Genetic Instruction Scheduling and Register Allocation

Fernanda Kri
Universidad de Santiago
Departamento de Ingeniería Informática
Av. Ecuador 3659
Santiago, Chile
fdakri@diinf.usach.cl

Marc Feeley
Univerisité de Montréal
Département d'Informatique et Recherche Opérationnelle
2920 Chemin de la Tour
Montréal, Canada
feeley@iro.umontreal.ca

Abstract

The construction of efficient compilers is very complex, since it has to contend with various optimization problems and depends on the characteristics of the architecture of the machine for which they generate code. Many of these problems are NP-hard. The genetics algorithms have been shown to be effective in the resolution of difficult problems, however, their use in compilation is practically non-existent. In this paper we propose a solution to the problems of register allocation and instruction scheduling. We carry out an analysis of performance by comparing with the more traditional approaches for these problems and we obtain profits on the speed of the generated code varying between -2% and 26%.

Keywords:: *optimizing compiler, genetics algorithms, instruction scheduling, register allocation.*

1. Introduction

In the compilation process it is necessary to solve several NP-hard optimization problems [1]. Generally, these problems are simplified and resolved using heuristics, which produces approximate solutions in a reasonable time. For example, the register allocation problem is often solved locally within each procedure or even within each basic block. A total solution using traditional heuristics is too complex, either because of the memory required, the length of execution time, or the poor solution quality for large problems [2, 3].

To define the heuristics, compiler designers base their work on the machine characteristics and the assumptions of the source programs. This results in the heuristics functioning well in some cases and badly in others, according to whether the assumptions are true or false. The use of genetics algorithms (GA) could make it possible to solve the optimization problems related to compilation in a more general way than the heuristics currently used.

The genetics algorithms are methods of research that are based on natural evolution [4, 5, 6]. In order to solve a problem with GA, it is necessary to find a representation for the (individual) solutions so that it is capable of encoding all of the possible solutions with the problem. Each solution has an associated numerical value, which is a measurement of its quality (fitness function). The GA starts with a set of random solutions (population) and forces them to evolve (from one generation to another) in the hope that at the end of the evolution process the quality of the population has improved. The evolution is carried out by using genetics operators (selection, crossover and mutation). We use a nontraditional fitness function: real execution time of the generated code. This choice is based on the importance of using a precise fitness function to ensure the GA produces positive performances. If the criterion of optimization is the execution time of the generated code, then the most exact fitness function is the real execution time.

This article is organized as follows. Section 2 introduces the problems of register allocation and instruction scheduling and shows the main techniques used for their resolu-

IEEE COMPUTER SOCIETY

tion. Section 3 shows the design of the GA in order to solve the problems of register allocation and instruction scheduling. Section 4 presents the performance evaluation. Finally, Section 6 presents our conclusions and describes some future work.

2. Related work

2.1. Register allocation problem

Typically, the compiler front-end generates an intermediate representation comprising a great number of variables. At code generation time, these variables must be assigned to either a register or the memory. On modern processors, the register instructions are executed more quickly than those implying memory access. Judicious use of the registers is very important at code generation time. It is essential to determine which variables are stored in the registers and to identify the registers to which they are assigned at each point of the program. This assignment must respect the conflicts in variables. The problem in determining the assignation of variables to the registers by keeping account of these conflicts is known as the register allocation problem [1].

The most important approaches to resolving register allocation problems are the graph coloring allocators and the on-the-fly allocators. Chaitin et al. [7] were the first to design a graph coloring allocator. This register allocator was useful as a basis for a large number of allocators (Chaitin style allocators). Briggs et al. [8, 9, 10] propose some modifications, which result in a major reduction of the number of variables which are spilled (conservative coalescing, optimistic coloring, rematerialization). Other works, like those of Bernstein [11] et al and Lueh [12], use the different heuristics to choose the nodes to spill.

