2012 31st International Conference of the Chilean Computer Science Society

Register Allocation by Evolutionary Algorithm

Carla Négri Lintzmayer
University of Campinas – Brazil
Email: carla0negri@gmail.com

Mauro Henrique Mulati
Midwestern State University – Brazil
Email: mhmulati@gmail.com

Anderson Faustino da Silva
State University of Maringá – Brazil
Email: anderson@din.uem.br

***Abstract*—Graph coloring is a highly effective approach to intraprocedural register allocation. In this paper, we describe a new algorithm for intraprocedural register allocation called HECRA, an algorithm that extends a classic graph coloring register allocator to use a hybrid evolutionary coloring algorithm. The experiments demonstrated that our algorithm is able to minimize the amount of spills, thereby improving the quality of the generated code. Besides, HECRA is interesting in contexts where compile time is a concern, and not only the quality of the generated code.**

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

I. Introduction

Register allocation is an important compiler optimizations and can be mapped as a graph coloring problem (GCP) [1], [2]. GCP essentially consists in finding the minimum value of k for which a graph is k-colorable. But, the k-GCP consists in trying to color a graph with k fixed colors by minimizing the amount of conflicts (adjacent vertices assigned with 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).

Finding a solution for k-GCP is a $\mathcal{NP}$-hard [3] problem. No exact algorithm in polynomial time is known, encouraging the use of alternative techniques to find good solutions for huge instances. This has resulted in several papers that exploit heuristic algorithms and metaheuristics [4], [5], [6].

In our previous paper, we investigated the application of the Ant Colony Optimization (ACO) metaheuristic [7] to the register allocation problem, leading us to develop *ColorAnt*$_3$*-RT* Register Allocator (CARTRA) [8]. CARTRA extends George-Appel Register Allocator (George-Appel) [9] to use our ACO-based algorithm *ColorAnt*$_3$*-RT* [10]. But, our results demonstrated that CARTRA is useful only in situations where compile time is not a concern, but code quality, such as compiler that generates code to embedded systems [11].

In this paper, we present our new register allocator – Hybrid Evolutionary Coloring Register Allocator (HECRA). HECRA also extends George-Appel, but in order to add a hybrid evolutionary algorithm [12]. Then, CARTRA and HECRA use George-Appel as a framework, where the main phase is a heuristic algorithm. The goal in developing HECRA is to obtain a register allocator that use an aggressive heuristic to color the interference graph but with short compile time, what is not the case of CARTRA. Although, the previous results demonstrated that CARTRA outperforms George-Appel in number of spills, this is achieved by a high compile time. The results with HECRA demonstrate that it has a similar performance to CARTRA in number of spills, but outperforms CARTRA when compile time is a concern.

This paper is organized as follows. Section II presents some related works. Section III describes the George-Appel Register Allocator. Section IV describes the CARTRA Register Allocator. Section V describes the HECRA Register Allocator. Section VI presents the results obtained with HECRA, CARTRA, and George-Appel. And, Section VII presents the conclusions and future works.

II. Related Works

Graph Coloring Register Allocation (GCRA) was proposed by Chaitin *et al.* [1]. 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. The most successful design for GCRA was developed by Briggs *et al.* [13]. 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 Chaitin's allocator, which allows it to be used for irregular architectures [14]. This work is an interesting framework for a retargetable graph-coloring allocator.

George and Appel [9] 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.* [15] presented a register allocation framework designed to address the embedded processor specificities, such as smaller number of registers, irregular and constrained

1522-4902/13 $31.00 © 2013 IEEE
DOI 10.1109/SCCC.2012.31

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 [16] proposed a hybrid metaheuristic algorithm for GCRA that combines several ideas from classic GCRA algorithms, besides evolutionary algorithms [17] and Tabu Search [18]. 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 George-Appel Register Allocator

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 [9]. 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 assumption in this approach is that the compiler is free to generate new temporaries and copies, because almost all copies will be coalesced. The phases of George-Appel register allocator are:

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.

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.

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.

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.

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.

Select *Select* removes 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 are no more nodes in the stack.

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.

The phases of George-Appel are organized as demostrated in Figure 1.

