)
     spillWebList: a list of spillWebs
     baseOffset: global static variable
     begin
       baseOffset := available storage space start offset
       spill: Boolean
      spill := Judge_NeedRegistersSpilling( )
        if spill then
          Find_SpillWebs( )
          Get_SpillWebsConflictShip( )
          Assign_StorgeSpaceToSpillWebs()
          Gen_SpillCode( )
        fi
end  || Optimise_SpillStorageSpce
```

We assume that optimistic heuristic method is used to prune the interference graph. To describe whether a node in an interference graph result should be spilled, we set a Boolean value for each node as a flag to mark this status. We traverse the webs whose spilling flags are true and put them into *spillWebs* and *spillWebList*. As for the conflict relation among the webs to be spilled, it can be obtained directly from the existing adjacent matrix and adjacent list and is stored in the *spillWebAdjList* object corresponding to *spillWebs*.

In the process of register spilling, the address of the storage units assigned to the webs in *spillWebList* increases from the base address for spilling. For each web *w*, the algorithm traverses the storage units starting from the base address until it finds a unit whose corresponding web is not conflict with *w*. When the algorithm finds out such a storage unit, it assigns this storage unit to the web *w* and ends the traversing process. It should be noted that some architectures have both scalar and vector operands, and hence there will be scalar and vector webs in *spillWebList*. In our algorithm, we assume that the increment of traversing storage units for a scalar web is *a*, while that for a vector web is *m*a*, both *m* and *a* being integer.

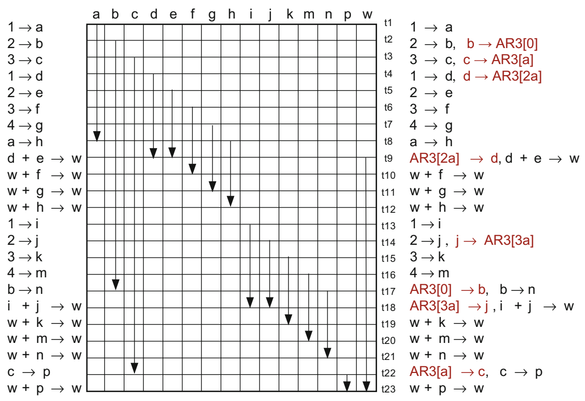
The corresponding algorithm is as follows:

```
   procedure Assign_StorgeSpaceToSpillWebs(spillWebList)
   begin
    i, j, minUnusedOffset: integer
    a: size of a storage unit
    oneSpillWeb: spillWeb
    offsetUseByNeighbors: map<uint, bool>
    pInstn: instruction
    map<uint, bool>::iterator it
    for i := 1 to n do
      for (it =
spillWebAdjList[spillWebList[i]].adjNodes.begin();
         it !=
spillWebAdjList[spillWebList[i]].adjNodes.end();
         ++ it) do
        oneSpillWeb := (*it).first
        for
(spillWebAdjList[oneSpillWeb].storageUnits.ToHead();
           spillWebAdjList[oneSpillWeb].
storageUnits.CurNotNull()
           spillWebAdjList[oneSpillWeb].
storageUnits.ToNext()) do
           offsetUseByNeighbors[spillWebAdjList[oneSpill
Web].storageUnits.Cur()]=true
        od
      od
      minUnusedOffset := baseOffset
      while(1) do
          if
(offsetUseByNeighbors.count(offsetUseByNeighbors) < 1)
           spillWebAdjList[spillWebList[i]].storageUnits
.AddTail(minUnusedOffset)
           break
          fi
          if (pInstn is scalar)
           minUnusedOffset = minUnusedOffset + a
          else (pInstn is vector)
           minUnusedOffset = minUnusedOffset + m*a
          fi
      od
    od
    baseOffset := max(minUnusedOffset)
   end || Assign_StorgeSpaceToSpillWebs
```

After all the webs in *spillWebs* being assigned storage units, the algorithm inserts corresponding spills and restores for them. This treatment is the same as that used in conventional graph coloring method, so we don't repeat it in this paper.