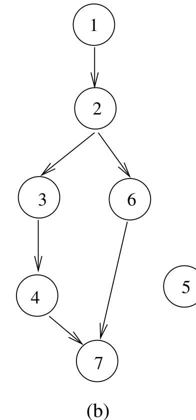
In the original Chaitin allocator, the STORE instructions are placed after each definition of the variable and the LOAD instructions before each use. Bergner et al. [13], Cooper and Simpson [14] and Lueh et al. [15] developed the heuristic versions for better placing of these instructions, in order to decrease the spill and avoid limiting the instruction level parallelism. Callahan and Koblenz [16] propose a graph coloring allocator that executes the program for coloring of segments considering its flow structure. This allocator places the spills in the least used parts of the program. Chow and Hennessy [17] introduce a graph coloring allocator that assigns the registers in function of the number of times that the variable is referred and the execution frequency of the basic block.

The on-the-fly allocators [18], [19] seek a compromise between allocation quality and compiler time. They are interesting when the compile time is an important factor (for example, in systems of dynamic compilation).

2.2. Instruction scheduling problem

(1) r1 = mem[10]
(2) r2 = mem[r1]
(3) r3 = r1 + r2
(4) r4 = r3 + 5
(5) r5 = mem[20]
*(6) r6 = r2 * 10*
(7) r7 = r6 + r4

(a)



(1) r1 = mem[10]
(2) r2 = mem[r1]
(3) r3 = r1 + r2
*(6) r6 = r2 * 10*
(4) r4 = r3 + 5
(5) r5 = mem[20]
(7) r7 = r6 + r4

(c)

Figure 1. (a) Program example. (b) Dependence graph. (c) Scheduler program.

The modern processors have several functional units that can work in parallel. This characteristic allows the program's implicit parallelism (instruction level parallelism, or ILP) to taken advantage of. So that two programs' instructions can be executed at the same time, they must be independent. For example, in the figure 1(a) instructions 1, 2 and 3 are dependent because they possess data dependence. They must therefore be executed in this order because if not, the result obtained will not be that desired. On the other hand, instruction 3 is independent of instruction 6, so these can be executed in any order, even in parallel. These dependences can be represented by a graph, as shown in figure 1(b).

To take advantage of ILP, the code must be rewritten so that the instructions are generated in the appropriate order. In our example, if we want instructions 3 and 6 to be executed in parallel, the code must be rewritten as figure 1(c) shows. This problem is called instruction scheduling [20]. The instruction scheduling problem is NP-hard [2]. Two main approaches exist to tackle the problem: loop scheduling and basic block scheduling. The loop scheduling orders instructions into the loop body. The Modulo Scheduling algorithm proposed by Rau [21] is most frequently used to solve this problem. Several modifications were presented in [22] and [23]. In [24], a comparison of several modulo scheduling techniques is presented.

Basic block scheduling consists of reordering the instructions in each basic block of the program. Since there are no loops, the dependences of the variables can be rep-

Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC'04)
0-7695-2185-1/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

resented by a DAG (Directed Acyclic Graph). The algorithm most often used to solve this problem is List Scheduling, which was proposed by Gibbons and Muchnick [25]. In Cooper et al. [26], Wilken et al. [27] we found different implementations of this algorithm.

The register allocation and the instruction scheduling problems are strongly interdependent [28]. Zalamea et al. [23], Eichenberger et al. [29], Ning and Gao [30], Gopal and Govindrajan [31] define different heuristics to solve both problems simultaneously.

Although the metaheuristic methods were largely used to solve NP-hard problems, their use in compilation is rare. We find some work on multiprocessor scheduling in [32, 33], but they tackle the problem in an abstract way. Very few works have really managed to use these methods in compilation. Williams [34] and Nisbet [35] use GA to carry out the automatic loop parallelization in a parallel language compiler. Beaty [36, 37] deals with the instruction scheduling problem using an algorithm of list scheduling, where the priority of each instruction is found by metaheuristics (GA and research taboo [36]). Beaty proposes an approach to carry out scheduling using a genetic algorithm but it does not produce results. In [37], he evaluates various genetics operators and vicinity generation functions. These results show that the two methods make it possible to find good solutions to the problem of instruction scheduling, but they do not show performance results. Finally, Stephenson et al. [38, 39] propose the use of genetic programming and machine learning to calculate the priorities for the traditional compiler heuristics.

