Solving the Register Allocation Problem for Embedded Systems Using a Hybrid Evolutionary Algorithm

Haluk Rahmi Topcuoglu, Betul Demiroz, and Mahmut Kandemir

Abstract—**Embedded systems are unique in the challenges they present to application programmers, such as power and memory space constraints. These characteristics make it imperative to design customized compiler passes. One of the important factors that shape runtime performance of a given embedded code is the register allocation phase of compilation. It is crucial to provide aggressive and sophisticated register allocators for embedded devices, where the excessive compilation time can be tolerated due to high demand on code quality. Failing to do a good job on allocating variables to registers (i.e., determining the set of variables to be stored in the limited number of registers) can have serious power, performance, and code size consequences. This paper explores the possibility of employing a hybrid evolutionary algorithm for register allocation problem in embedded systems. The proposed solution combines genetic algorithms with a local search technique. The algorithm exploits a novel, highly specialized crossover operator that takes into account domain-specific information. The results from our implementation based on synthetic benchmarks and routines that are extracted from well-known benchmark suites clearly show that the proposed approach is very successful in allocating registers to variables. In addition, our experimental evaluation also indicates that it outperforms a state-of-the-art register allocation heuristic based on graph coloring for most of the cases experimented.**

Index Terms—**Compilers, crossover, embedded systems, evolutionary algorithms (EAs), hybridization, register allocation.**

I. Introduction

THE SOFTWARE content in embedded systems continuously increasing as a result of increases in portability and flexibility requirements. In addition, implementing a component in software is typically much less expensive than implementing it in hardware from both cost and maintenance perspectives. Therefore, software-based techniques for improving performance and other aspects of embedded systems are gaining popularity. While it is conceivable that a knowledgeable programmer can optimize application software for a given embedded architecture, this solution is neither scalable nor easy to achieve. This is because, over the years, embedded applications exhibited almost an exponential growth in both complexity and dataset sizes. As a result, compiler-directed automatic code optimization techniques are very important.

The compilers for embedded systems face very different challenges than their counterparts in high-end computing environments, such as power consumption, memory space limitations, and form factor (device size) related issues. In addition, since many embedded environments execute only a small set of applications, code quality is of the utmost importance. Consequently, compilers designed for embedded systems can afford long execution cycles to generate an output code of the highest quality, optimized under multiple metrics. What this means, in practice, is that in each phase of the compilation process (e.g., instruction selection, instruction scheduling, register allocation, code-flow optimization), one can consider employing costly algorithms that generate good code quality.

Register allocation is the process of assigning program variables (and compiler-generated temporaries) into available registers in the architecture. If a variable cannot be kept in a register due to the limited number of registers available, there will be a runtime cost (called the spill cost) of the corresponding variable for loading from and storing in memory. The objective of the register allocation problem is to minimize the total spill cost of variables in a given program block. The register allocation phase of compilation is generally more challenging in embedded environments as compared with high-end systems, mainly due to the heterogeneity of register files in current embedded processors. Coupled with the fact explained above (that we can afford long compilation cycles), a possible solution to the register allocation problem is to use powerful techniques to do an excellent job. Failure to achieve this may lead to the following problems, none of which is desirable in an embedded environment.

- *Increase in the number of memory accesses.* Poor register allocation can cause compiler miss opportunities for placing the most critical program variables/temporaries into registers, and this can increase the number of memory accesses dramatically. Even if a small fraction of these accesses miss in cache memories and go to off-chip main memory, one can expect a significant degradation in the performance of the application.
- *Increase in code size.* In many of the RISC-style processor architectures, the only way to operate on a variable is to first bring it to a register (even if it is not assigned a permanent register). Consequently, failing to allocate registers for critical data can increase the number of memory load/store operations required for maintaining correct execution. These extra instructions, in turn, increase code size,

Manuscript received June 3, 2005; revised September 18, 2005 and June 30, 2006. This work was supported in part by the National Science Foundation (NSF) under Career Award 0093082.

H. R. Topcuoglu and B. Demiroz are with the Department of Computer Engineering, Marmara University, 34722 Istanbul, Turkey (e-mail: haluk@eng.marmara.edu.tr; bdemiroz@eng.marmara.edu.tr).

M. Kandemir is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802 USA (e-mail: kandemir@cse.psu.edu).

Digital Object Identifier 10.1109/TEVC.2007.892766

which has consequences on memory capacity, and overall form factor of the device, and instruction cache behavior.
- *Increase in power consumption*. Frequent main memory visits due to poor register allocation can also increase overall power consumption, as large off-chip memories present large capacitive loads to the system, which is one of the main contributors of dynamic power consumption [1], [2].

Motivated by these observations, this paper explores the possibility of employing a hybrid evolutionary algorithm (HEA) for register allocation problem in embedded systems. Evolutionary algorithms (EAs) [3]–[5] are general-purpose, stochastic search methods that use the principles inspired by natural selection and genetics. EAs operate by iteratively generating a population of chromosomes that are encoded forms of the candidate solutions. In order to reach an optimum solution, the EA typically uses various genetic operators (including selection, crossover, and mutation) by applying the "survival of the fittest" principle. EAs have been efficiently used in a wide variety of applications in the engineering, science, and business fields [3], [6].

However, simple EAs are generally poor at solving complex combinatorial problems [7]. EAs are usually strengthened with the domain-specific characteristics [5], [8]–[10], and they are combined with specialized heuristics to produce hybrid systems, which all have different names, including HEAs and memetic algorithms [11]–[13].

HEAs have been applied to several NP-hard combinatorial problems with impressive results, including the traveling salesman problem [14], [15], placement and packing problems [16], location-allocation problems [17], three-matching problem [18], bin-packing problem [19], and various constrained optimization and satisfaction problems [20], [21]. In addition to the numerical and combinatorial problems, HEA is also used for solving linear and partial differential equations [22].

In this paper, we propose a highly specialized crossover operator called conflict-free partition crossover (CFPX) that incorporates the specific information on the register allocation problem. The CFPX operator aims to Sproviding conflict-free register classes successively, by constructing a single class of offspring at each step. We propose an efficient local search method for checking the neighborhood solutions, where the search area decreases with the decrease in the number of conflicts in the offspring. The CFPX operator combined with a problem-specific local search leads to a very powerful method, which places our approach into the class of the HEAs.

To test the effectiveness of the proposed solution, we conducted experiments with both synthetic benchmarks and real application codes. The purpose of our experiments with the synthetic benchmark experiments is to study the different aspects of our scheme under various carefully tuned parameters. On the other hand, the main goal of experimenting with the application codes is to measure the success of our approach in practice. The results from our implementation clearly show that the proposed approach is very successful in allocating registers. In addition, our experimental evaluation also indicates that it outperforms a state-of-the-art register allocation heuristic [23], [24] based on graph coloring for most of the cases experimented.

The rest of this paper is structured as follows. Section II discusses the main issues regarding the register allocation problem in compilation. Section III presents the details of our hybrid algorithm. We present the results of our experimental evaluation in Section IV. Finally, Section V summarizes our main conclusions.

II. Problem Definition

One of the strong impacts on runtime performance of a given code is the register allocation phase of compilation. Registers are the fastest locations (in terms of access latency) for storing program variables (data) and computations involving only register operands are much faster than those involving memory-resident operands. Therefore, effective utilization of the registers provided by the target machine can significantly speed up program execution.

The register allocator of a modern compiler decides which values (symbolic registers) at each point of a given program reside in architectural registers (*register allocation*) and which register holds each of those values (*register assignment*). Both global register allocation problem and global register assignment problem are NP-complete [25]. In this paper, we consider these two problems together under the "register allocation" term, and propose a novel *global* register allocation (i.e., an allocator that operates on an entire procedure at a time as opposed to local allocators that work at a basic block granularity).

If the register allocator cannot keep a variable in a register throughout its lifetime due to lack of available architectural registers, the value is stored in memory for some or all of its lifetime, which is called *spilling*. The *spill cost* of a variable is the estimated runtime cost of the corresponding variable for loading from and storing in memory. In order to estimate spill cost of a variable, one needs to capture the cost of memory operation, and the estimated execution frequency [26]. For a given program block, a register allocator targets at minimizing the total spill cost of the variables in the block. In the perfect allocation, all variables are mapped to registers so that the total spill cost is equal to zero.

Most processors have distinct classes of registers for different kinds of variables, such as general-purpose registers and floating-point registers [26]. If there are limited interactions between these two register classes, allocation for them can be done independently. Both classes can be modeled in a single-allocation problem, if they overlap as in the case of keeping double-precision floating-point numbers in pairs of single-precision registers. In order to simplify the design, we consider a set of homogeneous registers in this study.

A. Register Allocation Based on Graph Coloring

Many register allocators consider *graph coloring* as a paradigm to model the register allocation problem, which is briefly presented in this section. Assume that $V = \{v_1, v_2, v_3 \ldots\}$ is the set of variables in a given intermediate representation[1] of a program.

[1]A compiler is typically organized as a series of phases (passes), each of which performs a particular task/optimization. The intermediate representation is a common representation used by these phases. It is possible, and sometimes necessary, for an optimizing compiler to accommodate more than one intermediate representation.

```
int main(){
    int a, b, i;

    a=10;
    b=1;
    i=0;

    while(i<=a){
       b+=b*i;
       i++;
       if (b>=100)
          break;
    }
    return 0;
}
```

(a)

```
main:
      pushl  %ebp
      movl   %esp, %ebp
      subl   $24, %esp
      andl   $-16, %esp
      movl   $0, %eax
      subl   %eax, %esp
      movl   $10, -4(%ebp)
      movl   $1, -8(%ebp)
      movl   $0, -12(%ebp)
.L2:
      movl   -12(%ebp), %eax
      cmpl   -4(%ebp), %eax
      jle    .L4
      jmp    .L3
.L4:
      movl   -8(%ebp), %eax
      movl   %eax, %edx
      imull  -12(%ebp), %edx
      leal   -8(%ebp), %eax
      addl   %edx, (%eax)
      leal   -12(%ebp), %eax
      incl   (%eax)
      cmpl   $99, -8(%ebp)
      jle    .L2
.L3:
      movl   $0, %eax
      leave
      ret
```

(b)

```
main:
      subl   $4, t1 -> t2
      movl   t3, (t2)
      movl   t2, t3 -> t4
      subl   $24, t2 -> t5
      andl   $-16, t5 -> t6
      movl   $0, t7
      subl   t7, t6 -> t8
      movl   $10, (t4)
      movl   $1, (t4)
      movl   $0, (t4)
.L2:
      movl   (t4), t7 -> t9
      cmpl   (t4), t9
.L4:
      movl   (t4), t9 -> t10
      movl   t10, t11
      imull  (t4),t11 -> t12
      leal   (t4), t10 -> t13
      addl   t12, (t13)
      leal   (t4), t13 -> t14
      incl   (t14) -> t15
      cmpl   $99, (t4)
.L3:
      movl   $0, t15 -> t16
      leave
      ret
```

(c)

Fig. 1. (a) A high-level source code. (b) Intermediate representation from the GAS (GNU Assembler). (c) Extended representation for identifying the live ranges.

