Soft Comput (2014) 18:1515–1527
DOI 10.1007/s00500-013-1157-9

METHODOLOGIES AND APPLICATION

# GRASP with ejection chains for the dynamic memory allocation in embedded systems

**Marc Sevaux · André Rossi · María Soto · Abraham Duarte · Rafael Martí**

Published online: 31 October 2013


**Abstract** In the design of electronic embedded systems, the allocation of data structures to memory banks is a main challenge faced by designers. Indeed, if this optimization problem is solved correctly, a great improvement in terms of efficiency can be obtained. In this paper, we consider the dynamic memory allocation problem, where data structures have to be assigned to memory banks in different time periods during the execution of the application. We propose a GRASP to obtain high quality solutions in short computational time, as required in this type of problem. Moreover, we also explore the adaptation of the ejection chain methodology, originally proposed in the context of tabu search, for improved outcomes. Our experiments with real and randomly generated instances show the superiority of the proposed methods compared to the state-of-the-art method.

Communicated by E. Viedma.

M. Sevaux · A. Rossi
Université de Bretagne-Sud, Lab-STICC, CNRS Centre de recherche , B.P. 92116, 56321 Lorient Cedex, France

M. Soto
Université de Technologie de Troyes, Troyes, France

A. Duarte
Dept. Ciencias de la Computación, Universidad Rey Juan Carlos, c/Tulipán s/n, 28933 Móstoles, Madrid, Spain

R. Martí (✉)
Dept. de Estadística e Investigación Operativa, Universidad de Valencia, c/ Dr. Moliner 50, 46100 Burjassot (Valencia), Spain
e-mail: rafael.marti@uv.es

## 1 Introduction

The continuous advances in nano-technology have made possible a significant development in embedded systems (such as smartphones) to surf the Web or to process HD pictures. While technology offers more and more opportunities, the design of embedded systems becomes more complex. Indeed, the design of an integrated circuit, whose size is calculated in billions of transistors, thousands of memories, etc., requires the use of competitive computer tools. These tools have to solve optimization problems to ensure a low cost in terms of silicium area and running time. There exist some computer assisted design (CAD) tools such as Gaut (Coussy et al. 2006) to generate the architecture of a circuit from its specifications. However, the designs produced by CAD softwares are generally not energy aware, which is of course a major drawback.

In the design of embedded systems, memory allocation is among the main challenges that electronic designers have to face. Indeed, electronics practitioners, to some extent, consider that minimizing power consumption is equivalent to minimizing the running time of the application to be executed by the embedded system (Chimientia et al. 2002). Moreover, the power consumption of a given application can be estimated using an empirical model as in Julien et al. (2003), and parallelization of data access is viewed as the main action point for minimizing execution time, and consequently power consumption.

This paper is focused on memory allocation in embedded systems because of its significant impact on power consumption as shown by Wuytack et al. (1996). We have addressed various simpler versions of the memory allocation problem: in Soto et al. (2011), we have proposed a mixed linear formulation and a variable neighborhood search algorithm for the static version, and studied an even more simplified version

of this problem in Soto et al. (2010). We have also dealt with the dynamic memory allocation problem in Soto et al. (2011) for which an integer linear formulation and two iterative approaches have been devised. Note that the term dynamic refers to the nature of the electronic design problem, but data are all known before the execution of the solution method. In this paper, we propose a GRASP with ejection chains for the dynamic memory allocation problem in embedded systems and compare it with the previous iterative approaches.

The considered memory architecture is similar to the one of a TI C6201 device Julien et al. (2003). It is composed of $m$ memory banks whose capacity is $c_j$ kilo Bytes (kB) for all $j \in \{1, \ldots, m\}$ and an external memory denoted by $m + 1$, which does not have a practical capacity limit. The processor needs $q$ milliseconds for accessing data structures located in a memory bank, and it spends $q$ times more (i.e., $p \times q$ milliseconds) when data structures are in the external memory.

Time horizon is split into $T$ time intervals whose durations may be different. The application to be implemented is assumed to be given as C source code, whose $n$ data structures (i.e. variables, arrays, structures) have to be loaded in memory banks or the external memory. The size of data structure $s_i$ for $i \in \{1, \ldots, n\}$ is expressed in kB. During each time interval $t$, the application requires accessing a given subset $A_t$ of its data structures. We denote with a pair $(a, b)$ when data structures $a$ and $b$ are simultaneously accessed. The set $D_t$ contains all these pairs in time period $t$.

The combinatorial nature of this problem comes from the fact that the processor can access all the memory banks simultaneously. For example, given a specific time interval, to compute $a + b$ we may access $a$ and $b$ simultaneously. If they are allocated to different memory banks, we can access both of them at the same time, with the associated time saving, but if they are allocated to the same memory bank, we have to perform two different accesses. Thus, the cost of accessing simultaneously data structures $a$ and $b$ is $d_{(a,b)}$, and if they are allocated to the same memory bank, the total access cost is $2 \times d_{(a,b)}$, i.e., accessing sequentially $a$ and $b$. However, if $a$ or $b$ are allocated to the external memory, the cost is $p \times d_{(a,b)}$, and if both are allocated to the external memory the cost is $2 \times p \times d_{(a,b)}$. Note that when we perform an operation such as $a = a + 1$, the accessing cost can be either $2 \times d_{(a,a)}$ or $2 \times p \times d_{(a,a)}$ depending whether $a$ is in a memory bank or the external memory. Finally, a data structure can be accessed in isolation (i.e., not in a pair) when for example, we perform the operation $a = 5$, then its cost is $d_{(a,0)}$ if it is in a memory bank, and $p \times d_{(a,0)}$ if it is in the external memory.

If we want to take into account the dynamic structure of the problem, the cost of changing a data structure between memory banks or with the external memory from the previous time interval to the current one is related to its size. In particular, if data structure $a$ is in a memory bank at time interval $t - 1$ and in a different memory bank at time interval $t$ we have a cost of $\ell \times s_a$ where $\ell$ is the duration of this physical move in milliseconds per kilo Bytes (ms/kB) and $s_a$ is the size of the data structure a. Alternatively, if we change the allocation of a data structure between a memory bank and the external memory, the cost is $v \times s_a$ where now the factor is given by $v$ ms/kB. The later cost is particularly relevant because of hardware requirements. All the data structures are initially (say in $t = 0$) allocated to the external memory and therefore we assume this as initial solution for each data structure allocated to a memory bank in $t = 1$. We assume that $v \geq \ell$ and $v < p$ because a Direct Memory Access controller is supposed to be part of the memory architecture, which allows for a direct access to data structures. Table 1 summarizes all the costs described above for a specific time interval.

**Table 1** Costs to evaluate a solution in a specific time interval

| Type | Value | Description |
|---|---|---|
| Access | $d_{(a,b)}$ | if $a$ and $b$ are in different memory banks |
| | $2 \times d_{(a,b)}$ | if $a$ and $b$ are in the same memory bank |
| | $p \times d_{(a,b)}$ | if $a$ or $b$ is in the external memory |
| | $2p \times d_{(a,b)}$ | if $a$ and $b$ are in the external memory |
| Change | $\ell \times s_a$ | $a$ changed between memory banks |
| | $v \times s_a$ | $a$ changed between a memory bank and the external memory |

This paper is organized as follows. Section 2 shows how to represent a solution and evaluate it on an illustrative example. Sections 3 and 4 present, respectively, the GRASP and the ejection chains methods. The proposed method is first tuned in our preliminary experimentation and then compared to previous iterative approaches in Sect. 5. Finally, Sect. 6 presents conclusions and future work for this problem.

## 2 Step by step example

For the sake of illustration, this section presents a detailed computation of the cost of a solution. From now, we represent a solution $x$ as a matrix with data structures in rows and time intervals in columns. Thus, given a data structure $i$ in $\{1, \ldots, n\}$, and a time interval $t$ in $\{1, \ldots, T\}$, $x(i, t) = j$ with $j$ in $\{1, \ldots, m + 1\}$ indicates that data structure $i$ is allocated to memory bank $j$ at time period $t$.