3. The Genetic Algorithms

3.1. The fitness function

One of the determining factors in obtaining a good performance from GA is the accuracy of the fitness function. The more precise the fitness function, the more likely a GA is to converge towards higher quality solutions. For example, if we want to obtain the smallest object code possible, the instruction number of the object code is a suitable fitness function. However, if our objective is to generate the object code with the shortest execution time, finding an adequate fitness function is not simple since this depends on a great number of hardware characteristics (memory speed, pipeline depth, structure of calculating unit, etc).

The use of the real execution time as a fitness function offers some interesting advantages. Firstly, since our objective consists of minimizing the execution time of the generated code, the real execution time is the most precise measurement we can have. Moreover, by using this fitness function, we need only very little information on target machine architecture, since we do not need to draw up estimates of the behavior of the machine. Another advantage of the use of the execution time is that we measure the real effect of optimizations of the target machine. This includes the indirect effects of optimizations that are very difficult to estimate, for example, the fault pages or the proprietary characteristics of the architecture.

In order to measure the quality of each individual, we must generate the object program described by the individual (allocation and scheduling), in order to carry it out and measure the real execution time. We measure the execution time by making an operating system call that determines the time used by the program.

3.2. The representation

We define an individual as the concatenation of 2 sub-individuals, one for each problem (figure 2).

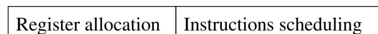


Figure 2. Individual representation

In order to choose a representation for each problem, it is necessary to take into account the interdependence of the problems. It is possible that, because of the interdependence between the problems, the instruction scheduling is not compatible with the register allocation, and yet the individual trained by the concatenation of these two sub-individuals is non-feasible. The design of sub-individuals must consider this dependency, with the objective always being to generate feasible individuals.

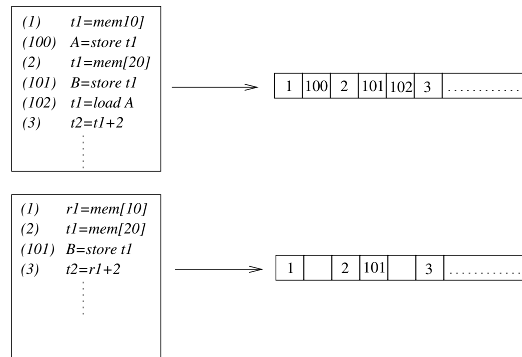


Figure 3. Instructions scheduling individual

Our aim is to train an individual (concatenation of the sub-individuals for each problem) that is always feasible. Thus, all the individuals in the population can be evaluated.

Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC'04)
0-7695-2185-1/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

For the register allocation problem, we chose the following representation (based on a graph coloring problem [40]): the individual is a chain of length L of positive or negative numbers. Each element of the individual represents a variable. The variables represented by negative values are directly assigned to the memory. GA uses a greedy function, which assigns the variables represented by positive values to the registers.

The function treats the variables sequentially in the order in which they appear in the individual, and when interferences between the variables allow, available registers are assigned to them. If not, the variable is assigned to the memory.

For the instruction scheduling, we use long fix L individuals, where L is the number of instructions, by assigning all the variables to the memory, i.e. L is the maximum length. The instructions are numbered with two distinct sequences. The first represents (using the numbers 0 to N) instructions that do not depend on the allocation and must be present regardless of the allocation. The second represents (using numbers larger than M, where $M > N$) the instructions that depend on the allocation (move instructions, storage and loading). Thus, without difficulty we can eliminate the instructions that are no longer necessary after a modification of the allocation without changing the order of the preserved instructions. Therefore, the evolution process is not disturbed since the relative order of the instructions in the individual is preserved. Figure 3 shows an example of an individual developed according to the above method.