A variable $v_i \in V$ is said to be *live* at a given point p in the program if it is defined above p and has not been used yet for the last time. Values stay in registers as long as they are live. Live range LR_i for a variable v_i is the region which begins with the definition of v_i and ends with the last use of v_i.

If any two live ranges LR_i and LR_j are simultaneously live at some point p in the program flow, they cannot be stored in the same architectural register. In this case, we say that LR_i interferes with LR_j. To model the allocation problem, a compiler constructs an *interference graph*, $G_I = G(V, E)$, where V is the set of individual live ranges and E is the set of edges that represent interferences between live ranges. Two terms, a variable and its corresponding node in the interference graph, are interchangeably used in this paper.

Fig. 1 includes three listings: (a) a sample high-level source code which is input to the GAS (GNU assembler); (b) its intermediate representation produced by GAS for Linux environment; and (c) an extended representation, which is generated by our parser program in order to determine the live ranges of registers. It should be noted that the register names are replaced with a uniform representation that starts with "t1" in the extended representation. Additionally, each "pushl" instruction for the stack given in Fig. 1(b) is replaced with the sequence of "subl" and "movl" instructions in Fig. 1(c).

A variable may have more than one live range and a register allocator can, and in most cases will, keep these live ranges in different physical registers. To implement this, if an instruction has the same register name in both destination and source fields, a different register name can be used in the destination field in order to provide a new live range. In Fig. 1(c), the register names given after "$->$" are the register names for the destinations

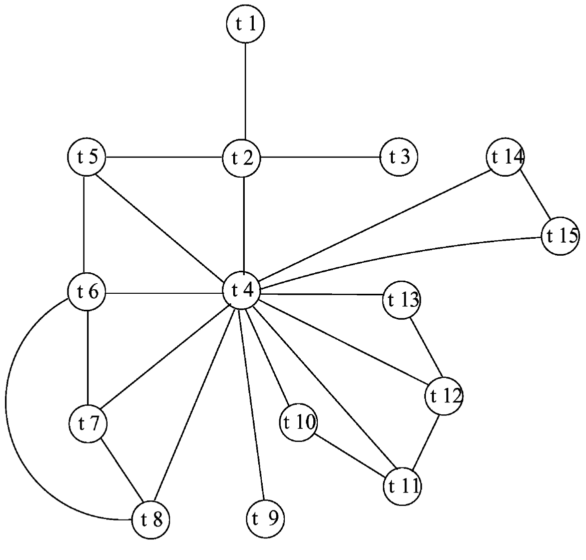


Fig. 2. Interference graph generated from the extended code given in Fig. 1(c).

due to the separation of source and destination registers. On the other hand, modern optimizing compilers can also provide a *coalescing* phase [26], which is basically the process of combining the two live ranges if the live ranges are connected by a copy or move instruction. Fig. 2 is the interference graph of the code fragment given in Fig. 1(c).

The register allocation problem is formally defined as the problem of mapping $f : V \rightarrow R = \{r_1, r_2, \ldots, r_k\}$, where R is the set that holds a maximum of k registers such that if $(v_i, v_j) \in E$, then $f(v_i) \neq f(v_j)$. When the interference graph

G_I is constructed, the compiler looks for an assignment of k colors to the nodes (vertices) of G_I, with distinct colors for adjacent nodes. This process corresponds to the well-known graph coloring problem of finding a k-coloring of the interference graph G_I.

Even simple formulations of the graph coloring problems are known to be NP-complete [25], consequently, a large set of heuristic approaches have been explored in the literature. Perhaps the simplest approach for graph coloring is the one that works on a permutation of the vertices and colors each one with the minimum number of colors. The permutations of vertices can be generated dynamically in some heuristics, such as in DSatur [27] algorithm. Simulated annealing, tabu search, and genetic algorithms (GAs) have been applied to the graph coloring problem extensively. It was shown by Fleurent and Ferland [28] that GAs that use standard crossover operators cannot outperform the heuristic algorithms based on tabu search and simulated annealing. Although, there are also parallel GAs [29], [30] for graph coloring, parallel algorithms are out of the scope of this paper.

The recent efforts [31]–[33] on the graph coloring problem consider HEAs that include problem-specific crossovers with powerful local search methods, and these hybrid methods outperformed the prior work on several sets of experiments. On the other hand, the HEAs presented for the graph coloring problem do not fully exploit the special characteristics of the register allocation problem. Specifically, solutions for the register allocation problem allocate registers for variables that have higher spill costs so that the overall spill cost could be minimized. However, techniques for solving the classical graph coloring problem do not differentiate between two vertices if they have the same number of outgoing edges. In this study, we present a new hybrid algorithm that exploits the specific nature of the register allocation problem.

B. Traditional Approaches for Register Allocation

The first widely used implementation of a register allocator based on graph coloring was Chaitin's heuristic [34], [35]. A later strategy based on a coloring paradigm is the priority-based method described by Chow and Hennessy [36], [37]. It was argued in [24] that all subsequent work on graph coloring-based allocators were derived from one of these two strategies. Although research efforts exist [38], [39] that target solving *register allocation* and *instruction scheduling* problems together, we do not consider this option in this paper.

The Chaitin-style allocator is illustrated as having seven phases in the literature [24]. The layout of the Chaitin's heuristic, which operates with a stack is given in Fig. 3 (adapted from [40]). In this figure, we only consider two out of the seven phases of the allocator: the *simplify phase* and the *select phase*. The simplify phase (captured in steps 2–12 in Fig. 3) moves a live range from the graph to the stack if its corresponding node has a degree less than k, where k is the number of available colors. It removes a node if the node can be assigned to a color (i.e., the corresponding variable can be assigned to a register). After the selected node is removed, the degrees of its neighbors are lowered, so that they may have a chance to be colored.

If the interference graph G_I is not empty and each node in G_I has a degree equal to or higher than k, a node is selected based on a predefined heuristic for spilling, which is called the *spill metric* (see Fig. 3). The original graph coloring allocator by Chaitin selects the node that minimizes the following spill metric function:

```
1. Initialize stack $S$ to empty.
2. while ($G_I \neq \emptyset$) do
3.     while $\exists\, v$ of $G_I$ such that $v^0 < k$    /* $v^0$ =degree of $v$
4.         Pick any node $v$ such that $v^0 < k$.
5.         Remove $v$ and its edges from $G_I$ and put $v$ on $S$.
6.     endwhile
7.     If ($G_I \neq \emptyset$) then
8.         Pick a node $v$ based on the given Spill Metric
9.         Spill the live range associated with $v$
10.        Remove $v$ and its edges from $G_I$.
11.    endif
12. endwhile
13. while ($S \neq \emptyset$) do
14.     $v$ = pop($S$)
15.     Color $v$ with the lowest color not used by any neighbor of $v$
16. endwhile
```

Fig. 3. The Chaitin's algorithm for register allocation (adapted from the outline given in [40]).

$$Spill\_Metric(i) = \frac{S\_Cost(i)}{C\_Degree(i)} \tag{1}$$

where $SCost(i)$ and $C\_Degree(i)$ are the spill cost and the current degree of the node i in G, respectively.

The select phase of the Chaitin's allocator assigns colors to the nodes (i.e., assigns registers to the variables) in reverse order, so that it tries to color the most constrained nodes first. Fig. 4 illustrates two sample interference graphs. Assume that we target 3-coloring (i.e., there are three registers) for the first graph. The simplify phase of Chaitin's algorithm removes all the nodes of this graph; so, it does not require any spilling. A possible stack content (starting from the bottom of the stack) is b, e, a, c, and d. After the select phase, we have the following register allocations: $d \rightarrow r_1$, $c \rightarrow r_2$, $a \rightarrow r_3$, $e \rightarrow r_3$, and $b \rightarrow r_1$.

Chaitin's heuristic is not guaranteed to find a k-coloring for a given graph, even if the graph is k-colorable [24]. The diamond graph (the second graph in Fig. 4) was used in the literature to show the limitations of the Chaitin's heuristic. Assume that we have two registers for the given diamond graph. The Chaitin's heuristic requires spilling for one of the nodes of this graph. An example register allocation is $b \rightarrow r_1$, $c \rightarrow r_2$, $d \rightarrow r_1$, and $a \rightarrow spilled$. However, in this case, a spill-free solution is possible with the following allocation: $a \rightarrow r_1$, $b- > r_2$, $c \rightarrow r_1$, and $d \rightarrow r_2$.

The *optimistic coloring heuristic* (OCH) [23], [24] is a Chaitin-style graph coloring register allocator that generates 2-coloring of the given diamond graph. The OCH algorithm requires modifications in both the simplify and the select phases of the Chaitin's heuristic. In the simplify phase, whenever a node is selected as a spill candidate using the given spill metric, the OCH (optimistically) puts the node into the stack for a

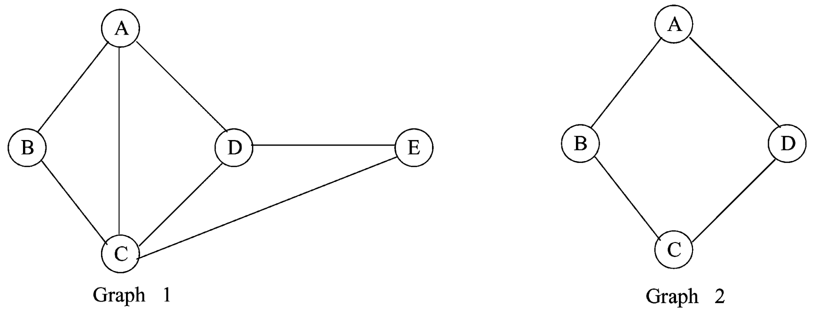


Fig. 4. Two sample interference graphs.

possibility of coloring, instead of marking it for spilling as in the Chaitin's heuristic. In the select phase of OCH, when it discovers that a node cannot be k-colored, it is left uncolored (i.e., unassigned to a register). The uncolored nodes are then spilled, and the sum of spill costs of the uncolored nodes is the total spill cost of the solution, as in the Chaitin's heuristic. Since OCH outperforms the other methods including the Chaitin's heuristic with respect to several benchmark routines [24], OCH is considered as the reference work in our experiments presented later in Section IV.

