ISSN 0361-7688, Programming and Computer Software, 2010, Vol. 36, No. 6, pp. 363–367. © Pleiades Publishing, Ltd., 2010.
Original Russian Text © D.S. Ivanov, 2010, published in Programmirovanie, 2010, Vol. 36, No. 6.

# Register Allocation with Instruction Scheduling for VLIW-Architectures

**D. S. Ivanov**

*Moscow Institute of Physics and Technology, Institutskii per. 9, Dolgoprudnyi, Moscow oblast, 141700 Russia*
*e-mail: dimanovs@gmail.com*

Received March 9, 2010

**Abstract**—Interaction between the phases of register allocation and instruction scheduling are often considered in publications devoted to optimizations for the final stage of compilation. Typically, it is proposed to adapt one of the phase for needs of another without their combination into a single unit. However, their integration can essentially reduce the time of operation and enhance the performance of the resulting code. This study describes an attempt to combine these phases as completely as possible with account for the features of static scheduling for VLIW-architectures.

**DOI:** 10.1134/S0361768810060058

## 1. INTRODUCTION

Currently, one of the most important problems in computer science is the low efficiency in the interaction between memory and microprocessor. Starting from 1980, the gap between speeds of memory channels and microprocessors has been steadily increasing [1]. Efficient allocation of registers in an optimizing compiler makes it possible to overcome this bottleneck by minimizing the number of memory access instructions.

## 2. EARLIER STUDIES

The conventional schemes based on the classical algorithm of interference graph coloring [2] disregard the link between the register allocation and instruction scheduling.

The most reasonable approach to integrate instruction scheduling with register allocation is the unified representation for resource allocation based on the parallel dependence graph (PDG) [3] with the use of the RAP—a register allocator that allocates registers in a hierarchical manner [4]. This mechanism has a number of advantages but requires the construction of the PDG, has a limited visibility range (within a region), and uses an additional pass for moving the memory access instructions.

In [5], the combined scheme is considered only within a basic block. In [6], it is proposed to allocate registers before scheduling and add into the interference graph arcs representing possible parallelism. This approach makes it possible to take into account to a certain degree the fact that the scheduling and register allocation phases affect each other; however, the relationship between the effect of the arcs reflecting the interference of variables and the effect of the arcs reflecting the parallelism remains unclear.

In recent studies like [7–9], the links between register allocation and instruction scheduling received little attention. In [9], the authors propose a register allocation algorithm with a scheduled code based on the principle of augmenting mosaic puzzles ignoring the influence of the scheduling phase on register allocation. In [7], an iterative allocator is described in which each iteration makes it possible to obtain a better allocation of registers in terms of the resulting code size. This algorithm also ignores the link between instruction scheduling and register allocation, and its performance is typically lower than the performance of the allocator based on interference graph coloring. In [8], a technique for reducing the register pressure before scheduling is proposed.

The neglect of the link between scheduling and register allocation is especially noticeable in architectures with a large instruction word. Register allocation before static instruction scheduling imposes strict limitations on the code parallelism, and the scheme with scheduling before allocation [10] leads to the insertion of new large instructions into the code if there is a deficit in registers.

The scheme with simultaneous instruction scheduling and register allocation has not been adequately investigated. This scheme would make it possible to better account of the mutual requirements of scheduling and allocation; however, although there has been continued research conducted in this field, no commonly accepted algorithms with reasonable complexity have been found.