With the selected representations for the two problems, all the individuals (concatenation of register allocation and instruction scheduling individuals) allow the generation of the correct object code and thus, the joint solutions of the problems.

3.3. The Genetic Operators

The crossover and mutation operators depend on the representation of the problem and they must be defined by taking into account the characteristics of each problem. Consequently, they will act independently on each sub-individual. On the other hand, the selection operator acts on the complete individual, because what interests us is measuring the quality of a complete individual, namely that made up of solutions to all the sub-problems. Then, we use the traditional selection operator, the roulette-wheel selection.

We empirically considered and evaluated operators of classic crossover and mutation that have been used successfully in similar problems. For the crossover we evaluated OBX and PBX and both showed a similar performance. However, as far as the quality of the solution, OBX allowed a faster convergence. For the mutation we used a random change in a portion of the individual.

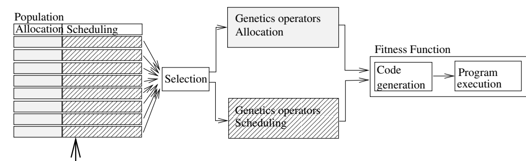


Figure 4. GA structure for the register allocation and instruction scheduling problems

Figure 4 shows the operation of the GA. The population is formed by individuals, where each individual is composed of solutions to the problems to be solved. With each generation, the selection operator chooses individuals according to their quality. Then, each individual is subdivided so that each sub-individual is processed by the corresponding operators. Finally, the individual is rebuilt and evaluated by the fitness function.

4. Results

The machine used for the experiments described below is an Ultra-1 SPARC, with an UltraSparc-I 167 MHz processor and 768 MB memory. We use the function of the operating system $time$ to measure the execution time. We add user and system times. All the results presented are averages of five executions. In all cases, the standard deviation is smaller than 0.005 seconds.

Name	Description
bj	Program that plays black-jack in order to measure the machine performance
bubble	GNU *bubble sort*
dhr	Dhrystone
fft	FFT implantation
fib	Calculate of fibonacci number
flow	Exhaustive algorithm implemantation to calculate the maximum flow of a network
fulk	Ford-Fulkerson implantation to calculate the maximum flow of a network
hanoi	Hanoi towers program
heap	GNU *heapsort*
heapsort	Program which carries out several times a heapsort in order to measure the machine performance
insertion	GNU *insertion sort*
merge	GNU *merge sort*
arithm	Program that executes aritmeticas operations in a cycle
queens	n queens program
quick	GNU *quick sort*
selection	GNU *selection sort*
shuffle	Checking of a generator of random random numbers
tfftdp	Another FFT implantation
whet	Whetstone simple precision

Table 1. Benchmark descriptions

The benchmarks used are described in the table 1. The table 2 indicates the number of procedures (NP), the number of variables used in each procedure (VpP), the number of basic blocks (NBB) and the size of the largest basic blocks (GBB) for each benchmark.

Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC'04)
0-7695-2185-1/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

Name	NP	VpP	NBB	GBB
arithm	1	19	12	139
bj	5	32-6-2-2-6	562	119
bubble	2	3-1	16	8
dhr	13	7-2-2-2-2 1-1-1-0-0-0-0-0	138	23
fft	5	3-14-3-3-3	119	154
fib	3	2-0-0	18	18
flow	10	4-4-3-3-3-3 2-1-0-0	150	25
fulk	10	4-4-4-4-3-3 3-2-1-0	168	35
hanoi	3	2-1-0	26	23
heap	3	3-2-1	28	10
heapsort	2	21-5	70	94
insertion	2	3-1	16	5
merge	4	4-1-1-0	31	9
queens	1	18	93	171
quick	3	3-1-0	26	6
selection	2	4-1	17	8
shuffle	3	8-6-6	68	112
tfftdp	2	16-10	83	130
whet	7	14-9-1-0-0-0-0	248	51

Table 2. Benchmark characteristiques

We incorporated the GA into the lcc compiler [41]. Lcc is a very simple C compiler that does not perform many optimizations.