3 Verification

To show the effectiveness of our method, we demonstrate the using of the method in an example (see the left side of Fig. 1) by comparing with the conventional storage assigning method. We assume that the aim architecture has 4 general purpose registers, i.e., *R0, R1, R2*, and *R3*, and has a register *AR3* as the base address register for register spilling. In Fig. 1, the arrows are live ranges corresponding to the variables, and *a* stands for the size of a storage unit. The result intermediate codes after the first register spilling pass are shown in the right part of Fig. 1.



```
1 → a           t1   1 → a
2 → b           t2   2 → b,  b → AR3[0]
3 → c           t3   3 → c,  c → AR3[a]
1 → d           t4   1 → d,  d → AR3[2a]
2 → e           t5   2 → e
3 → f           t6   3 → f
4 → g           t7   4 → g
a → h           t8   a → h
d + e → w       t9   AR3[2a] → d, d + e → w
w + f → w       t10  w + f → w
w + g → w       t11  w + g → w
w + h → w       t12  w + h → w
1 → i           t13  1 → i
2 → j           t14  2 → j , j → AR3[3a]
3 → k           t15  3 → k
4 → m           t16  4 → m
b → n           t17  AR3[0] → b,  b → n
i + j → w       t18  AR3[3a] → j , i + j → w
w + k → w       t19  w + k → w
w + m → w       t20  w + m → w
w + n → w       t21  w + n → w
c → p           t22  AR3[a] → c,  c → p
w + p → w       t23  w + p → w
```

Fig. 1. Schematic diagram of the first pass of register spilling for source codes.

The following physical register assigning process will find out that the assignment is not successful. Then a second register allocation pass with register spilling is needed, where *f* and *m* will be further spilled. The final allocation result is shown in the right part of Fig. 2. As a comparison, the allocation result based on the conventional storage unit assignment method for webs to be spilled is shown in the left part of Fig. 2.

According to this example, it is easy to see from Fig. 2 that 4 storage units is needed by our method but 6 storage units is needed by the conventional method. Of Course, it should be noted that the optimization effect of our method will be different for different codes because it depends on the conflict relations among register allocation candidates.

1 → a	t1	1 → a
2 → b	t2	2 → b
b → AR3[0]	t3	b → AR3[2a]
3 → c	t4	3 → c
c → AR3[a]	t5	c → AR3[a]
1 → d	t6	1 → d
d → AR3[2a]	t7	d → AR3[0]
2 → e	t8	2 → e
3 → f	t9	3 → f
f → AR3[4a]	t10	f → AR3[3a]
4 → g	t11	4 → g
a → h	t12	a → h
AR3[2a] → d	t13	AR3[0] → d
d + e → w	t14	d + e → w
AR3[4a] → f	t15	AR3[3a] → f
w + f → w	t16	w + f → w
w + g → w	t17	w + g → w
w + h → w	t18	w + h → w
1→ i	t19	1 → i
2→ j	t20	2 → j
j → AR3[3a]	t21	j → AR3[0]
3 → k	t22	3 → k
4 → m	t23	4 → m
m → AR3[5a]	t24	m → AR3[3a]
AR3[0] → b	t25	AR3[2a] → b
b → n	t26	b → n
AR3[3a] → j	t27	AR3[0] → j
i + j → w	t28	i + j → w
w + k → w	t29	w + k → w
AR3[5a] → m	t30	AR3[3a] → m
w + m → w	t31	w + m → w
w + n → w	t32	w + n → w
AR3[a] → c	t33	AR3[a] → c
c → p	t34	c → p
w + p → w	t35	w + p → w

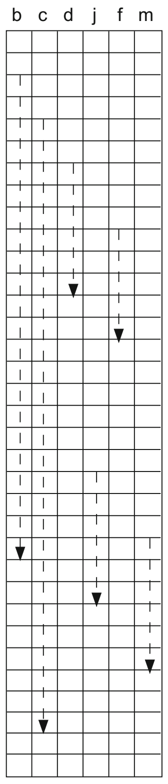


Fig. 2. Comparison diagram for the result code that the spilled data storing optimization is used and that the spilled data storing optimization is not used.