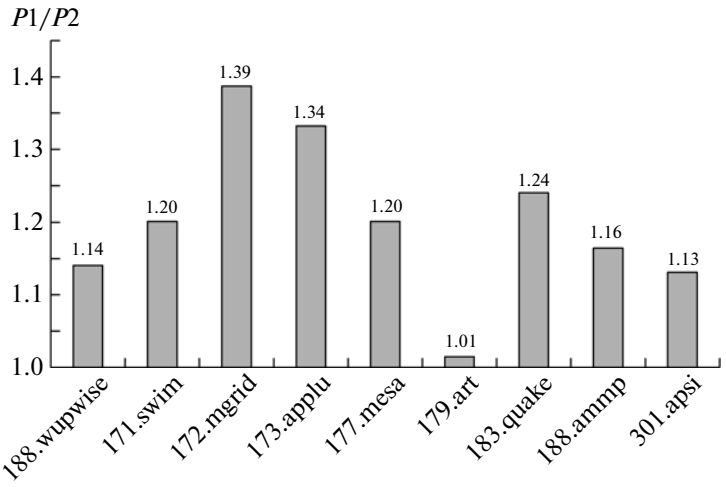


**Fig. 1.** Relative performance of the old and new mechanisms.

## 3. GENERAL DESCRIPTION OF THE ALGORITHM

Hereafter, we use the term web for a set of nodes and arcs of a procedure control flow graph [11] whose execution requires the intermediate values of a variable to be stored. In this case, the variable is said to live on the given set of arcs and nodes. Each web corresponds to a set of instructions that write or read the value of a variable. The first type of instructions (writing) is called the web definition, and the second type (reading) is called the web use.

If a variable lives within a single basic block, the corresponding set is called a local web; otherwise, it is a global web. Upon the allocation of registers, each variable must correspond to a single architectural register. It is difficult to support this property for global webs when the basic blocks are scheduled separately because it may happen that several webs want to use the same architectural register within the same block. Therefore, register allocation for global webs should be performed before scheduling.

The global web variables that are in short supply of registers are spilled into memory after the web definitions and read from memory before the web uses. Thus, each global web is decomposed into a set of local webs. Local webs are assigned registers directly in the instruction scheduling procedure.

### *3.1. Register Allocation for Global Webs*

In the register allocation procedure for global webs (hereafter, global allocation) performed before scheduling, one can use the Chaitin–Briggs interference graph coloring algorithm [12]; a generalized version of this algorithm is described in [13]. The difference between the generalized and the classical algorithms is that the number of colors (i.e., registers) available for each node of the graph (web) is different. Because the subsequent local allocation requires a certain number of registers to be reserved and this number may be different for different nodes of the control flow graph, the number of available registers for different global webs can be significantly different.

One of the shortcomings of the algorithm that uses the interference graph is its complexity; for that reason, we decided to construct a faster algorithm using bit vectors for the representation of registers and webs. Each bit in the vector corresponds to a node of the the procedure control flow graph. The zero value of a bit in the register vector means that no web is allocated to this register at the node corresponding to this bit. The zero value of a bit in the web vector means that this web's variable is not live in the basic block corresponding to the bit.

Let $N$ be the number of nodes of the control flow graph of the routine and $R$ be the number of architectural registers. Then, the register file in our algorithm is a bit matrix of size $N \times R$.

The web can certainly be allocated to a register $r$ if the conjunction of the $r$th column of the matrix and the bit vector of the web yields a zero vector. Otherwise, an additional analysis is needed for each node corresponding to the nonzero bit of the resulting vector. The allocation of several webs to a single register is possible if each of the webs is not live when any other web is defined [11].

If the additional analysis yields a positive result for all the nonzero bits, the candidate web can be allocated to the given register. When a global web is allocated to the register $r$, the unity bits of the web vector are copied into the $r$th column of the register matrix.

If the global web failed to find a register, the global web is divided into several local webs using the minimally required (see [14]) number of spill/load instructions.

