Partitioning Algorithms Analysis for Heterogeneous NoC based MPSoC

Igor K. Pinotti, Thais Webber*, Natanael Ribeiro, Carlos N. Fraga*, Rubem D. R. Fagundes*, César Marcon

PPGCC –Programa de Pós-Graduação em Ciência da Computação
*PPGEE –Programa de Pós-Graduação em Engenharia Elétrica
PUCRS – Pontifícia Universidade Católica do Rio Grande do Sul
Avenida Ipiranga, 6681 – Porto Alegre, Brazil – 90619-900
{igor.pinotti, thais.webber}@acad.pucrs.br, cesar.marcon@pucrs.br

***Abstract* - Several new applications have high complexity degree, requiring high processing rate and memory usage. Multiprocessor System-on-Chip (MPSoC) is a promising architecture to fulfill these requirements, due to its high parallelism that enables several tasks been executed at the same time. One problem in current heterogeneous MPSoC design is application's tasks partitioning aiming energy consumption minimization and load balance. In order to optimize partition problems, many algorithms have been applied to generate quality solutions. This work aims to analyze and compare stochastic and heuristic partitioning algorithms for obtaining low energy consumption and load balance when applied to tasks partitioning onto heterogeneous MPSoC.**

Keywords - MPSoC, NoC, Partitioning, Mapping.

I. INTRODUCTION

Planar network-on-chip (NoC), or 2D NoC, is an efficient communication infrastructure for multiprocessor system-on-chip (MPSoC) architectures. 2D NoC is typically composed by a set of routers interconnected by communication channels. In NoC topologies, each router connects to a module and both are placed inside a limited region of an integrated circuit called tile. Low energy consumption, performance, scalability, modularity, and communication parallelism, make NoCs powerful communication architecture for SoC [1].

In order to meet the ever-rising performance constraints, NoCs can integrate instruction set processors (ISPs), DSPs, FPGA fabric tiles, IPs and specialized memories on a single chip towards MPSoC development. In this context, homogeneous MPSoC consist of identical processing elements that can support some applications; and heterogeneous MPSoC consists of different types of processing elements that can support a variety of applications, i.e., the distinct features of different processing elements (PEs) are used to minimize the energy consumption improving performance.

Ogras *et al.*[2] have proposed to divide NoC architectural design into three dimensions, namely communication architecture synthesis, communication paradigm selection, and application partitioning/mapping optimization. In this paper we are interested in the application-partitioning problem, which consists in finding associations of tasks into groups, according to a given criterion that is normally expressed by a cost function. Each group of tasks is associated to a tile (a mapping) containing a processor to minimize some given cost function, which depends on the type of the processor. The *partitioning* of ***k*** tasks in groups generates Bell (***k***) possible solutions, and task groups *mapping* onto *n* processors can generate *n!* possible solutions. Considering a SoC containing hundreds of tiles, the solution is unfeasible if an exhaustive search is applied on the design space. Therefore, better SoC implementations require the development of efficient partitioning and mapping approaches.

In this paper we provide a comparative analysis of classical partitioning algorithms such as Simulated Annealing (SA) [5], Tabu Search (TS) [6], and Kernighan & Lin (KL) [8]. The main contribution of this work is the application of these algorithms in the context of heterogeneous MPSoC design, using energy consumption and load balance as cost functions. Sections II and III describe the theoretical background about application and NoC description, as well as presenting partitioning algorithms. Sections IV and V shows the experimental results. Finally, Section VI presents final considerations.

II. APPLICATION AND NOC DESCRIPTIONS

This section presents a theoretical background related to graph based representations for application's tasks and overall project requirements.

Definition 1: A TCG (*Task Communication Graph*) is a directed graph $<T, S>$, where $T = \{t_1, t_2, ..., t_n\}$ represents the set of *n* tasks in a parallel application, i.e. the set of TCG vertices. Assuming s_{ab} is the quantity of package bits sent from task t_a to task t_b, hence the set of edges *S* is $\{(t_a, t_b) \mid t_a, t_b \in T, s_{ab} \neq 0\}$, with each edge attached to its s_{ab} value, is the total communication amount between tasks of an application.