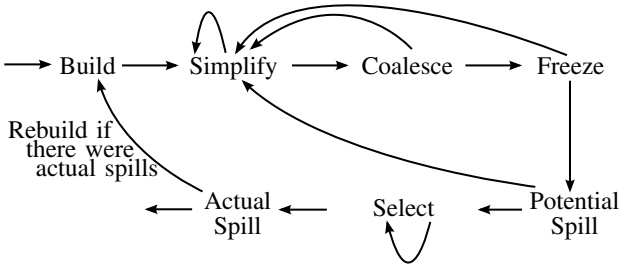

Fig. 1. The George-Appel Register Allocator [9]

IV. The ColorAnt$_3$-RT Register Allocator

CARTRA 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 3 shows the phases of CARTRA.

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

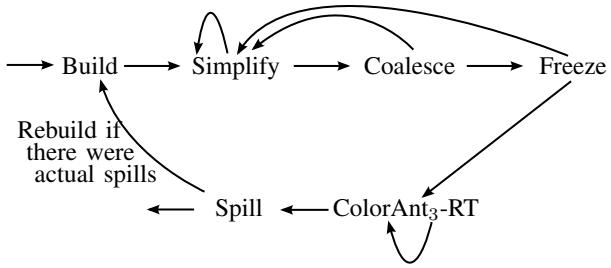

Fig. 2. *ColorAnt$_3$-RT* Register Allocator

GCRA problem. Firstly, we implemented an algorithm that was able to obtain satisfactory solutions, with respect to reducing the amount of conflicts [19]. But, our investigation demonstrated that changing the way to reinforce the pheromone trail results in a greater reduction in the amount of conflicts [20], [10]. Due to this fact, our research group has developed three *ColorAnt-RT* algorithms. The CARTRA uses our best version: *ColorAnt$_3$-RT*.

The three *ColorAnt-RT* algorithms use as constructive method (for each ant) an algorithm suggested along with *ANTCOL* [21], 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*.

Algorithm 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 $s(i) = 0 \quad \forall i \in V$; // s maps a vertex to a color
3 **while** $NC \neq \{\}$ **do**
4 choose a vertex v with the biggest degree of saturation in NC;
5 choose a color $c \in 1..k$ with probability p according to Equation 1;
6 $s(v) = c$;
7 $NC = NC \setminus \{v\}$;
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.

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

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

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

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

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

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-RT* are:

- *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-RT* algorithms utilize a local search method to improve the results of its solutions: the reactive tabu search *React-Tabucol* (RT) [22]. 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.

Algorithm 2 *ColorAnt$_3$-RT*.