Our system performance is compared with those of the existing approaches for the register allocation and instruction scheduling problems. For the register allocation, we use the lcc allocator (assigns the variables most used to the registers) [42] and a graph coloring allocator as described by Appel [2]. The heuristics used by this allocator, in order to decide which node must be spilled to the memory, are based on the number of times that the variable is used and on the number of node edges. Thus, the allocator favors spilling the variables which interfere with many others and that are seldom used.

For the instruction scheduling problem we have established a list scheduling using the node which maximizes parallelism as a heuristic, i.e. the node which maximizes the number of nodes that will then be ready to be carried out.

We modified lcc to control the number of registers available. We compile the benchmarks by assuming the presence of 4, 8 and 12 registers. This enables us to analyze the sensitivity of the various approaches to the number of registers available.

The curves of the figures 5 and 6 illustrate the convergence of GA for the benchmarks bj and $tfftdp$ with 8 registers.

We present the best solution of GA and we fix as a point of reference the constant curves representing the execution times while using:

- the lcc allocator and a list scheduling ($lcc + sch$);
- a coloring graph register allocator and a list scheduling ($color + sch$).
- the execution time in the worst case occurs with all the variables in memory and without instruction scheduling.

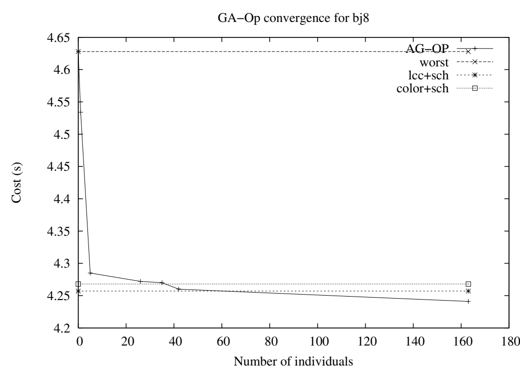


Figure 5. GA convergence for bj

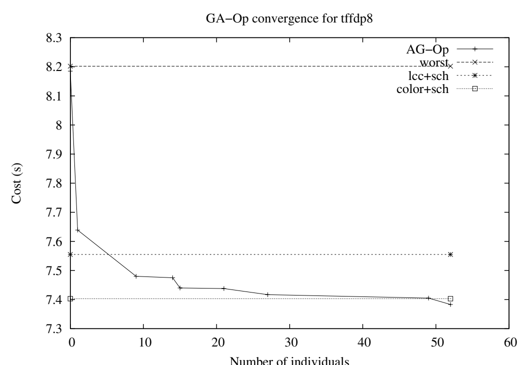


Figure 6. GA convergence for *fftdp*

According to the graphs of the figures 5 and 6, GA converges quickly with values close to those obtained by traditional approaches. This behavior is similar for all of the benchmarks. On the other hand, GA takes much more time to compile than lcc, therefore, the use of GA is viable only for users who can allow themselves an increase in compiling time in exchange for a gain in code execution time. Now, we pay attention to the analysis of the performance of our GA according to the speed of the generated code.

Table 3 presents the execution times for the benchmarks compiled with all of the approaches; we also have the results obtained using the worst case scenario, i.e., all the variables spilled to the memory and without instruction scheduling. For the benchmarks not included in this table, all the approaches obtained equivalent results.