III. A Hybrid Evolutionary Algorithm (HEA) for Register Allocation

EAs [3]–[5] is a broader class of metaheuristics that mimic the analogy of natural evolution and genetics. The GA [6], [41] is a classical method in this category. It has been studied extensively in the past and applied to various optimization problems. There have been several other methods proposed recently, which include genetic programming [42], particle swarm optimization [43], ant colony optimization [44], etc.

Although simple EAs can be easily developed for many optimization problems, they are not generally efficient for the complex combinatorial problems [7]. In order to improve the performance of EAs, they are usually enhanced with the addition of problem-specific knowledge [5], [8]–[10]. Specialized operators or algorithms are combined with EAs to generate complex hybrid systems, which are called different names in research papers, including hybrid GAs, HEAs, genetic local search algorithms, and memetic algorithms. Memetic algorithms [11], [13] are the most common and they cover a wide range of techniques. A comprehensive bibliography on memetic algorithms is given in [12].

In order to build HEAs, as stated in [5], there are four main methods to incorporate specialized operators and domain specific knowledge with EAs. They are:

- using problem-specific heuristics for initial population generation;
- hybridization within variation operators (intelligent crossover and mutation operators);
- applying local search on the output from variation operators, which is for all or a portion of the population;
- using domain-specific knowledge within decoder or repair function applied for genotype to phenotype mapping.

We consider all these methods in our HEA for the register allocation problem. The algorithm sets the initial population of solutions using a problem-specific heuristic. We propose a steady-state GA that takes two input strings, and our problem-specific crossover operator (CFPX operator) is applied to generate a single offspring. It is followed by the local search operator to improve the generated offspring before inserting it into the population. This process is repeated for a predefined number of iterations. In addition, two different methods are presented to map a genotype to a phenotype; then, the best fitness value among them is selected.

In our GA-based approach, we consider the *partition method* [33] for string representation. Each solution S_i partitions the variables into register classes $S_i = \{R_1, R_2, \ldots, R_k\}$, where each class R_i includes the live ranges of variables that are mapped to the register r_i; and k is the total number of registers.

A. Initial Population Generation

In order to generate a predefined number of strings for the initial population, we modified the DSatur algorithm [27], which is a graph coloring heuristic. The algorithm is so-called since it considers the *saturation degree* of each node, which can be defined as the number of different colors to which the node is adjacent to. In the DSatur algorithm, nodes are arranged by the decreasing order of the saturation degrees.

The graph coloring problem is one of assigning a color to each node in a given graph so that the total number of colors used is minimized and no pair of adjacent nodes have the same color [45]. It does not differentiate between any two nodes of a graph if they have the same number of outgoing edges. However, in the register allocation problem, both the spill costs and the degrees of the nodes in a given interference graph are considered. Therefore, we propose a new metric called the *spill degree* that can be used for ordering the nodes, instead of using the saturation degree as in the DSatur algorithm. The spill degree of a node i can be defined by one of the three equations given below

$$\begin{aligned} S\_Degree_1(i) &= S\_Cost(i) \times Degree(i) \\ S\_Degree_2(i) &= S\_Cost(i) \times Degree^2(i) \\ S\_Degree_3(i) &= S\_Cost(i). \end{aligned} \tag{2}$$

In these expressions, $S\_Cost(i)$ is the spill cost and $Degree(i)$ is the number of edges incident to node i. In order to generate the initial population, the spill degrees are set using a combination of the three equations given in (2). In our experiments, 40% of the initial population is set by considering the first equation, another 40% is set using the second one, and the remaining part is filled by considering the last equation. The spill degree term is considered only for generating the initial population.

In our approach, nodes are sorted in decreasing order of spill degrees. At each iteration, an unassigned node with the maximal spill degree is mapped to the register class with the lowest possible index value, where the selected node should be conflict-free with the other nodes in the same class. The nodes i and j in an interference graph are said to be conflict-free, when there is no edge connecting them. If tie-breaking becomes necessary, the node with the maximal degree is selected. If it is not possible to locate the node in a register class by preserving the conflict-free property, the algorithm selects any register class at random. This process is repeated until all the nodes (i.e., their corresponding variables) are mapped to one of the register classes.

```
Initialize conflict-free set $R_i^{CF}$ of register class $R_i$ with the set $R_i$.
Compute $CF(m)$ (total number of conflicts) for each variable $m$ in $R_i$.
$CF_i^{all} = \sum_{\forall\, m \in\, R_i} CF(m)$.
Initialize $j$ to 0.
while ( $j < \frac{CF_i^{all}}{2}$ )
    Select the variable $n$ where $CF(n) = \max_{m \in R_i^{CF}} \{CF(m)\}$.
    In case of tie-breaking, select a variable $n$ where
        $S\_Cost(n) = \min_{m \in R_i^{CF}} \{S\_Cost(m)\}$.
    Remove variable $n$ from $R_i^{CF}$.
    $j = j + CF(n)$.
endwhile
```

Fig. 5. The heuristic for determining the conflict-free set with the maximum number of nodes.

B. Crossover Operator

We introduce a new crossover operator in order to account for problem-specific information. The proposed operator is called *conflict-free partition crossover* (CFPX), as it aims to provide conflict-free register partitions. The crossover operator constructs an offspring iteratively, by generating a single partition (i.e., register class) of the offspring at each step.

Assume that our hybrid GA selects two parents, $S_1 = \{R_1^1, \ldots, R_k^1\}$ and $S_2 = \{R_1^2, \ldots, R_k^2\}$, for the crossover operator randomly from the population with uniform distribution. The partition from these parents that has the maximum number of *conflict-free nodes* is selected. In case tie-breaking is required in selecting a partition within a parent, a register class is selected randomly. If tie-breaking needs to be made between two parents, the register class is selected from the parent that has not been considered in the previous step.

Assume that R_i, the ith register class from one of the parents, is selected. The conflict-free subset of R_i (denoted by R_i^{CF}) becomes the base set of the first register class for the offspring. The base set is extended with one or more nodes by preserving its conflict-free property. This process determines the first register class of the offspring, which is subsequently repeated for the other register classes. If a node (i.e., variable) is not assigned to any class at the end of the crossover operation, it is assigned to the class that has the minimum number of conflicting nodes. After an offspring is generated, one of its parents which has the highest fitness value is replaced by the new offspring.

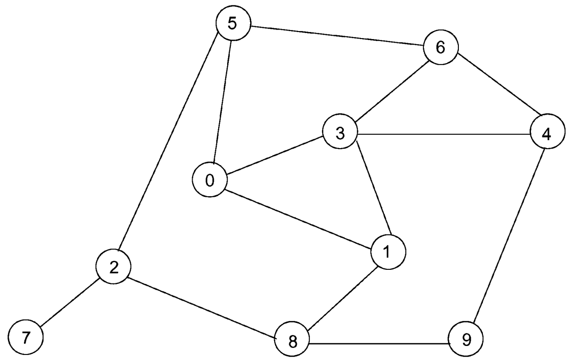
An important part of the CFPX operator is how to determine the conflict-free set of each register class R_i. We consider the heuristic given in Fig. 5. Initially, the conflict-free set includes all variables in R_i. The total number of conflicts of each variable m in class R_i, $CF(m)$, are determined. The sum of the $CF(m)$ values gives us the total number of conflicts in the register class R_i. Then, the variables from the partition are removed one by one in decreasing order of $CF(m)$ values, until the total number of conflicts in R_i becomes equal to 1/2 of its initial value.

Proposition: The conflict-free set of a register class R_i generated by the heuristic given in Fig. 5 is the conflict-free set with the maximum number of variables.

Sketch of Proof: We prove this by contradiction. Assume that R_i has n variables and the heuristic determines a conflict-free set S_1 which has $k < n$ variables. Assume further that another conflict-free set S_2 of the same register class has m variables, where $m > k$. If $m > k$, then, $n - m < n - k$, i.e., the number of nodes removed to generate S_2 is lower. If the total number of conflicts in a register class is z, a set of nodes are removed from the class until the total number of conflicts in the remaining set becomes less than or equal to $z/2$. Since our heuristic removes the nodes in the decreasing order of their number of conflicts, S_1 requires the minimum number of removals, and thus, we have the following inequality: $n - k <= n - m$, which causes a contradiction with the initial assumption.

Fig. 6 is an example interference graph with ten nodes, and it also shows spill costs of variables. Fig. 7 demonstrates the application of the CFPX operator to two selected parent solutions that are derived from the interference graph given in Fig. 6. When the first parent P1 in Fig. 7 is examined, the conflict-free subsets of the four partitions become {3}, {0,4}, {7,8}, and {9,1,5}. Then, the maximum number of conflict-free nodes in classes of P1 is equal to 3. The third register class of P2 has the maximum number of conflict-free nodes, which is also equal to 3. The total spill cost of P1 is equal to 7, which is due to nodes 6 (or 4) and 2. For P2, the total spill cost is equal to 10, which is due to nodes 3, 1, 2, and 9. The conflict-free subset with the maximum number of nodes ($R_4^{CF} = \{9, 1, 5\}$) becomes the base of the first register class of the result. This set is extended with node 7, since there is no edge between 7 and any node in the set R_4^{CF} (see Fig. 7). The complete offspring is generated when this process is repeated for the other register classes. When the generated offspring in Fig. 7 is examined, it has a total spill cost of zero, thanks to the lack of conflicts within its four register classes.