Consider an example in which we have to allocate $n = 12$ data structures in $m = 3$ memory banks and the external memory, with $T = 4$ time periods. Additionally, consider that the size of the data structures is given by $\{65, 18, 95, 88, 99, 12, 19, 81, 10, 4, 79, 80\}$ and each memory bank has a capacity of 111 kB. Then, a feasible solution $x$ is given by the following (12, 4)-matrix

$$x = \begin{pmatrix} 1 & 4 & 4 & 4 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 2 & 2 & 2 & 4 \\ 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 4 & 4 & 3 & 3 \\ 3 & 3 & 3 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 2 \\ 4 & 1 & 1 & 1 \end{pmatrix}.$$

As mentioned above, the entries in the matrix indicate the indexes of the memory banks (where 4 refers to the external memory). For example, in row 4, column 1, we have $x(4, 1) = 2$ which means that data structure 4 is allocated to memory bank 2 at time interval 1. If we consider the data structures at the same time interval (a column of the matrix) that are assigned to the same memory bank, we can see that the sum of their sizes do not exceed the capacity of this bank. For example, in the second column, we can see that memory bank 1 appears in rows 6, 7, and 12 (i.e. $x(6, 2) = 1$, $x(7, 2) = 1$, and $x(12, 2) = 1$). If we sum the sizes of these data structures, we obtain $s_6 + s_7 + s_{12} = 12 + 19 + 80 = 111$, which is exactly the capacity of memory bank 2. It is easy to check that solution $x$ verifies the capacity constraints of the three memory banks in the four time periods. Figure 1 illustrates the first two time periods ($t = 1$ and $t = 2$) of this solution in a diagram, in which memory banks appear as vertical boxes, labeled as $\mathtt{MB}_1$, $\mathtt{MB}_2$ and $\mathtt{MB}_3$ respectively, and the external memory as a rectangular horizontal box labeled as $\mathtt{MB}_4$.

Consider now that in the first time period we have to access the data structures $A_1 = \{1, 2, 3, 4, 7, 9\}$ with six pairs accessed simultaneously: $D_1 = \{(1, 2), (1, 3), (1, 4), (1, 7), (2, 3) (2, 9)\}$, with associated costs: 62, 68, 37, 83, 18, and 33. To compute the total cost associated with the first time period, we note in Fig. 1 that data structures 1 and 2 are in different memory banks (data structure 1 is in $\mathtt{MB}_1$ and data structure 2 is in $\mathtt{MB}_2$), therefore we consider their access cost of 62. Similarly for pairs (1,3), (1,4), (2,3), and (2,9) since their data structures are in different memory banks. On the contrary, the cost of pair (1,7) is $2 \times d_{(1,7)}$ since data structures 1 and 7 are both in $\mathtt{MB}_1$. The total access cost is therefore $62 + 68 + 37 + 2 \times 83 + 18 + 33 = 384$. To complete the cost computation at time period $t = 1$, we have to consider that initially (i.e., at $t = 0$), the $n = 12$ data structures are in the external memory and therefore, the data structures in the three memory banks, 1, 2, 3, 4, 6, 7, 9, and 10, have an associated cost of change of $v \times (65 + 18 + 95 + 88 + 12 + 19 + 10 + 4) = 311v$. Therefore, the total cost at $t = 1$ is $384 + 311v$.

Similarly, we compute the cost of period $t = 2$, considering that $A_2 = \{2, 3, 4, 5, 6, 10, 11, 12\}$ and the following five pairs are accessed simultaneously: $D_2 = \{(2, 10), (3, 11), (3, 12), (4, 5), (4, 6)\}$, with associated costs: 99, 45, 71, 17, and 98. To calculate the access cost in this time interval, we have to consider that data structure 4 is allocated to $\mathtt{MB}_2$ but data structure 5 is allocated to the external memory (as indicated with an arrow in Fig. 1), and therefore the cost of pair (4,5) is $p \times d_{(4,5)} = 17p$. Similarly with pair (3,11) given that data structure 11 is in the external memory. Therefore the total access cost is now $99 + 45p + 71 + 17p + 98 = 268 + 62p$. The cost of change computed at $t = 2$ accounts for the fact that data structure 1 is in the external memory (indicated with a circle in Fig. 1), and it is in $\mathtt{MB}_1$ at $t = 1$. This has an associated cost of $v \times s_1 = 65v$. Similarly, data structure 12 is allocated to the external memory at time $t = 1$, and to $\mathtt{MB}_1$ at time $t = 2$ (highlighted with a circle in the figure). Therefore the cost is $v \times s_{12} = 80v$. Then, the total cost generated at time period 2 is $268 + 62p + 145v$.

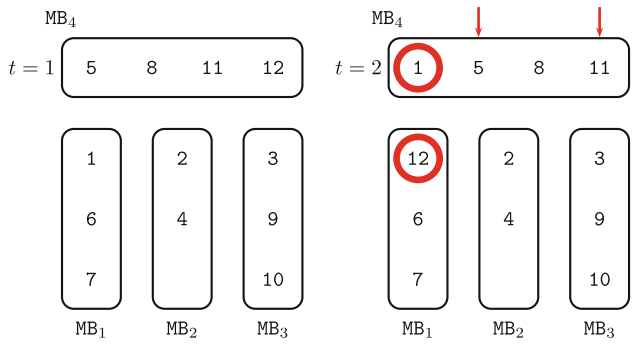


**Fig. 1** Example

## 3 GRASP

The GRASP metaheuristic was developed in the late 1980s (Feo and Resende 1989). Each GRASP iteration consists in constructing a trial solution with some greedy randomized procedure and then applying local search from the constructed solution. This two-phase process is repeated until some stopping condition is satisfied. The best local optimum found over all local searches is returned as the solution of the heuristic. Some successful and recent applications of this methodology can be fond in Duarte et al. (2011) or Resende et al. (2010). We refer the reader to Resende and Ribeiro (2010) for recent surveys of this metaheuristic.

Algorithm 1 shows the pseudo-code for a generic GRASP for minimization. The greedy randomized construction seeks to produce a diverse set of good-quality starting solutions from which to start the local search phase. Let $x$ be the partial solution under construction in a given iteration and let $C$ be the candidate set with all the remaining data structures that

**Algorithm 1:** GRASP algorithm

```
1  f* ← ∞
2  while stopping criterion not satisfied do
3      x ← ∅
4      Compute C with the candidate data structures that can be
       added to x
5      while C ≠ ∅ do
6          forall the c ∈ C do
7              compute g(c), g_min = min_{c∈C} g(c) and
               g_max = max_{c∈C} g(c)
8          Define RCL ← {c ∈ C | g(c) ≤ g_min + α(g_max − g_min)}
           with α ∈ [0, 1]
9          Select c* at random from RCL(C)
10         Add c* to partial solution: x ← x ∪ {c*}
11         Update C with the candidate data structures that can be
           added to x
12     x ← LocalSearch(x)
13     if f(x) < f(x*) then
14         x* ← x; f* ← f(x)
15 return x*
```

can be added to $x$. The GRASP construction uses a greedy function $g(c)$ to measure the contribution of each candidate data structure $c \in C$ to the partial solution $x$. A restricted candidate list $RCL$ is the subset of candidate data structures from $C$ with good evaluations according to $g$. In particular, if $g_{min}$ and $g_{max}$ are the minimum and maximum evaluations of $g$ in $C$ respectively, then $RCL = \{c \in C \mid g(c) \leq g_{min} + \alpha(g_{max} - g_{min})\}$, where $\alpha$ is a number in $[0, 1]$. At each step, the method randomly selects a data structure $c^*$ from the restricted candidate list and adds this data structure to the partial solution. The construction is repeated in the inner while loop (steps 4–10) until there are no further candidates. If $C = \emptyset$ and $x$ is infeasible, then a repair procedure needs to be applied to make $x$ feasible (steps 12–14). Once a feasible solution $x$ is available, a local search improvement is applied. The resulting solution is a local minimum. The GRASP algorithm terminates when a stopping criterion is met (typically a maximum number of iterations, time limit, or a target solution quality). The best overall solution $x^*$ is returned as the output of the heuristic.