To help us analyze the results, the figure 7 shows the percentage improvement with each approach compared to the worst case for each benchmark. We observe that the gains are significant in all cases, reaching up to 50% for $heapsort$ with 8 registers, whereas the differences between the methods of optimization are not very significant, except in the

Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC'04)
0-7695-2185-1/04 $ 20.00 IEEE
IEEE COMPUTER SOCIETY

Benchmark	worst (s)	lcc+sch (s)	color+sch (s)	AG-Op (s)
arithm4	13.4	12.7	12.7	10.6
arithm8	13.4	12.7	12.7	9.3
arithm12	13.4	12.8	12.8	10.6
bj4	4.6	4.4	4.3	4.2
bj8	4.6	4.3	4.3	4.2
bj12	4.6	4.3	4.3	4.2
dhr4	3.3	2.6	2.6	2.6
fft4	4.4	3.8	3.9	3.8
fft8	4.4	3.8	3.7	3.8
fft12	4.4	3.8	3.8	3.7
heapsort4	25.9	15.3	15.0	14.5
heapsort8	25.9	12.5	13.2	12.3
heapsort12	25.9	13.1	12.9	12.2
queens4	0.6	0.4	0.4	0.4
queens8	0.6	0.4	0.4	0.4
queens12	0.6	0.4	0.4	0.4
shuffe4	40.6	38.6	37.1	37.5
tfftdp4	8.2	7.7	7.7	7.6
tfftdp8	8.2	7.6	7.4	7.4
tfftdp12	8.2	7.4	7.3	7.2
whet4	22.8	20.4	21.5	20.5
whet8	22.8	20.4	21.4	20.4
whet12	22.6	19.1	19.1	19.1

Table 3. Execution time with every approach

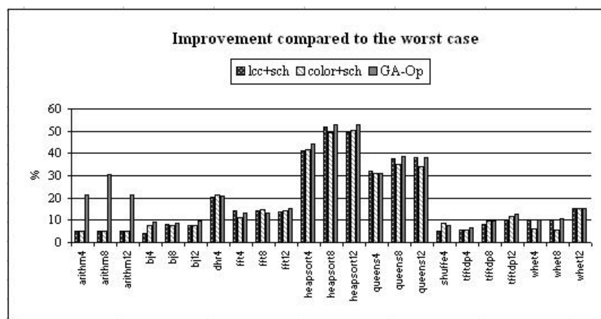


Figure 7. Improvement

case of $arithm$.

Note that GA performs better for $arithm$ than for the other benchmarks. Additionally, the traditional approaches ($lcc + sch$ and $color + sch$) show performances close to the worst case. This behavior is due to the fact that this benchmark has a structure that misleads the traditional heuristics. In $arithm$, there are variables which are much referred to and others very seldom. The traditional heuristics for the register allocation give priority to the variables according to the number of times to which they are referred. These approaches normally function well. However, in the execution, variables of $arithm$ that are seldom referred to are used more often than the others. Consequently, the traditional heuristic approaches give priority to the bad variables. By using the real execution time as a fitness function, GA converges easily towards the most suitable solution.

Table 4 shows the improvements obtained by GA compared to the worst case. The results are ordered according to percentage improvement.

If we consider the gains with respect to the worse case,

Benchmark	worst(s)	AG-Op (s)	AG/worst (%)
tfftdp4	8.2	7.6	7.3
shuffe4	40.6	37.5	7.6
bj8	4.6	4.2	8.7
bj4	4.6	4.2	8.7
bj12	4.6	4.2	8.7
tfftdp8	8.2	7.4	9.8
whet4	22.8	20.5	10.1
whet8	22.8	20.4	10.5
tfftdp12	8.2	7.2	12.2
fft4	4.4	3.8	13.6
fft8	4.4	3.8	13.6
fft12	4.4	3.7	15.9
whet12	22.6	19.1	15.5
dhr4	3.3	2.6	21.2
arithm4	13.4	10.6	20.9
arithm8	13.4	9.3	30.6
arithm12	13.4	10.6	20.9
queens4	0.6	0.4	33.3
queens12	0.6	0.4	33.3
queens8	0.6	0.4	33.3
heapsort4	25.9	14.5	44.0
heapsort8	25.9	12.3	52.5
heapsort12	25.9	12.2	52.9