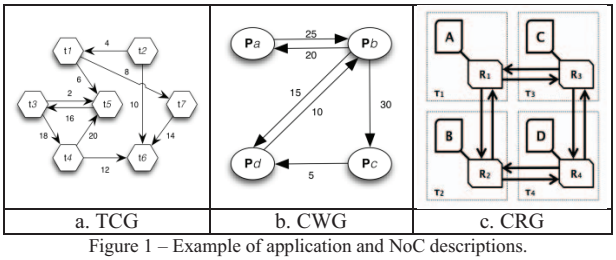
Definition 2: A CWG (*Communication Weighted Graph*) is a directed graph $<P, W>$, similar to TCG, but $P = \{p_1, p_2, ..., p_n\}$ represents the set of *n* processors involved with an application, also representing the number of *tiles* in the architecture. Furthermore, w_{ab} is the total communication amount (in bits) transmitted from processor p_a to processor p_b. The set of edges *W* is $\{(p_a, p_b) \mid p_a, p_b \in P, w_{ab} \neq 0\}$ with each edge attached to its w_{ab} value, representing all the communication between MPSoC processors. CWG reveals the relative communication volume of an application.

Definition 3: A CRG (*Communication Resource Graph*) is a directed graph $<\Gamma, L>$, where $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ represent the set of tiles, i.e., the set of CRG vertices. Also $L = \{(\tau_i, \tau_j), \forall \tau_i, \tau_j \in \Gamma\}$ corresponds to the set of CRG edges, i.e., the set of routing paths from tile τ_i to tile τ_j. The CRG vertices and edges represent, respectively, the routers and their physical links.

The communication behaviour of a given application in MPSoC can be expressed using a 2D direct mesh topology as communication infrastructure. Figure 1 illustrates an example of descriptions using TCG, CWG and CRG graphs.

Financial support granted by CNPQ and FAPESP to the INCTSEC (National Institute of Science and Technology Embedded Critical Systems Brazil), processes 573963/2008-8 and 08/57870-9; CAPES-Brazilian Ministry of Education (PNPD project 058792/2010).

978-0-7695-4929-3/12 $26.00 © 2012 IEEE
DOI 10.1109/SBESC.2012.42



a. TCG | b. CWG | c. CRG

Figure 1 – Example of application and NoC descriptions.

Figure 1a. shows a TCG, where T = $\{t_1, ..., t_7\}$, and S = $\{(t_1, t_5) \mid 6, (t_1, t_7) \mid 8, (t_2, t_1) \mid 4, (t_2, t_6) \mid 10, ...\}$. Figure 1b. shows a CWG with P = $\{p_A, \ldots, p_D\}$, and W = $\{(p_A, p_C) \mid 30, (p_C, p_D) \mid 25, \ldots\}$. Figure 1c. shows a CRG, where T = $\{\tau_1, ..., \tau_4\}$, and L = $\{(\tau_1, \tau_2), (\tau_1, \tau_3), (\tau_2, \tau_4), (\tau_3, \tau_4)\}$.

A. NoC Energy Model

Processors and communication infrastructure determine the energy consumption of a given application mapped on MPSoC. The sum of the consumed energy, from all grouped tasks executed in a given processor, enables to estimate individual processor energy consumption. In a heterogeneous scenario, the energy consumed in each processor, by a given task, could vary e.g. according to the architecture as well as optimized procedures considering task type in a processor. The amount of bits communicated between groups of tasks, mapped in different processors, provides the total energy consumption related to the communication architecture. The energy consumption originated by running tasks on processors, plus the energy applied to the communication architecture, determines the choice of partitions.

The NoC energy consumption model applied in this work is similar to [4]. The dynamic energy consumption is related to the packages exchange through the NoC, dissipating energy inside each router and on the links where the package passes by. E_{bit} is an estimation of the dynamic energy consumption for each bit, when the bit changes its value (i.e. polarity). E_{bit} is divided in three components: (i) ER_{bit} - dynamic energy consumed on the routers (e.g. wires, buffers and logic gates); (ii) ELH_{bit} and ELV_{bit} (EL_{bit}) – dynamic energy consumed on horizontal and vertical links between tiles, respectively; and (iii) EC_{bit} – dynamic energy consumed on links between each router and its local processor. For regular 2D mesh NoCs with square dimension tiles, it is reasonable to estimate that ELH_{bit} and ELV_{bit} have the same value. Due to this, we assume EL_{bit} as a simplified way to represent ELH_{bit} and ELV_{bit}.