4 Conclusion

In this paper, an optimization method of storage space optimization in register spilling is presented. This method is based on graph coloring register allocation method and consists of several main steps, such as identifying the register allocation candidates to be spilled, analyzing the conflict relation among these candidates, assigning storage units to these candidates, etc. It is demonstrated by an example that the method can reduce the storage units needed in register spilling, and that the correctness of graph coloring register allocation will not be affected. Consequently, more load/store instructions that directly access memory can be used in register spilling, and thus reducing the pressure on offset registers.

Acknowledgment. This work was supported by National Natural Science Foundation of China (Grant No. 61308001) and graduate student innovation fund project (Grant Nos.S140027 and CX2015B537).

References

1. Poletto, M.: Linear scan register allocation. ACM Trans. Program. Lang. Syst. **21**, 895–913 (1999)
2. Briggs, P., Cooper, K., Kennedy, K., Torczon, L.: Coloring heuristics for register allocation. ACM SIGPLAN Not. **39**, 275–284 (1989)
3. Colombet, Q., Brandner, F., Darte, A.: Studying optimal spilling in the light of SSA. In: 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems, Taipei, pp. 25–34 (2011)
4. Tavares, A., Colombet, Q., Bigonha, M.: Decoupled graph-coloring register allocation with hierarchical aliasing. In: Proceedings of the 14th International Workshop on Software and Compilers for Embredded Systems, Goar, Germany, pp. 1–10 (2011)
5. Colombet, Q., Boissinot, B., Brisk, P.: Graph-coloring and tree scan register allocation using repairing. In: 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems, Taipei, pp. 45–54 (2011)
6. Diouf, B., Cohen, A., Rastello, F.: A polynomial spilling heuristic: layered allocation. R. Research Report, Project-Teams Parkas and Compsys (2012)
7. Chaitin, G., Auslander, M., Chandra, A.: Register allocation via coloring. J. Comput. Lang. **6**, 47–57 (1981)
8. Carole, D.-G., Hugues, F., Eli, G., Leslie, L.: Adaptive register allocation with a linear number of registers. In: 27th International Symposium, DISC 2013, Jerusalem, Israel, 14–18 October (2013)
9. Steven, S.: Advanced Compiler Design and Implementation. Elsevier Science, Amsterdam (1997). M. USA
10. Salgado, M., Ragel, R.G.: Register spilling for specific application domains in ASIPs. In: 7th International Conference on Information and Automation for Sustainability. IEEE (2014)
11. Wu, C., Lu, C., Lee, J.: Register spilling via transformed interference equations for PAC DSP architecture. Concurrency Comput. Pract. Experience **26**, 779–799 (2014)
12. Pfenning, F., Simmons, R.: Lecture Notes on Register Allocation Optimizations (2015). http://www.cs.cmu.edu/~rjsimmon/15411-f15/lec/17-regopt.pdf
13. Yin, M., Steve, C., Rong, G.: Low-cost register-pressure prediction for scalar replacement using pseudo-schedules. In: 2004 International Conference on Parallel Processing, 0190–3918/04 (2004)
14. Shobaki, G., Shawabkeh, M., Rmaileh, N.: Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. ACM Trans. Archit. Code Optim. **10**, 14 (2013)
15. Philipp. K., Frankfurt, M.: Bytewise register allocation. In: 18th International Workshop on Software and Compilers for Embedded Systems, New York (2015). 978-1-4503-3593-5
16. Gaow, Z., Han, L., Pang, J.: Research on SIMD auto-vectorization compiling optimization. J. Softw. **26**, 1265–1284 (2015)