Table 4. Improvements obtained by GA compared to the worst case

we see that the best performances are obtained for the $heapsort$ and $queens$ benchmarks. These benchmarks have the characteristic that they include a very small number of procedures (1 and 2, respectively) and a large number of variables. Therefore, the register allocation is able to cause a more significant impact than in the more modular programs, which have many functions but few variables. This behavior is observed in all our benchmarks. The best performances of the GA are obtained on the benchmarks with few procedures and that use many variables. For the same reason, the fact that the basic block is larger allows instruction scheduling to make more significant improvements.

5. Conclusions

Our research establishes that the use of GA in compilation is viable, advantageous and also offers interesting advantages.

Even if at present the programs are increasingly modular, the behavior of GA is still very interesting because the compiler always has the option to create large basic blocks and extensive procedures by carrying out certain optimizations before the register allocation and the instruction scheduling. For example, loop unrolling and inlining of functions. These optimizations can be included with GA or carried out in advance. Moreover, certain programs generated by other programs (for example the Gambit-C compiler for Scheme) tend to have large functions.

In comparison to the traditional approaches, the GA demonstrates a positive performance when solving the problems of register allocation and instruction scheduling:

- compared to the worst case (all variables are in mem-

Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC'04)
0-7695-2185-1/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

ory and without instruction scheduling), the increases in speed vary between 7% and 53%.

- compared to the register allocation by graph coloring and instruction list scheduling, the increases in speed vary between -2.7% and 26%.
- compared to the register allocation of *lcc* and instruction list scheduling, the increases in speed vary between -0.5% and 26%.

The most important increases noted were for benchmarks that use a large number of variables, few procedures and larger basic blocks. This is due to the fact that in these programs optimizations have a greater total impact.

Thanks to the use of real time as fitness function, GA shows more homogeneous behavior than the heuristics for the entirety of the benchmarks. We observe that for a benchmark which deviates from the normal, the traditional heuristics do not manage to find a good solution; whereas, GA performs very well.

Finally, by using real execution times, fitness function GA needs very little information about the machine. Considering the increasing complexity of current architectures, this characteristic is very interesting because the compiler construction becomes much more simple. Likewise, the compiler will be able to adapt easily to the architectural changes.

The use of the real execution time also makes it possible to measure the side effects of various optimizations, which is not possible with the traditional heuristics.

References

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1978.

[2] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, 1997.

[3] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[4] John H. Holland. *Adaptation in Natural Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Mass., 1989.

[6] Lawrence Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[7] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation Via Coloring. *Computer Languages*, pages 47–57, 1982.

[8] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *ACM SIGPLAN Notices*, pages 311–321, 1992.

[9] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, pages 428–455, 1994.

[10] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring Heuristics for Register Allocation. *ACM SIGPLAN Notices*, pages 275–284, 1989.

[11] D. Bernstein, D. Q. Goldin, M. C. Golumbici, H. Krawczykand Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–263, 1989.

[12] Guei-Yuan Lueh. Issues in Register Allocation by Graph Coloring. Technical Report CMU-CS-96-171, School of Computer Science, 1996.

[13] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew T. O'Keefe. Spill Code Minimization Via Interference Region Spilling. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

[14] K. D. Cooper and L. T. Simpson. Live Range Splitting in a Graph Coloring Register Allocator. *Lecture Notes in Computer Science*, pages 174–187, 1998.

[15] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Global Register Allocation Based on Graph Fusion. *Languages and Compilers for Parallel Computing*, pages 246–265, 1996.

[16] David Callahan and Brian Koblenz. Register Allocation Via Hierarchical Graph Coloring. *Conference on Programming Language Design and Implementation*, pages 192–203, 1991.