In order to emphasize the use of problem-specific operators in an EA, the performance of the CFPX operator is compared with performance of the GPX operator [33], which is a competitive crossover operator used as part of a HEA for solving the graph coloring problem. Since some of the register allocators consider graph coloring as a paradigm to model the register allocation problem, and Galinier and Hao's work [33] outperforms



Vertex	Spill Cost
0	4
1	3
2	4
3	1
4	3
5	1
6	3
7	1
8	2
9	2

Fig. 6. An example interference graph with ten nodes with their spill costs.

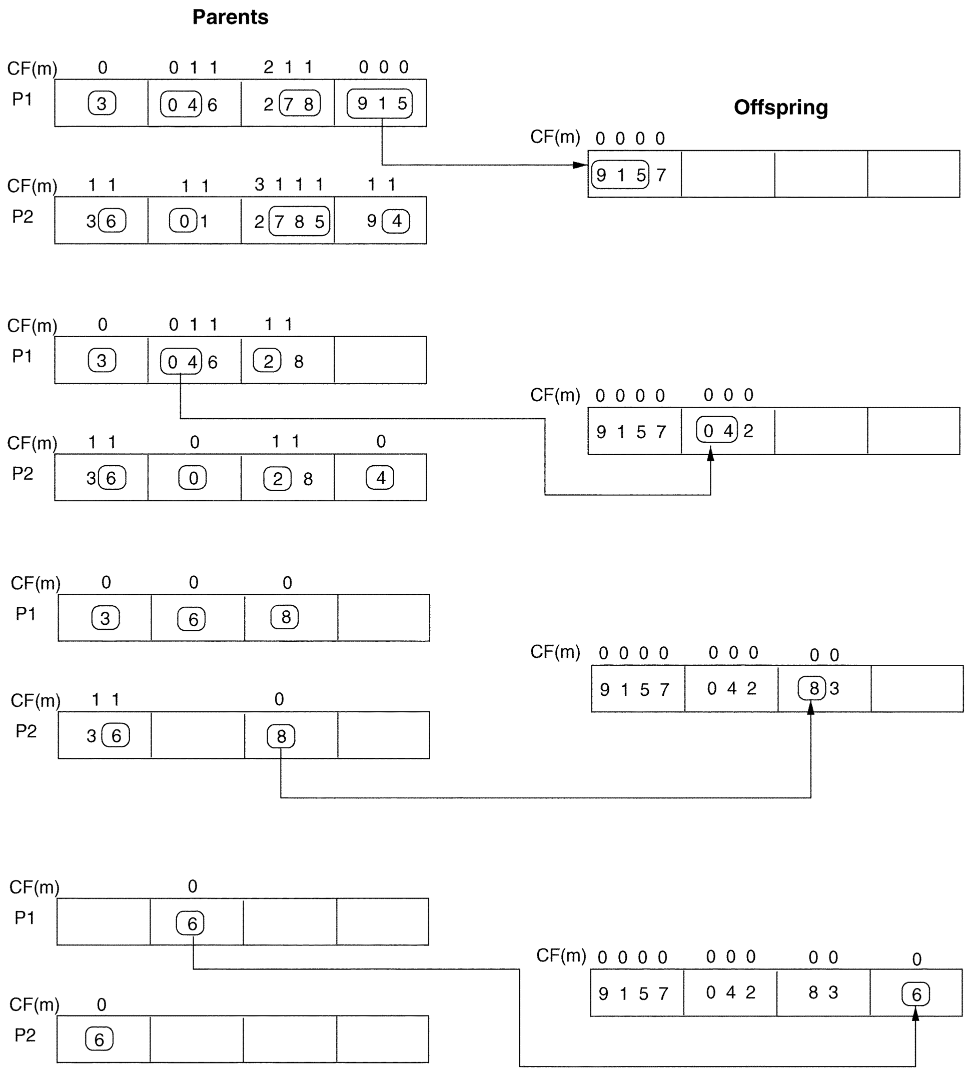


Fig. 7. Applying CFPX crossover on two selected parent configurations.

other state-of-the-art algorithms for solving the graph coloring problem, the GPX operator is considered in the computational experiments presented in Section IV.

The GPX operator considers the parents successively, and it chooses the class from the considered parent with the maximum number of unassigned vertices to become the next class of the offspring. It does not consider conflicts while selecting the partitions. Furthermore, the GPX operator does not include a second phase of extending each partition with the variables that belong to other partitions by preserving the conflict-free property, as in the case of the CFPX operator.

C. Local Search Phase

After a solution is generated using the crossover operator, the local search phase targets at improving it by checking a set of

TABLE I
COMPARISON OF ALGORITHMS WITH RESPECT TO THE NUMBER OF TEST GRAPHS (OUT OF 600) FOR VARIOUS EDGE DENSITY VALUES. THE SAME GRAPHS ARE CONSIDERED FOR ALL THREE PAIRWISE COMPARISONS. THE NUMBERS UNDER THE BETTER COLUMNS ARE THE NUMBER OF TESTS WHERE THE FIRST ALGORITHM IN THE PAIRWISE COMPARISON OUTPERFORMS THE SECOND ONE

		Pair-Wise Comparisons									**Unspilled Test Cases**		
		HEA-OCH			HEA-GPX			OCH-GPX					
n	α	better	equal	worse	better	equal	worse	better	equal	worse	HEA	OCH	GPX
	0.05	109	444	47	519	81	0	519	81	0	450	470	81
	0.10	296	300	4	599	1	0	594	1	5	400	400	1
100	0.25	397	200	3	600	0	0	519	0	81	294	200	0
	0.50	564	25	11	559	6	35	470	0	130	103	25	0
	0.75	594	0	6	519	50	31	395	1	204	0	0	0
	1.00	0	600	0	0	600	0	0	600	0	0	0	0
	0.05	100	500	0	600	0	0	600	0	0	500	500	0
	0.10	199	400	1	600	0	0	600	0	0	400	400	0
200	0.25	319	281	0	600	0	0	600	0	0	395	281	0
	0.50	500	100	0	600	0	0	588	0	12	200	100	0
	0.75	521	0	79	598	0	2	523	0	77	0	0	0
	1.00	0	600	0	0	600	0	0	600	0	0	0	0
	0.05	99	500	1	600	0	0	600	0	0	500	500	0
	0.10	200	400	0	600	0	0	600	0	0	469	400	0
300	0.25	300	300	0	600	0	0	600	0	0	400	300	0
	0.50	500	100	0	600	0	0	600	0	0	200	100	0
	0.75	498	0	102	600	0	0	569	0	31	37	0	0
	1.00	0	600	0	0	600	0	0	600	0	0	0	0
	0.05	98	500	2	600	0	0	600	0	0	500	500	0
	0.10	176	424	0	600	0	0	600	0	0	500	424	0
400	0.25	251	349	0	600	0	0	600	0	0	400	349	0
	0.50	461	139	0	600	0	0	600	0	0	200	139	0
	0.75	435	0	165	600	0	0	600	0	0	100	0	0
	1.00	0	600	0	0	600	0	0	600	0	0	0	0
	0.05	99	500	1	600	0	0	600	0	0	500	500	0
	0.10	106	494	0	600	0	0	600	0	0	500	494	1
500	0.25	201	399	0	600	0	0	600	0	0	400	399	0
	0.50	400	200	0	600	0	0	600	0	0	200	200	0
	0.75	407	0	193	600	0	0	600	0	0	100	0	0
	1.00	0	600	0	0	600	0	0	600	0	0	0	0

its neighborhood solutions.A neighbor of a given configuration S_1 is obtained by moving a single variable i from one register class to a different register class where i is already in conflict with one or more variables in the original register class of S_1. We introduce a new term, *spill factor*, for each variable j, which is computed using the following equation:

$$S\_Factor(j) = S\_Cost(j) \times CF(j) \tag{3}$$

where $S\_Cost(j)$ is the spill cost of j, and $CF(j)$ is the total number of conflicts of j in its assigned register class. Then, the variable i which has the maximum spill factor is selected. Assume that i is already assigned to class k. Equation (4) computes the difference in fitness values of the register class k for the cases with and without considering the variable i

$$\Delta\text{fitness}(k) = \text{fitness}(k) - \text{fitness}_{i \notin k}(k). \tag{4}$$

The Δfitness for each other register class m $(m \neq k)$ is computed similarly with the difference in fitness values for the cases with and without considering i. If the following inequality holds for any register class m:

$$\Delta\text{fitness}(m) < \Delta\text{fitness}(k) \quad \text{where} \quad \Delta\text{fitness}(m) = \min_{n \neq k}(\Delta\text{fitness}(n)) \tag{5}$$

then variable i is moved from class k to class m, since class m provides the best reduction in the partial fitness value. If we cannot find a register class that satisfies the inequality given in (5), the same process is repeated with a new variable according to the decreasing order of spill factors. The number of iterations (i.e., the number of variables examined) in the search phase is bounded by $\lceil 10\% \times CF^{\text{all}} \rceil$, where CF^{all} is the total number of conflicts in the given configuration. Therefore, the duration of the search phase decreases with a decrease in the total number of conflicts. In case of no improvement after the completion of the search phase, it considers the initial offspring generated after the crossover operation.

D. Computation of Fitness Function

The applications of GAs to real-world problems usually follow two main approaches, namely, the use of *direct* and *indirect* chromosome representations. For the direct chromosome representation, a chromosome represents a single solution. In the indirect version, one or more decoding algorithms are proposed in order to build the solution by considering the chromosome representation. Our GA-based framework can be considered in-between these two cases, i.e., it includes a *semi-direct* representation. The assigned register (and the register class) of each variable for an individual is already given in our chromosome representation. Additionally, we propose two decoding heuristics to select the spilled variables as part of our fitness function, since it is not encoded in the chromosome representation.

The two approaches used in our study to determine the variables to be spilled are referred to as the *conflict-based* and the *spill-cost-based* techniques. In the conflict-based technique, for each register class R_i of a given solution, the total number of conflicts of each variable j in R_i is determined. Then, the variables in R_i are put in a spill set in decreasing order of their number of conflicts, until R_i becomes conflict-free. In case of tie-breaking between two or more variables, the one with the minimum spill cost value is placed into the spill set. Here, the spill set of R_i includes only the variables that require spilling. The sum of spilled costs in this set gives the fitness value of the register class R_i, which is represented by $\mathrm{fitness}^C(i)$.

In the spill-cost-based case, as in the previous one, the total number of conflicts of each variable j in R_i is determined. After this, variables in R_i which are in conflict are put in a spill set in increasing order of their spill costs. In case of tie-breaking, the one with the maximum number of conflicts is considered. The sum of spilled costs in this set gives an alternative fitness value of the register class R_i, which is represented by $\mathrm{fitness}^S(i)$.

Then, the minimum of the fitness values derived from the two cases mentioned above becomes the fitness value of the given register class. This process is applied to all register classes in the solution; and the total sum of fitness values of classes gives the fitness value of the given individual. The following two equations finalize the computation of fitness function:

$$\mathrm{fitness}(i) = \min\left\{\mathrm{fitness}^C(i), \mathrm{fitness}^S(i)\right\} \tag{6}$$

$$\mathrm{fitness} = \sum_{i=1}^{k} \mathrm{fitness}(i). \tag{7}$$

IV. Experimental Evaluation

In this section, we present the results of experiments that evaluate the effectiveness of our algorithm. Our test suite has two parts: syntactically generated interference graphs, and a collection of nine routines extracted from several benchmarks. For the former part, we consider a large set of interference graphs that are generated at random. The latter part, on the other hand, includes a set of benchmark codes, which are processed to generate the corresponding interference graphs. Our experimental analysis includes three algorithms: our HEA with its original CFPX operator (called the HEA, henceforth), our HEA by replacing its CFPX operator with the GPX operator [33], as explained in Section III-B (called the GPX algorithm in this section), and the optimistic coloring heuristic (called the OCH algorithm). The experiments in this study were performed on a PC with 1.8 Ghz. Pentium IV processor and 512 MByte RAM.

A. Syntactically Generated Interference Graphs

We implemented a graph generator for developing random interference graphs, which takes three input parameters: the number of nodes (n) in the graph, edge density (α), and the mean spill cost value (γ) for the variables. The edge density is the probability with which an edge is present between any two nodes; in this study, we employ a uniform probability distribution. The total number of edges in a randomly generated interference graph is close to $\alpha \times (n \times (n-1)/2)$. There is no requirement that the resultant graph should be connected; in fact, our experimental evaluation includes disconnected interference graphs as well.

After a graph is constructed, the spill cost of each node (i.e., the variables) is set randomly using a uniform distribution with the range of $[1, \ldots, 2 \times \gamma]$. Each such generated graph is tested using various number of registers. For this purpose, the *register density* (β) term is introduced in the experiments. Note that the total number of registers in the system is equal to $r = \beta \times n$.

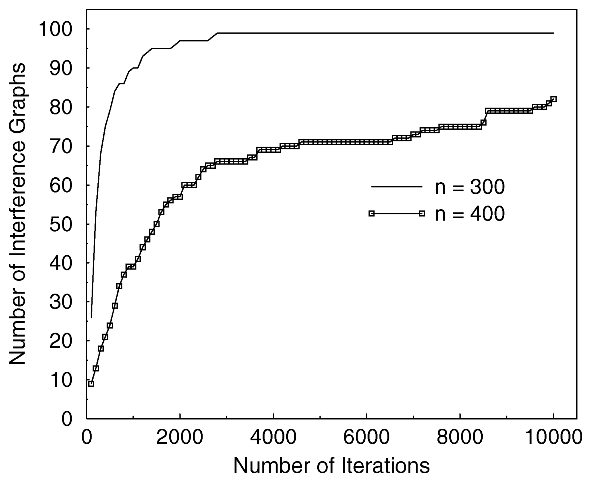


Fig. 8. Performance of HEA when number of iterations is varied.

TABLE II
Comparison of Algorithms With Respect to Total Costs and Number of Spilled Variables for Various Edge Densities. The Values in Cells Are the Average Values of 600 Interference Graphs That Are Considered for Table I

		Total Spill Cost			Number of Spilled Variables		
n	α	HEA	OCH	GPX	HEA	OCH	GPX
100	0.05	17.04	19.33	41.25	4.37	5.29	11.76
	0.10	41.15	49.65	87.33	9.38	11.92	21.92
	0.25	90.92	107.64	167.62	20.41	25.97	42.74
	0.50	167.44	192.96	254.33	37.50	45.39	59.36
	0.75	284.46	316.18	350.77	56.58	64.46	69.71
	1.00	456.16	456.16	456.16	90.19	90.19	90.19
200	0.05	12.10	18.31	108.97	3.9	6.46	31.27
	0.10	53.58	68.70	223.46	12.47	17.65	54.46
	0.25	148.91	171.33	393.23	33.62	41.81	94.28
	0.50	295.55	340.01	570.26	65.00	80.59	125.64
	0.75	504.46	543.32	713.73	104.20	120.17	145.73
	1.00	862.44	862.44	862.44	180.33	180.33	180.33
300	0.05	9.40	15.54	157.82	3.17	5.99	45.60
	0.10	44.64	73.34	307.87	12.03	20.83	82.11
	0.25	187.79	233.78	597.85	43.22	56.54	142.63
	0.50	196.08	437.13	796.20	86.18	112.33	190.38
	0.75	694.34	753.97	1039.21	146.02	172.76	218.80
	1.00	1358.54	1358.54	1358.54	270.50	270.50	270.50
400	0.05	5.62	10.86	185.99	2.04	4.93	57.13
	0.10	46.3	87.80	371.59	12.62	23.78	96.38
	0.25	234.6	310.57	769.34	51.49	70.55	175.90
	0.50	463.68	574.61	1049.57	106.69	142.05	246.07
	0.75	918.41	995.75	1368.80	188.09	224.64	284.92
	1.00	1775.99	1775.99	1775.99	360.66	360.66	360.66
500	0.05	2.59	6.51	269.78	0.99	3.43	78.06
	0.10	45.21	90.80	496.65	13.36	27.91	129.62
	0.25	255.07	340.15	923.09	60.64	84.97	220.07
	0.50	583.66	725.95	1328.58	127.40	171.83	288.61
	0.75	1093.94	1170.98	1559.55	229.88	272.18	331.95
	1.00	2434.16	2434.16	2434.16	450.83	450.83	450.83

TABLE III
COMPARISON OF ALGORITHMS WITH RESPECT TO THE NUMBER OF TESTS FOR VARIOUS REGISTER DENSITY VALUES. THE SAME TEST GRAPHS ARE CONSIDERED FOR ALL THREE PAIRWISE COMPARISONS. FOR EACH REGISTER DENSITY VALUE, 100 GRAPHS (OUT OF 600) ARE GENERATED BY SETTING THE EDGE DENSITY TO 1.0. DUE TO THEIR FULLY CONNECTIVITY CHARACTERISTIC AND LACK OF REGISTERS. ALL ALGORITHMS SPILL MOST OF VARIABLES BY GENERATING EQUAL TOTAL COSTS

		Pair-Wise Comparisons									**Unspilled Test Cases**		
		CFPX-OCH			CFPX-GPX			OCH-GPX					
n	β	better	equal	worse	better	equal	worse	better	equal	worse	CFFX	OCH	GPX
100	0.02	481	100	19	378	156	66	222	101	277	0	0	0
	0.04	413	144	43	500	100	0	369	100	131	50	70	0
	0.08	394	200	6	500	100	0	492	100	8	200	200	0
	0.10	297	300	3	500	100	0	496	100	4	294	200	0
	0.15	200	400	0	485	115	0	485	115	0	303	300	15
	0.20	175	425	0	433	167	0	433	167	0	400	325	67
200	0.02	499	100	1	498	100	2	413	100	87	0	0	0
	0.04	399	200	1	500	100	0	498	100	2	100	100	0
	0.08	300	300	0	500	100	0	500	100	0	295	200	0
	0.10	141	381	78	500	100	0	500	100	0	300	281	0
	0.15	200	400	0	500	100	0	500	100	0	400	300	0
	0.20	100	500	0	500	100	0	500	100	0	400	400	0
300	0.02	499	100	1	500	100	0	469	100	31	0	0	0
	0.04	399	200	1	500	100	0	500	100	0	169	100	0
	0.08	293	300	7	500	100	0	500	100	0	300	200	0
	0.10	105	400	95	500	100	0	500	100	0	300	300	0
	0.15	200	400	0	500	100	0	500	100	0	400	300	0
	0.20	100	500	0	500	100	0	500	100	0	437	400	0
400	0.02	498	100	2	00	100	0	472	100	28	0	0	0
	0.04	376	224	0	500	100	0	500	100	0	200	124	0
	0.08	185	349	66	500	100	0	500	100	0	300	249	0
	0.10	101	400	99	500	100	0	500	100	0	300	300	0
	0.15	161	439	0	500	100	0	500	100	0	400	339	0
	0.20	100	500	0	500	100	0	500	100	0	500	400	0
500	0.02	499	100	1	500	100	0	500	100	0	0	0	0
	0.04	306	294	0	500	100	0	500	100	0	200	194	0
	0.08	103	399	98	500	100	0	500	100	0	300	299	0
	0.10	105	400	95	500	100	0	500	100	0	300	300	0
	0.15	100	500	0	500	100	0	500	100	0	400	400	0
	0.20	100	500	0	500	100	0	500	100	0	500	400	0

1) Experiments for Identifying Parameter Settings: In this section, we present the results from our experiments for determining values of parameters in our algorithm. As a part of this work, we study the relationship between the problem size and the number of iterations in our algorithm. In order to capture the influence of the number of iterations on performance, we consider the cases where our algorithm requires the maximum number of iterations in order to outperform the OCH. After a set of experiments, it was observed that our algorithm requires the maximum number of iterations when the edge density and register density parameters are set to $\alpha = 0.75$ and $\beta = 0.08$, respectively. The experimentation is repeated for these values with two problem sizes ($n = 300$ and $n = 400$); and for each problem size, a total of 100 random graphs are generated. The spill cost of each variable is uniformly distributed in the range [1, . . . ,5].

The performance of the algorithms are measured by setting the number of iterations from the range [100, . . . ,10000] with an increment of 100 (see Fig. 8). The Y axis gives the number of graphs (out of 100) that our algorithm outperforms the OCH. When the graphs with 300 vertices are considered, the hybrid algorithm outperforms the OCH algorithm in 99 out of the 100 graphs tested when the number of iterations reaches 2800. For the case of 400 vertices, the hybrid algorithm outperforms (the OCH algorithm) in 82 graphs when the number of iterations is equal to 10 000. The same test is repeated with the smaller problem sizes ($n = \{100, 200\}$), where the hybrid algorithm outperforms OCH for all the generated graphs when the number of iterations reaches 400. For the experiments presented in the following subsections, we consider 1000 iterations when $n = \{100, 200\}$, 3000 iterations when $n = \{300\}$, and 10 000 iterations when $n = \{400, 500\}$.

Population size is another parameter for the GA-based methods that should be set properly. The performance of our algorithm is examined when the population size is in the range of $p = \{100, 200, 300, 400, 500\}$ for various problem sizes. It was observed that the performance of the algorithm does not depend on population size when the number of iterations is fixed. The best results in terms of the number of cases and average spill cost values are observed with the lowest population size, i.e., when p is equal to 100. This is due to the fact that the population of the individuals is not completely regenerated in every generation. Additionally, the proposed algorithm for the initial population (given in Section III-A) generates a limited number of different individuals; and when the population size is increased, there will be various replications of individuals in the population.

2) Results of Computational Experiments: For each problem size, the edge density (α) is varied using the values in the set {0.05,0.1,0.25,0.5,0.75,1.0} and the reg-

TABLE IV
COMPARISON OF ALGORITHMS WITH RESPECT TO THE TOTAL COSTS AND THE NUMBER OF SPILLED VARIABLES FOR VARIOUS REGISTER DENSITY VALUES. THE VALUES IN CELLS ARE THE AVERAGE VALUES OF 600 INTERFERENCE GRAPHS THAT ARE CONSIDERED FOR TABLE III

		Total Spill Cost			**Number of Spilled Variables**		
n	β	HEA	OCH	GPX	HEA	OCH	GPX
100	0.02	349.98	363.64	367.65	68.92	72.59	72.91
	0.04	246.63	265.29	295.19	50.42	40.81	61.67
	0.08	160.92	176.57	219.66	33.92	38.76	48.72
	0.10	136.35	150.54	195.88	28.93	33.56	44.51
	0.15	93.62	106.30	153.57	20.36	24.73	36.69
	0.20	69.66	78.69	125.53	15.85	18.72	31.17
200	0.02	625.05	663.01	738.64	126.41	136.72	445.49
	0.04	443.32	473.28	611.67	92.42	101.54	128.26
	0.08	285.62	304.23	472.13	61.93	70.17	104.98
	0.10	243.64	255.84	426.02	53.59	60.45	96.98
	0.15	159.58	175.23	341.63	36.60	44.02	81.84
	0.20	119.82	132.53	282.01	28.57	34.08	71.16
300	0.02	869.62	964.57	1098.02	178.33	196.63	223.39
	0.04	615.89	671.20	907.72	129.59	145.46	193.16
	0.08	406.66	433.75	697.67	87.03	99.72	158.02
	0.10	350.97	368.70	627.36	75.03	86.88	145.46
	0.15	231.79	258.49	503.91	50.92	61.58	122.86
	0.20	182.53	201.02	422.81	40.22	48.67	107.13
400	0.02	1140.57	1247.02	1493.97	229.02	255.33	297.67
	0.04	796.99	886.86	1212.02	166.26	188.82	255.24
	0.08	526.75	563.43	887.14	111.89	128.46	200.35
	0.10	451.16	473.92	787.91	121.15	111.96	182.70
	0.15	293.04	329.18	628.36	65.33	78.70	153.88
	0.20	236.11	255.16	511.86	53.33	63.34	131.24
500	0.02	1419.99	1601.43	1912.75	280.14	311.08	375.89
	0.04	1008.60	1255.55	1579.80	203.72	233.31	326.23
	0.08	682.15	722.64	1184.59	137.19	157.56	259.74
	0.10	580.78	612.67	1067.63	115.94	136.62	238.49
	0.15	393.40	437.89	864.03	79.45	96.39	200.40
	0.20	329.77	347.42	404.51	66.66	76.19	98.35

TABLE V
COMPARATIVE RESULTS OF ALGORITHMS FOR NINE DIFFERENT BENCHMARK ROUTINES

	Interference Graph			**Total Spill Cost**			**Number of Spilled Variables**		
Routine	Vertices	Edges	Maximum Spill Cost	HEA	HEA*	OCH	HEA	HEA*	OCH
vpenta	1923	14927	577	274.55	264.09	320.00	158.09	172.36	272.73
wss	408	2594	61	67.30	66.80	70.40	27.90	29.10	49.50
bmcm	324	1901	67	48.90	48.55	53.18	28.36	28.54	45.73
eflux	1489	15151	156	164.29	154.57	162.36	74.29	80.71	82.93
tomcatv	799	4499	190	68.27	66.45	82.90	55.91	54.45	57.00
btrix	2496	23081	80	371.93	355.43	362.50	131.21	141.36	203.57
adi	305	1262	79	41.42	41.00	41.71	32.57	32.14	33.14
apsi	251	1628	21	39.22	39.44	41.22	21.67	21.56	28.89
tsf	285	2398	7	45.46	45.46	47.46	18.85	18.85	26.85

ister density (β) is varied over the values from the set $\{0.02, 0.04, 0.08, 0.1, 0.15, 0.2\}$. A set of 100 different interference graphs are generated with a fixed edge density value for each problem size; and each generated graph is tested using various number of registers by varying the register density. The total number of cases considered for each edge density is equal to 600, since there are 100 different random graphs for each six different register densities. The spill costs of variables are uniformly distributed across the range of $[1, \ldots, 10]$.

Each row in Tables I and II present the performance of algorithms for a given problem size (n) and edge density (α) pair. The first set of columns in Table I are the results of pairwise comparison of the three algorithms. For each pairwise comparison, three columns are considered to present the total number of test cases (out of 600) that the first algorithm in the comparison produces better, equal, and worse total spill cost values than the second algorithm. As an example, for HEA-OCH pairwise comparison when $n = 100$ and $\alpha = 0.50$, the HEA outperforms the OCH algorithm for 564 test cases; and they give equal performance for 25 test cases. Since there are three algorithms considered, three separate column sets are provided.

We observe from these results that the solution proposed in this paper (the HEA) significantly outperforms the other two methods for all n and α pairs and the GPX algorithm gives the worst performance results for each case tested. When the edge density is very low, the test graphs become sparse graphs, and

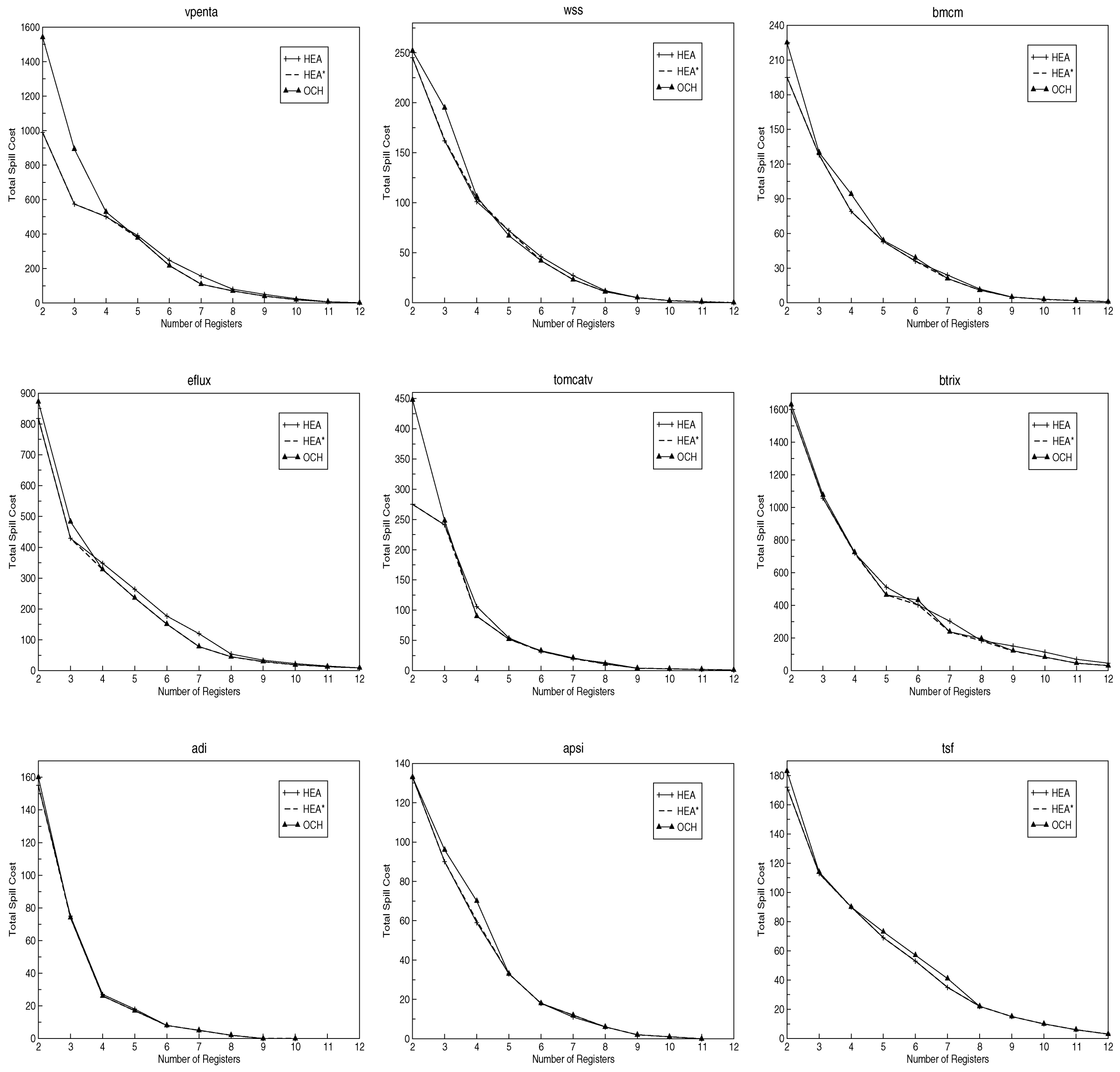


Fig. 9. The change of total spill costs versus the number of registers for various benchmark routines.

may include disconnected nodes. A graph of this type is for the code where the live ranges rarely overlap. When $\alpha = 0.05$, both the algorithms provide the same results for 81% of test cases; and our algorithm outperforms OCH for 17% of the test cases, which occurs in the test cases with the lower register density values.

The number of cases that our algorithm outperforms the other algorithm increases with an increase in edge density. When $\alpha = 1.0$, both the algorithms give the same results for all test cases. Due to full connectivity of the graphs, both algorithms tend to spill most of the variables in the code, since the register density is set only from the range of [0.02,. . .,0.20]. Therefore, the number of test cases are equal for all three algorithms, when α is equal to 1.0. Each row of Table I also presents the number of the unspilled test cases for each algorithm. The number of test cases that our algorithm does not require spilling is significantly higher than that of OCH and GPX, under several problem size and edge density pairs. For both algorithms, an increase in edge density decreases the number of unspilled cases.

The HEA significantly outperforms the related work with respect to the total spill costs and the number of spilled variables, for all n and α pairs (see Table II). The GPX algorithm gives the highest total cost values (especially for lower edge density values), which is due to the significantly higher number of spills it generates. If we examine the case when $n = 100$ and $\alpha = 1.0$, both algorithms require a spill with an average of more than 90 variables. In this case, the number of spilled variables should be between 80 and 98, since the register density is between 0.02 and 0.20. The total spill cost value is equal to 456.16 for both algorithms, which is very close to spilling 90 variables with a mean spill cost value of 5.5, since the spill costs are distributed over the range $[1, \ldots, 10]$. The difference in the total

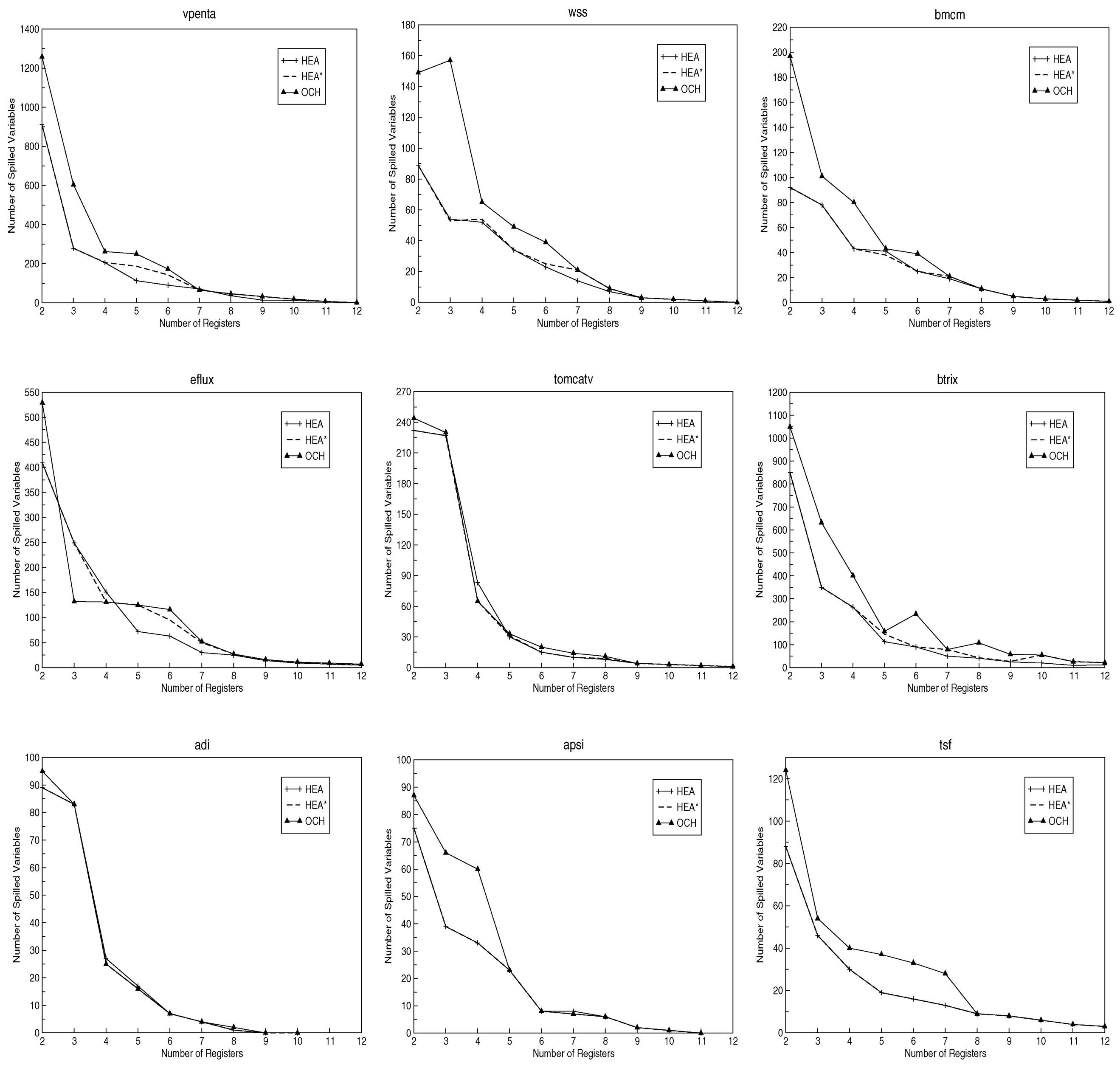


Fig. 10. The change of number of spilled variables versus the number of registers for various benchmark routines.

spill costs decreases with an increase in edge density. For the case of $n = 500$, the OCH has a 151% higher average spill cost than our HEA under $\alpha = 0.05$. This difference decreases to 7% when α is set to 0.75.

In another experiment, the performance of algorithms are compared for various register density values. The results of the pairwise comparisons and the number of unspilled test cases are given in Table III, where the unit of each cell in the table is the number of test graphs. Table IV gives the total spill costs and the number of spilled variables for each register density and graph size pairs, which are computed by averaging 600 randomly generated interference graphs. The HEA significantly outperforms the OCH and GPX algorithms, with respect to all performance metrics given in the tables.

It is expected that an increase in register density causes more number of test cases to be completed without spilling, so that the number of spilled variables for both the algorithms will decrease with an increase in register density. When Tables III and IV are examined, one can see that this expectation does not hold true for some graph size and register density pairs. Since the spill costs are not uniform and varied in the range [1, . . . ,10], in order to minimize the total spill cost, the algorithms tend to spill a set of variables with low-spill costs instead of spilling a single variable with the maximum spill cost. We have already tested our preliminary work [46] with a lower range of spill costs, where the above expectation was held for all cases. It can be concluded that an increase in register density may not come up with a decrease in the number of spilled variables, if there is a high variance among the spill costs of the variables.

Since running time of algorithms for various edge density and graph sizes is not a critical issue in our study, it is not investigated in detail. As a specific example on running time, if

$n = 100$ and $\alpha = 1.0$, the HEA takes 3.70 s and the GPX algorithm takes 2.95 s, whereas the OCH algorithm needs 0.03 s to generate the allocation of variables. Since the OCH algorithm is the fastest heuristic (among all tested), it is considered in the phase of generating the initial population for the HEA, in the remaining comparative experiments.

B. *Routines From Benchmark Suites*

This section presents the results from our experiments of the algorithms using a collection of nine routines, that are extracted from Perfect Club [47], Spec [48], and Livermore [49] benchmark suites.[2] All nine routines are written in C programming language. First, the intermediate representation of each routine is generated by GAS (GNU Assembler). Then, our parser inputs the intermediate representation of each routine and it generates an extended version of the routine which is used to construct its interference graph (as explained earlier in Section II-A).

The two important characteristics of the interference graphs of these benchmarks (i.e., the number of nodes and the number of edges), are given in Table V. It should be noted that these numbers are not the best possible numbers, and if an advanced technique such as *coalescing* [26] is considered, the number of nodes and the number of edges can further decrease. The edge density of some of the constructed graphs is lower than the values considered in the previous set of experiments. Specifically, α is equal to 0.008 for the *vpenta* routine, and it is equal to 0.007 for the *btrix* routine. For each routine given, the spill cost of each variable is set to the number of occurrences of the variables in the extended version of the routine; and the maximum spill cost value of variables in each routine is given under the third column of the table.

The number of variables spilled and the corresponding total spill costs given in Table V are the average values across various register densities. The GPX algorithm gives the highest total spill costs for each routine with significant differences with the other two algorithms, therefore, its results are not presented in the table. The HEA is extended by generating a single individual of initial population with the OCH algorithm and this extension is called as HEA^* in Table V and Figs. 9 and 10. The HEA outperforms the OCH algorithm for seven routines in terms of the total spill costs. The HEA^* improves the performance of the HEA for all cases; but, it also requires a higher number of spilled variables than the HEA for some of the tests. The HEA^* outperforms the OCH algorithm for all the nine cases tested.

The detailed performance results of the nine routines for various register sizes with respect to total spill costs and the number of spilled variables are given in Figs. 9 and 10, respectively. Our algorithms outperforms the OCH with respect to total spill costs for all register sizes; and the HEA^* may require a higher number of spilled variables than others when low register density values are considered. Some of the benchmark routines (as in eflux, btrix and tsf routines) require more than 12 registers (up to 15) in order to achieve perfect register allocation.

[2]Specifically, vpenta, tomcatv, and btrix are from Spec benchmarks; wss, bmcm, eflux, apsi, and tsf are from Perfect Club benchmarks; adi is from Livermore kernels.

V. Conclusion

The register allocation phase of compilation shapes the runtime performance of a given application code and this phase is generally more challenging in embedded environments as compared with high-end systems. In this study, we introduced a HEA for assigning program variables into available registers with the objective of minimizing the total spill cost.

The experimental evaluation based on synthetic benchmarks and real application codes reveals that our HEA significantly outperforms a widely used register allocation heuristic (the OCH algorithm) and a crossover operator of a well-known HEA for the graph coloring problem (the GPX operator given in [33]), with respect to all given metrics of solution quality. This success is mainly due to its highly specialized crossover operator, a domain-specific local search technique, and two different decoding techniques employed for evaluating fitness values of individuals.