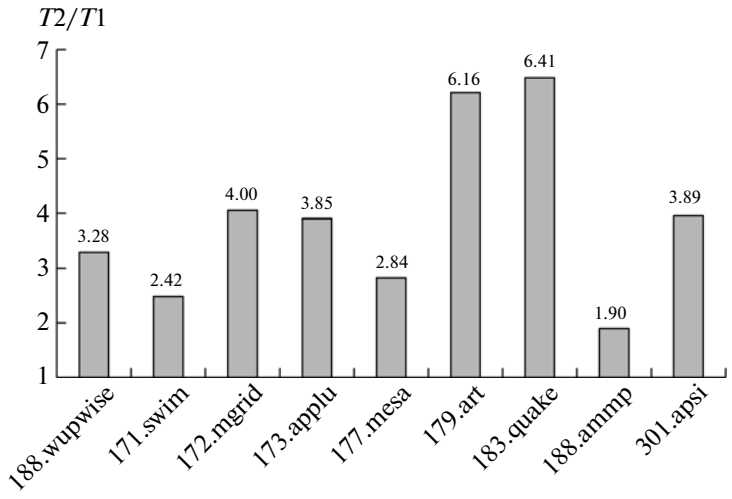


**Fig. 2.** Ratio of the total operation time of scheduling and allocation phases.

### *3.2. Register Allocation for Local Webs*

Register allocation for local webs occurs when their corresponding instructions are scheduled. In this study, we use the List Scheduling algorithm [15]; however, there are no obstacles for using any other other scheduling algorithm in the proposed scheme.

Local webs are allocated to free registers when their first definitions are scheduled. When the last use is scheduled, the register is released.

If a deficit of registers occurs during scheduling, spill/load instructions are created, which are scheduled on general grounds (i.e., with necessary delays). When no free registers are available, one can also suspend the scheduling of the instruction that needs a register rather than spill one of the occupied registers. As a rule, the delay in scheduling makes sense if the scheduling of another instruction results in releasing of one or more registers. In this case, the critical path must not increase [1]; otherwise, the the instruction delay must be less than the load instruction delay that restores the spilled register; otherwise; the register spill is preferable.

### *3.3. Keeping the Register Balance*

When registers for the local and global webs are allocated separately, it is necessary to determine as accurately as possible how many registers are to be allocated for each type in each basic block of the code.

In frequently executed (hot) code blocks, the spill of the local web inevitably leads to the appearance of a pair of spill/load instructions. A global web can be live on a hot block but have neither uses no definitions there. Spilling such a web into memory (i.e., its transformation into several local webs in rarely executed (cold) basic blocks) does not lead to a large degradation of performance. In this case, local webs should be given preference in hot blocks.

The situation is somewhat different with global webs that have both uses and definitions in a given hot basic block. Spill such a web into memory can lead to the appearance of spill/load instructions in other hot blocks of the code, where there is no deficit of registers. In this case, it is more preferable to allocate a register the global web; however, hereafter, we will assume for simplicity that local webs in hot blocks have higher priority.

The spill of local webs in cold blocks does not lead to the significant degradation of performance; however, the allocation of a large number of registers to local webs can lead to spilling global webs into memory in hot blocks. Therefore, local webs in cold blocks should be assigned a minimum number of registers needed for the convergence of the algorithm (i.e., equal to the maximum allowable number of register operands in the instruction).

The number of registers to be reserved for local webs in hot blocks can be accurately evaluated by calculating the degree of parallelism of he interference graph [15] for webs. The degree of parallelism is actually the maximum number of local webs living simultaneously. This value can be evaluated using the formula

$$LWN = \frac{\sum_i (TU(i) - TD(i))}{H},$$

where $LWN$ is the number of local webs living simultaneously, $TU(i)$ is the arithmetic mean of early and late scheduling times of the last use of the local web $i$, $TD(i)$ is the arithmetic mean of early and late scheduling times of the first definition of the local web $i$, and $H$ is the height of the interference graph of the basic block.

The reservation of particular registers for the local allocation can lead to a considerable degradation in performance. For example, reserving three registers

for local allocation, one can easily come across the situation when none of them is used and spill/load instructions indicating the lack of registers are created. This is explained by the fact that the local allocation had enough registers that were not needed for the global webs at the corresponding nodes. Therefore, the number of registers reserved for local allocation should be controlled dynamically for global allocation.

## 4. BENEFITS OF THE ALGORITHM

### *4.1. Taking into Account the Register Pressure*

The pressure on register file (or register pressure) is the number of variables stored in the registers at a particular time. Instruction scheduling that does not take into account the register pressure can lead to an unreasonably extended lifetime of registers and to the generation of redundant spill/load instructions. This problem can be worked out by adding the construction of webs and a register pressure control mechanism to the scheduling phase. However, if there are intermediate phases between scheduling and register allocation, one has to rebuild the webs.

