Hybrid Evolutionary Algorithm based solution for Register Allocation for Embedded Systems

Anjali Mahajan
G H Raisoni College of Engineering, Nagpur, India
Email : armahajan@rediffmail.com

M S Ali
Prof. Ram Meghe Institute of Technology and Research,
Badnera, Amravati, India
Email : softalis@hotmail.com

Abstract— **Embedded systems have an ever-increasing need for optimizing compilers to produce high quality codes with a limited general purpose register set. Either memory or registers are used to store the results of computation of a program. As compared to memory, accessing a register is much faster, but they are scarce resources and have to be utilized very efficiently. The optimization goal is to hold as many live variables as possible in registers in order to avoid expensive memory accesses. We present a hybrid evolutionary algorithm for graph coloring register allocation problem based on a new crossover operator called crossover by conflict-free sets(CCS) and a new local search function.**

Index Terms—**compilers, compiler optimization, register allocation, hybrid evolutionary algorithm, embedded systems**

I. Introduction

Register allocation is one of the most important optimizations a compiler performs and is becoming increasingly important as the gap between processor speed and memory access time widens. Its goal is to find a way to map the temporary variables used in a program into physical memory locations (either main memory or machine registers).

Accessing a register is much faster than accessing memory; therefore one tries to use registers as much as possible. Of course, this is not always possible, thus some variables must be transferred (spilled) to and from memory. This has a cost, the cost of *load* and *store* operations, which should be avoided as much as possible. Typically, this degrades runtime performance and increases power consumption. Therefore, an efficient mapping should generally minimize the register requirements and the number of spilling instructions. The critical applications, especially in embedded computing, industrial compilers are ready to accept longer compilation times if the final code gets improved.

This approach attempts to combine the flexibility of general-purpose programmable processors with the performance achieved by domain-specific architecture optimizations. Compilers for embedded processors must cope with these architectural optimizations and be able to exploit them.

This paper presents a heuristic algorithm for graph coloring register allocation problem for embedded systems based on a new crossover operator and a new local search function.

Graph coloring abstracts the problem of assigning registers to live ranges in a program into the problem of assigning colors to nodes in an *interference graph*. The register allocator attempts to “color” the graph with a finite number of machine registers, with one constraint that any two nodes connected by an interference edge must be colored with different registers.

To model register allocation as a graph coloring problem, the compiler first constructs an interference graph G. The nodes in G correspond to live ranges, and the edges represent interferences. Thus, there is an edge in G from node i to node j if live range of i interferes with live range of j, that is, if they are simultaneously live at some point and cannot occupy the same register.

To find an allocation from G, the compiler looks for a k-coloring of G, that is, an assignment of k colors to the nodes of G such that adjacent nodes always have distinct colors. If we choose k to match the number of machine registers, then we can map a k-coloring of G into a feasible register assignment for the underlying code. Because graph coloring is NP-complete, the compiler uses a heuristic method to search for a coloring; it is not guaranteed to find a k-coloring for all k-colorable graphs. If a k-coloring is not discovered, some values are spilled; the values are kept in memory rather than in registers.

Spilling one or more live ranges creates a new and different interference graph. The compiler proceeds by iteratively spilling some live ranges and attempting to color the resulting new graph.

In the context of embedded processors, the most important restriction of the graph coloring approach is that it is based on the assumption of a homogenous register set. The different phases of register allocation are

© 2008 ACADEMY PUBLISHER

differently impacted by embedded processor irregularities.

II. Related work

Register allocation has been widely addressed in literature, and many approaches have been proposed. Most of them are related to Chaitin's [6] approach, but very few address problems raised by embedded processor's architecture like support for irregular registers via register classes and register set concatenation. Although [13] proposed an approach using integer-programming supporting irregular register sets, this approach requires modeling of register constraints by inequations, making it difficult to implement in an industrial compiler. Briggs[3,4] proposes an approach to model register pairs in the interference graph. He introduced an improved coloring strategy that produces better allocations for many graphs on which Chaitin's method fails. The difference lies in the timing of spill decision. George and Apple [9] introduced iterating coalescing where they eliminated aggressive coalescing completely.

Embedded processors are often characterized by a small number of registers. Algorithms and heuristics have been adapted in our infrastructure to limit spill when few registers are available. Ref. [17] presents a generalization of the *degree* < *K* test , called the $\langle p, q \rangle$ test, to handle irregular register sets and register classes. VPO [13] is a portable optimizer for DSPs, based on the Zephyr retargetable compiler infrastructure. VPO address the problem of heterogeneous register classes by computing and propagating along the IR tree the correct reduced register class (register class resulting from the intersection of the target/source operand of two instructions) that allows registers to be allocated without inserting an extra *move* operation. The PROPAN postpass optimizer [14] relies on integer linear programming to perform register allocation and scheduling.

Some researchers attempt to use non-graph-coloring methods. Koes[15] proposes a progressive register allocator which uses a multi-commodity networkflow mode tp represent the intricacies of irregular architectures. Scholz[18] formulates register allocation as a partitioned boolean quadratic optimization problem that allows generic modeling of processors peculiarities. Hirnschrott [11] used Partitioned Boolean Quadratic Programming for register allocation shows better results in spill costs and coalescing benifits. Mahajan [16] present a new hybrid evolutionary algorithm (HGR) for graph coloring register allocation problem for embedded systems as one of the most efficient algorithms with highly specialized, domain specific crossover and local search function.

There are many local search algorithms for graph coloring, such as simulated annealing [7], tabu search[10],etc Galinier and Hao[8] introduced a algorithm based on hybrid evolutionary algorithm. An important feature of this algorithm is a specialized crossover but the mutation operator of the GA is replaced by an LS operator.

III. A hybrid evolutionary algorithm

Evolutionary algorithms involve natural evolution and genetics. The genetic algorithm is a classical method in this category. It has been applied to various optimization problems. There are several other methods like genetic programming, ant colony optimization, etc. The simple evolutionary algorithms are not generally efficient for complex combinatorial problems. The performance of the evolutionary algorithms is improved with the addition of problem specific knowledge. Specialized operators are combined with evolutionary algorithms to generate complex hybrid systems called hybrid genetic algorithms, hybrid evolutionary algorithms, genetic local search algorithms and memetic algorithms.

An approach for combinatorial optimization is to embed local search into the framework of population based evolutionary algorithm, leading to hybrid evolutionary algorithm. HEA is based on two elements: an efficient local search (LS) operator and a highly specialized crossover operator. The basic idea consists in using the crossover operator to create new and potentially interesting configurations which are then improved by the LS operator.

We present a hybrid evolutionary algorithm for graph coloring based on a new crossover operator for register allocation problem and a new local search function. This section presents our algorithm- Hybrid evolutionary Algorithm for Graph coloring Register Allocation with proposed operator called crossover by conflict-free sets(CCS).

A. The Algorithm

The HEA consists of a genetic component and a local search (LS). The genetic component initializes and evolves a population of solutions. The general algorithm is as given below:

Input *: Interference graph , IG = (V,E) ; number of registers , k*
Output *: best configuration*
begin
P = generate_population(|P|)
iter = 0
while *(iter ≤ MaxIter* ***or*** *popu-diversity>0* ***)*** ***do***
(p1, p2) = select_parents(P)
p = crossover (p1, p2)
p = local_search(p, L)
P = update_population(P,p)
iter =iter + 1
endwhile
end

The algorithm first builds an initial population of configurations (*generate_population)* and then performs a series of cycles called generation. At each generation, two configurations *p1* and *p2* are chosen in the population (*select_parents).* A crossover is then used to produce an offspring *p* from *p1* and *p2 (crossover).* The LS operator is applied to improve *p* for a fixed number L of iterations *(local_search).* Finally, the improved

© 2008 ACADEMY PUBLISHER

configuration p is inserted in the population by replacing another configuration (*update_population*). This process repeats until the value of *iter* is less than or equal to a prefixed number *MaxIter* is reached or population diversity (*popu-diversity*) is greater than zero.

Population diversity is calculated as the average distance between all configurations in the population. For two configurations p1 and p2, the distance between p1 and p2 is the minimum number of elementary transformations necessary to transform p1 into p2.

In our approach, we consider the partition method for string representation [8]. Each solution P_i partitions the variables into register classes, $P_i = \{R_1 , R_2 , ... R_k\}$ where each class R_i includes the live ranges of variables that are mapped to the registers r_i and k is the total number of registers. Given two parents p1= $\{ R^1_1 , R^1_2 , ... R^1_k\}$ and p2= $\{ R^2_1 , R^2_2 , ... R^2_k\}$, the partial configuration will be a set $\{ R_1 , R_2 , ... R_k\}$ of disjoint sets of nodes where each subset R_i is included in a class of one of the two parents, and all R_i are conflict-free sets. (the nodes i and j in an interference graph are said to be conflict-free, when there is no edge connecting them)

B. Initial population generation

Generally the initial population is generated using the DSatur algorithm [5], which is a graph coloring heuristic. 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 register allocation problem, both the spill costs and the degree of the nodes in a given interference graph are considered. We adopt the new metric called the *spill degree* proposed by [17] that can be used for ordering the nodes. The spill degree of a node i can be defined by one of the three equations given below

$$SDegree_1(i) = SCost(i) \times Degree(i)$$
$$SDegree_2(i) = SCost(i) \times Degree^2(i)$$
$$SDegree_3(i) = SCost(i) \qquad (1)$$

In these expressions, *SCost(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 (1). The nodes are sorted in decreasing order of spill degrees. At each iteration, an unsigned node with the maximum 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. This process is repeated until all the nodes (i.e., their corresponding variables) are mapped to one of the register classes.

C. The crossover operator

The crossover used here is the new proposed operator Crossover by Conflict-free Sets (CCS). Given two parent configurations p1= $\{ R^1_1 , R^1_2 , ... R^1_k\}$ and p2= $\{ R^2_1 , R^2_2 , ... R^2_k\}$, chosen randomly by the *select_parents* operator from the population, the algorithm *crossover (p1, p2)* builds an offspring *p*= $\{R1 , R_2 , ... R_k\}$ as follows

```
Input: configurations p1= { R^1_1 , R^1_2 , .... R^1_k}
       and p2= { R^2_1 , R^2_2 , .... R^2_k}
Output: configuration p= { R_1 , R_2 , .... R_k}
begin
// consider p1
if conflict(p1)>0 then
                call function conflict_free(p1)
// consider p2
if conflict(p2)>0 then
                call function conflict_free(p2)
  for i (1≤ i ≤ k ) do
CfR^1_n= max q∈p1 Cf_Individual(q)
// CfR^1_n  is the partition with maximum number of
//conflict-free variables from p1
CfR^2_n= max q∈p2 Cf_Individual(q)
// CfR^2_n  is the partition with maximum number of
// conflict-free variables from p2
 choose j such that | CfR^1_n ∩ CfR^2_j| is maximum
 R_i = |CfR^1_n ∪ CfR^2_j|
 SpillQuality(R_i)=Spillcost(R_i)*Reg_class_Conflict(R_i)
 remove the nodes of R_i from p1 and p2
   endfor
   assign randomly the nodes of R- (R_l U…U R_k)
end

function conflict_free(p)
begin
 Cf_Individual=∅
for  i ( 1≤ i ≤ k) do
   if Reg_class_Conflict(i) >0 then
// Reg_class_Conflict(i) is the total number of conflicts
// in i^th  register class
begin
initialize t to 0
while (t < (Reg_class_Conflict(i)/2)) do
        select a variable m from R_i where
        Conflict(m)=max c∈Ri{Conflict(c)} or
        spillcost(m)=min c∈Ri{spillcost(c)}
       remove variable m from R_i
       t=t+Conflict(m)
endwhile
CfR(i)= R_i
endif
Cf_Individual= Cf_Individual ∪ CfR(i)
  endfor
end
```

The algorithm builds step by step the k classes R_1 , R_2 … R_k of the offspring. If any register set of the parent has conflict, the algorithm first determines the conflict free sets of all the register classes of each parent. The function to find the conflict free set is based on the heuristic from [20] for determining the conflict-free set with the maximum number of nodes of each register class. In that, initially the conflict-free set includes all variables in R^l. The total number of conflicts of each variable m in class R^l, *conflict(m)* are determined. The sum of the *conflict(m)* values gives us the total number of conflicts in the register class R^l, *Reg_class_Conflict(l).* Then, the variables from the partition are removed one by

© 2008 ACADEMY PUBLISHER

one in decreasing order of *conflict(m)* values, until the total number of conflicts in R^l becomes equal to ½ of its initial value. The function gives us the conflict free register classes of each individual parent. We then consider the partition CfR^1_n , which has *maximum number of conflict-free nodes from p1* . It then selects a partition CfR^2_j from p2 that has largest number of nodes in common with CfR^1_n. The algorithm unifies the pair of these two partitions. It calculates the conflict factor of the produced partition. If it is zero, the partition is conflict free. It then removes all the nodes belonging to offspring register partition from the parents p1 and p2. In case of tie-breaking, any partition is taken randomly. This new partition becomes the base set of the first register class for the offspring. The process is subsequently repeated for the other register classes.

At the end of k steps, some nodes may remain unassigned. These are then assigned to a class which has minimum *conflict factor*.

D. The Local Search LS operator

After a solution is generated using the crossover operator, the local search phase improves it before inserting it into the population for a maximum of L iterations. We have used a Local search operator LS. It uses a 1-exchange neighborhood. Formally, given a partition { R_1 , R_2 , …. R_k}, a neighboring partition is obtained by assigning a single variable to other register set, i.e., a variable v_i is removed from the original register class R_i to which it belongs to and where v_i is already in conflict with one or more variables in that class. It is moved into a new register class R_j.

A variable v_i which has maximum *conflict factor* is selected. Assume that v_i is already assigned to class k. If the fitness value of the register class k with v_i is more than the fitness of register class j without considering v_i, then v_i is moved from register class k to register class j. This process is repeated for other register classes also. The register class which has minimum value is the final register class for variable v_i.

The same process is repeated with a new variable according to the decreasing order of *conflict factor.* In this way the quality of generated offspring is improved. It is then inserted in the population by replacing the worst of the two parents.

E. Fitness function calculation

To solve a register allocation problem, we consider a set of k - register classes. In the interference graph IG = (V, E), all variables (i.e., nodes) v_i are assigned to k - register classes. The nodes i and j in an interference graph are said to be conflict-free, when there is no edge connecting them. *Conflict(m)* denotes the conflict of a variable *m* in register class R_i. *Spillcost* is the spill cost of variable *m*. The fitness of a register class is given as

$$fitness(R_i)=\sum_{\forall m\in R_i} Conflict(m)*Spillcost(m)/degree(m) \quad (3)$$

The total sum of fitness values of all register classes gives the fitness value of the given individual

$$Fitness = \sum_{i=1}^{k} fitness(i) \quad (4)$$

The goal of the optimization process is to minimize fitness until zero.

IV. Experimental Evaluation

The experiments are based on MachineSUIF [19] compiler research framework. The register allocator of MachineSUIF implements George-Appel's 'Iterated Register Coalescing' algorithm [9]. Our experimental analysis includes three algorithms We compare our algorithm with these algorithms. The first is the MachineSUIF's default algorithm[9] with extenstions by Smith-Ramsey and Holloway [19] for handling register aliasing. We denote it by IRC. The second is based on GPX operator[8]. The third is optimal register allocation algorithm ORA [11] is based on PBQP[17] . The algorithm runs in worst-case exponential time and does optimal spilling with respect to a set Boolean constraint generated from the program.

We have implemented our HEA allocator with the proposed crossover operator CCS (called HEA henceforth), our HEA by replacing its CCS with GPX operator. We replace the existing original allocator in SUIF/MachineSUIF compiler research framework by these allocators; other parts of the framework are not changed. We have used x86architecture with its limited register file and register usage constraint. Our machine is a 2.8 Ghz Pentium 4 with 1 GB of RAM.

We applied the algorithms to 6 embedded and real time applications. For each application, the population size is set to 20 and LS iterations are all set from 500 to 2000. For each application, we run the allocator five times and average the results.

Performance evaluation was done with respect to the following parameters : number of memory accesses required , spill loads, spill costs, the compile time needed by the allocator, the execution time of the generated code, including the number of load/store generated , size of allocator itself.

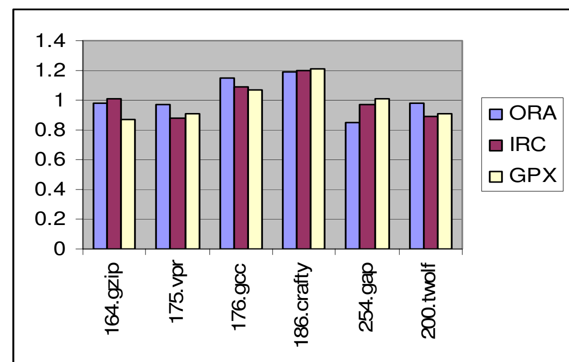

Figure 1. Number of memory accesses

© 2008 ACADEMY PUBLISHER

TABLE I
SPILL COST OF INSTRUCTION

Bench marks	**Vert ices**	**Edg es**	**Total spill cost**			
			IRC	***GPX***	***HEA***	ORA
164.gzip	165	1592	76	58	43	56
175.vpr	169	1688	82	72	49	67
176.gcc	201	1263	124	97	62	88
186.crafty	721	4529	93	66	41	59
254.gap	220	2448	17	13	9	15
200.twolf	318	1900	28	27	26	28

Comparison of number of memory accesses is the comparison of total static number of load and store instructions inserted by each register allocator. Figure 1. compares the number of load/store instructions in the assembly code. The HEA inserts fewer memory- access instruction than ORA, 1.6 % fewer memory- access instruction than IRC and 8.9 % fewer memory- access instruction than GPX.

Table I give the spill costs of all the algorithms. For lowest population size the spill cost is less. For each benchmark given, the spill cost of each variable is set to the number of occurrences of the variables. The spill costs of the variables are the average values. The IRC algorithm gives the highest total spill cost. The HEA algorithm produce less spill cost than the maximum in four tests and it outperforms the IRC and GPX algorithms in all tests. Genetic operator systematically eliminates low-quality solutions from the population, preserve diversity between solutions, and provide better input for local search. A small amount of spill cost is due to function callers and callees saving many contents of registers in order to preserve correct program semantics.

Spill loads refers to additional number of loads incurred by the allocation algorithm. Spill loads give an indication of how well the allocator is able to perform the task. The number of spill loads is highly correlated with application running time.

We calculate the dynamic number of spill loads added to each module of the program by multiplying the number of spill loads added to each block by the number of times that block is executed. Then we sum the number of dynamic spill loads added to each block. We obtain the dynamic number of spill loads for the entire program by summing the number of dynamic spill loads added to each module.

Table II shows the spill loads for each allocator as a ratio to spill loads generated by IRC allocator (considered as base allocator for comparison). The numbers are given as geometric mean. We see improvements of HEA over other allocator.

TABLE II
RATIO OF THE SPILL LOADS PRODUCED

Benchmarks	**GPX**	**HEA**	**ORA**
164.gzip	1.07	1.04	1.11
175.vpr	1.33	1.16	1.24
176.gcc	1.00	1.00	1.09
186.crafty	1.46	1.23	1.34
254.gap	2.03	1.34	1.98
200.twolf	1.09	1.02	1.10

Table III gives results in terms of compile time and run time. We observe that in most of the cases, the performance of different allocators were almost similar in terms of compile times; however, the execution time of the code generated by our method is less than others in most cases.

Code quality is measured in terms of number of loads/stores generated for of the program and the number of static instructions generated. Table IV give the total number of loads/stores generated for 8 registers. Our algorithm is superior to all other algorithms.

V. CONCLUSION

Register allocation for embedded systems is complex having to deal with irregular architectural characteristics and to meet strict requirements.

In this paper, we propose a novel crossover operator for graph coloring register allocation problem for embedded systems, based on hybrid evolutionary algorithm. Compared to classical graph coloring register allocation algorithms, this technique can deal with irregular architectural characteristics of embedded processors more uniformly. Our experimental results indicate that compared to other algorithms, HEA is one of the most efficient algorithms with highly specialized, domain specific crossover and local search function. It outperforms the IRC algorithm with minimized spill costs. Being hybrid algorithm, it is not computationally expensive to obtain high quality solutions. The HEA can be easily parallelized, thus the computing time can be largely reduced.

Population size of the problem should be set properly. Compared to the size of population, the length of iterations L of LS is more critical on the performance of the algorithm.

In the future, we intend to test our HEA heuristic on graphs using coalescing and on chordal interference graphs generated from a large set of applications. HEA can also be applied to other compiler optimization problems such as instruction scheduling and phase coupling of the compiler.

© 2008 ACADEMY PUBLISHER

TABLE III
RESULTS OF EXECUTION TIME

benchmarks	execution time in milliseconds											
	IRC			***GPX***			***HEA***			***ORA***		
	ct	***rt***	***tt***	***ct***	***rt***	***tt***	***ct***	***rt***	***tt***	***ct***	***rt***	***tt***
164.gzip	0.2	0.033	0.233	0.2	0.029	0.229	0.2	0.003	0.203	0.2	0.013	0.213
175.vpr	0.3	0.354	0.654	0.2	0.657	0.857	0.3	0.153	0.453	0.3	0.358	0.658
176.gcc	0.1	0.014	0.114	0.1	0.121	0.221	0.1	0.011	0.111	0.2	0.239	0.439
186.crafty	0.6	0.235	0.835	0.5	0.233	0.733	0.4	0.202	0.602	0.7	0.295	0.995
254.gap	0.2	0.983	1.183	0.2	1.203	1.403	0.2	0.936	1.136	0.2	0.923	1.123
200.twolf	0.6	0.340	0.940	0.4	0.123	0.523	0.4	0.140	0.540	0.6	0.221	0.821

(ct : compile time; rt : run time ; tt : total time)

TABLE IV
RESULT OF LOAD/ STORE GENERATED

benchmarks	Total Number of Loads / Stores Generated											
	IRC			***GPX***			***HEA***			***ORA***		
	load	***store***	***total***	***load***	***store***	***total***	***load***	***store***	***total***	***load***	***store***	***total***
164.gzip	29	23	52	39	25	64	16	7	23	34	19	53
175.vpr	291	234	525	345	135	480	152	129	281	122	131	253
176.gcc	18	12	30	15	13	28	10	8	18	19	12	31
186.crafty	57	19	76	66	57	123	57	114	171	64	68	132
254.gap	432	124	556	149	163	312	58	110	168	149	74	223
200.twolf	15	12	27	28	28	56	17	9	26	26	26	52

ACKNOWLEDGMENT

We thank the reviewers for their useful suggestions.

REFERENCES

[1] P. Bergner, P. Dahl, D. Engebretsen, and Matthew T. O'Keefe. "Spill code minimization via interference region spilling". *In SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

[2] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. "Spill code minimization techniques for optimizing compliers". *In Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 258–263. ACM Press, 1989.

[3] P. Briggs. "Register Allocation via Graph Coloring". *Ph.d Thesis*, Rice University, April 1992.

[4] P. Briggs, K.Cooper, and L. Torczon. "Improvements to graph coloring register allocation". *ACM Transactions on Programming Languages and Systems*, 16(3):428-455, May 1994.

[5] D. Brelaz, " New methods to color the vertices of a graph. Communication". *ACM*, vol. 22, no 4, pages 251-256, 1979.

[6] G.J. Chaitin. "Register Allocation and Spilling via Graph Coloring". *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices* 17(6):98-105, June 1982.

[7] M. Chams, A. Hertz, and D. de Werra. "Some experiments with simulated annealing for coloring graphs". *European Journal of Operational Research,*(32): 260-266, 1997

[8] P. Galinier and J.K. Hao. "Hybrid evolutionary algorithms for graph coloring". *Journal of Combinatorial Optimization,*(3):379-397,1999

[9] L. George, A. Appel. "Iterated Register Coalescing". *ACM Transactions on Programming Languages and Systems, 18(3):300-324, May 1996*.

[10] A. Hertz, and D. de Werra. "Using Tabu search technique for coloring graphs". *Computing,*(39):345-351,1987

[11] U. Hirnschrott, A.Krall, B. Scholz. "Graph Coloring vs. Optimal Register Allocation for Optimizing Compliers". In Proceeding of the Joint Modular Languages Conference (JMLC'03), Lecture Notes in Computer Science (LNCS), Vol. 2789, pp. 202-213, Springer Press, Klagenfurt, Austria August 2003.

[12] M. S. Johnson and T. C. Miller. "Effectiveness of a machine-level, global optimizer". In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 99–108, New York, NY, USA, 1986. ACM Press.

[13] S. Jung, Y. Paek. "The Very Portable Optimizer for Digital Signal Processors". *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01),November 2001, Pages 84-92*.

[14] T. Kong, K.D. Wilken. "Precise Register Allocation for Irregular Architecture". *Proceedings of the 31st Annual ACM/IEEE International Symposium on Micro architecture (MICRO '98), November 1998, Pages 297-307.*

[15] D. Koes and S.C.Goldstein. "A progressive register allocation for irregular architectures". *In 3rd IEEE/ACM International Symposium on Code generation and optimization (CGO 2005), San Jose, CA, USA,* pages 269-280, IEEE Computer Society, 2005

[16] A. Mahajan, M.S.Ali. "Hybrid Evolutionary Algorithm for Graph Coloring Register Allocation". *Proceedings of the 2008 IEEE Congress on Evolutionary Computation (WCCI-CEC '08),Hong kong June 2008.*

[17] J. Runeson, S-O Nystrom. "Retargetable Graph-Coloring Register Allocation for Irregular Architectures". *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES'03), September 2003, Pages 240-254.*

[18] B. Scholz and E. Eckstein. "Register allocation for irregular architectures". *Proceedings of Joint Conference*

© 2008 ACADEMY PUBLISHER

on Languages, Compilers and Tools for Embedded systems & software and Compilers for Embedded systems (LCTES 02-SCOPES 02), Berlin , Germany, pages 139-148, ACM, 2002

[19] M.D Smith, N. Ramsey, G. Holloway. " A generalized algorithm for graph coloring register allocation." *Proceedings of PLDI,* pages 277-288, 2004.

[20] H. R. Topcuoglu, B. Demiroz, and M. Kandemir. "Solving the register allocation problem for embedded systems using a hybrid evolutionary algorithm." *IEEE Transaction on Evolutionary Computation,*(11, 5): October 2007, pages 620-234

[21] http://www.eecs.harvard.edu/hube/research/machsuif.html

[22] http://archi.snu.ac.kr/realtime/benchmark/. SNU real-time benchmarks suit.

© 2008 ACADEMY PUBLISHER