A mathematical model and a metaheuristic approach for a memory allocation problem

María Soto André Rossi
Marc Sevaux
Université de Bretagne-Sud – Lab-STICC, CNRS, UMR 3192
Centre de recherche – BP 92116 – F-56321 Lorient Cedex, FRANCE
Corresponding author: marc.sevaux@univ-ubs.fr

Received: date / Accepted: date

Abstract

Memory allocation in embedded systems is one of the main challenges that electronic designers have to face. This part, rather difficult to handle is often left to the compiler with which automatic rules are applied. Nevertheless, an optimal allocation of data to memory banks may lead to great savings in terms of running time and energy consumption. This paper introduces an exact approach and a VNS-based metaheuristic for addressing a memory allocation problem. Numerical experiments have been conducted on real instances from the electronic community and on DIMACS instances expanded for our specific problem.

keywords: Electronic design Memory allocation MILP VNS-TS

1 Introduction

This paper deals with the optimization of cache memory allocation; this choice is motivated by the fact that processor cache memory management deeply impacts the performances and power consumption of electronic devices [28]. The continuous advances in microelectronic technology have made possible the development of miniaturized chips that allow new embedded products to be enhanced (smart phones and high definition image processing are typical examples). As technology empowers the integration of more and more functionalities into these electronic chips, their design becomes more and more challenging. Furthermore, these high-tech products have to hit the market within a very short amount of time as innovation in the field of microelectronics also makes them subject to rapid obsolescence.

For a long time, electronic chips used to be designed manually by experts who mastered a given technology, and were able to control the complexity of

the whole product. Such a line of design is no longer possible, and computer-aided design softwares like Gaut [9] have been developed to generate chips from their specifications. While the design process is significantly faster with these types of software, the generated layouts are considered to be poor on power consumption and surface compared to human expert designed circuits. This is a major drawback as embedded products have to feature low-power consumption. The need for optimization is expected to become even more stringent in the future, as embedded systems will run heavy computations. As an example, some cell phones already support multithreading operating systems.

Furthermore, designers of electronic devices want to find a trade-off between architecture cost and power consumption [1]. Generally, electronic practitioners consider that to some extent, minimizing power consumption is equivalent to minimizing application running time on a given chip [6]. Moreover, power consumption of a chip can be estimated through consumption models based on measurements as shown in [18]. As a consequence, memory allocation must be such that loading operations are performed in parallel as often as possible.

It is assumed that the application to be implemented on such systems (e.g. MPEG encoding, filtering or any other signal processing algorithms) is provided as a C source code, and the data structures involved have to be mapped into memory bank. Another hypothesis is the absence of an operating system (OS) in the chip. Indeed, OS manage memory with methods of their own, so the results presented in this work would require to change it, which is generally impossible. The term *data structures* refers to the variables (scalars, arrays, structures) of the application. Due to cost and technological reasons, the number and the capacity of memory banks is limited; an external memory with unlimited capacity is then assumed to be available for storing data (it models mass memory storage), but the data stored in that external memory are accessed p times slower than if they were in a memory bank.

An access time is spent by the processor for loading data structures (this time is referred to as access cost). All memory banks can be accessed simultaneously; then, data structures `a` and `b` can be loaded at the same time when an operation in which they are jointly involved is to be performed (e.g. `a+b`) provided that `a` and `b` are allocated to two different memory banks. If they are allocated to the same memory bank, then they must be loaded sequentially and more time is needed; so, `a` and `b` are said to be *conflicting* if they are involved in the same operation.

Each conflict has a cost, which is proportional to the number of times the corresponding operation is performed in the application. Such a situation happens when operations appear in a loop. When the number of iterations of a loop cannot be forecasted (as in a `while` loop), code profiling tools can be used for assessing conflict costs on a statistical basis [17, 20]. Thus, a

conflict is said to be *closed* if its data structures are allocated to two different memory banks, and it is said to be *open* if both data structures are allocated to the same memory bank, or if at least one data structure is allocated to the external memory. In such a case, its conflict cost is multiplied by p because accessing the external memory bank is p times longer. Finally, if both structures are allocated to the external memory, then the conflict cost is multiplied by $2p$.

The problem addressed in this paper is referred to as MemExplorer, it is stated as follows: for a given number of capacitated memory banks and an external memory, we search for a memory allocation for data structures such that the time spent loading these data is minimized. We have studied a simplified version of MemExplorer in [26]. A mixed integer linear program is designed for this purpose, it is introduced in section 2. Some metaheuristics are proposed in section 3, and exact and heuristic approaches are compared in section 4, conclusions are drawn in section 5.

2 A Mixed integer linear programming formulation for MemExplorer

The number of data structures is denoted by n and the number of memory banks is denoted by m. Memory bank $m+1$ refers to the external memory. The size of data structure i is denoted by s_i for all $i \in \{1, \ldots, n\}$. The capacity of memory bank j is c_j for all $j \in \{1, \ldots, m\}$ (it is recalled that the external memory is not subject to capacity constraint). Sizes and capacities are expressed in the same memory unit (typically Kbytes). Besides its size, each data structure i is also characterized by an access cost denoted by e_i for all $i \in \{1, \ldots, n\}$, that represents the time required to access data structure i if it is mapped to a memory bank. If a data structure is mapped to the external memory its access cost is multiplied by factor p. Any conflict k is associated with conflict cost d_k for all $k \in \{1, \ldots, o\}$, where o is the number of conflicts. If a data structure is allocated to the external memory then all the conflicts in which it is involved have their cost multiplied by p. More formally, conflict k between two data structures a_k and b_k has one of the four following statuses:

- Status 1: a_k and b_k are mapped in two different memory banks. The conflict does not generate any cost.
- Status 2: a_k and b_k are mapped in the same memory bank. The conflict generates a cost d_k.
- Status 3: a_k and b_k are such that one of these data structures is mapped in a memory bank and the other one is mapped in the external memory. The conflict generates a cost pd_k.

- Status 4: a_k and b_k are mapped in the external memory. The conflict generates a cost $2pd_k$.

Conflict costs and access costs are expressed in the same time unit (typically milliseconds).

It must be stressed that a data structure can be conflicting with itself. This case arises when two elements of the same data structure are involved in the same instruction in the original C code, for example in `a[i] = a[i+1]`. This particular case is taken into account in our approach.

The number of memory banks with their capacities and factor p describe the architecture of the chip. The number of data structures, their size and access cost describe the application, whereas the conflicts and their costs carry information on both the architecture and the application.

The problem decision variables represent the allocation of data structures to memory banks. These variables are modeled as a binary matrix X, where:

$$x_{i,j} = \begin{cases} 1, \text{ if data structure } i \\ \quad \text{is mapped to memory bank } j \\ 0, \text{ otherwise} \end{cases}, \quad \begin{array}{l} \forall i \in \{1,\dots,n\}, \\ \forall j \in \{1,\dots,m+1\} \end{array} \tag{1}$$

The vector of real nonnegative variables Y models the conflict statuses; so variable y_k associated with conflict k has four possible values: 0, 1, p and $2p$.

The mixed integer program for that problem is the following:

$$\text{Min} \sum_{k=1}^{o} y_k d_k + \sum_{i=1}^{n}\sum_{j=1}^{m} \left(e_i x_{i,j}\right) + p \sum_{i=1}^{n} \left(e_i x_{i,m+1}\right) \tag{2}$$

$$\sum_{j=1}^{m+1} x_{i,j} = 1, \quad \forall i \in \{1,\dots,n\} \tag{3}$$

$$\sum_{i=1}^{n} x_{i,j} s_i \leq c_j, \quad \forall j \in \{1,\dots,m\} \tag{4}$$

$$x_{a_k,j} + x_{b_k,j} \leq 1 + y_k, \quad \forall j \in \{1,\dots,m\},\ \forall k \in \{1,\dots,o\} \tag{5}$$

$$x_{a_k,j} + x_{b_k,m+1} \leq 1 + \frac{1}{p} y_k, \quad \forall j \in \{1,\dots,m\},\ \forall k \in \{1,\dots,o\} \tag{6}$$

$$x_{a_k,m+1} + x_{b_k,j} \leq 1 + \frac{1}{p} y_k, \quad \forall j \in \{1,\dots,m\},\ \forall k \in \{1,\dots,o\} \tag{7}$$

$$x_{a_k,m+1} + x_{b_k,m+1} \leq 1 + \frac{1}{2p} y_k, \quad \forall k \in \{1,\dots,o\} \tag{8}$$

$$x_{i,j} \in \{0,1\} \quad \forall (i,j) \in \{1,\dots,n\} \times \{1,\dots,m\} \tag{9}$$

$$y_k \geq 0 \quad \forall k \in \{1,\dots,o\} \tag{10}$$

The *cost function* of the problem, equation (2) is the total time spent accessing the data structures and storing them in the appropriate registers to perform the required operations listed in the C file. It is the sum of three terms. The first one is the total cost generated by the conflicts. The second term is the access cost of all data structures mapped to a memory bank, whereas the last term is the access cost of all data structures placed in the external memory.

The constraints are as follows. First, equation (3) enforces that each data structure is allocated either to a unique memory bank or to the external memory. Equation (4) is used for ensuring that the total size of the data structures allocated to a memory bank does not exceed its capacity. Third, for any conflict k, variable y_k must be set appropriately. This is enforced through equations (5) to (8). Inequality (5) prevents y_k from being less than 1 if conflict k has status 2. Inequalities (6) and (7) prevent y_k from being less than p if conflict k has status 3, and inequality (8) prevents y_k from being less than $2p$ if conflict k has status 4. Fourth, $x_{i,j}$ is a binary variable, for all (i, j) and y_k is nonnegative for all k.

Note that this problem is similar to the *k-weighted graph coloring problem* [4] (this problem is also referred to as the *generalized graph coloring problem* in [19]) if memory banks are not subject to capacity constraints, or if their capacity is large enough for holding all the data structures. Indeed, in that case the external memory is no longer used and the size, as well as the access cost of data structures can be ignored. The k-weighted graph coloring problem is to color the vertices of an undirected weighted graph with at most k colors so as to minimize the sum of the weighted edges, the end points of which have the same color. In this problem, the vertices represent data structures and each edge represents a conflict between a pair of structures [26]. The k-weighted graph coloring problem has been addressed using Local Search Programming in [27].

A close related problem is to determine the minimum number of memory banks with infinite capacity so as to have no open conflicts, this turns out to be the classical graph coloring problem [10, 25].

An optimal solution to MemExplorer problem can be computed by using a solver like GLPK [15] or Xpress-MP [12]. However, as shown by the computational tests in Section 4, an optimal solution cannot be obtained in a reasonable amount of time for medium size instances. This is the reason why metaheuristics are proposed in the next section.

3 Metaheuristics for MemExplorer

In this section, we describe the design of the different metaheuristics used for addressing this problem. Before presenting the metaheuristics for MemExplorer, we present the algorithms used for generating initial solutions, as well

as two neighborhoods. Then, a Tabu Search-based approach is introduced with the two neighborhoods for exploring the solution space. At the end of this section, a Variable Neighborhood Search-based approach hybridized with a Tabu Search-inspired method (VNS-TS called `Vns-Ts MemExplorer`) is also presented.

3.1 Generating initial solutions

Below, we present two ways for generating initial solutions. The first one generates feasible solutions at random, and the second one builds solutions using a greedy algorithm.

Random initial solutions

Fig. 1 presents the procedure `RandomMemex` for generating random feasible initial solutions. At each iteration, a data structure is allocated to a random memory bank (or the external memory) provided that capacity constraints are satisfied.

Output: $[X^*, f^*]$
`Initialization:`
Capacity used: $u_j \leftarrow 0, \forall j \in \{1, \ldots, m+1\}$
Allocation: $x^*_{ij} \leftarrow 0, \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m+1\}$
$f^* \leftarrow 0$
`Assignment:`
for $i \leftarrow 1$ **to** n **do**
  repeat
    Generate j at random in $\{1, \ldots, m+1\}$
  until $u_j + s_i \leq c_j$;
  $x^*_{i,j} \leftarrow 1$
  $u_j \leftarrow u_j + s_i$
  Compute g_{ij}, the cost generated from allocating the data i to memory bank j
  $f^* \leftarrow f^* + g_{ij}$
end

Figure 1: Pseudo-code for `RandomMemex`

Greedy initial solutions

`GreedyMemex` is a greedy algorithm for MemExplorer, this kind of algorithm makes locally optimal choices at each stage in the hope of finding the global

optimum [3, 8]. Generally, greedy algorithms do not reach an optimal solution as they are trapped in local optima, but they are easy to implement and can provide initial solutions to more advanced approaches.

```
Input: A ← {a_1, ..., a_n}
Output: [X*, f*]
Initialization:
Capacity used: u_j ← 0, ∀j ∈ {1, ..., m+1}
Allocation: x*_ij ← 0, ∀i ∈ {1, ..., n}, ∀j ∈ {1, ..., m+1}
f* ← 0
Assignment:
for i ← 1 to n do
    h* ← ∞        // (auxiliary variable for the partial greedy
       solution)
    for j=1 to m+1 do
        if u_j + s_{a_i} < c_j then
            Compute g_ij, the cost for allocating data a_i to memory bank j
            if g_ij < h* then
                b ← j
                h* ← g_ij
            end
        end
    end
    x*_{a_i,b} ← 1
    u_b ← u_b + s_{a_i}
    f* ← f* + h*        //(total cost of the solution)
end
```

Figure 2: Pseudo-code for `GreedyMemex`

`GreedyMemex` is described in pseudocode of Fig. 2, where A is a permutation of the set $\{1, \ldots, n\}$ that models data structures, used for generating different solutions. Solution X^* is the best allocation found by the algorithm, where $(x^*_{i,j})$ variables have the same meaning as in equation (1), and $f^* = f(X^*)$. Matrix G is used to assess the cost when data structures are moved to different memory banks or to the external memory. More precisely, $g_{i,j}$ is the sum of all open conflict costs produced by the affectation of data structure i to memory bank j. If data structure i is moved to external memory ($j = m + 1$), $g_{i,j}$ is the sum of all open conflict costs multiplied by p plus its access cost multiplied by $(p - 1)$. The numerical value of $g_{i,j}$ depends on the current solution because the open conflict cost depends on the allocation of the other data structures.

At each iteration, `GreedyMemex` completes a partial solution which is initially empty by allocating the next data structure in A. The allocation for the current data structure is performed by assigning it to the memory bank leading to the minimum local cost denoted by h^*, provided that no memory

bank capacity is exceeded. The considered data structure is allocated to the external memory if no memory bank can hold it. Allocation cost f^* is returned when the all data structures have been allocated.

`GreedyMemex` has a computational complexity of $\mathcal{O}(nm)$. Both algorithms require very few computational efforts, but return solutions that may be far from optimality. However, these procedures are not used as standalone algorithms, but as subroutines called in the algorithm of Fig. 3 for generating initial solutions for a Tabu search-based procedure introduced in Section 3.2.

Input: A
Output: $[X^*, f^*]$
if $A \leftarrow \emptyset$ **then**
  $(X^*, f^*) \leftarrow$`RandomMemex`
else
  $(X^*, f^*) \leftarrow$`GreedyMemex`(A)
end

Figure 3: Pseudo-code for `InitialMemex`

3.2 A Tabu Search Procedure for MemExplorer

Tabu search is a metaheuristic that relies on a single local search procedure: it iteratively moves from the current solution to another one in its neighborhood [14]. Generally, local search procedures stop when a local optimum is found, then it becomes necessary to escape from the local optimum to explore other regions of the search space.

We introduce tabu search for MemExplorer in Fig. 4, which is based on *TabuCol*, an algorithm for graph coloring introduced in [16]. The main difference with a classic tabu search is that the size of the tabu list is not constant over time. This idea is introduced in [2] and also used in the work of Porumbel, Hao and Kuntz on the graph coloring problem [23]. In TabuMemex, the size of the tabu list NT is set to $a + NTmax \times t$ every $NTmax$ iterations, where a is a fixed integer and t is a random number in $[0, 2]$.

A pair (i, j) means that data structure i is in memory bank j. A move is a trio (i, h, j), this means that data structure i, which is currently in memory bank h, is to be moved to memory bank j. As a consequence, if the move (i, h, j) is performed, then the pair (i, h) is appended to the tabu list. Thus, the tabu list contains the pairs that have been performed in the recent past and it is updated on the FIFO basis (First In First Out).

The algorithm takes an initial solution X as input that can be returned by `InitialMemex`. Its behavior is controlled by some calibration parameters, such as the number of iterations, $Niter$, and the number of iterations for

```
Input: Initial solution X and number of neighborhood k
Output: [X*, f*]
Initialization:
Capacity used u_j ← 0   ∀j ∈ {1, ..., m}
NT ← NTmax
f* ← ∞
Iterative phase:
Iter ← 0
while Iter < Niter and f(X) > 0 do
    [X', (i, h, j)] ← Explore-Neighborhood-N_k(X)
    X ← X'
    if f(X') < f* then
        f* ← f(X')
        X* ← X'
    end
    Update the size of tabu list NT
    Iter ← Iter + 1
end
Update the tabu list with pairs (i, j) and (i, h)
```

Figure 4: Pseudo-code for `TabuMemex`

changing the size of the tabu list, $NTmax$. The result of this algorithm is the best allocation found X^* and its cost f^*.

The iterative phase searches for the best solution in the neighborhood of the current solution. The neighborhood exploration is performed by calling `Explore-Neighborhood-`$\mathcal{N}_k(X)$ which calls the corresponding procedure with only one neighborhood used at a time. Two neighborhoods, denoted by $\mathcal{N}_0$ and $\mathcal{N}_1$ are considered; they are introduced in the next section. The fact that the new solution may be worse than the current solution does not matter because each new solution allows unexplored regions to be reached, and thus to escape local optima. This procedure is repeated for $Niter$ iterations, but the search stops if a solution without any open conflict, and for which the external memory is not used is found. Indeed, such a solution is necessarily optimal because the first and third terms of Equation (2) are zero because no conflict cost has to be paid, and no data structure is in the external memory. Consequently, the objective function assumes its absolute minimum value, the second term of Equation (2), and so is optimal. A new solution is accepted as the best one if its total cost is less than the current best solution.

This tabu search procedure is used as a local search procedure in a VNS-based algorithm introduced in section 3.4.

3.3 Exploration of neighborhoods

In this section, we present two algorithms which explore two neighborhoods for MemExplorer. Both of them return the best allocation (X') found along with the corresponding move (i, h, j) performed from a given solution X. In these algorithms, a move (i, h, j) is said to be non tabu if pair (i, j) is not in the tabu list. The first one explores a neighborhood which is generated by performing a feasible allocation change of a single data structure, it is shown in Fig. 5.

Input: X
Output: $[X', (i, h, j)]$
Find non tabu min cost move (i, h, j), such that $h \neq j$ and $u_j + s_i \leq c_j$
Build the new solution X' as follows:
$X' \leftarrow X$
$x'_{i,h} \leftarrow 0$
$x'_{i,j} \leftarrow 1$
$u_j \leftarrow u_j + s_i$
$u_h \leftarrow u_h - s_i$

Figure 5: Pseudo-code for `Explore-Neighborhood-`$\mathcal{N}_0$

Algorithm `Explore-Neighborhood-`$\mathcal{N}_1$ is presented in Fig. 6. It explores solutions that are beyond $\mathcal{N}_0$ by allowing for unfeasible solutions before repairing them.

The first phase of `Explore-Neighborhood-`$\mathcal{N}_1$ performs a move that may make the current solution X' unfeasible by violating the capacity constraint of a memory bank. However, this move is selected to minimize the cost of the new solution, and is not tabu. The second phase restores the solution by performing a series of reallocations for satisfying capacity constraints, but also trying to generate the minimum allocation cost. Then, it allows both feasible and infeasible regions to be visited successively. This way of using a neighborhood is referred to as *Strategic Oscillation* in [14].

3.4 A hybrid Variable Neighborhood Search (VNS-TS)

Since both neighborhoods have their own utility (confirmed by preliminary tests), it seems clear that they should be used together in a certain way. The general Variable Neighborhood Search scheme is probably the most appropriate method to properly deal with several neighborhoods.

Fig. 7 presents the VNS-based algorithm ([21]) for MemExplorer. The number of neighborhoods is denoted by $kmax$, and the algorithm starts exploring $\mathcal{N}_0$ as $\mathcal{N}_0 \subset \mathcal{N}_1$.

At each iteration, `Vns-Ts MemExplorer` generates a solution X' at random from X. It copies the affectation of 60% of the data structures in the initial solution (the 60% of data structures is selected randomly), and the

Input: X
Output: $[X', (i, h, j)]$
`First phase: considering a potentially unfeasible move`
Find non tabu min cost move (i, h, j), such that $h \neq j$
Build the new solution X' as follows:
  $X' \leftarrow X$
  $x'_{i,h} \leftarrow 0$
  $x'_{i,j} \leftarrow 1$
  $u_j \leftarrow u_j + s_i$
  $u_h \leftarrow u_h - s_i$
`Second phase: repairing the solution`
while $u_j > c_j$ **do**
  Find non tabu min cost move (l, j, b), such that $l \neq i$, $j \neq b$ and $u_b + t_l \leq c_b$
  Update solution X' as follows:
    $x'_{l,j} \leftarrow 0$
    $x'_{l,b} \leftarrow 1$
    $u_b \leftarrow u_b + s_l$
    $u_j \leftarrow u_j - s_l$
end

Figure 6: Pseudo-code for `Explore-Neighborhood-`$\mathcal{N}_1$

`GreedyMemex` is used for mapping the remaining 40% of unallocated data structures for producing a complete solution X'.

This VNS algorithm relies on two neighborhoods. $\mathcal{N}_0$ is the smallest neighborhood, as it is restricted to feasible solutions only. If `TabuMemex` improves the current solution, it keeps searching for new solutions in that neighborhood. Otherwise, it does not accept the new solution and changes the neighborhood (*i.e.* by applying `Explore-Neighborhood-`$\mathcal{N}_1$ to the current solution).

4 Computational results

This section presents the relevant aspects of implementation of the algorithms, and the results reached by algorithms over two set of instances on an Intel Pentium IV processor system at 3 GHz and 1 Gbyte RAM. Algorithms have been implemented in C++ and compiled with `gcc 4.11`.

4.1 Instances used

For testing our algorithms we have used two sets of instances. The first one is called LBS, it is a collection of real instances provided by Lab-STICC laboratory for electronic design purposes. The second set of instances is called DMC, it originates from DIMACS [22], a well-known collection of online graph

```
output: X* and f*
Initialization:
Generate A
(X*, f*) ←InitialMemex(A)
k ← 0
Iterative phase:
i ← 0
while i < Nrepet do
    // (Make a new initial solution X from X*)
    X ← 60%X* complete the solution with GreedyMemex
    Apply (X', f') ←TabuMemex(X, k) using Explore-Neighborhood-N_k
    if f' < f* then
        X* ← X'
        f* ← f'
        i ← 0
        k ← 0
    else
        if k = kmax then
            k ← 0
        else
            k ← k + 1
        end
        i ← i + 1.
    end
end
```

Figure 7: Pseudo-code for `Vns-Ts MemExplorer`

coloring instances. The instances in DMC have been enriched by generating edge costs at random so as to create conflict costs, access costs and sizes for data structures, and also by generating a random number of memory banks with random capacities.

Although real-life instances available today are relatively small, they will be larger and larger in the future as market pressure and technology tend to integrate more and more complex functionalities in embedded systems. Thus, we tested our approaches on current instances and on larger (but artificial) ones as well, for assessing their practical use for forthcoming needs.

4.2 Results

To our best knowledge, there are no alternative approaches for MemExplorer in the literature. The k-weighted graph coloring problem can be addressed by Local Search Programming [27], so we have tested the local search on instances of MemExplorer. To this end, we have used LocalSolver 1.0 [11] which is a solver for combinatorial optimization entirely based on local search. This solver address a combinatorial optimization problem by performing autonomous moves which can be viewed as a structured ejec-

tion chains applied to the hypergraph induced by boolean variables and constraints [24]. Results of that method are also reported.

In our experiments, in the tabu search procedure the size of the tabu list is set every $NTmax = 50$ iterations to $NT = 5 + NTmax \times t$, where t is a real number selected at random in interval $[0, 2]$. The maximum number of iterations has been set to $Niter = 50000$.

Table 1 presents `Vns-Ts MemExplorer` performances, *i.e.*, the best results obtained over all the combinations of different initial solutions and different greedy algorithms for generating a solution X'. `Vns-Ts MemExplorer` results are compared with Local Solver Programming and the MILP formulation solved by Xpress-MP. The MILP formulation is used as a heuristic when the time limit of one hour is reached: the best solution found so far is then returned by the solver. A lower bound found by the solver was also calculated but the value for non-optimal solutions was useless.

In Table 1 the first three columns show the main features of the instances (the source, the name, n: the number of data structures, o: the number of conflicts and m: the number of memory banks). Instance name with a "*" are ones of LBS set which are the real life instances. For a clear view of the difficulty, the instances have been sorted in non-decreasing order of number of conflicts. The next two columns report the cost and CPU time (in s) of `Vns-Ts MemExplorer`, the two following columns the cost and CPU time of Local Solver, and the last three columns the results of the MILP model: lower bound, cost and CPU time.

Bold figures in Table 1 represent the best known solutions over all methods. In the MILP columns, the cost with a star has been proved optimal by Xpress-MP. `Vns-Ts MemExplorer` reaches the optimal solution for all of instances for which the optimal cost is known. The optimal solution is known for 88% of real-electronic instances and for 31% of DIMACS' instances. Furthermore, `Vns-Ts MemExplorer` always finds a better allocation cost than Xpress-MP. The number of best solutions reported by our approach is 38, compared to 16 with Local Solver and 24 with the MILP model.

Indeed, on average the MILP cost is improved by 35.29% using the VNS algorithm. CPU time comparison of `Vns-Ts MemExplorer` and MILP shows that our algorithm remains significantly faster than MILP in most cases. On average, the time spent by Xpress-MP is 1700 times longer than the time spent by VNS algorithm. When no optimal solution is found with Xpress-MP, the lower bound on the objective value seems to be of poor quality, as it is 37% of the best solution found on average. This may suggest that after one hour of computation, the optimal solution would still require a very long time to be found. For the instances for which the optimal solution is not known, the lower bound is often far from the best known solution. It is also important to note that the MILP performs well on small size instances (up to 250 conflicts) since it benefits from very performant advances in its code (like internal branch-and-cut, cut pool generation and presolver).

Table 1: `Vns-Ts MemExplorer`, Local Solver and MILP results

Instances			`Vns-Ts MemExplorer`		Local Solver		MILP		
Set	Name	$n\backslash o\backslash m$	Cost	Time	Cost	Time	L. bound	Cost	Time
LBS	`compress`*	6 \6 \2	**511232**	0.09	**511232**	1.00	511232	**511232***	0.03
LBS	`volterra`*	8 \6 \2	**1**	< 0.01	**1**	1.00	1	**1***	0.33
LBS	`adpcm`*	10 \7 \2	**224**	< 0.01	**224**	1.00	224	**224***	0.08
LBS	`cjpeg`*	11 \7 \2	**641**	0.2	**641**	1.00	641	**641***	0.05
LBS	`lmsb`*	8 \7 \2	**3140610**	0.18	16745739	200	3140610	**3140610***	0.50
LBS	`lmsbv`*	8 \8 \2	**2046**	< 0.01	**2046**	1.00	2046	**2046***	0.03
LBS	`spectral`*	9 \8 \2	**640**	< 0.01	**640**	1.00	640	**640***	0.03
LBS	`gsm`*	19 \17 \2	**86132**	0.34	**86132**	1.00	86132	**86132***	0.06
LBS	`lpc`*	15 \19 \2	**790**	0.42	**790**	200	790	**790***	0.19
DMC	`myciel3`	11 \20 \2	**377**	0.68	**377**	1.00	377	**377***	0.17
LBS	`turbocode`*	12 \22 \3	**2294**	0.43	**2294**	300	2294	**2294***	0.34
LBS	`treillis`*	33 \61 \2	**12.06**	1.43	**12.06**	200	12.06	**12.06***	0.28
LBS	`mpeg`*	68 \69 \2	**786.5**	0.88	**786.5**	1641	786.5	**786.5***	0.36
DMC	`myciel4`	23 \71 \3	**2853**	1.94	**2930**	1.00	2853	**2853***	16.30
DMC	`mug88_1`	88 \146 \2	**1020**	6.33	**1379**	3596	1020	**1020***	31.23
DMC	`mug88_25`	88 \146 \2	**918**	7.00	**1263**	3483	918	**918***	13.71
DMC	`queen5_5`	25 \160 \3	**1338**	2.47	**8507**	140	1338	**1338***	1616
DMC	`mug100_1`	100 \166 \2	**2652**	6.74	**2788**	2810	2652	**2652***	2392
DMC	`mug100_25`	100 \166 \2	**2661**	5.40	**2791**	1198	2661	**2661***	1165
DMC	`r125.1`	125 \209 \3	**346**	8.94	361	31.00	260.33	**346**	3600
LBS	`mpeg2enc`*	127 \236 \2	**32.09**	7.21	39.2	6.00	32.09	**32.09***	6.48
LBS	`mpeg2enc2`*	180 \236 \2	**32.09**	8.93	36.3	892	32.09	**32.09***	4.69
DMC	`myciel5`	47 \236 \3	**2990**	4.56	3254	11	1420.54	3098	3600
DMC	`queen6_6`	36 \290 \4	**8656**	14.63	9029	1940	4213.43	8871	3600
LBS	`mpeg2`*	191 \368 \2	**61476.52**	8.78	61480.1	740	61476.52	**61476.52***	12.00
DMC	`queen7_7`	49 \476 \4	**13951**	10.93	14414	10.00	4708.61	14972	3600
DMC	`queen8_8`	64 \728 \5	**15132**	10.48	15389	7.00	482.77	17183	3600
LBS	`mpeg2x2`*	382 \736 \4	122831.26	0.05	**122828.7**	834	122826.97	122831.26	3600
DMC	`myciel6`	95 \755 \2	**9135**	5.54	10532	2065	9135	**9135***	1437
LBS	`ali`*	192 \960 \6	**7951**	248.45	7965	3600	4738.9	8009	3600
DMC	`myciel7`	191 \2360 \4	**3347**	37.15	9001	269	6.17	5140	3600
DMC	`zeroin_i3`	206 \3540 \15	**707**	26.80	757	2936	15	962	3600
DMC	`zeroin_i2`	211 \3541 \15	**575**	51.67	878	1396	15	829	3600
DMC	`r125.5`	125 \3838 \18	**20502**	36.67	47403	3572	61.33	85026	3600
DMC	`mulsol_i2`	188 \3885 \16	1470	91.59	**1255**	3299	31.61	5722	3600
DMC	`mulsol_i1`	197 \3925 \25	543	944.49	**520**	3183	30	543	3600
DMC	`mulsol_i4`	185 \3946 \16	1149	30.19	**1047**	1325	30.19	1169	3600
DMC	`mulsol_i5`	186 \3973 \16	**730**	53.17	2022	1383	15	1840	3600
DMC	`zeroin_i1`	211 \4100 \25	716	50.07	**497**	2816	15	1050	3600
DMC	`r125.1c`	125 \7501 \23	**91433**	44.55	266463	3210	15	289868	3600
DMC	`fpsol2i3`	425 \8688 \15	**1921**	52.50	2313	3571	19.29	3468	3600
DMC	`fpsol2i2`	451 \8691 \15	**1006**	89.38	1813	3563	30	2059	3600
DMC	`inithx_i1`	864\18707\27	**739**	204.28	1154	3590	15	2878	3600
Number of optimal sol.			23		11			23	
Number of best sol.			38		16			24	
Avg. impr. on MILP:			35.29%						
Avg. CPU time (s):				48.27		1349.44			1881.95

For the initial solutions, we have used three different sorting procedures for permutation A of data structures; then, we have three `GreedyMemex` algorithms: in the first one, A is not sorted; in the second one, A is sorted by decreasing order of the maximum conflict cost involving each data structure and in the last one, A is sorted by decreasing order of the sum of the conflict cost involving each data structure. Hence, we have four initial solutions (random initial solutions and greedy solutions) and three ways of mapping the 40% of solution X' in VNS algorithm.

However, other tests showed that the benefit of using different initial solutions and different greedy algorithms to generate X' is not significant. In fact, this benefit is visible only for the most difficult instances with a low value of 1.2% on average, and for the other instances, VNS algorithm finds the same solutions no matter the initial solution or greedy algorithm.

In the VNS, the search is intensified by using `TabuMemex` as a local search procedure in the research space. To assess the benefit of this strategy, we have tested our VNS with a classic tabu search method (*i.e.*, without changing the size of the tabu list), and we have also tested `TabuMemex` with each neighborhood. Table 2 shows the comparison between `Vns-Ts MemExplorer` performances, a VNS variant with the classical tabu search and the tabu search alone with each of the two neighborhoods. The first two columns of Table 2 are the same as Table 1, the next four columns report the cost value of each variant of the approach.

The costs reached by the other variants of VNS are worse in most cases, in fact the solution cost of `Vns-Ts MemExplorer` with classic tabu search is on average 35% higher than with `TabuMemex`; in addition the tabu searches with each neighborhood (namely $\mathcal{N}_0$ and $\mathcal{N}_1$) are on average 56% and 21% worse than `Vns-Ts MemExplorer`, respectively. This shows the benefit of the joint use use of different neighborhoods and an advanced tabu search method.

4.3 Statistic analysis

We have used the Friedman test [13] to detect differences in the performance of three heuristics (`Vns-Ts MemExplorer`, local search, MILP formulation) using the results presented by Table 1. As the result over instances are mutually independent and costs as well as CPU times can be ranked, we have applied the Friedman test for costs and CPU times. This allows us to compare separately (univariate model [5]) the performance in terms of solution quality and running time.

For each instance, the CPU times of the three approaches are ranked as follows. The smallest CPU time is ranked 1, the largest one is ranked 3. If two CPU times are equal, their rank is computed as the average of the two candidate ranks (*i.e.*, if two CPU times should be ranked 1 and 2, the rank is 1.5 for both). The same is performed for solution objective value.

Table 2: Intensity of some local search variants

Instances		Vns-Ts M.	VNS with	Tabu search neighborhood	
Name	$n\backslash o\backslash m$	cost	classic tabu	$\mathcal{N}_0$	$\mathcal{N}_1$
compress*	6 \6 \2	511232	511232	511232	511232
volterra*	8 \6 \2	1	1	1	1
adpcm*	10 \7 \2	224	224	224	224
cjpeg*	11 \7 \2	641	641	641	641
lmsb*	8 \7 \2	3140610	16745700	16745700	16745700
lmsbv*	8 \8 \2	2046	2046	2046	2046
spectral*	9 \8 \2	640	640	640	640
gsm*	19 \17 \2	86132	86132	86132	86132
lpc*	15 \19 \2	790	790	790	790
myciel3	11 \20 \2	377	2167	377	377
turbocode*	12 \22 \3	2294	2294	2294	2294
treillis*	33 \61 \2	12.06	12.06	12.06	12.06
mpeg*	68 \69 \2	786.5	790.88	786.5	790.5
myciel4	23 \71 \3	2853	2853	2877	2853
mug88_1	88 \146 \2	1020	1068	1036	1020
mug88_25	88 \146 \2	918	1095	918	950
queen5_5	25 \160 \3	1338	1342	1342	1342
mug100_1	100 \166 \2	2652	2735	2901	2662
mug100_25	100 \166 \2	2661	2734	2661	2661
r125.1	125 \209 \3	346	349	429	347
mpeg2enc*	127 \236 \2	32.09	36.59	32.2	32.47
mpeg2enc2*	180 \236 \2	32.09	38.48	32.2	33.22
myciel5	47 \236 \3	2990	3033	3281	2990
queen6_6	36 \290 \4	8656	8810	9257	8754
mpeg2*	191 \368 \2	61476.52	61480.2	61476.5	61479.3
queen7_7	49 \476 \4	13951	14186	15120	14107
queen8_8	64 \728 \5	15132	15480	15455	15360
mpeg2x2*	382 \736 \4	122831.26	122831.26	122831.26	122831.26
myciel6	95 \755 \2	9135	9706	9135	9135
ali*	192 \960 \6	7951	8123	8053	8088
myciel7	191 \2360 \4	3347	3741	4116	3548
zeroin_i3	206 \3540 \15	707	754	2233	791
zeroin_i2	211 \3541 \15	575	632	954	607
r125.5	125 \3838 \18	20502	22735	22993	22609
mulsol_i2	188 \3885 \16	1470	1779	3651	1480
mulsol_i1	197 \3925 \25	543	755	955	792
mulsol_i4	185 \3946 \16	1149	1085	1382	1197
mulsol_i5	186 \3973 \16	730	800	3729	732
zeroin_i1	211 \4100 \25	716	661	841	1516
r125.1c	125 \7501 \23	91433	94479	96528	94358
fpsol2i3	425 \8688 \15	1921	1973	3125	2121
fpsol2i2	451 \8691 \15	1006	1015	2184	1106
inithx_i1	864 \18707 \27	739	820	1698	850
Avg. worsening:			35%	56%	21%

The number of instances is denoted by r, the number of compared metaheuristic is dented by q and the Friedman test statistic is denoted by Q. The null hypothesis suppose that for each instance the ranking of the metaheuristics is equally likely. The null hypothesis is rejected at the level of significance α if Q is greater than the $1-\alpha$ quantile of the $F_{(q_1,q_2)}$-distribution (Fisher-Snedecor distribution) with $q_1 = q-1$ and $q_2 = (q-1)(r-1)$ degrees of freedom.

The test statistic Q is 21.86 for the running time, and 13.52 for the cost. Moreover, the value for the $F_{(2,84)}$-distribution with a significance level $\alpha = 0.01$ is 4.90. Then, we reject the null hypothesis for running time and cost at the level of significance $\alpha = 0.01$.

We can conclude that there exists at least one metaheuristic whose performance is different from at least one of the other metaheuristics. To know which metaheuristics are really different, it is necessary to perform an appropriate post-hoc paired comparisons test.

4.4 Post-hoc paired comparisons

As the null hypothesis of Friedman test was rejected, we can use the following method for knowing if two metaheuristics are different [7].

Let A_2 be the total sum of squared ranks, B_2 the total sum of squared sum of ranks of metaheuristics and R_i the sum of ranks of metaheuristics i for all i in $\{1, \ldots, q\}$. We say that two metaheuristics are different if:

$$|R_i - R_j| > \sqrt{\frac{2r(A_2 - B_2)}{(r-1)(q-1)}} t_{(1-\frac{\alpha}{2},q_2)} \tag{11}$$

where $t_{\left(1-\frac{\alpha}{2},q_2\right)}$ is the $1-\frac{\alpha}{2}$ quantile of the t-distribution with $(r-1)(q-1)$ degrees of freedom.

For $\alpha = 0.01$, $t_{(0.095,84)}$-distribution is 2.64; then, the left-hand side of equation (11) for the running time is 20.06 and for the cost is 17.44. Table 3 summarizes the paired comparisons for the cost and running time. The bold values means the metaheuristics are different.

Table 3: Paired comparisons

Cost paired test			Running time paired test		
$\lvert R_i - R_j \rvert$	MILP	Local search	$\lvert R_i - R_j \rvert$	MILP	Local search
`Vns-Ts MemExplorer`	**26**	**32.5**	`Vns-Ts MemExplorer`	**42**	**45**
MILP	-	6.5	MILP	-	3
Critical value	17.44		Critical value	20.06	

The post-hoc test shows that MILP and local search have the same performance in terms of solution cost and CPU time, while `Vns-Ts MemExplorer` is the best approach in terms of solution cost and computational time.

5 Conclusion

In this work, an exact approach and a VNS-based metaheuristic are proposed for addressing a memory allocation problem. `Vns-Ts MemExplorer` takes advantage of some features of tabu search methods initially developed for graph coloring, which is efficient as relaxing capacity constraints on memory banks lead to the k-weighted graph coloring problem. `Vns-Ts MemExplorer` appears to be performing well because of its reasonable CPU time for large instances, and because it returns an optimal memory allocation for all instances for which the optimal cost is known. These results allow one to hypothesize that the solutions found for the instances for which the optimal solution is unknown are of good quality. The improvements over a classic tabu search approach, like the implementation of a variable tabu list, have a significant impact on solution quality. These features have `TabuMemex` exploring the search space efficiently.