When scheduling and register allocation are combined, the information about the register pressure is always available. The number of memory calls (beginning with some critical pressure) can be minimized by assigning scheduling priorities to all the instruction, taking into account the instructions' contribution to the register pressure. Thus, the scheduling operates in two modes: the normal mode, when there is no lack of registers, and the pressure mode, when such a deficit exists.

### *4.2. Optimization of Memory Access Delays*

The speed of a memory reading instruction depends on the hierarchy level of the memory where the requested data is located. When memory reading instructions are scheduled, one normally makes an optimistic assumption that the data is in the first-level cache memory; i.e., the delay is 1–3 cycles [16]. The insertion of a memory reading instruction in the scheduled code leads to additional difficulties and the degradation in performance. If a memory reading instruction is placed immediately before the data usage leads to blocking the pipeline until the instruction result is obtained. The insertion of a memory reading instruction earlier so as to take into account the delay requires additional analysis. For example it is impossible to move a read instruction through a wide instruction (WI) using the given register for its value would be spoiled, which makes degradation inevitable.

However, even if the dependences make it possible to take into account the memory access delay, the WI can be out of resources for coding the memory read instruction. Then, a new large instruction must be inserted into the code with only a single memory reading instruction rather than several tens of possible instructions. In this case, if the delay is not respected, the number of the empty WIs is equal to the delay; i.e., in the general case, the loss caused by this kind of scheduling can reach

$$IL_{max} = DT_{max} * WIS_{max} - 1$$

instructions, where $IL$ is the number of instructions that might be scheduled, $DT$ is the delay, and $WIS$ is the number of instructions in a WI.

Similar reasoning can also be applied to scheduling memory writing instructions.

The simultaneous instruction scheduling and register allocation makes it possible to respect the necessary delays in a natural way due to the construction of dependences between the original web instructions and the spill/load instructions. These dependences are taken into account by the scheduling algorithm due to which the spill/load instructions take part in the scheduling on equal terms and do not require the creation of new large instructions.

### *4.3. Reduction of the Algorithmic Complexity*

The classical register allocation algorithm constructs the interference graph and its generally iterative coloring by $R$ colors, where $R$ is the number of available registers. This technique is described in detail in [2, 12]. The complexity of the algorithm in the Briggs version is experimentally estimated as $O(N * \log(N))$ [11], where $N$ is the total number of webs in the program.

When the registers are allocated by graph coloring, the graph must include all (both global and local) webs. The alternative is to fill the bit register matrix; this algorithm is not iterative, and it does not require any structures to be rebuilt when there is a lack of registers. The assignment of registers to local webs is integrated into the instruction scheduling. This makes it possible to significantly reduce the total operation time of the register allocation and instruction scheduling phases. The complexity of this algorithm judging by the results of experimental runs is $O(N)$, where $N$ is the number of webs in the code.

## 5. EXPERIMENTAL RESULTS

The experiments were performed for an Elbrus microprocessor [17, 18] with VLIW architecture. The microprocessor has 256 registers that can be used both for integer and floating-point computations and 16 syllables for coding instructions (depending on their type, one or two instructions can be coded into one syllable).

The main advantages of register allocation combined with instruction scheduling manifest themselves on problems requiring a larger number of registers than offered by the architecture because the performance largely gains from the reduction in the number of memory access instructions and from taking into

account the delays occurring in scheduling these instructions. The efficiency of the algorithm was assessed on problems chosen from the SPEC CFP2000 packages. All results were obtained for the register window of a routine consisting of 48 registers.

Let us denote by $P1$ the execution time of the problems compiled using the new algorithm and by $P2$ the execution time of the same problems with register allocation on the scheduled code. The relative performance $P2/P1$ on SPEC CFP2000 problems is shown in Fig. 1.

Let us denote by $T1$ the operation time of the allocation phase in instruction scheduling and by $T2$ the total time of operation of the scheduling and allocation phases. Figure 2 shows the relative acceleration of the operation on SPEC CFP2000 problems for the combined mechanism as compared to the allocation on the scheduled code (i.e., $T2/T1$).