Eq. (1) computes the dynamic energy consumed by a bit passing through the NoC from tile τ_i to tile τ_j, with η being the number of routers that the bit passes through.

$$(1) \quad E_{bitij} = \eta \times ER_{bit} + (\eta - 1) \times EL_{bit} + 2 \times EC_{bit}$$

Being τ_i and τ_j the tiles to which p_a and p_b are respectively mapped, the dynamic energy consumed by all communications traffic $p_a \rightarrow p_b$ is given by $E_{bitab} = W_{ab} \times E_{bitij}$. The total amount of NoC energy consumption (*ENoC*) related to all communication traffic between processors (|W|) is given by Eq. (2).

$$(2) \quad ENoC = \sum_{i=1}^{|W|} E_{Bitab}(i), \ \forall p_a, p_b \in P$$

The partitioning cost functions use the NoC energy model parameters stated by Eq. (1), only exploring the communication needs without the exact processor position into NoC, thus the number of hops between two communicating processors is unknown. Due to this fact, partitioning cost function uses the *average of hops* concept, which allows computing the average energy consumption of all possible paths.

Let both X and Y be the number of tiles in horizontal and vertical dimensions of a NoC, respectively. Therefore Eq. (4) computes the total number of hops of paths that all processors have regarding to XY routing algorithm. The *average of hops* is computed dividing the summation of all hops, of all paths, of all processors, by the total number of communications, which is following stated by Eq. (3), (4), (5), and (6).

$$(3) \quad Hops_{total} = \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} \sum_{i=0}^{X-1} \sum_{j=0}^{Y-1} (|x - i| + |y - j|)$$

$$(4) \quad Num_{Processors} = X \times Y$$

$$(5) \quad Max_{Comm} = Num_{Processor} \times (Num_{Processors} - 1)$$

$$(6) \quad Hops_{Average} = {}^{Hops_{Total}}/_{Max_{Comm}}$$

The $Hops_{Average}$ value is applied on Eq. (1) replacing the value η, resulting on an average value of E_{Bitij}. Thus, the energy consumption estimation, of each communication used during the partitioning process, is the result of a multiplication of E_{Bitij} by the communication volume.

B. Energy Reduction Model

An energy reduction model was implemented to attend the partitioning process aiming energy consumption minimization. Figure 2 shows an algorithm developed to cope with energy constraints. Remark that all implemented partitioning algorithms use this model as cost function to check if the partitioning process respects energy constraints.

```
1.  double communicationCost ← 0;
2.  for(int s ← 0; s < size(); s++){
3.    Association as ← getAssociation(s);
4.    if(as.getTasksList() == null)
5.      continue;
6.    for(int t ← 0; t < size(); t++){
7.      if(t == s)
8.        continue;
9.      Association at ← getAssociation(t);
10.     if(at.getTasksList() == null)
11.       continue;
12.     for(Task ts: as.getTasksList()){
13.       for(Task tt: at.getTasksList())
14.         communicationCost ← communicationCost +
            ts.communicationVolumeSentToTask(tt);
15.     }
16.   }
17. }
```

Figure 2 – Pseudo-code of energy reduction model algorithm.

The algorithm is composed by four nested loops. The outer loop (lines 2 to 17) searches for all processor/tasks associations

related to source associations. Meanwhile, the inner loop (lines 6 to 16) searches for all processor/tasks associations related to target associations. The communication value obtained from the communication between source tasks associations and target tasks associations are computed inside two nested loops (from line 12 to 15). The computed cost value is stored in the variable *communicationCost*.

C. LOAD BALANCE MODEL

A partitioning process aims to distribute application's tasks in a given number of groups, attending some criteria. However, this process needs a load balance supervising to avoid tasks concentration in just a couple of processors while other processors are idle. To cover this problem, a load balance model was developed to apply in this work, based on minimizing the *mean square error* (*MSE*) as stated in Eq. (7).

$$MSE = \frac{\sum_{t=1}^{n} e_t^2}{n} \quad (7)$$

The load balance algorithm (Figure 3) first computes the total CPU usage of all processors, storing in totalCpuUse. The average CPU usage per processor is stored in mediumValue. The loop (lines 11 to 16) computes the absolute error (ABSE), i.e. the difference between the average CPU usage of processors and the CPU usage of each processor. This value is squared and gathered in MSE, in such a way that all computed ABSE are stored inside MSE. Finally, the MSE value is divided by the number of involved processors, reflecting the mean square error. This approach is a fast and simple way to detect errors, usually leading to a good load balance on the partitioning result.

```
1.  double totalCpuUse ← 0;
2.  for(Association as: associations){
3.    if(as.getTasksList() == null)
4.      continue;
5.    procType ← as.getProcessor();
6.    for(Task task: as.getTasksList())
7.      totalCpuUse ← totalCpuUse+task.getProcessorUse();
8.  }
9.  double mediumValue ← totalCpuUse / size();
10. double MSE ← 0;
11. for(Association as: associations) {
12.   if(as.getTasksList() == null)
13.     continue;
14.   double ABSE  ← mediumValue - as.getProcessorUse();
15.   MSE ← MSE + ABSE x ABSE;
16. }
17. MSE ← MSE / size();
18. return MSE;
```

Figure 3 – Pseudo-code of load balance model algorithm.

III. PARTITIONING ALGORITHMS

The following algorithms are used as base for comparison for solving the partitioning problem in heterogeneous MPSoC: two stochastic search algorithms (SA and TS) and a heuristic graph partitioning algorithm (KL).

A. SA ALGORITHM

Simulated Annealing (SA) algorithm [5] is a special class of randomized local search algorithms with probabilistic characteristics, which can be applied in various domains. The partitioning process optimization, considering a very large number of application tasks and selected criteria, is analogous to the annealing process used for metals, where the value of temperature is decreased slowly till it approaches the freezing point. The energy within the material corresponds to the partition placement score.

An *annealing schedule* specifies (i) a beginning temperature, (ii) a temperature decrement function, (iii) an equilibrium condition at each temperature, and (iv) a convergence (or frozen) condition. The simulated annealing method begins with a random initial partitioning. An altered partitioning is generated, and the resulting change in score Δs is calculated. If $\Delta s < 0$, (the system level went to a lower energy level), then the move is accepted. If, $\Delta s \geq 0$, then the move is accepted with probability $e^{-\Delta s/t}$. As the simulated temperature t decreases, the probability of accepting an increased score decreases. This algorithm can climb out of local minima to find a global optimum if the proper condition on the annealing schedule are satisfied.

The SA implemented explores several partitioning solutions by using two nested loops. The external loop (lines 3 to 27) tries to localize well-defined partitions reaching global minimum. The inner loop (lines 10 to 26) explores small changes inside the obtained partition (from the external loop), aiming a global optimum; i.e. it does a sharpening at the partitioning solution provided by the external loop. Figure 4 shows the pseudo-code with the detailed nested loops, illustrating SA algorithm.

```
1.   globalMinimumCost ← Maximum double value
2.   interaction ← Interaction parameter
3.   while(interaction > 0) {
4.     interaction--
5.     if(randomBigMove() == false)
6.       continue
7.     localMinimumCost ← computedCost()
8.     saveComputedMoveInLocalMinimumOne()
9.     temperature = Temperature parameter
10.    while(temperature > 0) {
11.      temperature--
12.      if(randomSmallMove() == false)
13.        continue
14.      if(costFunctionComparison(localMinimumCost,
             computedCost())){
15.        localMinimumCost ← computedCost()
16.        saveComputedMoveInLocalMinimumOne()
17.      }
18.      else {
19.        if(!acceptableTreshold(temperature,
               computedCost(), localMinimumCost))
20.          restoreLocalMinimumMoveToComputedOne()
21.      }
22.    }
23.    if(costFunctionComparison(globalMinimumCost,
           localMinimumCost)){
24.      globalMinimumCost = localMinimumCost
25.      saveLocalMinimumMoveInGlobalOne()
26.    }
27.  }
```

Figure 4 – Pseudo-code of SA algorithm.

A random search in a wide spectrum of possibilities is the effect of the two nested loops. These results allow SA to find partitions that minimize the *computedCost()* function, the main objective of the partitioning process. The internal loop begins with a given random partitioning provided by the external loop,

and a resultant value is stored in *localMinimumCost*. At each end of the internal loop, the *localMinimumCost* is compared with the previous stored. If the current value is smaller than the stored value, the current value is stored as a new best partitioning, and it becomes the new current partitioning. At the same time, some worse partitioning values can be accepted as new current partitioning due to the *temperature* parameter used to control the stochastic acceptance procedure [4]. This parameter, when loaded with higher values, implies in a greater probability to accept worse partitioning. Conversely, lower values applied to this parameter implies in a smaller probability to accept worse partitioning. The *temperature* parameter is decremented after each internal loop execution, and restarted in each external loop. This procedure is done to avoid local minimum partitioning values.

The *randomBigMove()* and *randomSmallMove()* functions, lines 5 and 12 respectively, explore the random partitioning and stand for returning a Boolean status if a given partitioning result attends the current constraints configured (e.g. energy consumption, load balance, CPU occupation). If the current partitioning could not attend current constraints, the *iteration* parameter is decremented, and a new partitioning set is explored. The best partitioning obtained at the end of the internal loop is compared with the best global partitioning, and the one with smaller cost is stored as the new best global partitioning. In the external loop, several modules are randomly swapped to produce a widely different partitioning [4].

D. TS ALGORITHM

Tabu Search (TS) algorithm [6] is a general combinatorial optimization search technique applied on a variety of problems. This algorithm's search is similar to iterative improvement in which moves are required, transforming the current solution to its best neighbouring solution. TS maintains a *tabu list* of its *r* most recent moves (e.g. pairs of tasks that have been swapped recently), with *r* a prescribed constant that determines the *tabu list* size; and moves using elements that are part of the *tabu list* cannot be performed [3].

The *tabu list* is the core of TS, preventing cycling near local minimum and also enabling uphill moves. In other words, *tabu list* keeps the process from cycling in one neighbourhood of the solution space. At each iteration, solutions are checked against the tabu list. A solution that is on the list will not be chosen for the next iteration (unless it overrules its tabu condition by what is called an *aspiration condition*). Additionally, at each iteration, a steepest-descent solution that does not violate the tabu condition is chosen. If no non-tabu improving solution exists, the best non-improving solution is taken. The combination of memory and gradient descent allows for diversification and intensification of the search. Local minima are avoided while good areas are well explored [6]. In contrast to SA that exploits random moves, TS exploits data structures of the search history as a condition for next moves. More generally, TS is a search method designed to cross boundaries of feasibility, normally treated as barriers, and it systematically imposes and releases constraints to allow the exploration of forbidden regions [6][7].

Similar to SA, TS is implemented with two nested loops. The TS internal loop (lines 7 to 13) generates at most one pair of modules candidate to swapping. The swap is done at the external loop (lines 2 to 23).

```
1.  interaction ← Interaction parameter
2.  while(interaction > 0){
3.    interaction--
4.    if(randomBigMove() == false)
5.      continue
6.    temperature ← Temperature parameter
7.    while(temperature > 0){
8.      temperature--
9.      if(randomSmallMove() == true){
10.       if(candidateIsNotOnTabuList())
11.         addCandidateToCandidateList()
12.     }
13.   }
14.   if(locateBestCandidate()){
15.     if(candidateBetterThanBest(){
16.       candidateIsNewBest()
17.       saveComputedMoveInLocalMinimumOne()
18.     }
19.     addFeaturesDifferences()
20.     if(tabuListIsFull())
21.       expireFeaturesOfTabuList()
22.   }
23. }
```

Figure 5 – Pseudo-code of TS algorithm.

The computation of energy consumption is incremental as in SA. The internal loop randomly searches module pairs to swap (*randomSmallMove()* function), looking for the pair that better attends the given requirement, e.g. energy consumption (*locateBestCandidate()* function). To optimize the search, there is a swap list with all distinct pairs of 'swappable' modules. The search process (*addCandidateToCandidateList()* function) randomly produces indices to access the swap list. If a swap saves more energy than a previous one, this pair is stored in the swap list. When the internal loop ends, if a pair that saves energy was found, it is inserted in the *tabu* list, removed from the swap list (lines 14 to 21), and the current partitioning is modified with the selected swap. If no pair exists, no action is taken, and a new internal loop is executed.

The main TS parameters are *neighborhood* and *iteration*. The parameter *neighborhood* was renamed to *temperature* to maintain similarity with the SA. Thus, *temperature* denotes the number of pairs evaluated at each swap step; and iteration, which controls the external loop, denotes the number of performed swaps.

E. KL* ALGORITHM

Kernighan & Lin (KL) [8] proposed a bisection algorithm for a graph, starting with a random initial partition, and then using pairwise swapping of vertices between partitions to reduce the cutsize until no improvement is possible. Schweikert and Kernighan [9] proposed the use of a net model in order to handle hypergraphs. Fidducia and Mattheyses [10] reduced time complexity of KL algorithm from $O(n^3)$ to $O(n \log n)$ with some operational modifications.

The KL classical algorithm starts by initially partitioning the graph into two subsets of equal sizes. Vertex pairs are exchanged across the bisection if the exchange improves the cutsize. The procedure is carried out iteratively until no further improvement can be achieved. This algorithm belongs to the

group migration method which uses a deterministic method that is often trapped at a local optimum. Thus, it is avoided to proceed further, then being one of the most successful heuristics for partitioning problems. Moreover, the method is particularly well suited for bisection (dividing the graph into two modules), but can be generalized as well to partitioning into unequal pieces, becoming the basis of a hierarchical partitioning scheme.

However, KL algorithm does not totally comply with the application's partitioning process. The partitioning needs to set a group related to a given processor, for each task of all application's tasks. Depending on the number of processors, the number of groups will usually be higher than two. Due to the fact that KL works with a steady initial *bisection* of two modules (i.e. two groups) and requires a predefined size of partitions, modifications are needed for applying to application's tasks partitioning on MPSoC. A modified KL algorithm (KL*) was implemented (Figure 6), based on the classical KL idea with some improvements from [9] and [10]. KL* is also implemented with two nested loops, but includes modifications such as (i) only a single vertex is moved across the cut in a single move, and (ii) when no moves are possible, only those moves that give the best cutsize are actually carried out.

```
1.  interaction ← Interaction parameter
2.  while(interaction > 0){
3.    interaction--
4.    createInitialBipartition()
5.    addListInListOfLists()
6.    while(listOfListsNotFull()){
7.      if(verifyBipartition()){
8.        if(depth_search)
9.          depth_bipartition()
10.       else
11.         width_bipartition()
12.       copyLists()
13.       while(copyListsNotEmpty){
14.         tripleMovement()
15.       }
16.       refreshListOfLists()
17.     }
18.     checkListOfProcessors()
19.     if(verifyPartition()){
20.       partitionIsNewBest()
21.       saveComputedMoveInLocalMinimumOne()
22.     }
23.     clearAll()
24.   }
25. }
```

Figure 6 – Pseudo-code of KL* algorithm.

KL* external loop (lines 2 to 25) is responsible for creating a random initial *bisection/bipartition*, from the set of application's tasks. The *createInitialBipartition()* function tries to attend a *bisection* of equal sizes, and also controls the number of times the algorithm tries to achieve a satisfactory partitioning result, managed by the parameter *interaction*. The internal loop (line 6 to 20) verifies if the bipartition created attend the constraints (e.g. load balance and energy consumption) related to the given group of processors (*verifyBipartition()* function); if it does not attend them, another *bisection* is created. At this step, two *bisection* approaches have been developed to explore a better movement. The algorithm can search for a good partition analysing the communications edges of processors groups in two ways: (i) *depth_bipartition()* function (line 9) - a node of the current *bisection* performs a *bisection*, increasing the tree size firstly in just one side; however, once reaching the *bisection* limit, it restarts on the first remaining *bisection* node; (ii) *width_bipartition()* function (line 11) - both nodes of the current *bisection* perform a *bisection*, generating a binary search tree.

The bisection approach is set before partitioning in the algorithm, as a parameter, and the chosen approach is not changeable during the process. The *bisection* process goes (i.e. limit of the *bisection*) until it reaches a partition to each group of related processors, trying to allocate all tasks in those partitions. The partitions generated after the *bisection* exchanges tasks (*tripleMovement()* function) to attend each partition constraints (i.e. group related to a processor). After all possible exchanges, the better movement is carried out. If a partition attends the constraint, it is stored as best partition (lines 20 and 21) and moved aside from bisection. If, at the end of all possible *bisections*, there is no partition for each group that attends the constraints, new processes begin. The whole *bisection* process is controlled by the *interaction* parameter, and auxiliary lists are used to verify the available idle groups.