[17] F. C. Chow and J. L. Hennessy. The Priority-based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, pages 501–536, 1990.

[18] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and Speed in Linear-Scan Register Allocation. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.

[19] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 895–913, 1999.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, second edition, 1996.

[21] B. Ramakrishna Rau. Iterative Modulo Scheduling: an Algorithm for Software Pipelining Loops. *27th Annual International Symposium on Microarchitecture*, pages 63–74, 1994.

[22] Richard A. Huff. Lifetime-Sensitive Modulo Scheduling. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–267, 1993.

[23] J. Zalamea, J. Llosa, E. Ayguad'e, and M. Valero. Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures. *34th International Symposium on Microarchitecture (MICRO-34)*, pages 160–169, 2001.

[24] Josep M. Codina, Josep Llosa, and Antonio Gonzlez. A Comparative Study of Modulo Scheduling Techniques. *16th International Conference on Supercomputing*, pages 97–106, 2002.

[25] Philip B. Gibbons and Steven S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. *SIGPLAN Symposium on Compiler Contruction*, pages 11–16, 1986.

Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC'04)
0-7695-2185-1/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

[26] Keith Cooper, Philip Schielke, and Devika Subramanian. An Experimental Evaluation of List Scheduling. Technical Report TR98-326, Rice University, 1998.

[27] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal Instruction Scheduling Using Integer Programming. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2000.

[28] J. R. Goodman and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. *2nd International Conference on Supercomputing*, pages 442–452, 1988.

[29] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Minimizing Register Requirements of a Modulo Schedule Via Optimum Stage Scheduling. *International Journal of Parallel Programming*, pages 103–132, 1996.

[30] Qi Ning and Guang R. Gao. A Novel Framework of Register Allocation for Software Pipelining. *20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, 1993.

[31] Madhavi Gopal Valluri and R. Govindarajan. Evaluating Register Allocation and Instruction Scheduling Techniques in Out-of-Order Issue Processors. *International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 78–83, 1999.

[32] Arthur L. Corcoran and Roger L. Wainwright. A Parallel Island Model Genetic Algorithm for the Multiprocessor Scheduling Problem. *Selected Areas in Cryptography*, pages 483–487, 1994.

[33] A. Schoneveld, J.F. de Ronde, and P.M.A. Sloot. Task Allocation by Parallel Evolutionary Computing. *Journal of Parallel and Distributed Computing*, pages 91–97, 1997.

[34] K. Williams and S. William. Genetic Compilers: A New Technique for Automatic Parallelisation. *2nd European School of Parallel Programming Environments (ESPPE 96)*, pages 27–30, 1996.

[35] Andy P. Nisbet. GAPS: Genetic Algorithm Optimised Parallelisation. *7th Workshop on Compilers for Parallel Computing*, pages 172–183, 1998.

[36] Steven J. Beaty. Genetic Algorithms and Instruction Scheduling. *24th Annual International Symposium on Microarchitecture*, pages 206–211, 1991.

[37] S. J. Beaty. Genetic Algorithms Versus Tabu Search for Instruction Scheduling. *International Conference on Neural Network and Genetic Algorithms*, pages 496–501, 1993.

[38] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta-optimization: Improving compiler heuristics with machine learning. Technical report, Technical Report MIT-LCS-TM-634, 2002.

[39] Mark Stephenson, Una-May O'Reilly, Martin C. Martin, and Saman Amarasinghe. Genetic programming applied to compiler heuristic optimization. *Genetic Programming, Proceedings of EuroGP'2003*, 2003.

[40] C. Fleurent and J. A. Ferland. Genetic and Hybrid Algorithms for Graph Coloring. *Annals of Operations Research*, pages 437–461, 1997.

[41] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., 1995.

[42] Christopher W. Fraser and David R. Hanson. Simple Register Spilling in a Retargetable Compiler. *Software - Practice and Experience*, pages 85–99, 1992.

Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC'04)
0-7695-2185-1/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY