2011 30th International Conference of the Chilean Computer Science Society

Register Allocation with Graph Coloring by Ant Colony Optimization

Carla Negri Lintzmayer, Mauro Henrique Mulati and Anderson Faustino da Silva
State University of Maringá – Brazil
Email: {carla0negri,mhmulati}@gmail.com, anderson@din.uem.br

***Abstract*—The goal of register allocation is to allocate an unbounded number of program values to a finite number of machine registers. In this paper, we describe a new algorithm for intraprocedural register allocation called CA-RT-RA, an algorithm that extends a classic graph coloring register allocator to use our graph coloring algorithm *ColorAnt-RT*. The experiments demonstrated that our algorithm is able to minimize the amount of spills, thereby improving the quality of the generated code. CA-RT-RA is interesting in applications where compile time is not a concern, but the code quality.**

***Keywords*-Register Allocation; Graph Coloring; Ant Colony Optimization; ColorAnt-RT.**

I. Introduction

Register allocation is among the most important compiler optimizations affecting the performance of compiled code [1]. It determines which of the program values (variables and temporaries) should be in machine registers (or memory) during the execution of a program.

In a real machine, registers are usually few and fast to access [2], [3], so the problem addressed here is how to minimize the traffic between registers and memory hierarchy. Therefore, the challenge is to relegate the least program values to memory.

Register allocation can be mapped as a graph coloring problem (GCP) [4], [5]. The traditional GCP consists in finding the minimum value of k for which a graph is k-colorable. The k-GCP consists in trying to color a graph with k fixed colors by minimizing the amount of conflicts (adjacent vertices assigned the same color). Note that, in register allocation the k-GCP has a slight variation: it is forced to eliminate conflicting edges besides coloring the graph with just k colors (registers).

Graph coloring [1] is a highly effective approach to intraprocedural register allocation, and can be briefly described as follows. During code generation, the compiler [6] uses infinite symbolic registers to hold program values, and determines what values are candidates for allocation to registers. Firstly, the register allocator generates a so-called interference graph [1], whose vertices/nodes represent the program values and the machine real registers and whose edges represent interferences. In this graph, an edge (interference) is added either if two values are simultaneously live or a value cannot be or should not be allocated to that register. After that, it will color the interference graph nodes with k colors, so that any two adjacent nodes have different colors. And, finally, the allocator will allocate each value to the register that has the same color that was assigned to it.

In applications where compile time is a concern, such as dynamic compilation systems [7], [8], [9], researchers try to balance the time necessary for allocation against the resulting code quality. And, in this context, they do not choose a register allocation algorithm based on graph coloring, because it is a complex algorithm, besides being a time-consuming register allocator. However, allocators [10], [11], [12], [13] that are considered faster than those based on graph coloring result in code that is almost as efficient as that obtained by a graph coloring register allocator (GCRA) [1], [14], [15], [16].

A problem with some GCRA approaches is the fact that some of them apply simple heuristic methods resulting often in a poor allocation. In this case, there will be constant data traffic between processor and memory causing a performance loss.

Finding a solution for k-GCP is a $\mathcal{NP}$-complete problem [17]. No exact algorithm in polynomial time is known, encouraging the use of heuristic algorithms and metaheuristics to find good solutions.

Ant Colony Optimization (ACO) is a metaheuristic used to create heuristic algorithms to find good solutions for $\mathcal{NP}$-hard combinatorial optimization problems [18]. ACO is inspired on the effective behavior present in some species of ants of exploring the environment to find and transport food to the nest. Several works propose using ACO algorithms to solve problems [19] such as: vehicle routing, frequency assignment, scheduling and graph coloring.

In this paper, we describe an intraprocedural register allocation algorithm, called CA-RT-RA, that is based on graph coloring and ACO. This algorithm extends George-Appel's Graph Coloring [16] to use our ACO-based *ColorAnt-RT* algorithm [20].

We evaluate both the resulting code and the compile-time performance of our algorithm, and compare it to George-Appel's GCRA. In order to do this, we implemented CA-RT-RA and George-Appel algorithms in a research compiler, and compared the resulting code. The CA-RT-RA outperforms George-Appel's algorithm in terms of program values that are effectively represented in memory, besides in code size. Moreover, our results demonstrated that CA-RT-RA is useful in situations where compile time is not important, but code quality, such as compiler that generates code to embedded systems [21], [22].

1522-4902/12 $26.00 © 2012 IEEE
DOI 10.1109/SCCC.2011.32

The rest of this paper is organized as follows. Section II summarizes related work on Ant Colony Optimization and register allocation. Section III outlines the George-Appel's GCRA algoritm. The details of our register allocation algorithm appear in Section IV. Section V presents measurements of the algorithm's performance. Finally, Section VI summarizes our conclusions and directions for future work.

II. Related Work

GCRA was proposed by Chaitin *et al.* [4], [23]. This allocator was used in an experimental IBM 370 PL/I compiler. Currently, versions of it and allocators derived from it have been used in mainstream compilers. Subsequently, several works added improvements to Chaitins allocator [24], [25]. The most successful design for GCRA was developed by Briggs *et al.* [26]. Their work redesigned the Chaitin *et al.* allocator to delay spill decisions until later on in the allocation process. Runeson and Nyström proposed a generalization of Chaitins allocator, which allows it to be used for irregular architectures [27]. This work is an interesting framework for a retargetable graph-coloring allocator.

George and Appel [28], [16] designed a GCRA that interleaves Chaitin-style simplification steps with Briggs-style conservative coalescing. They ensure this approach eliminates more move instructions than Briggs's one, while still guaranteeing not to introduce spills.

Daveou *et al.* [29] presented a register allocation framework designed to address the embedded processor specificities, such as smaller number of registers, irregular and constrained register sets, and instructions operating on short or long data types. This allocator is based on Briggs's one, with two new components developed to improve performance, namely: a spill manager that optimizes spill operations, and a code manager that optimizes the move operations inserted by the allocator.

Wu and Li [30] proposed a hybrid metaheuristic algorithm for GCRA that combines several ideas from classic GCRA algorithms, besides evolutionary algorithms [31] and Tabu Search [32], [33]. The main idea of this approach is to exploit the interplay between intensification and diversification of the solution space. The authors argue it is a good solution to prevent searching processes from cycling, i.e., from endlessly revisiting the same solutions set, besides can impart additional robustness to the search.

III. The Iterative Register Allocation Algorithm

Based on the observation that a good graph coloring register allocator should not only assigning different colors to interfering program values, but also trying to assign the same color to temporaries related by copies, George and Appel developed a Iterative Register Allocation Algorithm [28], [16].

This algorithm iterates until there are no spills. The results demonstrated how to interleave coloring reductions with coalescing heuristic, leading to an algorithm that is safe and aggressive.

The Iterative Register Allocation Algorithm has the following phases:

1) Build: build the interference graph.
2) Simplify: simplify the interference graph.
3) Coalesce: perform conservative coalescing.
4) Freeze: freeze some move-related nodes.
5) Spill: select a node for spilling.
6) Select: assign colors to the vertices.

The assumption in this approach is that the compiler is free to generate new temporaries and copies, because almost all copies will be coalesced. Figure 1 shows how the phases are organized.

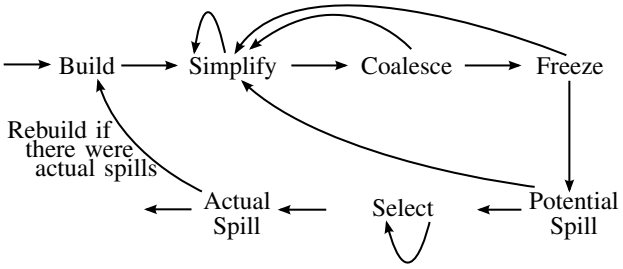

Fig. 1. The Iterative Register Allocation Algorithm [16].

The next subsections outline the algorithm phases.

A. Build

In this phase the interference graph is constructed by using dataflow analysis and its nodes are categorized as either related or not related to moves. A move instruction means that the node is either the source or the destination of that move.

B. Simplify

George-Appel's algorithm uses a simple heuristic to simplify the graph. If the graph G contains a node n with less than k (number of registers) neighbors, then G' is built by doing $G' = G - \{n\}$. Then, if G' can be colored, then G can be as well. This phase repeatedly removes the non-move-related nodes from the graph if they have low degree ($< k$), by pushing them on a stack.

C. Coalesce

This phase tries to find moves to coalesce in the reduced graph obtained in *Simplify* phase. If two temporaries $T1$ and $T2$ do not interfere, is desirable these temporaries be allocated into the same register. This phase eliminates all possible move instructions by coalescing source and destination into a new node. If it is possible, this phase also removes the redundant instruction from the target program.

Simplify and *Coalesce* phases are repeated while the graph contains non-move-related nodes or nodes of low degree.

D. Freeze

Sometimes, neither *Simplify* nor *Coalesce* can be applied. In this case, the algorithm *freezes* a move-instruction node of low degree by considering it a non-move-related, and enabling more simplification. After this, *Simplify* and *Coalesce* are resumed.

E. Potential Spill

If the graph, at some point, has only nodes of degree $\geq k$, these nodes are marked for spilling (they probably will be represented in memory). But at this point, they are just removed from the graph and pushed on the stack.

F. Select

Select remove the nodes from the stack, and tries to color them by rebuilding the original graph. This process does not guarantee that the graph will be k-colorable. If the adjacent nodes were already colored with k colors, the current node cannot be colored and will be an *actual spill*. This process will continue until there is no more nodes in the stack.

G. Actual Spill

In case of *Select* phase identifies an *actual spill*, the program is rewritten to fetch the spilled node from memory before each use, and store it after each definition. Now, the algorithm needs to be repeated on this new program.

IV. The ColorAnt-RT Register Allocation Algorithm

CA-RT-RA algorithm modifies the George-Appel's algorithm in order to add an ACO metaheuristic phase. Two modifications were made, namely:

1) The *Select* phase was substituted by our *ColorAnt$_3$-RT* algorithm, now it is a more aggressive phase than George-Appel's optimistic coloring; and
2) The strategy used for selecting spill is not based on node degree, but based on conflicting edges.

Figure 2 shows the phases of the CA-RT-RA algorithm.

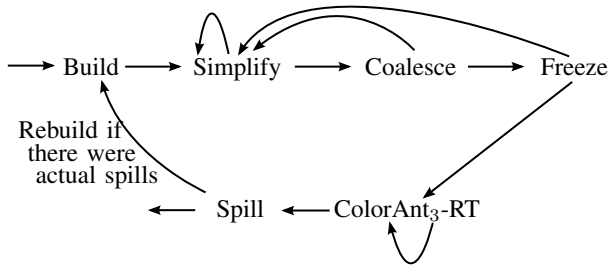

Fig. 2. The Iterative Register Allocation with *ColorAnt$_3$-RT*.

Firstly, the George-Appel's classic phases construct an interference graph and reduces the graph. After, our *ColorAnt$_3$-RT* algorithm colors the interference graph. And finally, the new *Spill* phase selects an appropriate node to be represented in memory. The next two sections detail these two modifications.

A. The ColorAnt$_3$-RT Phase

Our approach is to use an heuristic algorithm based on artificial colonies of ants with local search designed for the GCRA problem. Firstly, we implemented an algorithm that was able to obtain satisfactory solutions, with respect to reducing the amount of conflicts [34]. But, our investigation demonstrated that changing the way to reinforce the pheromone trail results in a greater reduction in the amount of conflicts [35], [36]. Due to this fact, our research group has developed three *ColorAnt-RT* algorithms. The CA-RT-RA uses our best version: *ColorAnt$_3$-RT*.