## 6. CONCLUSIONS AND FURTHER RESEARCH

On the average, the combination of instruction scheduling and register allocation significantly increases the performance, especially for architectures with a large instruction word. However, scheduling and allocation are highly affected also by other phases as well (for example, loop unrolling [15]). The detection of these phases and the development of efficient heuristics for them that take into account their interaction with the final phases is a key problem for further studies.

When the interference graph is ignored, the combined algorithm performs several times faster. However, the practical application of the global register allocation using a bit matrix requires further investigations.

## REFERENCES


1. Hennessy, J.L. and Patterson, D.A., *Computer Architecture: a Quantitative Approach*, San Francisco: Morgan Kaufmann, 2003.
2. Chaitin, G.J., Register Allocation and Spilling via Graph Coloring, *Proc. of the 1982 SIGPLAN Symp. on Compiler Construction*, June 23–25, 1982, Boston, MA, United States, pp. 98–105.
3. Berson, D.A., Gupta, R., and Soffa, M.L., GURRR: A Global Unified Resource Requirements Representation, *ACM SIGPLAN Notices*, 1995, vol. 30, no. 3, pp. 23–34.
4. Norris, C. and Pollock, L.L., Register Allocation over the Program Dependence Graph, *Proc. of the 2004 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2004, pp. 266–277.
5. Motwani, R., Palem, K.V., Sarkar, V., and Reyen, S., Combined Register Allocation and Instruction Scheduling Problem (CRISP), *Tech. Rep., Courant Institute, TR 698, July 1995.*
6. Pinter, S.S., Register Allocation with Instruction Scheduling: a New Approach, *ACM SIGPLAN Notices,* 1993, vol. 28, no. 6, pp. 248–257.
7. Koes, D.K. and Goldstein, S.C., A Global Progressive Register Allocator, *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation, June 11–14, 2006, Ottawa, Ontario, Canada*, 2006, pp. 204–215.
8. Xu, W. and Tessier, R., Tetris: a New Register Pressure Control Technique for VLIW Processors, *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems*, 2007, pp. 113–122.
9. Pereira, F.M.Q. and Palsberg, J., Register Allocation by Puzzle Solving, *Proc. of the 2008 ACM SIGPLAAN Conf. on Programming Language Design and Implementation*, *June, 2008*, pp. 216–226.
10. Bokhanko, A.S., Register Allocation by the Method of Interference Graph Coloring for Modern Computing Systems, *Cand. Sci. (Tech.) Dissertation*, Moscow: MTsST, 2005.
11. Muchnick, S.S., *Advanced Compiler Design and Implementation*, San Francisco: Morgan Kaufman, 1997.
12. Briggs, P., Cooper, K.D., Kennedy, K., and Torczon, L., Coloring Heuristics for Register Allocation, *Proc. of the 1989 SIGPLAN Conf. on Programming Language Design and Implementation*, 1989, pp. 275–284.
13. Bernstein, D., Goldin, D., Golumbic, M., Krawczyk, H., Mansour, Y., Nahshon, I., and Pinter, R., Spill Code Minimization Techniques for Optimizing Compilers, *ACM SIGPLAN Notices*, 1989, vol. 24, pp. 258–263.
14. Allen, R. and Kennedy, K., *Optimizing Compilers for Modern Architectures*, San Francisco: Mogan Kaufman, 2002.
15. Korneev, V. and Kiselev, A., *Sovremennye mikroprotsessory* (Modern Microprocessors), St.-Petersburg: BHV-Peterburg, 2003.
16. Babayan, B., E2K Technology and Implementation, *Proc. of the 6th Int. Conf. on Parallel Processing (Euro-Par 2000), January, 2000*, vol. 1900/2000, pp. 18–21.
17. Galazin, A.B., Stupachenko, E.V., and Shlykov, S.L., A Software Instruction Prefetching Method in Architectures with Static Scheduling, *Programmirovanie*, 2008, no. 1, pp. 67–74 [*Programming Comput. Software* (Engl. Transl.), 2008, vol. 34, no. 1, pp. 49–53].
18. Fisher, J.A. and O'Donnel, J.J., VLIW Machines: Multiprocessors We Can Actually Program, *Proc. of CompCon'84 Conf.*, IEEE, 1984, pp. 299–305.