## 3.1 Constructive methods

To construct a solution we have to allocate the data structures, one by one, to the memory banks or the external memory in each time period. In this section we propose two different approaches; the first one sequentially constructs a solution starting with time period $t = 1$ and moving to next time period once the allocation of all data structures in the current one is completed. On the other hand, the second constructive method gives priority to the allocation of data structures that might generate the largest cost, regardless of their time period.

### *3.1.1 Sequential approach*

In the sequential approach, SEQ, we first complete the assignments in a period and then we start the assignments in the next one. From the costs in Table 1, it is clear that the best option for a pair of data structures is to be allocated to different memory banks; however, they have a limited capacity and some data structures have to be eventually allocated to the same bank or in the external memory.

Consider a partial solution in which we have assigned the data structures to the memory banks in time periods 1 to $t-1$ and we have already assigned some of the data structures in period $t$. Let $C_t$ be the candidate set of unassigned data structures at time period $t$. To determine the data structure and memory bank for the next assignment in our construction process, we compute for each data structure $i \in C_t$ its evaluation $g(i, t, j)$ to any possible memory bank $j$, or to the external memory (for which $j = m + 1$).

To compute $g(i, t, j)$, we consider two types of costs. The first one is the cost related to the access to data structure $i$ in the memory bank $j$ at time interval $t$. This cost is denoted by $total\_access(i, t, j)$ and defined by:

$$total\_access(i, t, j) = \sum_{k=1}^{|N_{i,t}|} access(j, x(a_k, t), d_{(i,a_k),t}) \tag{1}$$

where $N_{i,t}$ is the set of data structures which are in conflict with data structure $i$ at time interval $t$, thus $a_k \in N_{i,t}$. Two data structures $i$ and $a_k$ are said to be in conflict at time period $t$ if $(i, a_k) \in D_t$. Function $access(j_1, j_2, d)$ defined in Eq. (2) computes the access cost produced by a conflict whose cost is $d$ and where its corresponding data structures are allocated to memory banks $j_1$ and $j_2$ respectively. Thus $access(j, x(a_k, t), d_{(i,a_k),t})$ is the access cost generated by conflict $(i, a_k)$ at time interval $t$ where data structure $i$ is allocated to memory bank $j$ and $x(a_k, t) \in \{1, \ldots, m + 1\}$ is the memory bank where $a_k$ is allocated.

$$access(j_1, j_2, d) = \begin{cases} d, & \text{if } j_1 \neq j_2 \text{ and } j_k \neq m+1, \\ & \quad \forall k = 1, 2 \\ 2 \times d, & \text{if } j_1 = j_2 \neq m+1 \\ 2p \times d, & \text{if } j_1 = j_2 = m+1 \\ p \times d, & \text{otherwise} \end{cases} \tag{2}$$

The other cost involved in the evaluation function $g(i, t, j)$ is the cost related to the change of a data structure between memory banks or between the external memory and a memory bank in consecutive time intervals. It is computed as follows:

$$change(i, j_1, j_2) = \begin{cases} 0, & \text{if } j_1 = j_2 \\ \ell \times s_i, & \text{if } j_1 \neq j_2 \text{ and } j_k \neq m+1, \forall k = 1, 2 \\ v \times s_i, & \text{otherwise} \end{cases} \quad (3)$$

Thus, the evaluation function $g(i, t, j)$ is given by:

$$g(i, t, j) = total\_access(i, t, j) + change(i, x(i, t-1), j) \quad (4)$$

when the data structure $i$ is not accessed during time period $t$, we have to consider the cost of change from the allocation in time interval $t-1, x(i, t-1)$, and its trial allocation in $t$. Thus, the evaluation function is $g(i, t, j) = change(i, x(i, t-1), j)$ for all $i \notin A_t$ .

Equation (4) computes the increment in the cost of the current partial solution under construction, if data structure $i \in A_t$ is assigned to a memory bank or to the external memory. However, feasible assignments only are considered; i.e., those for which data structure $i$ can be added to a memory bank without exceeding its capacity. Once all feasible assignments have been evaluated, we compute the minimum and maximum of those values as:

$$g_{\min}(t) = \min_{i \in C_t} \{g(i, t, j) \quad \forall j \in \{1, \ldots, m+1\}\}$$
$$g_{\max}(t) = \max_{i \in C_t} \{g(i, t, j) \quad \forall j \in \{1, \ldots, m+1\}\}$$

Then, as shown in Algorithm 1, we compute the restricted candidate list $RCL(t)$ with the pairs of data structures in conflict and memory banks for which the evaluation is within the customary GRASP limits. Specifically,

$$RCL(t) = \{(i, j) : i \in C_t \text{ and } g(i, t, j) \leq g_{\min}(t) + \alpha(g_{\max}(t) - g_{\min}(t))\},$$

where $\alpha$ is a number in [0, 1]. At each step, the method randomly selects a pair $(i, j)$ from the restricted candidate list and performs the associated assignment (i.e., it assigns $i$ to $j$ at time interval $t$). The constructive algorithm, called SEQ, terminates when all the data structures have been assigned, for all the time periods.

#### *3.1.2 Conflict priority approach*

Our second constructive method, CPA, has two stages. In the first one we allocate the pairs of data structures from $D_t$ for any time period $t$, and in the second one we allocate the remaining elements not in $A_t$ for any time period $t$.

In the first stage we define $D$ as the ordered set of pairs in $\bigcup_{t=1}^{T} D_t$ in decreasing values of the cost $d_{(a,b),t}$. The restricted candidate list RCL is here simply computed as a fraction of the pairs in $D$. Specifically, we consider a percentage $\alpha$ of the largest elements in $D$. We then select one pair $(a^\star, b^\star)$ in RCL at random and allocate its elements in their time period $t$. To do that, we compute $g(a^\star, t, j)$ and $g(b^\star, t, j)$ $\forall j \in \{1, \ldots, m+1\}$ and perform the best allocation. Once a pair has been allocated, it is removed from $D$. This operation is repeated as long as $D \neq \emptyset$. Note that the construction of the restricted candidate list does not follow the conventional adaptive scheme. In other words, the $d_{(a,b),t}$ values do not change from one construction step to the next. However, note also that finding the best allocation for the selected pair involves the computation of an adaptive value since $g(i, t, j)$ depends on the previous assignments.

In the second stage we allocate the remaining elements (i.e., those not present in the pairs). As previously defined, let $C_t$ be the candidate set of unassigned data structures at time period $t$. We define $C$ as the ordered set of pairs $(i, t)$ in $\bigcup_{t=1}^{T} C_t$ in decreasing values of the size $s_i$. As in the first stage, RCL contains a fraction $\alpha$ of the largest elements, from which we select one pair $(a^\star, t)$ at random. Finally, we allocate $a^\star$ in time $t$ in the best memory bank according to $g(i, t, j)$ and update $C$. The method finishes when all the elements in $C$ have been allocated.

The randomized nature of the constructive process permits to generate different solutions in each construction. We have empirically found that in some instances the allocation of all the elements in the external memory banks produces a trivial solution with a relatively good value. Therefore, we have included this trivial solution as the first construction of this method.

### 3.2 Local search

Since there is no guarantee that a randomized greedy solution is optimal, local search is usually applied after each construction step to attempt to find a local optimal solution, or at least to obtain an improved solution with smaller cost than that of the constructed solution. This idea originates in the seminal paper by Feo and Resende (1989) for set covering and was later referred to as GRASP (Feo and Resende 1995).

Given a feasible solution $x$ of a combinatorial optimization problem, we define a *neighborhood* $N(x)$ of $x$ to consist of all feasible solutions that can be obtained by making a predefined modification to $x$. We say a solution $x^*$ is *locally optimal* if $f(x^*) \leq f(x)$ for all $x \in N(x^*)$, where $f$ is the objective function that we are minimizing. Given a feasible solution, a local search procedure finds a locally optimal solution by exploring a sequence of neighborhoods, starting from it. At the $i$-th iteration it explores the neighborhood of solution $x_i$. If there exists some solution $y \in N(x_i)$ such that $f(y) < f(x_i)$, it sets $x_{i+1} = y$ and proceeds to iteration $i+1$. Otherwise, $x^* = x_i$ is declared a locally optimal solution and the procedure stops.

Insertions are used as the primary mechanism to move from a solution to another in the local search for the memory allocation problem. Specifically, given a solution $x$ and a data structure $i$ allocated to memory bank $j$ at time period $t$ (i.e.,

$x(i,t) = j$), we define $move(i,t,j')$ as the removal of $i$ from its current memory bank $j$, followed by its insertion in memory bank $j'$ at time period $t$. This operation results in the solution $y$, as follows:

$$\begin{aligned} y(i,t) &= j' \\ y(i,t) &= x(i,t) \quad \text{for } t \neq t' \\ y(i',t) &= x(i',t) \quad \text{for } i' \neq i \text{ and for all } t \end{aligned}$$

Given a solution $x$ and a data structure $i$ in memory bank $j$ at time period $t$, we define $h(i,t,j)$ as its contribution to the solution value. If data structure $i$ is accessed in one or more pairs in $t$, it can be computed as:

$$\begin{aligned} h(i,t,j) = total\_access(i,t,j) + change(i,x(i,t-1),j) \\ + change(i,x(i,t+1),j) \end{aligned} \tag{5}$$

This expression is similar to (4) in which we compute the cost for accessing data structure $i$ in the memory bank $j$ to execute the operation at time interval $t$. The second and third terms in (5) compute respectively the cost of change of $i$ from period $t-1$ to $t$ and from $t$ to $t+1$. The later term was not involved in (4) since we computed there the cost of a partial solution under construction in which no data structure was assigned to period $t+1$ at that stage. Therefore, $h(i,t,j)$ sums all the costs generated with the assignment of data structure $i$ in $\{1,\ldots,n\}$ to its current location $j$ in $\{1,\ldots,m+1\}$ at time interval $t$ in $\{1,\ldots,T\}$.

Considering that we have $n$ data structures at a given time period $t$ (where we start with $t = 1$), there are $n$ possible candidates for an insertion move. In other words, we can consider an allocation change for any data structure in this time period for improving the solution. However, instead of enumerating all these possibilities (i.e. scanning the entire neighborhood of the solution), we implement the so-called *first strategy*, in which we perform the first insertion move that improves the solution. In order to study first those data structures that are more likely to provide improving moves, our local search method first computes the contribution of all the data structures in a time period and orders them according to these values. Then, it scans the data structures in this order, where the data structure with the largest contribution comes first. As a result, it first explores those data structures with a large contribution since we can consider them as "badly allocated" and tries to re-allocate them in a different, and better, memory bank.

Algorithm 2 shows the main steps of the local search method. In line 1, the time period $t$ is initialized to 1. In line 4, we order the elements in conflict (accessed simultaneously) in time period $t$, $A_t$, according to their contribution. Once we select a data structure $i \in A_t$ allocated to memory bank $j$ at time period $t$, we compute $h(i,t,j')$ for all memory banks. In line 6, we select the best memory bank $j^\star$. If $j^\star \neq j$ (line 8), which indicates that $h(i,t,j^\star) < h(i,t,j)$, we move (in line 9) data structure $i$ from memory bank $j$ to $j^\star$ if there is room in bank $j^\star$, since it results in a reduction of the solution value. Specifically, we perform $move(i,t,j^\star)$ and the value of the solution is reduced by $h(i,t,j) - h(i,t,j^\star)$. Once we have explored the possible assignments of data structure $i$ to a different memory bank, and eventually moved it, we resort to the next data structure in the ordered list to study its associated moves. When we explore all the data structures with a positive contribution, we stop the scan of the candidate list (there is no point in moving data structures with a null contribution). At this stage, we set $t = t-1$ if $t > 1$ in line 12 and explore the previous time period. The rational behind this strategy is to perform a backward step to re-allocate the elements linked with those recently moved in the current time period. Alternatively, if we have not found any improvement move in $t$, we set $t = t+1$ in line 14, and apply the improvement method in the next time period. In any case, we compute the contribution of the data structures and proceed in the same fashion. The local search LS terminates when no data structure is moved after scanning all the time periods. It returns the associated local optimum.

**Algorithm 2:** Local search

```
1  t ← 1
2  while t ≤ T do
3      improved ← false
4      A_t ← SortElementsInConflicts(t)
5      forall the i ∈ A_t do
6          j* ← arg min_{1≤j'≤m} h(i, t, j')
7          j ← CurrentMemoryBank(i)
8          if j ≠ j* then
9              move(i, t, j*)
10             improved ← true
11     if improved = true and t ≠ 1 then
12         t ← t − 1
13     else
14         t ← t + 1
```

## 4 Ejection chains

Consider the example in Sect. 2 in which we have to allocate $n = 12$ data structures in $m = 3$ memory banks and the external memory, in $T = 4$ time periods. We apply the greedy randomized constructive method to this instance with parameter values $q = 1$, $p = 16$, $\ell = 1$ and $v = 4$, and we obtain a solution with a value of 15,692. Now we apply our local search method based on insertions to improve this trial solution, and we observe that the data structure with the largest contribution is data structure 6, allocated to the external memory (with index 4) at time period $t = 3$. Figure 2 shows a representation of the allocation of the data struc-

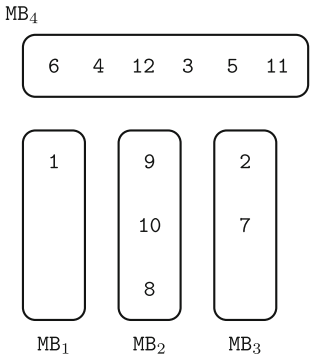


**Fig. 2** Memory banks at $t = 3$

tures in memory banks at this time period. Data structure 6 has a contribution value of $h(6, 3, 4) = 2{,}384$, and the local search computes its alternative assignments at time period 3: $h(6, 3, 1) = 533$, $h(6, 3, 2) = 622$, and $h(6, 3, 3) = 545$, which are significantly lower than the value of its current assignment. Since there is room in memory bank 1 for data structure 6, we perform the move, *insert*$(6, 3, 1)$ and obtain a new solution with value $15{,}692 - 2{,}384 + 533 = 13{,}841$. We now resort to the next data structure in the ordered list according to their contribution. It is data structure 4 allocated to the external memory (with index 4) at $t = 3$ and a contribution to the solution value of $h(4, 3, 4) = 1{,}424$. We would move this data structure to memory bank 1 considering that its associated contribution is $h(4, 3, 1) = 793$; however, there is no room in memory bank 1 for this data structure (it has a remaining capacity of 34 units and this data structure has a size of 88). As a matter of fact, we cannot move this data structure to any other memory bank due to the associated evaluation or the size constraints. We therefore resort to the next data structure in the list and continue in this fashion.

The previous example illustrates that some insertions cannot be performed because there is no room in the *destination* memory bank. This suggests that we could consider to make room there by moving one of its data structures elsewhere, implementing what is known in local search as a compound move or ejection chain. Glover and Laguna (1997) introduced compound moves, often called variable depth methods, constructed from a series of simpler components. As it is well-known, one of the pioneering contributions to this kind of moves was Lin and Kernighan (1973). Within the class of variable depth procedures, a special subclass called ejection chain procedures has recently been proved useful. An ejection chain EC is an embedded neighborhood construction that compounds the neighborhoods of simple moves to create more complex and powerful moves. It is initiated by selecting a set of data structures to undergo a change of state (e.g., to occupy new positions or receive new values). The result of this change leads to identifying a collection of other sets, with the property that the data structures of at least one must be "ejected from" their current states. State-change steps and ejection steps typically alternate, and the options for each depends on the cumulative effect of previous steps (it is usually impacted by the latest step). In some cases, a cascading sequence of operations may be triggered, representing a domino effect. Successful applications of this strategy can be found in Lozano et al. (2012); Martí et al. (2009, 2011).

In the memory allocation problem, when data structure $i$ in memory bank $j$ at time period $t$ has a relatively large contribution to the objective function, the local search selects it to evaluate its insertion in a different memory bank, say $j'$, in order to reduce this contribution. In some cases however, this move is not feasible because $j'$ does not have enough capacity for $i$ (i.e. because the difference between its capacity and the sum of the sizes of the data structures in $j'$ is lower than the size of this data structure). The EC local search then considers to move one of the data structures in $j'$, say $i'$, to another memory bank on time period $t$, say $j''$, to make room in $j'$ for $i$. We may say that the insertion of $i$ in $j'$ caused $i'$ to be ejected to $j''$, or, in other words, that we apply the ejection chain $move(i, t, j') + move(i', t, j'')$ of depth 2. Clearly, the benefit of moving a data structure in $j''$ to another memory bank, say $j'''$, could also be evaluated if one would like to consider chains of depth three. Longer chains are possible by applying the same logic.

In EC, we define $chain(i, dplimit)$ as the ejection chain that starts from moving vertex $i$ and performs a maximum of $dplimit$ consecutive insertion moves. Once a data structure $i$ with a relatively large contribution at time period $t$ is identified, EC starts by scanning alternative memory banks, in a random order, to allocate it. The chain then starts by making $chain(i, dplimit) = \{move(i, t, j')\}$ where $j'$ is the first alternative memory bank considered. If this depth-1 move is improving and feasible (there is room in $j'$ for $i$), it is executed and the chain stops. Otherwise, we search for $move(i', t, j'')$ associated with a data structure $i'$ in $j'$. If the compound move of depth-2 is an improving and feasible move (there is room in $j'$ for $i$ and in $j''$ for $i'$), the move is executed and the chain stops; otherwise the chain continues until the compound move becomes improving and feasible or the depth of the chain reaches the pre-specified limit $dplimit$. If none of the compound moves from depth-1 to $dplimit$ examined is an improving and feasible move, alternative memory banks (values of $j$ and $j'$) and data structures (values for $i'$ and associated trial data structures) are considered in a recursive exploration. If none of them is improving and feasible, no move is performed and the exploration continues with the next $i$ data structure in the candidate list.

Algorithm 3 shows the pseudo-code of the EC method. It starts by setting $t = 1$ at line 1. Then, at line 4, it orders data structures $i$ in $D_t$ according to their contribution. Line 6 sets the depth parameter $d$ to 1, and defines at line 7 an auxiliary variable, $i_{eject}$, with the element to be moved (ini-

**Algorithm 3:** Ejection chain

```
1  t ← 1
2  while t ≤ T do
3      improved ←false
4      D_t ← SortElementsInConflicts(t)
5      forall the i ∈ C do
6          d ← 1
7          i_eject ← i
8          impEC ←false
9          while (d ≤ dplimit) and impEC =false do
10             j* ← arg min_{j∈MB} h'(i_eject, t, j)
11             j ←CurrentMemoryBank(i_eject)
12             if j ≠ j* then
13                 S ←EjectedElements(i_eject, t, j*)
14                 if S = ∅ then
15                     impEC ←true
16                     improved ←true
17                 else
18                     d ← d + 1
19                     i_eject ← arg max_{k∈S} h'(k, t, j*)
20     if improved = true and t ≠ 1 then
21         t ← t − 1
22     else
23         t ← t + 1
```

tially equal to $i$). The while loop at line 9 allows the method to perform ejection chain steps as long as it improves, or the maximum depth limit, $dplimit$ is reached. The best memory bank $j^\star$ to move $i_{eject}$, including those banks with no room for it, is identified in line 10. The move $move(i_{eject}, t, j^\star)$ is performed at line 13 and the method computes the set $S$ with all the elements in memory bank $j^\star$ that create enough room to allocate $i_{eject}$ when removing them from $j^\star$. The variables are then updated, including the $i_{eject}$, which now is the best element in $S$ in terms of its contribution (line 19). Lines 20–23 apply the same logic to scan the time periods described in the LS method.

EC is a local optimizer and hence it performs only improving chains, although it is worth mentioning that these chains are compound of several insertion moves, and all of them are improving moves. Although the general design of the compound moves permits the implementation of non-improving single moves, for the sake of reducing the CPU time, we restrict ourselves to improving moves.

## 5 Computational results

In this section, we first perform some preliminary experiments to set the appropriate search parameters and to compare our different methods. Then, we compare our best variant with the iterative approaches and ILP formulation in Soto et al. (2011), which are considered the state of the art and the only methods when solving the Dynamic Memory Allocation Problem. All algorithms have been implemented in Java SE 6 and run on an Intel Core i7 2600 CPU at 3.4 GHz with 3 GB RAM.

We have tested our metaheuristics using 45 real and artificial instances previously reported in Soto et al. (2011). Real instances originate from electronic design problems addressed in the Lab-STICC laboratory. Artificial instances originate from DIMACS (Porumbel 2009) and they have been enriched by randomly generating conflict costs, number and capacity of memory banks, sizes and number of access to data structures. Artificially large instances allow us to assess our metaheuristics for the practical use for forthcoming needs, as technology tends to integrate more and more functionalities in embedded systems. In line with previous studies, the problem parameters are set as in the real electronic applications. The time $q$ spent by the processor to access data structures to memory banks is set to 1 ms, the factor $p$ to access data structures to external memory is set to 16, as with TI C6201. The cost $v$ for moving a data structure from the external memory to memory banks and vice versa is set to 4 ms/kB and cost for moving data structures between memory banks is equal to 1 ms/kB.

Table 2 shows the main features of the instances: name, number of data structures ($n$), number of conflicts ($D$), memory banks ($M$), and time intervals ($T$). Instances are sorted by non decreasing sizes (by considering first the number of data structures and then the number of conflicts). This ordering gives a rough estimation about the hardness of solving a specific instance. All these instances can be downloaded from http://www.optsicom.es/dmap/dmap.zip.

We perform our preliminary experimentation with 8 representative instances with different sizes and characteristics. They are: fpsol2i2dy, mpeg2enc2dy, mug100-25dy, myciel7dy, queen5-5dy, r125.1cdy, treillisdy, and zeroin-i1dy. In the first experiment we compare the two constructive methods described in Sect. 3.1, SEQ and CPA. Table 3 shows, for each variant, the average objective function value (Value), the average percent deviation from the best solutions obtained within this experiment (Dev. Exp.), and the number of instances (Best) in which the method is able to match the best solutions obtained within this experiment (out of 8 instances). These measures are local in the sense that the best solutions are those found within the experiment. They are used because they allow us to discriminate among the procedures being tested and identify the better alternatives. However, as a point of reference, Table 3 also includes the average deviation achieved by each variant against the best-known solutions (Dev. BK.). This is a global measure in the sense that the best-known solutions are those found across all experiments in this paper, which as far as we know represents the best-known published solutions. Finally, the last row of the table summarizes the information, reporting the average

**Table 2** Mean features of the instances

| Name | $n$ | $D$ | $M$ | $T$ | Name | $n$ | $D$ | $M$ | $T$ |
|---|---|---|---|---|---|---|---|---|---|
| gsm_newdy | 6 | 5 | 3 | 2 | mpegdy | 68 | 69 | 3 | 8 |
| compressdy | 6 | 6 | 3 | 3 | mug88_1dy | 88 | 146 | 3 | 6 |
| lmsbdy | 8 | 8 | 3 | 3 | mug88_25dy | 88 | 146 | 3 | 6 |
| lmsbv01dy | 8 | 8 | 3 | 4 | myciel6dy.col | 95 | 755 | 3 | 11 |
| lmsbvdy | 8 | 8 | 3 | 3 | mug100_1dy | 100 | 166 | 3 | 7 |
| lmsbvdyexpdy | 8 | 8 | 3 | 4 | mug100_25dy | 100 | 166 | 3 | 7 |
| volterrady | 8 | 6 | 3 | 2 | r125.1dy.col | 125 | 209 | 4 | 6 |
| spectraldy | 9 | 8 | 3 | 3 | r125.5dy.col | 125 | 3,838 | 18 | 38 |
| adpcmdy | 10 | 8 | 3 | 3 | r125.1cdy.col | 125 | 7,501 | 23 | 75 |
| cjpegdy | 11 | 7 | 3 | 4 | mpeg2enc2dy | 130 | 239 | 3 | 12 |
| myciel3dy.col | 11 | 20 | 3 | 4 | mulsol_i4dy.dat | 185 | 3,946 | 16 | 39 |
| turbocodedy | 12 | 22 | 4 | 4 | mulsol_i5dy.dat | 186 | 3,973 | 16 | 40 |
| lpcdy | 15 | 19 | 3 | 4 | mulsol_i2dy.dat | 188 | 3,885 | 16 | 39 |
| gsmdy | 19 | 18 | 3 | 5 | myciel7dy.col | 191 | 2,360 | 4 | 24 |
| gsmdycorrdy | 19 | 18 | 3 | 5 | alidydy | 192 | 960 | 6 | 48 |
| myciel4dy.col | 23 | 71 | 4 | 7 | mulsol_i1dy.dat | 197 | 3,925 | 25 | 39 |
| queen5_5dy | 25 | 160 | 4 | 5 | zeroin_i3dy.dat | 206 | 3,540 | 15 | 35 |
| treillisdy | 33 | 61 | 3 | 6 | zeroin_i2dy.dat | 211 | 3,541 | 15 | 35 |
| queen6_6dy | 36 | 290 | 5 | 10 | zeroin_i1dy.dat | 211 | 4,100 | 25 | 41 |
| myciel5dy.col | 47 | 236 | 4 | 6 | fpsol2i3dy.dat | 425 | 8,688 | 15 | 87 |
| queen7_7dy | 49 | 476 | 5 | 16 | fpsol2i2dy | 451 | 8,691 | 16 | 87 |
| queen8_8dy | 64 | 728 | 6 | 24 | inithx_i1dy | 864 | 18,707 | 28 | 187 |

**Table 3** Constructive methods

| Instances | SEQ | | | | | CPA | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Name | Value | CPU | Dev.Exp. (%) | Dev.BK. (%) | Best | Value | CPU | Dev.Exp. (%) | Dev.BK. (%) | Best |
| fpsol2i2dy | 3,577,023.00 | 35.0 | 2.49 | 27.99 | 0 | 3,489,953.00 | 37.0 | 0.00 | 24.87 | 1 |
| mpeg2enc2dy | 14,548.81 | 0.0 | 0.00 | 48.28 | 1 | 14,548.80 | 0.0 | 0.00 | 48.28 | 1 |
| mug100-25dy | 57,462.00 | 0.0 | 0.00 | 88.41 | 1 | 57,650.00 | 0.0 | 0.33 | 89.02 | 0 |
| myciel7dy | 986,793.00 | 1.0 | 1.08 | 120.76 | 0 | 976,216.00 | 1.0 | 0.00 | 118.39 | 1 |
| queen5-5dy | 57,595.00 | 0.0 | 11.41 | 110.24 | 0 | 51,698.00 | 0.0 | 0.00 | 88.71 | 1 |
| r125.1cdy | 3,538,925.00 | 71.0 | 3.42 | 188.86 | 0 | 3,421,905.00 | 73.0 | 0.00 | 179.31 | 1 |
| treillisdy | 4,138.81 | 0.0 | 0.00 | 129.23 | 1 | 4,138.81 | 0.0 | 0.00 | 129.23 | 1 |
| zeroin-i1dy | 1,942,869.00 | 43.0 | 6.24 | 237.12 | 0 | 1,828,682.00 | 44.0 | 0.00 | 217.30 | 1 |
| Summary | 1,272,419.32 | 18.7 | 3.08 | 118.86 | 3 | 1,230,598.95 | 19.4 | 0.04 | 111.89 | 7 |

Value, CPU, Dev. Exp., Dev. BK., and the sum of the Best values.

Table 3 shows that constructive method CPA clearly outperforms the sequential method, SEQ. Specifically, CPA presents an average deviation with respect to the best solution found in the experiment and with respect to the best known solution of 0.04 and 111.89 %, respectively, obtained in 19.4 s, which compares favorably with the average deviation of 3.08 and 118.86 % obtained with SEQ in 18.7 s. Moreover, CPA is able to obtain seven best solutions in this experiment while SEQ obtains three out of the eight considered instances.

In our second experiment, we compare the two complete GRASPs formed by the constructive methods coupled with the local search. We denote them as CPA+LS and SEQ+LS. Table 4 shows for each method and each of the eight instances considered, the value of the best solution found across ten constructions with local search, Value, as well as the other four statistics described above: CPU, Dev.Exp., Dev. BK, and Best.

Table 4 shows that the CPA+LS method obtains better results than the SEQ+LS in shorter running times. In particular CPA+LS presents an average percent deviation of

**Table 4** GRASP methods

| Instances | SEQ+LS | | | | | CPA+LS | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Name | Value | CPU | Dev.Exp. (%) | Dev. BK. (%) | Best | Value | CPU | Dev.Exp. (%) | Dev. BK. (%) | Best |
| fpsol2i2dy | 2,814,803.00 | 86.0 | 0.00 | 0.72 | 1 | 2,816,210.00 | 66.0 | 0.05 | 0.77 | 0 |
| mpeg2enc2dy | 10,808.34 | 0.0 | 0.01 | 10.15 | 0 | 10,807.80 | 0.0 | 0.00 | 10.14 | 1 |
| mug100-25dy | 32,673.00 | 0.0 | 0.12 | 7.13 | 0 | 32,633.00 | 0.0 | 0.00 | 7.00 | 1 |
| myciel7dy | 631,631.00 | 2.0 | 32.67 | 41.30 | 0 | 476,107.00 | 1.0 | 0.00 | 6.51 | 1 |
| queen5-5dy | 35,210.00 | 0.0 | 28.03 | 28.53 | 0 | 27,501.00 | 0.0 | 0.00 | 0.39 | 1 |
| r125.1cdy | 3,130,709.00 | 81.0 | 0.00 | 155.54 | 1 | 4,282,700.00 | 14.0 | 36.80 | 249.58 | 0 |
| treillisdy | 3,847,81 | 0.0 | 60.11 | 113.11 | 0 | 2,403.30 | 0.0 | 0.00 | 33.10 | 1 |
| zeroin-i1dy | 871,604.00 | 50.0 | 48.53 | 51.24 | 0 | 586,823.00 | 8.0 | 0.00 | 1.82 | 1 |
| Summary | 941,410.77 | 27.4 | 21.18 | 50.96 | 2 | 1,029,398.10 | 11.1 | 4.61 | 38.66 | 6 |

**Table 5** GRASP with ejection chains

| Method | Value | CPU | $Dev.Exp.(\%)$ | Dev.BK (%) | Best |
|---|---|---|---|---|---|
| GRASP(10)+EC(2) | 655,395.44 | 90.5 | 8.47 | 17.27 | 3 |
| GRASP(10)+EC(3) | 655,009.06 | 100.25 | 8.44 | 17.25 | 4 |
| GRASP(10)+EC(4) | 654,848.31 | 95.625 | 8.44 | 17.24 | 5 |
| GRASP(10)+EC(5) | 654,848.31 | 93.375 | 8.44 | 17.24 | 5 |
| GRASP(10) | 911,946.14 | 28.25 | 33.84 | 45.81 | 0 |
| GRASP(100) | 868,135.70 | 219.75 | 27.88 | 39.67 | 2 |

4.61 % (Dev. Exp.) and 38.66 % (Dev. BK.) obtained in 11.1 s, while SEQ+LS exhibits an average of 21.18 % (Dev. Exp.) and 50.96 % (Dev. BK.) obtained in 21.18 s. Additionally, CPA+LS matches six best solutions in this experiment while SEQ+LS obtains two out of the eight instances considered. This experiment confirms again the superiority of CPA over the SEQ algorithm. Therefore, in the following experiments we considered the CPA+LS as the GRASP method.