COLORANT$_3$-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;
```

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 describe an algorithm that preserves coalesced nodes found before the potential spill was discovered. Our algorithm uses the same strategy for coalescing, but CARTRA uses an different approach to choose the nodes in the graph that will be represented in memory.

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.

CARTRA uses 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. THE HYBRID EVOLUTIONARY COLORING REGISTER ALLOCATOR

Hybrid Evolutionary Coloring Register Allocator (HECRA), as CARTRA, modifies George-Appel in order to add a metaheuristic phase. While CARTRA is based on ACO algorithm, HECRA is based on Hybrid Evolutionary Algorithm (HCA) based on local search and a highly specialized crossover operator [12]. Both allocators use the same strategy to select spills. Therefore, they have the same structure but with a different coloring phase, while HECRA uses HCA as coloring phase CARTRA uses *ColorAnt$_3$-RT*. Thus, Figure 3 shows the phases of HECRA.

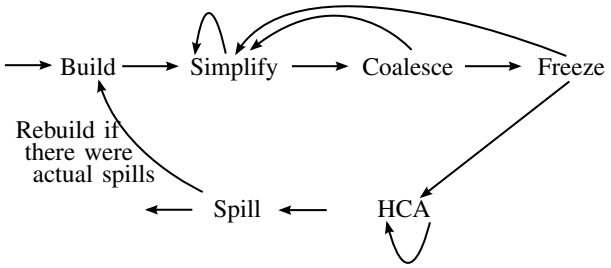

Fig. 3. Hybrid Evolutionary Coloring Register Allocator

The motivation to develop HECRA was to investigate a different strategy to coloring the interference graph that was able to reduce the compile time. Although, the results with CARTRA demonstrated that it is able to reduce the amount of spills, we need to pay the price of a high compile time [8].

A. The HCA Phase

The HCA phase begins with a population and an iterative process is repeated for a number of generations. In the members of the population (*parents*) the crossover operator is applied to generate a new configuration (*child*), in which the local search will be applied to improve it before it be returned to the population (instead of having a mutation). The HCA is as shown in Algorithm 3.

Algorithm 3 *HCA*, adapted of [12].

HCA($G = (V, E)$, k, p)
1 P = INITIAL_POPULATION(G, p);
2 **while** not(reached stop criteria) **do**
3 (s_1,s_2) = PICK_PARENTS(P);
4 s = CROSSOVER(s_1, s_2);
5 s = TABUCOL$_{HCA}$(s, L);
6 **if** $f(s_1) > f(s_2)$ **then**
// *the worst parents are replaced*
7 $P = (P \backslash \{s_1\}) \cup s$;
8 **else**
9 $P = (P \backslash \{s_2\}) \cup s$;
10 Find the best solution s^* in final population P;
11 **return** s^*;

HCA works with a fixed population of size p and its structure is described below. An initial population is constructed according to the Algorithm 4. In each generation, two parents configurations are chosen randomly by the method *Pick_Parents* and in these are applied the crossover operator. The configuration is enhanced by the generated children applying a local search, *Tabucol*$_{HCA}$, for a fixed number of L iterations before it is included in the population, replacing the worst parent.

Algorithm 4 HCA's Initial Population

INITIAL_POPULATION($G = (V, E)$, p)
1 $P = \{\}$;
2 **for** $i = 1$ **to** p **do**
3 $ncolor = 0$;
4 **while** $ncolor < |V|$ **do**
5 choose vertex v not colored with major saturation degree; if possible, c will color with the minor color; if not, c is chosen randomly among 1 and k;
6 $C_c = C_c \cup \{v\}$;
7 $ncolor$++;
8 $s = \{C_1, ..., C_k\}$;
9 s = TABUCOL$_{HCA}$(s, L);
10 $P = P \cup s$;
11 **return** P;

The crossover operator builds k color classes of children. For each color c, one parent is chosen (alternately) and in this is chosen the largest class of vertices to become the class C_c of the child. The vertices of this class are removed from both parents and, ultimately, the vertices not yet colored receive a color randomly. The crossover operator is presented in Algorithm 5, in which C_i^A represents the C_i class of the father A.

The population diversity is calculated as the average distance among all individuals. The distance between the two solutions is the number of changes that must be made so that it is equal to another. Such as, in a graph with three vertices, and a solution $s_1 = \{(0,1),(1,2),(2,1)\}$ (i.e., the vertex 0 is colored with color 1, and so on) and another solution $s_2 = \{(0,1),(1,2),(2,2)\}$, the distance between s_1 and s_2 is one, because just changing the color of the vertex 2.

Algorithm 5 HCA's Crossover Operator

CROSSOVER($G = (V, E)$, $s_1 = \{C_1^1, ..., C_k^1\}$, $s_2 = \{C_1^2, ..., C_k^2\}$)
1 **for** $c = 1$ **to** k **do**
2 **if** $c \mod 2 == 1$ **then** $A = 1$;
3 **else** $A = 2$;
4 choose i which $|C_i^A|$ is the major;
5 $C_c = C_i^A$;
6 remove vertices in C_i^A of s_1 and s_2;
7 color randomly the vertices $V \backslash (C_1 \cup ... \cup C_k)$;
8 $s = \{C_1, ..., C_k\}$;
9 **return** s;

B. The Tabucol$_{HCA}$

The local search used by HCA, *Tabucol*$_{HCA}$, is an improved tabu local search algorithm [12] and is presented in Algorithm 6.

Algorithm 6 *Tabucol* algorithm, adapted from [23].

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 $(v, c) \notin lista\_tabu$ with the minimum value for $\delta(v, c)$;
// where $\delta(v, c) = f(s \cup (v, c)) - f(s)$
5 $s = (s \cup (v, c)) \backslash (v, s(v))$;
6 update tl according to reactive tabu scheme;
7 $lista\_tabu = lista\_tabu \cup \{(v, s(v))\}$;
// for tl iterations
8 **if** $f(s) < f(s^*)$ **then**
9 $s^* = s$;
10 **return** s^*;

The Tabucol local search is as described below. With an objective function f which returns the number of conflicting edges, a solution space S which each solution is formed by k color classes and all vertices are colored (probably with conflicting edges) and an initial solution $s_0 \in S$, f should be minimized on S. At each iteration, the best neighbor solution is chosen to replace the current solution. A neighbor solution is obtained by moving a vertex v of its current color class ($C_{s(v)}$) to a new class C_c, this *moving* is represented by (v, c). The vertex v must be conflicting with at least one vertex that belongs to its class. When the movement (v, c) occurs, the pair $(v, s(v))$ is classified as *tabu* for the next tl iterations,

ensuring that v does not belong the class $C_{s(v)}$ in this period again.

This search then generates a sequence $s_1, s_2, \ldots, s_n$ of solutions S, where s_{i+1} is nearby s_i, must be generated by a non-tabu moving and must have the least number of conflicts between possible solutions of s_i, unless it leads to an objective function value better than the best found so far during the search (aspiration criterion).

The tl parameter is called *tabu tenure*, or, sometimes, *tabu list length*. It uses a dynamic scheme of tabu tenure, which value depends on the current solution and the movement that was executed. The *dynamic tabu tenure* is given by $tl = \alpha f(s) + \text{RANDOM}(A)$, which A and α are algorithm parameters. The adjustment of the *tabu tenure* depends on the function order evolves during the search. Three parameters aid in this update: φ (frequency), η (increase) and δ (threshold). Each φ iterations δ is determined, which is the difference between the maximum and minimum objective function values achieved in the last φ iterations. The new tl value is:

$$tl = \begin{cases} tl + \eta & \text{if } \delta \leq \delta \\ tl - 1 & \text{otherwise} \end{cases}$$

The basic difference between CARTRA's local search and HCRA's local search is that Tabucol$_{HCA}$ uses *dynamic tabu tenure*, while React-Tabucol uses *reactive tabu tenure*, which value is determined based on the search history. Therefore, the difference between two local search algorithms is the tl update mechanism. In React-Tabucol tl is updated according the reactive tabu scheme, and in Tabucol$_{HCA}$ tl is updated according to a dynamic scheme.

VI. Results and Discussion

We conducted a series of experiments to evaluate HECRA. To perform such experiments, we add HECRA in a research compiler framework in which our previous work we implemented CARTRA and George-Appel. It compiler framework generates code to Intel's IA32, and was executed in a Intel Xeon E5620 of 2.40 GHz, 8GB RAM running Rocks Cluster Linux operating system.

The benchmark consists of eleven programs from SNU-RT [24], and *Queens*. For each program, we run the allocators ten times to measure the performance. The parameters of CARTRA are: $nants = 80$, $\alpha = 3$, $\beta = 16$, $\rho = 0.7$ and $max\_cycles = 625$, and the tabu search was limited by a maximum of 300 cycles. CARTRA stops if there is no improvement in reducing the number of conflicting edges for more than $max\_cycles/4$. And, three parameters of HECRA are: $p = 10$, $L = 2000$, $max\_cycles =$, $diversity = 20$, and the tabu search was limited by a maximum of 2000 cycles.

We conduct several experiments to measure the performance of our algorithm. The experiments have the following goals: (1) measure the amount of spills; (2) analyse the convergence; (3) measure the code size; and (4) analyse the compilation time. Besides, we want to analyse if the amount of nodes influences the result. The next sections present our experiments and some discussions.

A. Compilation Time

Table III shows the compilation time of all allocators. George-Appel is faster than CARTRA from 5.43 to 606.52 times. George-Appel is also faster than HECRA, but in this case from 0.19 to 3.08 times. In the context which compilation time should be address our allocators can be a problem, for example, in dynamic systems. On the other hand, in a standalone compilation system, the compilation time should not be a problem.

TABLE III
Compilation Time

Program	HECRA		CARTRA		Appel	
	Average	SD	Average	SD	Average	SD
Binary Search	0.272	0.005	26.059	0.322	0.168	0.001
FFT	0.657	0.037	43.351	4.461	0.213	0.008
Fibonacci	0.087	0.020	2.431	0.483	0.448	0.004
FIR	1.421	0.034	235.817	18.697	1.317	0.053
Insert Sort	0.285	0.106	22.495	7.287	0.110	0.000
Jfdctint	1.705	0.360	634.870	386.634	1.402	0.045
LMS	0.810	0.089	84.079	11.072	0.358	0.002
Quick sort	1.326	0.247	327.153	242.891	1.180	0.000
Queens	0.339	0.206	21.322	3.621	0.112	0.002
Qurt	0.674	0.059	144.351	45.332	0.238	0.006
Select	1.327	0.079	363.318	180.627	1.510	0.065
Sqrt	0.079	0.000	3.413	0.019	0.047	0.008

These results also demonstrated the instability of ACO algorithm. Note that the standard deviation (SD) 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 problems, the runtime is a cost that must be paid in some cases.

Note that CARTRA 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 CARTRA 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.

It is very important to note that HECRA is able to address the goals that CARTRA addresses, besides minimizing the compilation time. HECRA is faster than CARTRA from 27.94 to 372.36 times. These results demonstrated that changing the strategy for coloring the interference graph the new allocator was able to maintain the code quality, but in a low compilation time.

B. Spill and Fetch

The implementation of both allocators attempts to minimize the number of spills. As it can be seen in Table I, our allocators outperform George-Appel's register allocator. Our proposed allocators tend to spill less temporaries, because they try to find the best approach to color the graph, so that the number of conflicting edges is zero. In this case, they are able to use less registers per function. They minimize 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 allocators tend to spill fewer temporaries and to use fewer registers in

TABLE I
RESULTS OBTAINED BY HECRA, CARTRA AND GEORGE-APPEL.

Program		HECRA						CARTRA						George-Appel	
Name	Function	Worst		Average		Best		Worst		Average		Best		S	F
		S	F	S	F	S	F	S	F	S	F	S	F		
Binary Search	bs	17	18	15.3	16.5	14	15	17	17	15.5	16.0	16	15	28	27
	main	3	3	3.0	3.0	3	3	3	3	3.0	3.0	3	3	98	115
	Total	**20**	**21**	**18.3**	**19.5**	**17**	**18**	**20**	**20**	**18.5**	**19.0**	**19**	**18**	**126**	**142**