`Vns-Ts MemExplorer` achieves encouraging results for addressing the MemExplorer problem due to its intensive search. The search is intensified by exploring the largest neighborhood when a local optimum is found, in addition the local search method (`TabuMemex`) gives a more in-depth search because of the significant improvements over a classic tabu search procedure. Using methods inspired by graph coloring problems can be successfully extended to more complex allocation problems for embedded systems, thereby assess the gains made by using these methods to specific cases in terms of energy consumption. Moreover, it gives good perspectives for using metaheuristics in the field of electronic design.

Finally, if the exact approach is suitable for today's applications, it is clearly not for tomorrow's needs. Indeed, the best solution returned by the solver is generally very poor even after a long running time, and the quality of the lower bound is too bad for being helpful at all. The proposed metaheuristics appear to be suitable for the needs of today and tomorrow. The very modest CPU time compared to the exact method is an additional asset for integrating them to CAD tools, letting designers test different options in a reasonable amount of time.

Acknowledgements
This research was partially supported by the Région Bretagne and the grant R2 - Allocations de recherche doctorale - ARED,211-B2-9/ARED,2008,CG2M.

References

[1] D. Atienza, S. Mamagkakis, F. Poletti, J. Mendias, F. Catthoor, L. Benini, and D. Soudris. Efficient system-level prototyping of power-aware dynamic memory managers for embedded systems. *Integration, the VLSI Journal*, 39(2):113–130, 2006.

[2] R. Battiti. The reactive tabu search. In *ORSA Journal on Computing*, volume 6, pages 126–140, 1994.

[3] P.E. Black. Greedy algorithm. Dictionary of Algorithms and Data Structures, U.S. National Institute of Standards and Technology, 2005.

[4] R.C. Carlson and G.L. Nemhauser. Scheduling to minimize interation cost. *Operations Research*, 14(1):52–58, 1966.

[5] M. Chiarandini, . Paquete, M. Preuss, and E. Ridge. Experiments on metaheuristics: Methodological overview and open issues. Technical Report DMF-2007-03-003, The Danish Mathematical Society, Denmark, 2007.

[6] A. Chimientia, L. Fanucci, R. Locatellic, and S. Saponarac. VLSI architecture for a low-power video codec system. *Microelectronics Journal*, 33(5):417–427, 2002.

[7] W.J. Conover. *Practical nonparametric statistic*. Wiley, New York, USA, 1999. Third edition.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter Greedy Algorithms, pages 370–404. The Massachusetts Institute of Technology, second edition, 1990.

[9] P. Coussy, E. Casseau, P. Bomel, A. Baganne, and E. Martin. A formal method for hardware IP design and integration under I/O and timing constraints. *ACM Transactions on Embedded Computing System*, 5(1):29–53, 2006.

[10] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, Heidelberg, Germany, 2005.

[11] Bouygues e-lab Innovation & Optimisation. Localsolver 1.0, 2010. `http://e-lab.bouygues.com/?p=693`.

[12] FICO. Xpress-MP, 2009. `http://www.dashoptimization.com/`.

[13] M. Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32:675–701, 1937.

[14] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publisher, Dordrecht, The Netherlands, 1997.

[15] GNU. GLPK linear programming kit, 2009. `http://www.gnu.org/software/glpk/`.

[16] A. Herz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.

[17] M. Iverson, F. Ozguner, and L. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Transactions on Computers*, 48(12):1374–1379, 1999.

[18] N. Julien, J. Laurent, E. Senn, and E. Martin. Power consumption modeling and characterization of the TI C6201. *IEEE Micro*, 23(5):40–49, 2003.

[19] A.W.J. Kolen and J.K. Lenstra. *Handbook of Combinatorics*, chapter Combinatorics in operations research, pages 1875–1910. Elsevier Science, Amsterdam, The Netherlands, 1995.

[20] W. Lee and M. Chang. A study of dynamic memory management in C++ programs. *Computer Languages Systems and Structures*, 28(3):237–272, 2002.

[21] N. Mladenović and P. Hansen. Variable neighbourhood decomposition search. *Computers and Operations Research*, 24(11):1097–1100, 1997.

[22] D. Porumbel. DIMACS graphs: Benchmark instances and best upper bound, 2009. `http://www.info.univ-angers.fr/pub/porumbel/graphs/`.

[23] D. Porumbel, J-K. Hao, and P. Kuntz. Diversity control and multi-parent recombination for evolutionary graph coloring algorithms. In *Proc. of the 9th EvoCOP conference on Evolutionay Computation in Combinatorial Optimization*, pages 121–132, Tübingen, Germany, 2009.

[24] C. Rego and F. Glover. Local search and metaheuristics. In Ding-Zhu Du, Panos M. Pardalos, Gregory Gutin, and Abraham Punnen, editors, *The Traveling Salesman Problem and Its Variations*, volume 12 of *Combinatorial Optimization*, pages 309–368. Springer US, 2004.

[25] M. Soto, A. Rossi, and M. Sevaux. Two upper bounds on the chromatic number. In *Proc. of the CTW09 Cologne-Twente Workshop on Graphs and Combinatorial Optimization*, volume 8, pages 191–194, Paris, France, 2009.

[26] M. Soto, A. Rossi, and M. Sevaux. Métaheuristiques pour l'allocation de mémoire dans les systèmes embarqués. In *Proc. ROADEF 11eme congrès de la société Française de Recherche Opérationelle et d'Aide à la Décision*, pages 35–43, Toulouse, France, 2010.

[27] T. Vredeveld and J.K Lenstra. On local search for the generalized graph coloring problem. *Operations Research Letters*, 31(1):28–34, 2003.

[28] S. Wuytack, F. Catthoor, L. Nachtergaele, and H. De Man. Power exploration for data dominated video application. In *Proc. IEEE Symposium on Low Power Design*, pages 359–364, Monterey, CA , USA, 1996.