In our final preliminary experiment, we compare the contribution of the ejection chain post-processing (EC) and the influence of the depth limit parameter, $dplimit$, in this method. Table 5 reports the average results of value, CPU, Dev.Exp, Dev. BK, and the number of best solutions found with the GRASP (CPA+LS) runs for 10 and 100 iterations and the same GRASP method run for 10 iterations in which we apply the EC($dplimit$) post-processing, with a given value of the depth parameter, after the application of the LS. We have tested the values 2, 3, 4, and 5 for the depth limit parameter.

Results in Table 5 indicate that the ejection chain post-processing (EC) significantly improves the GRASP method. Specifically, comparing GRASP(10) with GRASP(10) +EC(4) it is clear that EC drastically reduces the GRASP deviation (GRASP(10) exhibits an average Dev. BK. of 45.81 % while GRASP(10)+EC(4) is able to drop this value to 17.24 %). However, the EC post-processing has an associated running time and therefore the total running time of GRASP(10)+EC(4) is, as expected, larger than GRASP(10). This is why we include in this table GRASP(100) to show that even if we run GRASP for a longer running time than GRASP+EC, it is not able to reach by itself the high quality solutions obtained with the combination of both methods. In particular, GRASP(100) obtains an average Dev. BK. of 39.67 % in 219.75 s while GRASP(10)+EC(4) only needs 95.62 s to obtain a Dev. BK. of 17.24 %. On the other hand, comparing the results obtained with GRASP+EC with different values of the $dplimit$, we observe small differences with a slight improvement in the number of best solutions found when it is equal to 4 or 5.

In our final experiment, we compare our best algorithms, CPA, GRASP and GRASP+EC with the iterative metaheuristic (IM), proposed in Soto et al. (2011), which is the best published method so far. The objective of this experiment is two-fold. On the one hand, to compare our proposals with the state-of-the-art method, and on the other hand, to experimentally test that the inclusion of more elaborated strategies drive us to better outcomes when executing all the methods for the same computing time. Table 6 summarizes the results of this experiment over the entire set of instances. The individual results of each method and each instance can be found in Table 7 in the Appendix.

Table 6 reports the average value of the objective function, the CPU time in seconds, the deviation value with respect to

**Table 6** Final comparison

| Method | Value | CPU | Dev.BK (%) | Best |
|---|---|---|---|---|
| IM | 1,378,609.24 | 300.72 | 62.66 | 14 |
| CPA | 1,375,151.61 | 300.48 | 461.34 | 3 |
| GRASP | 1,110,087.76 | 300.45 | 20.12 | 9 |
| GRASP+EC | 1,060,597.74 | 130.55 | 6.30 | 21 |

the best known value, and the number of instances in which the method is able to match the best known solution (Best). In order to have a fair comparison, IM, CPA, and GRASP were executed for the same computing time on each instance (with an average computing time of about 300 s). GRASP+EC was executed for shorter running times (about 130 s) since this method requires less time to obtain higher quality solutions.

Results in Table 6 clearly indicate the superiority of GRASP and GRASP+EC with respect to the previous method IM. In particular, GRASP presents an average percentage deviation of 20.12 % and 9 best solutions, GRASP+EC 6.30 % and 21 best solutions, while the previous IM method presents an average percentage deviation of 62.66 % and 14 best solutions on average. Finally, the CPA method obtains, as expected, low quality solutions which indicates that simple approaches do not work well on this difficult problem. It is worth mentioning that if we run IM, CPA or GRASP for shorter running times they obtain lower quality solutions than those reported in this table; on the contrary, the combination of GRASP with EC is able to obtain very good solutions (the best known in most of the cases) in significantly shorter running times, requiring less than half of their CPU time.

We applied the *non-parametric Friedman test* for multiple correlated samples to the best solutions obtained by each of the four methods. This test computes, for each instance, the rank value of each method according to solution quality (where rank 1 is assigned to the best method and rank 4 to the worst one). Then, it calculates the average rank values of each method across all the instances solved. If the averages differ greatly, the associated $p$-value or significance will be small. The resulting $p$-value of 0.000 obtained in this experiment clearly indicates that there are statistically significant differences among the methods tested. Specifically, the rank values produced by this test are 1.82 (GRASP+EC), 2.15 (GRASP), 2.59 (IMA), and 3.44 (CPA).

We finally compare our best variant (GRASP+EC) with the previous best method (IM) with the well-known *Wilcoxon test* for pairwise comparisons, which answers the question: do the two samples (solutions obtained with GRASP+EC and IM in our case) represent two different populations? The resulting $p$-value of 0.001 indicates that the values compared come from different methods (using the typical significance level of $\alpha = 0.05$ as the threshold between rejecting or not rejecting the null hypothesis).

## 6 Conclusion

In this paper we propose several heuristics based on GRASP and ejection chains for the dynamic memory allocation problem. The proposed GRASP heuristics consist of two randomized greedy construction procedures and a local search procedure. An ejection chain intensification algorithm was also proposed and tested as a post-processing of the GRASP.

We performed a computational comparison of our proposals and a previous method. It clearly shows that our GRASP with ejection chains is able to improve the previous method for the problem considered. The performance of our method is definitely enhanced by the context-specific strategies described in Sects. 3 and 4 that we developed for this problem. However, we hope other researchers might find effective and GRASP with ejection chains could become a standard hybrid method in future implementations.

**Acknowledgments** This research was partially supported by the grant-invited -Professors-UBS-2012 of France, and by the the Ministerio de Economía y Competitividad of Spain (TIN2009-07516 and TIN2012-35632-C02), and the Generalitat Valenciana (Prometeo 2013/049).

## Appendix: Best known solutions

Table 7 shows in the first column the name of the instance, in the second column the best known value, which appears in bold when our new methods are able to improve it in this experiment w.r.t the best previously identified. The next column presents the solution value reached by the ILP formulation in Soto et al. (2011) solved with Xpress-MP, that is used as a heuristic when the time limit of 1 h is reached: the best solution found so far is then returned by the solver. Note that in some large instances, this method is not able to provide a solution within the 3,600 s of time limit considered. The following three columns show the deviation value with respect to the best known value for the IM, CPA, and GRASP methods, and the associated CPU time in seconds which is the same for the three algorithms. Finally, the last two columns show the Dev. BK value and the CPU time for the GRASP+EC method, respectively.