FFT	sin	15	24	12.9	21.3	12	19	12	19	12.9	21.0	12	19	12	21
	init_w	8	12	6.1	8.5	5	7	12	25	6.1	9.3	5	7	17	20
	fft	33	57	31.0	55.2	30	53	30	52	30.0	52.0	30	52	29	51
	main	7	10	6.1	8.6	6	7	6	8	6.1	9.2	6	8	10	11
	Total	**63**	**103**	**56.1**	**84.6**	**53**	**86**	**60**	**104**	**55.1**	**91.5**	**53**	**86**	**68**	**103**
Fibonacci	fib	9	8	5.5	4.3	4	3	6	5	4.5	3.6	4	3	5	5
	main	0	0	0.0	0.0	0	0	0	0	0.0	0.0	0	0	0	0
	Total	**9**	**8**	**5.5**	**4.3**	**4**	**3**	**6**	**5**	**4.5**	**3.6**	**4**	**3**	**5**	**5**
FIR	sin	15	25	10.9	21.3	9	17	15	27	14.0	24.8	13	23	9	21
	sqrt	12	19	9.4	13.4	9	11	13	21	11.8	15.9	11	13	15	23
	fir_filter	25	33	18.9	26.6	13	19	16	24	14.0	20.9	12	19	13	19
	gaussian	13	16	8.0	10.6	6	8	5	11	5.0	11.0	5	11	18	22
	main	8	82	7.1	72.5	6	42	8	81	8.1	66.8	8	46	7	43
	Total	**73**	**175**	**54.3**	**144.4**	**43**	**97**	**57**	**164**	**52.9**	**139.4**	**49**	**112**	**68**	**128**
Insert Sort	main	19	38	15.6	35.0	13	32	16	36	13.2	32.9	12	32	21	39
	Total	**19**	**38**	**15.6**	**35.0**	**13**	**32**	**16**	**36**	**13.2**	**32.9**	**12**	**32**	**21**	**39**
Jfdctint	fdct	99	191	89.3	183.5	84	175	91	185	86.9	178.6	81	168	79	157
	main	11	11	9.2	9.6	7	6	14	14	7.5	8.0	8	10	8	8
	Total	**110**	**202**	**98.5**	**193.1**	**91**	**181**	**105**	**199**	**94.4**	**186.6**	**89**	**178**	**87**	**165**
LMS	sqrt	10	13	9.2	12.8	9	11	12	15	11.6	15.7	11	14	15	23
	sin	213	221	35.7	46.2	9	20	15	27	14.4	25.7	13	23	9	21
	gaussian	12	15	8.5	12.1	6	10	5	11	5.0	11.0	5	11	18	22
	lms	34	60	32.5	56.0	30	53	29	50	30.9	52.1	31	51	69	88
	main	24	34	22.3	29.3	21	27	27	50	24.8	32.7	23	28	25	32
	Total	**293**	**343**	**108.2**	**156.4**	**75**	**121**	**88**	**153**	**86.7**	**137.2**	**83**	**127**	**136**	**186**
Quick Sort	sort	46	107	41.7	103.4	37	101	48	105	39.5	101.2	38	98	50	116
	main	2	2	2.0	2.0	2	2	2	2	2.0	2.0	2	2	121	161
	Total	**48**	**109**	**43.7**	**105.4**	**39**	**103**	**50**	**107**	**41.5**	**103.2**	**40**	**100**	**171**	**277**
Queens	print	9	13	8.8	11.8	8	10	10	13	8.5	11.3	7	9	10	16
	tree	9	32	8.2	29.6	8	29	8	32	8.0	31.7	8	32	7	27
	main	1	1	1.0	1.0	1	1	1	1	1.0	1.0	1	1	1	1
	Total	**19**	**46**	**18**	**42.4**	**17**	**40**	**19**	**46**	**17.5**	**44.0**	**16**	**42**	**18**	**44**
Qurt	sqrt	13	20	10.0	14.2	9	13	11	18	8.7	11.9	8	11	12	19
	qurt	21	30	19.3	27.8	16	25	20	29	18.7	26.8	17	24	28	34
	main	2	2	2.0	2.0	2	2	2	2	2.0	2.0	2	2	55	73
	Total	**36**	**52**	**31.3**	**44.0**	**27**	**40**	**33**	**49**	**29.4**	**40.7**	**27**	**37**	**95**	**126**
Select	select	54	96	46.1	88.6	40	83	50	93	43.4	86.1	35	80	70	104
	main	2	2	2.0	2.0	2	2	2	2	2.0	2.0	2	2	121	161
	Total	**56**	**98**	**48.1**	**90.6**	**42**	**85**	**52**	**95**	**45.4**	**88.1**	**37**	**82**	**191**	**265**
Sqrt	sqrt	12	20	9.7	14.0	8	10	11	18	9.1	12.6	8	11	12	19
	main	0	0	0.0	0.0	0	0	0	0	0.0	0.0	0	0	0	0
	Total	**12**	**20**	**9.7**	**14.0**	**8**	**10**	**11**	**18**	**9.1**	**12.6**	**8**	**11**	**12**	**19**

the allocation, they are able to find more opportunities for coalescing.

In ten applications, our allocators achieve reductions from 0% to 85.58% on number of spills in the average, when they are compared with George-Appel. Only for one application the George-Appel obtained better results, namely: `Jfdctint`. Besides, our allocators achieve reductions from 3.64% to 86.27% on number of fetches. However, for fetches, the George-Appel obtained best results for `FIR` and `Jfdctint`. In summary, only for one application our allocators did not achieve a better performance than George-Appel. It demonstrated that the strategy for coloring the interference graph and selecting spill, used by our allocators are a better approach to minimize the number of spill.

In the worst case, CARTRA is able to get better results than HECRA and George-Appel. On the other hand, it is necessary to run our allocators several times to get the best result. This does not occur with the George-Appel's algorithm, because it has no random feature like HECRA and CARTRA. In other words, George-Appel's algorithm is deterministic, while HECRA and CARTRA provides a different solution for each run (nondeterministic). The ideal is to run our allocators 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 allocators. 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.

C. Convergence

It is important to note that all allocators are iterative, i.e., each register allocator ends only when there are no spills (see Figures 1 and 3 – *rebuild if there were actual spills*). The Table II shows the convergence in average. For each interference graph is presented a list containing the amount

TABLE II
CONVERGENCE

Program		Allocator		
Name	Function	HECRA	CARTRA	George-Appel
Binary Search	bs	[9,0](2)	[9,0](2)	[9,3,1,1,1,2,1,1,1,0](10)
	main	[3,0](2)	[3,0](2)	[18,16,16,16,16,0](6)
FFT	sin	[5,0](2)	[5,1,0](3)	[5,0](2)
	init_w	[6,1,0](3)	[5,0](2)	[6,3,3,1,1,1,1,0](8)
	fft	[21,2,0](3)	[22,0](2)	[18,3,0](3)
	main	[4,2,0](3)	[4,2,0](3)	[4,2,2,2,0](5)
Fibonacci	fib	[2,1,0](3)	[2,1,0](3)	[2,1,1,0](4)
	main	[0](1)	[0](1)	[0](1)
FIR	sin	[5,0](2)	[6,0](2)	[6,0](2)
	sqrt	[7,0](2)	[8,1,0](3)	[8,1,1,0](4)
	fir_filter	[8,2,0](3)	[8,1,0](3)	[9,1,0](3)
	gaussian	[5,0](2)	[5,0](2)	[5,1,1,2,1,1,0](7)
	main	[7,0](2)	[8,0](2)	[7,0](2)
Insert Sort	main	[9,0](2)	[8,0](2)	[7,7,8,1,0](5)
Jfdctint	fdct	[39,0](2)	[33,0](2)	[24,0](2)
	main	[5,2,1,1,1,1,0](7)	[5,1,1,0](4)	[6,0](2)
LMS	sqrt	[7,0](2)	[8,0](2)	[8,1,1,0](4)
	sin	[5,0](2)	[6,0](2)	[6,0](2)
	gaussian	[5,0](2)	[5,0](2)	[5,1,1,2,1,1,0](7)
	lms	[23,4,1,0](4)	[22,3,1,0](4)	[18,10,9,10,5,3,1,0](8)
	main	[16,0](2)	[17,0](2)	[15,2,1,1,0](5)
Quick Sort	sort	[16,2,1,0](4)	[15,0](2)	[16,2,2,2,2,2,0](7)
	main	[2,0](2)	[2,0](2)	[3,0](2)
Queens	print	[6,1,0](3)	[6,0](2)	[8,0](2)
	tree	[7,0](2)	[7,0](2)	[6,0](2)
	main	[1,0](2)	[1,0](2)	[1,0](2)
Qurt	sqrt	[7,0](2)	[7,0](2)	[7,0](2)
	qurt	[11,4,1,0](4)	[11,2,0](3)	[11,2,3,2,3,1,1,1,0](9)
	main	[2,0](2)	[2,0](2)	[10,9,9,9,9,0](5)
Select	select	[19,1,1,0](4)	[16,0](2)	[19,1,6,4,5,4,3,2,0](9)
	main	[2,0](2)	[2,0](2)	[21,20,20,20,20,0](6)
Sqrt	sqrt	[7,0](2)	[7,0](2)	[7,0](2)
	main	[0](1)	[0](1)	[0](1)

of spills at each iteration and the list size.

The results demonstrated that both HECRA and CARTRA find a coloring that eliminates the number of spills in fewer iterations (rebuilding) than George-Appel. In general, the number of iterations required by George-Appel's algorithm is up to 5 times the amount needed by our allocators.

The approach based on *defs-uses* used by George-Appel 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.

HECRA and CARTRA do 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. Besides, the HECRA performance is similar to CARTRA.

D. Code Size

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

The reduction of the number of assembly instructions ranges from 2.79% to 35.35% in HECRA, and 0% to 35,35% in CARTRA, comparing our allocators with George-Appel. These reductions cause a reduction in the code size between 1.38% and 20.56%, and 0% and 20,56%, in HECRA and CARTRA respectively. These results are very important for systems that use embedded microprocessors, due to the fact that their components usually consist of limited computational power and limited memory. Only for `Jfdctint` our allocators do not outperform the generated code by George-Appel, due to the amount of spills.

TABLE IV
CODE SIZE - ASSEMLBY INSTRUCTIONS (A), CODE SIZE IN BYTES (CS)

Program	HECRA		CARTRA		George-Appel	
	A	CS	A	CS	A	CS
Binary Search	523	4884	523	4884	809	6148
FFT	809	6320	797	6276	840	6412
Fibonacci	43	860	43	860	47	872
FIR	1801	15208	1732	15104	1759	15192
Insert Sort	348	3508	337	3496	358	3564
Jfdctint	1568	10704	1525	10276	1501	10204
LMS	898	6336	867	6352	1000	6996
Quick sort	1355	12212	1314	12132	1676	13840
Queens	401	3748	398	3740	398	3740
Qurt	836	7536	802	7496	981	8180
Select	1254	11468	1216	11392	1618	13324
Sqrt	113	1452	108	1436	119	1472

Note that the traditional goal of a compiler is either to generate code that results on improved processor performance, or to minimize the compilation time to an acceptable level of processor performance. Wireless Sensor Netwoks (WSN) [25], 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 [26]. 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. HECRA and CARTRA are good allocators to address these issues.

VII. Conclusion and Future Works

Register allocation determines what values in a program must reside in registers, due to instructions involving register operands are faster than those involving memory access. Therefore, register allocation is a very important compiler optimization technique and can be mapped as a graph coloring problem, which is $\mathcal{NP}$-complete.

Due the nature of this problem, register allocators based on graph coloring algorithm apply some heuristic method to find a good coloring. But these allocators do not guarantee that the coloring is the best.

In our previous work we demonstrated that ACO algorithms are able to provide excellent solutions, but at a cost of a high compilation time. Therefore, there is a tradeoff here: solution quality versus compile time. In this new work we presented the HECRA, a graph coloring register allocator based on a hybrid evolutionary algorithm. Comparing our new allocator with the previous, the new one has a similar performance in generated code, but the new outperforms the old one if we are interested in compilation time.

There are several research directions in the future. First, we are interested in comparing our register allocators with the proposed by Wu and Li. Second, we will focus on implementing others heuristic phases.

References

[1] 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.

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

[3] 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.

[4] M. Plumettaz, D. Schindl, and N. Zufferey, "Ant local search and its efficient adaptation to graph colouring," *Journal of the Operational Research Society*, vol. 61, no. 5, pp. 819–826, 2010.

[5] P. Galinier and A. Hertz, "A survey of local search methods for graph coloring," *Computers & Operations Research*, vol. 33, no. 9, pp. 2547–2562, 2006.

[6] M. Laguna and R. Martí, "A grasp for coloring sparse graphs," *Computational Optimization and Applications*, vol. 19, no. 2, pp. 165–178, 2001.

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

[8] C. N. Lintzmayer, M. H. Mulati, and A. F. da Silva, "Register Allocation with Graph Coloring by Ant Colony Optimization," in *XXX International Conference of the Chilean Computer Science Society*, Curico, Chile, 2011.

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

[10] 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.

[11] 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.

[12] P. Galinier and J.-K. Hao, "Hybrid evolutionary algorithms for graph coloring," *Journal of Combinatorial Optimization*, vol. 3, no. 4, pp. 379–397, 1999.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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 SBPO - Simpósio Brasileiro de Pesquisa Operacional 2011*, Ubatuba, SP, BRA, 2011.

[20] C. N. Lintzmayer, M. H. Mulati and A. F. da Silva, "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.

[21] D. Costa and A. Hertz, "Ants can colour graphs," *The Journal of the Operational Research Society*, vol. 48, no. 3, pp. 295–305, 1997.

[22] I. Blöchliger and N. Zufferey, "A graph coloring heuristic using partial solutions and a reactive tabu scheme," *Computers & Operations Research*, vol. 35, no. 3, pp. 960–975, 2008.

[23] 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.

[24] F. Group, "SNU Real-Time Benchmarks," 2012, http://www.cprover.org/goto-cc/examples/snu.html.

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

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