Acknowledgment

The authors would like to thank the anonymous referees for their constructive suggestions in improving the quality of the paper. Preliminary work of this research [46] was presented at the Fourth European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP'04), Coimbra, Portugal, 2004.

References

[1] A. Chandrakasan and R. Brodersen, *Low Power Digital CMOS Design*. Norwell, MA: Kluwer, 1995.
[2] A. Chandrakasan, W. J. Bowhill, and F. Fox, *Design of High-Performance Microprocessor Circuits*. Piscataway, NJ: IEEE Press, 2001.
[3] T. Back, D. B. Fogel, and Z. Michalewicz, *Handbook of Evolutionary Computation*. New York: Inst. Physics and Oxford Univ. Press, 1997.
[4] T. Back, *Evolutionary Algorithms in Theory and Practice*. New York: Oxford Univ. Press, 1996.
[5] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Berlin, Germany: Springer-Verlag, 2003.
[6] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.
[7] Z. Michalewicz and D. B. Fogel, *How to Solve It: Modern Heuristics*. Berlin, Germany: Springer-Verlag, 2000.
[8] B. R. Fox and M. B. McMahon, "Genetic operators for sequencing problems," in *Foundations of Genetic Algorithms, First Workshop on the Foundation of Genetic Algorithms and Classifier Systems*, G. Rawlins, Ed., 1990, Morgan Kaufmann.
[9] J. J. Greffenstette, "Incorporating problem specific knowledge into a genetic algorithm," in *Genetic Algorithms and Simulated Annealing*, L. Davis, Ed. San Mateo, CA: Morgan Kaufmann, 1987.
[10] G. Raidl, "Hybrid evolutionary algorithms for combinatorial algorithms," Habilitation thesis, Vienna Univ. Technology, Vienna, Austria, 2002.
[11] P. Moscato, "On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms," Caltech, Pasadena, CA, Tech. Rep. Caltech Concurrent Computation Program Rep. 826, 1989.
[12] P. Moscato, "Memetic algorithms," Home Page. [Online]. Available: http://www.ing.unlp.edu.ar/cetad/mos/memetic_home.html .
[13] N. J. Radcliffe and P. D. Surry, "Formal memetic algorithms," in *Proc. Evol. Comput.: AISB Workshop*, 1994, vol. 865, LNCS, pp. 1–16.
[14] R. Baraglia, J. I. Hidalgo, and R. Perego, "A hybrid heuristic for the traveling salesman problem," *IEEE Trans. Evol. Comput.*, vol. 5, no. 6, pp. 613–622, Dec. 2001.
[15] B. Freisleben and P. Merz, "New genetic local search operators for the traveling salesman problem," *Lecture Notes in Computer Science*, vol. 1141, pp. 890–899, 1996.

[16] V. Schnecke and O. Vornberger, “Hybrid genetic algorithms for constrained placement problems,” *IEEE Trans. Evol. Comput.*, vol. 1, no. 4, pp. 266–277, 1997.

[17] C. R. Houck, J. A. Joines, and M. G. Kay, “Comparison of genetic algorithms, random search and two-opt switching for solving large location-allocation problems,” *Comput. Oper. Res.*, vol. 23, no. 6, pp. 587–596, 1996.

[18] G. Magyar, M. Johnsson, and O. Nevalainen, “An adaptive hybrid genetic algorithm for the three-matching problem,” *IEEE Trans. Evol. Comput.*, vol. 4, no. 2, pp. 135–146, 2000.

[19] E. Falkenauer, “A hybrid grouping genetic algorithm for bin packing,” *J. Heuristics*, vol. 2, no. 1, pp. 5–50, 1996.

[20] G. Dozier, J. Bowen, and A. Homaifar, “Solving constraint satisfaction problems using hybrid evolutionary search,” *IEEE Trans. Evol. Comput.*, vol. 2, no. 1, pp. 23–33, 1998.

[21] A. Ruiz-Andino, L. Araujo, F. Saenz, and J. Ruz, “A hybrid evolutionary approach for solving constrained optimization problems over finite domains,” *IEEE Trans. Evol. Comput.*, vol. 4, no. 4, pp. 353–372, 2000.

[22] J. He, J. Xu, and X. Yao, “Solving equations by hybrid evolutionary computation techniques,” *IEEE Trans. Evol. Comput.*, vol. 4, no. 3, pp. 295–304, 2000.

[23] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon, “Coloring heuristics for register allocation,” in *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, Portland, OR, 1989, pp. 275–284.

[24] P. Briggs, K. Cooper, and L. Torczon, “Improvements to graph coloring register allocation,” *ACM Trans. Programming Languages and Systems*, vol. 16, no. 3, pp. 428–455, 1994.

[25] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

[26] K. D. Cooper and L. Torczon, *Engineering a Compiler*. San Mateo, CA: Morgan Kaufmann, 2004.

[27] D. Brelaz, “New methods to color the vertices of a graph,” *Commun. ACM*, vol. 22, no. 4, pp. 251–256, 1979.

[28] C. Fleurent and J. Ferland, “Genetic and hybrid algorithms for graph coloring,” *Ann. Oper. Res.*, vol. 63, pp. 437–461, 1996.

[29] Z. Kokosinski, M. Kolodziej, and K. Kwarciany, “Parallel genetic algorithm for graph coloring problem,” in *Proc. 4th Int. Conf. Comput. Sci.*, 2004, vol. 3036, Lecture Notes in Computer Science, pp. 217–224.

[30] Z. Kokosinski, K. Kwarciany, and M. Kolodziej, “Efficient graph coloring with parallel genetic algorithms,” *Comput. Inform.*, vol. 24, pp. 1001–1025, 2005.

[31] R. Dorne and J. Hao, *A New Genetic Local Search Algorithm for Graph Coloring*. Berlin, Germany: Springer-Verlag, 1998, vol. 1498, Lecture Notes in Computer Science, pp. 745–754.

[32] D. A. Fotakis, S. D. Likothanassis, and S. K. Stefanakos, “An evolutionary annealing approach to graph coloring,” *Lecture Notes in Computer Science*, vol. 2037, pp. 120–129, 2001.

[33] P. Galinier and J.-K. Hao, “Hybrid evolutionary algorithms for graph coloring,” *J. Combinatorial Opt.*, vol. 3, no. 4, pp. 379–397, 1999.

[34] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopplings, and P. W. Markstein, “Register allocation via coloring,” *Comput. Languages*, vol. 6, pp. 47–57, 1981.

[35] G. J. Chaitin, “Register allocation and spilling via graph coloring,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1982, pp. 98–105.

[36] F. C. Chow and J. L. Hennessy, “Register allocation by priority-based coloring,” in *Proc. SIGPLAN Symp. Compiler Construction*, Montreal, QC, Canada, 1984, pp. 222–232.

[37] ——, “The priority-based coloring approach to register allocation,” *ACM Trans. Programming Languages and Systems*, vol. 12, no. 4, pp. 501–536, 1990.

[38] P. Pinter, “Register allocation with instruction scheduling: A new approach,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1993, pp. 248–257.

[39] ——, “Register allocation with instruction scheduling: A new approach,” *J. Programming Language*, vol. 4, pp. 21–38, 1996.

[40] K. D. Cooper and L. Torczon, Lecture Slides. [Online]. Available: http://www.owlnet.rice.edu/comp412/Lectures/

[41] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press, 1998.

[42] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[43] J. Kennedy and R. C. Eberhart, “Particle swarm optimization,” in *Proc. IEEE Int. Conf. Neural Netw.*, 1995, vol. IV, pp. 1942–1948.

[44] M. Dorigo and G. Di Caro, “The ant colony optimization meta-heuristic,” in *New Ideas in Optimization*, D. Corne, M. Dorigo, and F. Glover, Eds. New York: McGraw-Hill, 1999.

[45] T. R. Jensen and B. Toft, *Graph Coloring Problems*. New York: Wiley, 1995.

[46] B. Demiroz, H. Topcuoglu, and M. Kandemir, “A hybrid evolutionary algorithm for solving the register allocation problem,” *Lecture Notes in Computer Science*, vol. 3004, pp. 62–71, 2004.

[47] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, and C. Hsiung *et al.*, “The PERFECT club benchmarks: Effective performance evaluation of supercomputers,” *Int. J. Supercomput. Appl.*, vol. 3, no. 3, pp. 5–40, 1988.

[48] Spec Benchmarks, Home Page. [Online]. Available: http://www.spec.org

[49] F. McMohan, The Livermore Fortran Kernels: A computer test of the numerical performance range Lawrence Livermore Nat. Lab., Livermore, CA, Tech. Rep. UCRL-53745, Dec. 1986.

Haluk Rahmi Topcuoglu (S’95–M’03) received the B.Sc. and M.Sc. degrees in computer engineering from Bogazici University, Bogazici, Istanbul, in 1991 and 1993, respectively, and the Ph.D. degree in computer science from Syracuse University, Syracuse, NY, in 1999.

He is an Associate Professor in the Computer Engineering Department, Marmara University, Istanbul, Turkey. His main research interests include task scheduling and mapping for parallel and distributed systems, parallel computing, evolutionary algorithms, and evolutionary computing for dynamic environments.

Dr. Topcuoglu is a member of the IEEE Computer Society and the Association for Computing Machinery (ACM).

Betul Demiroz received the B.Sc. and M.Sc. degrees in computer engineering from Marmara University, Istanbul, Turkey, in 2002 and 2004, respectively. She is currently working towards the Ph.D. degree in computer engineering at Bogazici University, Istanbul, Turkey, where she is a Research Assistant in the Computer Engineering Department.

Her research interests include task scheduling and mapping algorithms, chip multiprocessing, and genetic algorithms.

Mahmut Kandemir (S’98–M’03) received the B.Sc. and M.Sc. degrees in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively, and the Ph.D. degree in electrical engineering and computer science from Syracuse University, Syracuse, NY, in 1999.

He is an Associate Professor in the Department of Computer Science and Engineering, Pennsylvania State University, University Park. His main research interests are optimizing compilers, I/O intensive applications, and power-aware computing. His research is currently funded by the National Science Foundation (NSF) (including CAREER Award), SRC, and the Defense Advanced Research Projects Agency (DARPA).

Dr. Kandemir is a member of the Association for Computing Machinery (ACM).