The three *ColorAnt* algorithms use as constructive method (for each ant) an algorithm suggested along with *ANTCOL* [37], which tries to color a graph with k fixed colors. Such algorithm will be called here *Ant_Fixed_k*, and it is presented in Algorithm 1. The *Ant_Fixed_k* was suggested as a constructive method for a version of *ANTCOL* for the k-GCP, here called *k-ANTCOL*.

Algoritmo 1 *Ant_Fixed_k* Algorithm.

```
ANT_FIXED_K(G = (V, E), k)    // V: vertices; E: edges
1  NC = V;                    // set of non-colored vertices
2  s(i) = 0  ∀i ∈ V;          // s maps a vertex to a color
3  while NC ≠ {} do
4      choose a vertex v with the biggest degree of
       saturation in NC;
5      choose a color c ∈ 1..k with probability p according
       to Equation 1;
6      s(v) = c;
7      NC = NC \{v};
8  return s;   // return solution constructed
```

At each construction step, *Ant_Fixed_k* has to choose a vertex v non colored yet with the biggest degree of saturation[1] and choose a color c to assign v. The color c is chosen with probability p, presented in Equation 1, which is calculated based on pheromone trail τ, presented in Equation 2, and based on heuristic information η, presented in Equation 3.

$$p(s,v,c) = \frac{\tau(s,v,c)^{\alpha} \cdot \eta(s,v,c)^{\beta}}{\sum_{i \in \{1,\ldots,k\}} \tau(s,v,i)^{\alpha} \cdot \eta(s,v,i)^{\beta}} \quad (1)$$

where α and β are parameters of the algorithm and control the influence of the values associated to them in the equation, and

$$\tau(s,v,c) = \begin{cases} 1 & \text{if } C_c(s) = \{\} \\ \dfrac{\sum_{u \in C_c(s)} P_{uv}}{|C_c(s)|} & \text{otherwise} \end{cases} \quad (2)$$

[1]Degree of saturation is the number of different colors that were already assigned to the adjacent of a vertex.

$$\eta(s, v, c) = \frac{1}{|N_{C_c(s)}(v)|} \quad (3)$$

where P_{uv} is the pheromone trail between vertices u and v, explained next, $C_c(s)$ is the color class c of solution s, that is, the set of vertices already colored with c in that solution, and $N_{C_c(s)}(v)$ are the vertices $x \in C_c(s)$ adjacent to v in solution s.

The pheromone trail, stored on matrix $P_{|V|\times|V|}$, is initialized with 1 for each edge between non-adjacent vertices and with 0 for each edge between adjacent vertices. Its update involves the persistence of the current trail (by a ρ factor, meaning that $1 - \rho$ is the evaporation rate) and the reinforce by using the experience obtained by the ants (edges between pairs of non-adjacent nodes are reinforced when they receive the same color). The evaporation is presented in Equation 4 and the general form of depositing pheromone is presented in Equation 5.

$$P_{uv} = \rho P_{uv} \quad \forall u, v \in V \quad (4)$$

$$P_{uv} = P_{uv} + \frac{1}{f(s)} \quad \forall u, v \in C_c(s) \mid (u, v) \notin E, c = 1..k \quad (5)$$

where $C_c(s)$ is the set of vertices colored with c in solution s and f is the objective function, which returns the number of conflicting edges of that solution.

The mainly difference between the three versions of *ColorAnt* and *k-ANTCOL* (besides the use of a local search method) are:

- *k-ANTCOL*: utilizes all ants of colony to reinforce the pheromone trail;
- *ColorAnt*$_1$*-RT*: besides each ant of colony is used to reinforce the trail, the solution of best ant of colony in a cycle (s') and the solution of best ant so far in the execution (s^*) also reinforce it;
- *ColorAnt*$_2$*-RT*: only s' and s^* are used to reinforce the trail;
- *ColorAnt*$_3$*-RT*: s' and s^* do not reinforce the trail simultaneously, so that initially s' does it more often than s^*. A gradual exchange on this frequency is done based on the maximum number of cycles of the algorithm: at each interval of a fixed number of cycles, the amount of cycles in which s^* will reinforce the trail (instead of s') is increased by one.

The three *ColorAnt* algorithms utilize a local search method to improve the results of its solutions: the reactive tabu search *React-Tabucol* (RT) [32]. In *ColorAnt*$_1$*-RT* and *ColorAnt*$_2$*-RT*, the local search is applied only to the best ant of colony, at the end of a cycle. In *ColorAnt*$_3$*-RT*, the local search is applied to all ants of colony every cycle. *ColorAnt*$_3$*-RT* is presented in Algorithm 2.

The local search *React-Tabucol* is described next. Given the objective function f which returns the number of conflicting edges, a solution space S where each solution is a set of k color classes and all the vertices are colored (with or without conflicting edges) and a initial solution $s_0 \in S$, f must be minimized over S. A *move* consists in changing the color

Algoritmo 2 *ColorAnt*$_3$*-RT*.

```
COLORANT3-RT(G = (V, E), k)   // V: vertices; E: edges
 1  P_uv = 1  ∀(u,v) ∉ E;
 2  P_uv = 0  ∀(u,v) ∈ E;
 3  f* = ∞;      // best value for objective function so far
 4  while cycle < max_cycles and time < max_time
    and f* ≠ 0 do
 5      f' = ∞;      // best value function in a cycle
 6      for a = 1 to nants do
 7          s = ANT_FIXED_K(G, k);
 8          s = REACT_TABUCOL(G, k, s);
 9          if f(s) == 0 or f(s) < f' then
10              s' = s; f' = f(s');
11      if f' < f* then
12          s* = s'; f* = f(s*);
13      P_uv = ρP_uv  ∀u,v ∈ V;
            // according to Equation 4
14      if cycle mod √max_cycles == 0 then
15          phero_counter = cycle ÷ √max_cycles;
16      if phero_counter > 0 then
17          P_uv = P_uv + 1/f(s*)
                ∀u,v ∈ C_c(s*) | (u,v) ∉ E, c = 1..k;
                    // according to Equation 5
18      else
19          P_uv = P_uv + 1/f(s')
                ∀u,v ∈ C_c(s') | (u,v) ∉ E, c = 1..k;
                    // according to Equation 5
20      phero_counter = phero_counter−1;
21      cycle = cycle+1;
```

of only one vertex and it occurs between two *neighbors* solutions. When it is performed, the inverse of that move is stored in a *tabu list*, meaning that for the next tl (*tabu tenure*) iterations that move cannot be performed again. The next solution must be generated by a non-tabu move and it must have the minimum value of conflicts between all the possible neighbors solutions. *React-Tabucol* is presented in Algorithm 3.

The tabu tenure tl for *React-Tabucol* is a *reactive scheme*. Usually, a *dynamic* tabu tenure is used, which is a scheme that depends on the current solution and on the executed move. The reactive tabu tenure depends on the search history. When a solution is visited twice, the size of tl is modified [32].

B. The Spill Phase

George and Appel showed that the Briggs *et al.* conservative coalescing criterion could be relaxed to allow more aggressive coalescing without introducing extra spilling. Besides, they describes an algorithm that preserves coalesced nodes found before the potential spill was discovered. Our algorithm uses the same strategy for coalescing, but CA-RT-RA uses an different approach to choose the nodes in the graph that will be represented in memory.

Algoritmo 3 *React-Tabucol* algorithm, adapted from [38].

REACT-TABUCOL($G = (V, E)$, k, $s_0 = \{C_1, ..., C_k\}$)

```
1  s = s_0; s* = s; lista_tabu = {};
2  initialize tl;
3  while  stop conditions not found  do
4      choose a move (v, c) ∉ lista_tabu with the
       minimum value for δ(v, c);
                 // where δ(v, c) = f(s ∪ (v, c)) − f(s)
5      s = (s ∪ (v, c))\(v, s(v));
6      update tl according to reactive tabu scheme;
7      lista_tabu = lista_tabu ∪ {(v, s(v))};
                 // for tl iterations
8      if  f(s) < f(s*)  then
9          s* = s;
10 return  s*;
```

In George-Appel's algorithm, if there is no opportunity for *Simplify* or *Freeze*, the node will be spilled. In this case, the *Potential Spill* phase will calculate spill priorities for each node using the Equation 6.

$$P_n = \frac{(uses_{out} + defs_{out}) + 10 \times (uses_{in} + defs_{in})}{degree} \quad (6)$$

where $uses_{out}$ is the set of temporaries that the node uses outside a loop; $defs_{out}$ is the set of temporaries that it defines outside a loop; $uses_{int}$ is the set of temporaries that it uses within a loop; $defs_{in}$ is the set of temporaries that it defines within a loop; and $degree$ is the number of edges incident to the node.

The node that has the lowest priority will be select to be spilled first. George-Appel's approach is an optimistic approximation: the node removed from the graph does not interfere with any of the others nodes in the graph.

CA-RT-RA use a different approach to select a spill node. Since the resulting graph given by *ColorAnt$_3$-RT* phase may have conflicting edges, the *Spill* phase selects the node with more frequency in the set of conflicting ones, in other words, considering each color c, the node colored with c which has the biggest number of incident conflicting edges is removed from the graph and considered as an *actual spill*. If there is *actual spill*, the program will be rewritten as George-Appel's algorithm, and a new iteration will take place. Therefore, the algorithm finishes when there are no more conflicting edges in the graph.

V. EXPERIMENTAL RESULTS

The experiments are based on a compiler research framework. We have implemented George-Appels allocator [28], [16] and our proposed CA-RT-RA allocator on a compiler research framework that generates code to Intel's IA32 architecture. Also, the compilers were executed in a Intel Xeon E5620 of 2.40 GHz, 8GB RAM running Rocks Cluster Linux operating system.

The benchmark consists of fourteen programs from SNU-RT [39], except *Merge Sort*, and *Queens*. Table I outlines the programs, and their characteristics. For each program, we run the allocators ten times to measure the performance. The parameters of CA-RT-RA were chosen in a relatively arbitrary fashion. They are: $nants = 80$, $\alpha = 3$, $\beta = 16$, $\rho = 0.7$ and $max\_cycles = 625$. But in general, for small instances the best results are obtained for α smaller than β. The Tabu Search was limited by a maximum of 300 cycles. Our algorithm stops if there is no improvement in reducing the number of conflicting edges for more than $max\_cycles/4$.

TABLE I
PROGRAMS

Name	Characteristics		
	Function	Interference Graph	
		Nodes	Edges
Binary Search	bs	60	581
	main	173	1029
FFT	sin	50	382
	fft	306	5112
	main	38	288
FFT Complex	sin	49	370
	init_w	50	395
	fft	138	2303
	main	52	377
Fibonacci	fib	20	120
	main	11	40
FIR	sin	52	461
	sqrt	40	376
	fir_filter	53	498
	gaussian	53	455
	main	585	7456
Insert Sort	main	149	1285
Jfdctint	fdct	652	9555
	main	28	186
LMS	sqrt	40	376
	sin	52	461
	gaussian	53	455
	lms	145	2076
	main	70	956
Merge Sort	isdigit	28	149
	skipto	30	157
	readint	30	181
	readlist	22	124
	merge	51	386
	f	18	88
	printint	18	85
	printlist	21	106
	main	21	108
Quick Sort	sort	462	5199
	main	158	1065
Queens	print	36	261
	tree	117	1055
	main	27	140
Qurt	sqrt	38	329
	qurt	254	2081
	main	95	603
Select	select	416	5339
	main	154	1041
Sqrt	sqrt	38	329
	main	11	40

A. Spill and Fetch

The implementation of both algorithms attempts to minimize the number of spills. As it can be seen in Table II,

TABLE II
RESULTS OBTAINED BY CA-RT-RA AND APPEL'S ALGORITHM.

Program		CA-RT-RA								Appel	
Name	Function	Worst		Average		Best		Standard Deviation		Spill	Fetch
		Spill	Fetch	Spill	Fetch	Spill	Fetch	Spill	Fetch		
Binary Search	bs	17	17	15.500	16.000	16	15	0.707	0.816	28	27
	main	3	3	3.000	3.000	3	3	0.000	0.000	98	115
	Total	**20**	**20**	**18.500**	**19.000**	**19**	**18**	**0.707**	**0.816**	**126**	**142**
FFT	sin	14	26	10.800	20.600	10	17	1.398	3.471	11	22
	fft	67	116	65.500	114.900	64	110	2.273	6.500	67	117
	main	10	18	8.100	15.000	6	11	1.197	6.361	13	20
	Total	**91**	**160**	**84.400**	**150.500**	**80**	**138**	**3.204**	**7.307**	**91**	**159**
FFT Complex	sin	12	19	12.900	21.000	12	19	1.101	2.404	12	21
	init_w	12	25	6.100	9.300	5	7	2.424	5.736	17	20
	fft	30	52	30.000	52.000	30	52	0.000	0.000	29	51
	main	6	8	6.100	9.200	6	8	0.316	0.789	10	11
	Total	**60**	**104**	**55.100**	**91.500**	**53**	**86**	**2.601**	**5.778**	**68**	**103**
Fibonacci	fib	6	5	4.500	3.600	4	3	0.707	0.966	5	5
	main	0	0	0.000	0.000	0	0	0.000	0.000	0	0
	Total	**6**	**5**	**4.500**	**3.600**	**4**	**3**	**0.707**	**0.966**	**5**	**5**
FIR	sin	15	27	14.000	24.800	13	23	1.414	3.155	9	21
	sqrt	13	21	11.800	15.900	11	13	1.317	3.542	15	23
	fir_filter	16	24	14.000	20.900	12	19	1.826	2.424	13	19
	gaussian	5	11	5.000	11.000	5	11	0.000	0.000	18	22
	main	8	81	8.100	66.800	8	46	0.568	18.177	7	43
	Total	**57**	**164**	**52.900**	**139.400**	**49**	**112**	**2.685**	**18.404**	**68**	**128**
Insert Sort	main	16	36	13.200	32.900	12	32	1.229	1.370	21	39
	Total	**16**	**36**	**13.200**	**32.900**	**12**	**32**	**1.229**	**1.370**	**21**	**39**
Jfdctint	fdct	91	185	86.900	178.600	81	168	5.724	6.004	79	157
	main	14	14	7.500	8.000	8	10	2.759	2.449	8	8
	Total	**105**	**199**	**94.400**	**186.600**	**89**	**178**	**6.786**	**6.670**	**87**	**165**
LMS	sqrt	12	15	11.600	15.700	11	14	1.265	2.830	15	23
	sin	15	27	14.400	25.700	13	23	0.966	2.111	9	21
	gaussian	5	11	5.000	11.000	5	11	0.000	0.000	18	22
	lms	29	50	30.900	52.100	31	51	1.449	1.853	69	88
	main	27	50	24.800	32.700	23	28	1.989	6.800	25	32
	Total	**88**	**153**	**86.700**	**137.200**	**83**	**127**	**2.263**	**6.861**	**136**	**186**
Merge Sort	isdigit	6	5	6.000	5.000	6	5	0.000	0.000	6	5
	skipto	5	3	2.500	2.200	2	2	1.080	0.422	4	3
	readint	5	6	5.400	6.400	5	6	1.265	1.265	9	10
	readlist	5	6	3.200	3.300	3	3	0.632	0.949	5	4
	merge	6	16	6.000	16.100	6	16	0.000	0.316	6	18
	f	8	11	8.000	11.000	8	11	0.000	0.000	8	11
	printint	3	6	3.000	6.000	3	6	0.000	0.000	3	6
	printlist	5	7	5.000	7.000	5	7	0.000	0.000	7	11
	main	2	2	2.000	2.000	2	2	0.000	0.000	2	2
	Total	**45**	**62**	**41.000**	**58.800**	**40**	**58**	**2.108**	**1.687**	**50**	**70**
Quick sort	sort	48	105	39.500	101.200	38	98	3.408	2.530	50	116
	main	2	2	2.000	2.000	2	2	0.000	0.000	121	161
	Total	**50**	**107**	**41.500**	**103.200**	**40**	**100**	**3.408**	**2.530**	**171**	**277**
Queens	print	10	13	8.500	11.300	7	9	0.850	1.494	10	16
	tree	8	32	8.000	31.700	8	32	0.000	0.483	7	27
	main	1	1	1.000	1.000	1	1	0.000	0.000	1	1
	Total	**19**	**46**	**17.500**	**44.000**	**16**	**42**	**0.850**	**1.491**	**18**	**44**
Qurt	sqrt	11	18	8.700	11.900	8	11	0.949	2.234	12	19
	qurt	20	29	18.700	26.800	17	24	1.252	1.619	28	34
	main	2	2	2.000	2.000	2	2	0.000	0.000	55	73
	Total	**33**	**49**	**29.400**	**40.700**	**27**	**37**	**1.838**	**3.199**	**95**	**126**
Select	select	50	93	43.400	86.100	35	80	5.275	5.587	70	104
	main	2	2	2.000	2.000	2	2	0.000	0.000	121	161
	Total	**52**	**95**	**45.400**	**88.100**	**37**	**82**	**5.275**	**5.587**	**191**	**265**
Sqrt	sqrt	11	18	9.100	12.600	8	11	1.101	2.914	12	19
	main	0	0	0.000	0.000	0	0	0.000	0.000	0	0
	Total	**11**	**18**	**9.100**	**12.600**	**8**	**11**	**1.101**	**2.914**	**12**	**19**

our algorithm outperforms George-Appel's register allocator. Our proposed algorithm tends to spill less temporaries, because it tries to find the best approach to color the graph, so that the number of conflicting edges is zero. In this case, it is able to use less registers per function. It minimizes the function cost by reducing the amount of memory access instructions, instructions that typically have a higher cost when compared to other instructions classes. Also, because our algorithm tends to spill fewer temporaries and to use fewer registers in the allocation, it is able to find more opportunities for coalescing.

In thirteen applications, our algorithm achieves reductions from 2.778% to 85.3175% on number of spills. Only for one application the George-Appel's algorithm obtained better results, namely: `Jfdctint`. Besides, our algorithm achieves reductions from 11.165% to 86.620% on number of fetches. However, for fetches, the George-Appel's algorithm obtained best results for `FIR` and `Jfdctint`. In summary, only for one application our algorithm did not achieve a better performance than George-Appel's algorithm. It demonstrated that the strategy for selecting spill, used by our algorithm is a better approach to minimize the number of spill.

In the worst case, CA-RT-RA is still able to get better results than George-Apple's algorithm. On the other hand, it is necessary to run it several times to get the best result. This does not occur with the George-Appel's algorithm, because it has no random feature like CA-RT-RA. In other words, George-Appel's algorithm is deterministic, while CA-RT-RA provides a different solution for each run (nondeterministic). The ideal is to run the algorithm as many times as possible to ensure that good results are obtained.

The analysis of the interference graphs does not give some insight about the performance of our algorithm. Neither the number of nodes nor the number of edges influenced the performance, except for Binary Search.

All benchmarks spill some temporaries. Besides, the number of store instructions is almost equal to the number of fetch instructions, suggesting that the nodes that have been spilled may have just few definitions and uses.

B. Convergence

It is important to note that both algorithms are iterative, i.e., the register allocator ends only when there are no spills (see Figures 1 and 2 – *rebuild if there were actual spills*).

The Table III shows the convergence. For each interference graph is presented a list containing the amount of spills at each iteration and the list size.

The results demonstrated that CA-RT-RA finds a coloring that eliminates the number of spills in fewer iterations (rebuilding) than George-Appel's algorithm. In general, the number of iterations required by George-Apple's algorithm is up to 5 times the amount needed by CA-RT-RA.

The approach based on *defs-uses* used by George-Appel's algorithm causes a gradual decrease in the number of spills until this number reaches zero. On the other hand, the approach based on conflicts leads to a faster convergence.

CA-RT-RA does not need more than three rounds to finish, while George-Appel needs in many cases more than five rounds. Besides, some rounds do not minimize the number of spills, resulting in more iterations.

TABLE III
CONVERGENCE

Program		Algorithm	
Name	Function	CA-RT-RA	Appel
Binary Search	bs	[9,0](2)	[9,3,1,1,1,2,1,1,1,0](10)
	main	[3,0](2)	[18,16,16,16,16,0](6)
FFT	sin	[5,0](2)	[5,0](2)
	fft	[35,1,0](3)	[32,2,1,1,0](5)
	main	[6,0](2)	[6,3,2,0](4)
FFT Complex	sin	[5,1,0](3)	[5,0](2)
	init_w	[5,0](2)	[6,3,3,1,1,1,1,0](8)
	fft	[22,0](2)	[18,3,0](3)
	main	[4,2,0](5)	[4,2,2,2,0](5)
Fibonacci	fib	[2,1,0](3)	[2,1,1,0](4)
	main	[0](1)	[0](1)
FIR	sin	[6,0](2)	[6,0](2)
	sqrt	[8,1,0](3)	[8,1,1,0](4)
	fir_filter	[8,1,0](3)	[9,1,0](3)
	gaussian	[5,0](2)	[5,1,1,2,1,1,0](7)
	main	[8,0](2)	[7,0](2)
Insert Sort	main	[8,0](2)	[7,7,8,1,0](5)
Jfdctint	fdct	[33,0](2)	[24,0](2)
	main	[5,1,1,0](4)	[6,0](2)
LMS	sqrt	[8,0](2)	[8,1,1,0](4)
	sin	[6,0](2)	[6,0](2)
	gaussian	[5,0](2)	[5,1,1,2,1,1,0](7)
	lms	[22,3,1,0](4)	[18,10,9,10,5,3,1,0](8)
	main	[17,0](2)	[15,2,1,1,0](5)
Merge Sort	isdigit	[3,2,0](3)	[3,2,0](3)
	skipto	[2,0](2)	[2,1,0](3)
	readint	[3,2,0](2)	[3,2,1,0](4)
	readlist	[3,0](2)	[3,1,0](3)
	merge	[6,0](2)	[6,0](2)
	f	[2,1,1,0](4)	[2,1,1,0](4)
	printint	[2,1,0](3)	[2,1,0](3)
	printlist	[2,3,0](3)	[3,2,1,1,0](5)
	main	[2,0](2)	[2,0](2)
Quick Sort	sort	[15,0](2)	[16,2,2,2,2,2,0](7)
	main	[2,0](2)	[3,0](2)
Queens	print	[6,0](2)	[8,0](2)
	tree	[7,0](2)	[6,0](2)
	main	[1,0](2)	[1,0](2)
Qurt	sqrt	[7,0](2)	[7,0](2)
	qurt	[11,2,0](4)	[11,2,3,2,3,1,1,1,0](9)
	main	[2,0](2)	[10,9,9,9,9,0](5)
Select	select	[16,0](2)	[19,1,6,4,5,4,3,2,0](9)
	main	[2,0](2)	[21,20,20,20,20,0](6)
Sqrt	sqrt	[7,0](2)	[7,0](2)
	main	[0](1)	[0](1)

C. Code Size

Table IV shows the number of assembly instructions and code size in bytes for each program.

The reduction of the number of assembly instructions ranges from 8.511% to 35.352%, which causes a reduction in the code size between 1.76% and 20.559%. Therefore, these results are very important for systems that uses embedded microprocessors, due to the fact that their components usually consist of limited computational power and limited memory.

Note that the traditional goal of a compiler is either to generate code that results on improved processor performance,

TABLE IV
CODE SIZE

Program	CA-RT-RA		Appel	
	Assembly Instructions	Code Size (Bytes)	Assembly Instructions	Code Size (Bytes)
Binary Search	523	4884	809	6148
FFT	991	7596	1021	7720
FFT Complex	797	6276	840	6412
Fibonacci	43	860	47	872
FIR	1732	15104	1759	15192
Insert Sort	337	3496	358	3564
Jfdctint	1525	10276	1501	10204
LMS	867	6352	1000	6996
Merge Sort	501	4424	519	4488
Quick sort	1314	12132	1676	13840
Queens	398	3740	398	3740
Qurt	802	7496	981	8180
Select	1216	11392	1618	13324
Sqrt	108	1436	119	1472

TABLE V
COMPILATION TIME

Program	CA-RT-RA		Appel	
	Average	SD	Average	SD
Binary Search	26.059	0.322	0.168	0.001
FFT	178.103	53.394	0.581	0.004
FFT Complex	43.351	4.461	0.213	0.008
Fibonacci	2.431	0.483	0.448	0.004
FIR	235.817	18.697	1.317	0.053
Insert Sort	22.495	7.287	0.110	0.000
Jfdctint	634.870	386.634	1.402	0.045
LMS	84.079	11.072	0.358	0.002
Merge Sort	21.629	1.313	0.106	0.001
Quick sort	327.153	242.891	1.180	0.000
Queens	21.322	3.621	0.112	0.002
Qurt	144.351	45.332	0.238	0.006
Select	363.318	180.627	1.510	0.065
Sqrt	3.413	0.019	0.047	0.008

or to minimize the compilation time to an acceptable level of processor performance. Wireless Sensor Netwoks (WSN) [40], [41], [42], on the other hand, often require careful attention towards program storage, memory restrictions and energy consumption.

In this context, we consider that memory constraints should be addressed, i.e., generating compact code is not an option. Generating such code depends on the algorithm, the language, and the actual compiler. The algorithm determines the number of hardware instructions executed. The programming language affects the instruction count, since statements are translated to hardware instructions. The efficiency of the compiler affects both the instruction count and average cycles per instructions, since the compiler determines the translation of the source language instructions into hardware instructions [1], [6]. During the strategy used to translate the source code into native code, the compiler can save storage space by using optimizations, such as register allocation.

A traditional system software development goal is concerned in minimizing the execution time to an acceptable level. A WSN development, in contrast, often requires careful attention towards program storage space. Therefore, in this context, performance is also related to memory and storage restrictions, besides processor performance.

CA-RT-RA is a good algorithm to address these issues.

D. Runtime

Table V shows the compilation time of both algorithms.

George-Appel's algorithm is faster than CA-RT-RA from 5.426 to 606.517 times. It is a problem when the compilation time should be address, for example, in dynamic systems. On the other hand, in a standalone compilation system, the compilation time should not be a problem.

These results also demonstrated the instability of ACO algorithm. Note that the standard deviation is very high.

A relatively high runtime is usually a problem on ACO algorithms. Although these algorithms are able to find satisfactory solutions to many problem, the runtime is a cost that must be paid. Thus, many researchers use different approaches avoiding ACO algorithms.

Note that CA-RT-RA is able to reduce the number of spills. This reduction eliminates the clock cycles and code size, which is a very important issue in WSN. Although CA-RT-RA has a very high runtime, it is able to address several goals, such as: reduce code size, reduce the number of memory accesses, and consequently reduce the amount of energy needed.

VI. CONCLUSION AND FUTURE WORK

Register allocation determines what values in a program must reside in registers. Instructions involving register operands are faster than those involving memory access. Therefore, register allocation is a very important compiler optimization technique.

Register allocation problem can be mapped as a graph coloring problem, which is $\mathcal{NP}$-complete. Register allocators based on graph coloring algorithm apply some heuristic method to find a good coloring. But these algorithm do not guarantee that the coloring is the best.

ACO algorithms are able to provide excellent solutions, but at a cost of a high runtime. Therefore, there is a tradeoff here: solution quality versus compile time.

In this paper we presented the CA-RT-RA algorithm, an algorithm for graph coloring register allocation. Compared to a classic graph coloring register allocation algorithm, CA-RT-RA can generate better object code.

There are several research directions in the future. First, we are interested in comparing our algorithm with other graph coloring algorithms. Second, we will focus on implementing our algorithm in GCC-AVR compiler.

REFERENCES

[1] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[2] D. A. Patterson and J. L. Hennessy, *Computer Organization And Design: The Hardware/software Interface*. The Morgan Kaufmann, 2008.

[3] W. Stallings, *Computer Organization and Architecture*. Prentice Hall, 2010.

[4] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation via Coloring," *Computer Languages*, vol. 6, no. 1, pp. 47 – 57, 1981.

[5] F. Glover and M. Laguna, *Tabu Search*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[6] C. N. Fischer, R. K. Cytron, and R. J. LeBlanc, *Crafting a Compiler*. Addison Wesley, 2010.

[7] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive Optimization in the Jalapeno JVM," *SIGPLAN Notices*, vol. 46, pp. 65–83, May 2011.

[8] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani, "Effectiveness of Cross-platform Optimizations for a Java Just-in-Time Compiler," *SIGPLAN Notices*, vol. 38, pp. 187–204, October 2003.

[9] T. Suganuma, T. Ogasawara, K. Kawachiya, M. Takeuchi, K. Ishizaki, A. Koseki, T. Inagaki, T. Yasue, M. Kawahito, T. Onodera, H. Komatsu, and T. Nakatani, "Evolution of a Java Just-in-Time Compiler for IA-32 Platforms," *IBM Journal of Research and Development*, vol. 48, pp. 767–795, September 2004.

[10] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21, pp. 895–913, September 1999.

[11] E. Johansson and K. F. Sagonas, "Linear Scan Register Allocation in a High-Performance Erlang Compiler," in *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, ser. PADL '02. London, UK, UK: Springer-Verlag, 2002, pp. 101–119.

[12] H. Mössenböck and M. Pfeiffer, "Linear Scan Register Allocation in the Context of SSA Form and Register Constraints," in *Proceedings of the International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2002, pp. 229–246.

[13] C. Wimmer and H. Mössenböck, "Optimized Iinterval Splitting in a Linear Scan Register Allocator," in *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2005, pp. 132–141.

[14] M. D. Smith, N. Ramsey, and G. Holloway, "A generalized algorithm for graph-coloring register allocation," *SIGPLAN Not.*, vol. 39, pp. 277–288, June 2004.

[15] K. D. Cooper and A. Dasgupta, "Tailoring graph-coloring register allocation for runtime compilation," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 39–49.

[16] A. W. Appel, *Modern Compiler Implementation in C*. New York, NY, USA: Cambridge University Press, 1998.

[17] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds. New York, NY, EUA: Plenum Press, 1972, pp. 85–103.

[18] M. Dorigo, M. Birattari, and T. Sttzle, "Ant Colony Optimization - Artificial Ants as a Computational Intelligence Technique," *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, 2006.

[19] M. Dorigo and T. Sttzle, *Ant Colony Optimization*, ser. Bradford Books. Cambridge, Massachusetts: MIT Press, 2004.

[20] C. N. Lintzmayer, M. H. Mulati, and A. F. da Silva, "Toward Better Performance of ColorAnt ACO Algorithm," in *XXX International Conference of the Chilean Computer Science Society*, Curico, Chile, 2011.

[21] P. Marwedel, *Embedded System Design*. Springer Verlag, 2010.

[22] M. Wolfe, "How Compilers and Tools Differ for Embedded Systems," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY, USA: ACM, 2005, pp. 1–1.

[23] G. J. Chaitin, "Register Allocation & Spilling via Graph Coloring," *SIGPLAN Notices*, vol. 17, pp. 98–101, June 1982.

[24] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe, "Spill code minimization via interference region spilling," *SIGPLAN Not.*, vol. 32, pp. 287–295, May 1997.

[25] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon, "Spill Code Minimization Techniques for Optimizing Compilers," *SIGPLAN Notices*, vol. 24, pp. 258–263, June 1989.

[26] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 16, pp. 428–455, May 1994.

[27] J. Runeson and S.-O. Nyström, "Retargetable Graph-Coloring Register Allocation for Irregular Architectures," in *Proceedings of the Software and Compilers for Embedded Systems*. Springer, 2003, pp. 22–8.

[28] L. George and A. W. Appel, "Iterated register coalescing," *ACM Transactions on Programming Languages and Systems*, vol. 18, pp. 300–324, May 1996.

[29] J.-M. Daveau, T. Thery, T. Lepley, and M. Santana, "A Retargetable Register Allocation Framework for Embedded Processors," *SIGPLAN Notices*, vol. 39, pp. 202–210, June 2004.

[30] S. Wu and S. Li, "Extending Traditional Graph-Coloring Register Allocation Exploiting Meta-heuristics for Embedded Systems," in *Proceedings of the Third International Conference on Natural Computation*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 324–329.

[31] D. Ashlock, *Evolutionary Computation for Modeling and Optimization*. Springer, 2005.

[32] I. Blöchliger and N. Zufferey, "A Graph Coloring Heuristic Using Partial Solutions and a Reactive Tabu Scheme," *Computers & Operations Research*, vol. 35, pp. 960–975, Mar. 2008.

[33] A. Hertz and D. Werra, "Using Tabu Search Techniques for Graph Coloring," *Computing*, vol. 39, pp. 345–351, 1987.

[34] C. N. Lintzmayer, M. H. Mulati, and A. F. da Silva, "RT-ColorAnt: Um Algoritmo Heurístico Baseado em Colônia de Formigas Artificiais com Busca Local para Colorir Grafos," in *XLIII Simposio Brasileiro de Pesquisa Operacional 2011*, Ubatuba, SP, BRA, 2011.

[35] ——, "Algoritmo Heurístico Baseado em Colônia de Formigas Artificiais ColorAnt2 com Busca Local Aplicado ao Problema de Coloração de Grafo," in *X Congresso Brasileiro de Inteligência Computacional*, Fortaleza, SP, BRA, 2011.

[36] ——, "Toward Better Performance of ColorAnt ACO Algorithm," in *XXX International Conference of the Chilean Computer Science Society*, Curico, Chile, 2011.

[37] D. Costa and A. Hertz, "Ants Can Colour Graphs," *The Journal of the Operational Research Society*, vol. 48, no. 3, pp. 295–305, 1997.

[38] A. Hertz and N. Zufferey, "A New Ant Algorithm for Graph Coloring," in *Workshop on Nature Inspired Cooperative Strategies for Optimization NICSO*. Granada, Espanha: David Alejandro Pelta and Natalio Krasnogor, 2006, pp. 51–60.

[39] F. Group, "SNU Real-Time Benchmarks," http://www.cprover.org/goto-cc/examples/snu.html.

[40] I. F. Akyildiz and M. C. Vuran, *Wireless Sensor Networks*. John Wiley & Sons Inc, 2010.

[41] M. Ilyas and I. Mahgoub, Eds., *Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems*. CRC PRESS, 2004.

[42] A. Hać, *Wireless Sensor Network Designs*. San Francisco, CA, USA: John Wiley, 2003.