**Table 7** Final comparison

| Instance | Best known | ILP | IM (%) | CPA (%) | GRASP (%) | CPU(s) | GRASP+EC (%) | CPU(s) |
|---|---|---|---|---|---|---|---|---|
| adpcmdy.dat | 44,192 | 44,192 | 0.00 | 45.44 | 0.00 | 0.01 | 0.00 | 0 |
| alidydy.dat | 108,699 | 108,699 | 927.33 | 146.37 | 156.48 | 160.48 | 48.75 | 85 |
| cjpegdy.dat | 4,466,800 | 4,466,800 | 0.00 | 1.93 | 0.00 | 0.01 | 0.00 | 0 |
| compressdy.dat | 342,592 | 342,592 | 0.00 | 7.77 | 2.67 | 0.01 | 0.00 | 0 |
| fpsol2i2dy.dat | **2,794,787** | * | 52.29 | 23.67 | 0.00 | 1,015.13 | 0.00 | 1,000 |
| fpsol2i3dy.dat | **2,762,059** | * | 49.81 | 25.12 | 0.00 | 1,062.37 | 0.00 | 1,000 |
| gsm_newdy.dat | 7,808 | 7,808 | 0.00 | 17, 295.02 | 0.00 | 0.01 | 0.00 | 0 |
| gsmdy.dat | 1,355,390 | 1,355,390 | 0.00 | 0.00 | 0.13 | 0.01 | 0.13 | 0 |
| gsmdycorrdy.dat | 494,118 | 494,118 | 0.00 | 0.00 | 0.35 | 0.04 | 0.36 | 0 |
| inithx_i1dy.dat | 6,280,430 | * | 63.69 | 11.81 | 0.00 | 700 | 0.21 | 1,000 |
| lmsbdy.dat | 7,409,669 | 7,409,669 | 0.00 | 0.36 | 0.33 | 0.29 | 0.14 | 0 |
| lmsbv01dy.dat | 4,350,640 | 4,350,640 | 0.00 | 3.77 | 1.13 | 0.01 | 1.88 | 1,000 |
| lmsbvdy.dat | 4,323,294 | 4,323,294 | 0.00 | 2.27 | 0.00 | 0.01 | 1.14 | 0 |
| lmsbvdyexpdy.dat | 4,367,024 | 4,367,024 | 0.00 | 3.38 | 2.63 | 0.01 | 1.88 | 1,000 |
| lpcdy.dat | 26,888 | 26,888 | 0.00 | 43.67 | 22.02 | 0.02 | 26.19 | 0 |
| mpeg2enc2dy.dat | 9,812 | 9,886.81162 | 0.00 | 45.99 | 9.46 | 0.75 | 10.14 | 0 |
| mpegdy.dat | 10,613.625 | 10,613.625 | 0.15 | 41.75 | 4.62 | 0.13 | 26.54 | 0 |
| mug100_1dy.col | 28,890 | * | 0.00 | 109.98 | 25.49 | 14.71 | 21.47 | 0 |
| mug100_25dy.col | 30,499 | * | 0.00 | 87.51 | 5.98 | 11.89 | 7.00 | 0 |
| mug88_1dy.col | 25,527 | * | 0.00 | 93.13 | 9.23 | 11.43 | 12.74 | 0 |
| mug88_25dy.col | 24,310 | * | 0.91 | 73.59 | 0.91 | 7.78 | 0.00 | 0 |
| mulsol_i1dy.dat | **518,278** | * | 146.34 | 186.46 | 54.15 | 1,096.13 | 0.00 | 66 |
| mulsol_i2dy.dat | **654,533** | 764,693 | 94.33 | 165.68 | 24.94 | 1,086.69 | 0.00 | 71 |
| mulsol_i4dy.dat | **570,529** | * | 106.05 | 186.03 | 22.22 | 1,057.35 | 0.00 | 56 |
| mulsol_i5dy.dat | **574,723** | 748,781 | 121.00 | 178.61 | 27.15 | 1,080.78 | 0.00 | 59 |
| myciel3dy.col | 6,379 | 6,379 | 89.14 | 6.88 | 8.42 | 1.24 | 11.26 | 0 |
| myciel4dy.col | 18,455 | 18,455 | 44.88 | 18.73 | 18.11 | 6.07 | 10.21 | 0 |
| myciel5dy.col | 41,938 | 41,938 | 31.20 | 67.22 | 32.98 | 28.86 | 8.55 | 0 |
| myciel6dy.col | 108,077 | 108,077 | 66.31 | 60.45 | 40.01 | 94.96 | 15.27 | 1 |
| myciel7dy.col | **447,000** | 486,449 | 79.11 | 90.06 | 20.25 | 377.08 | 0.00 | 18 |
| queen5_5dy.col | 27,395 | * | 34.16 | 0.00 | 0.00 | 4.76 | 0.05 | 0 |
| queen6_6dy.col | **47,174** | * | 64.55 | 24.60 | 5.38 | 284 | 0.00 | 0 |
| queen7_7dy.col | **81,102** | * | 130.07 | 62.73 | 31.14 | 42.82 | 0.00 | 0 |
| queen8_8dy.col | **154,499** | * | 150.35 | 50.23 | 14.95 | 82.56 | 0.00 | 2 |
| r125.1cdy.col | **1,225,115** | * | 90.93 | 178.96 | 0.00 | 700.00 | 0.00 | 120 |
| r125.1dy.col | 61,537 | 61,537 | 83.24 | 17.83 | 15.61 | 33.38 | 12.26 | 0 |
| r125.5dy.col | **741,388** | * | 105.27 | 112.88 | 84.18 | 1,028.86 | 0.00 | 26 |
| spectraldy.dat | **15,472** | 15,472 | 6.72 | 25.44 | 6.31 | 0.01 | 0.00 | 0 |
| treillisdy.dat | 1,805.5625 | 1,805.5625 | 0.02 | 129.23 | 113.11 | 0.03 | 33.10 | 0 |
| turbocodedy.dat | 3,195 | 3,195 | 33.49 | 84.32 | 20.09 | 0.13 | 20.09 | 0 |
| volterrady.dat | 178 | 178 | 7.87 | 7.87 | 7.87 | 0.01 | 7.87 | 0 |
| zeroin_i1dy.dat | **576320** | * | 49.26 % | 219.00 % | 39.85 % | 1091.16 | 0.00 % | 79 |
| zeroin_i2dy.dat | **557295** | * | 67.91 % | 176.54 % | 25.79 % | 1086.34 | 0.00 % | 103 |
| zeroin_i3dy.dat | **620385** | 750128 | 60.81 % | 186.83 % | 31.29 % | 1063.72 | 0.00 % | 58 |

## References


Chimientia A, Fanucci L, Locatellic R, Saponarac S (2002) VLSI architecture for a low-power video codec system. Microelectron J 33(5):417–427

Coussy P, Casseau E, Bomel P, Baganne A, Martin E (2006) A formal method for hardware IP design and integration under I/O and timing constraints. ACM Trans Embed Comput Syst 5(1):29–53

Duarte A, Martí R, Resende MGC, Silva RMA (2011) Grasp with path relinking heuristics for the antibandwidth problem. Networks 58(3):171–189

Feo TA, Resende MGC (1989) A probabilistic heuristic for a computationally difficult set covering problem. Oper Res Lett 8:67–71

Feo TA, Resende MGC (1995) Greedy randomized adaptive search procedures. J Glob Optim 6:109–133

Glover F, Laguna M (1997) Tabu search. Kluwer Academic Publishers, New York

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

Lin S, Kernighan B (1973) An effective heuristic algorithm for the traveling salesman problem. Operat Res 21:498–516

Lozano M, Duarte A, Gortzar F, Martí R (2012) Variable neighborhood search with ejection chains for the antibandwidth problem. J Heuristics 18:919–938

Martí R, Duarte A, Laguna M (2009) Advanced scatter search for the max-cut problem. INFORMS J Comput 21(1):26–38

Martí R, Pantrigo JJ, Duarte A, Campos V (2011) Scatter search and path relinking: a tutorial on the linear arrangement problem. Int J Swarm Intell Res 2(2):1–21

Porumbel D (2009) DIMACS graphs: benchmark instances and best upper bound.

Resende MGC, Martí R, Gallego M, Duarte A (2010) Grasp and path relinking for the max–min diversity problem. Comput Operat Res 37:498–508

Resende MGC, Ribeiro CC (2010) Handbook of Metaheuristics. In: Potvin JY, Gendrau M (eds) Greedy randomized adaptive search procedures, 2nd edn. Kluwer Academic Publishers, New York, pp 283–320

Soto M, Rossi A, Sevaux M (2010) Métaheuristiques pour l'allocation de mémoire dans les systèmes embarqués. In: Proceedings of ROADEF 11e congrès de la société Française de Recherche Opérationelle est d'Aide à la Décision. Toulouse, France, pp 35–43

Soto M, Rossi A, Sevaux M (2011) A mathematical model and a metaheuristic approach for a memory allocation problem. Journal of Heuristics 18(1):149–167

Soto M, Rossi A, Sevaux M (2011) Two iterative metaheuristic approaches to dynamic memory allocation for embedded systems. In: Merz P, Hao JK (eds) Evolutionary computation in combinatorial optimization—11th European Conference, EvoCOP 2011, Torino, Italy. Proceedings, vol 6622 of Lecture Notes in Computer Science Springer, 250–261

Wuytack S, Catthoor F, Nachtergaele L, De Man H (1996) Power exploration for data dominated video application. In: Proceedings of IEEE International Symposium on Low Power Electronics and Design. Monterey, USA, pp 359–364