IV. EXPERIMENTAL SETUP

Two MPSoC architectures were simulated over CAFES framework [4]. In order to evaluate SA, TS and KL* algorithms for partitioning process over application's tasks, two test sets are applied over the framework. Each test set is composed by four sets of applications, and a different architecture as follows.

(i) Test-Set 1: four synthetic applications (10, 20, 30, 40 tasks), MPSoC with a 2x3 NoC size composed by a set of processors: 2x ARM 800Mhz, 2x PowerPC 400Mhz, 1x MIPS 200Mhz, 1x Intel x86 1800MHz;

(ii) Test-Set 2: four synthetic applications (10, 20, 30, 40 tasks), MPSoC with a 2x3 NoC size composed by a set of processors: 2x ARM 400Mhz, 2x ARM 800Mhz, 1x ARM1200Mhz, 1x ARM 600Mhz.

The applications sets are composed of synthetic applications with random characteristics, following intervals of: (i) number of communications - from 1 to 7; (ii) communication quantity – from 100 phits to 400 phits; (iii) power dissipation – from 5 uW to 15 uW; (vi) processor occupation – from 8% to 20%. The partitioning process also respects 100% CPU occupation and maximum energy of 150 watts per processor. Parameters related to the algorithms, such as *temperature* and *interaction*, are configured to 500 and 1500, respectively. The partitioning results are compared in energy consumption, computation time, and memory consumed.

V. EXPERIMENTAL RESULTS

The experiments have evaluated quality and influence of partitioning algorithms on energy consumption minimization. Other constraints such as load balance (MSE), computation time, and memory usage, are presented in the results for providing a wide comparison among all algorithms. The results related to energy consumption, load balance and elapsed computation time are presented for Test-set 1 (Figure 7, Figure 8 and Figure 9) and for Test-set 2 (Figure 10, Figure 11, and Figure 12), respectively.

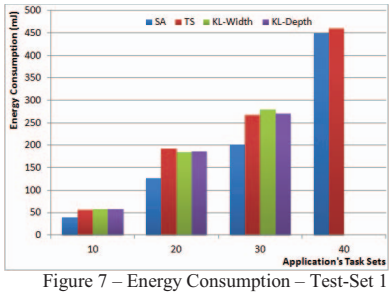


Figure 7 – Energy Consumption – Test-Set 1

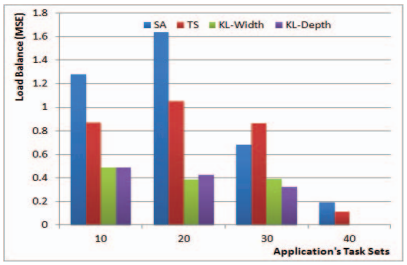


Figure 8 – Load Balance (MSE) – Test-Set 1

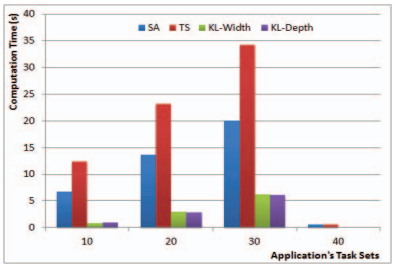


Figure 9 – Computation Time – Test-Set 1

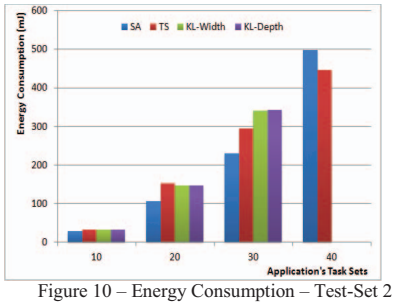


Figure 10 – Energy Consumption – Test-Set 2

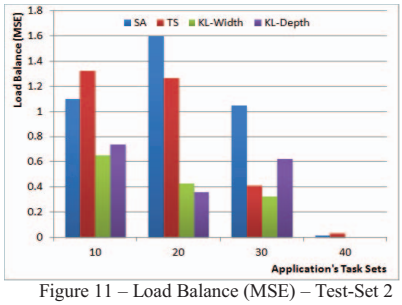


Figure 11 – Load Balance (MSE) – Test-Set 2

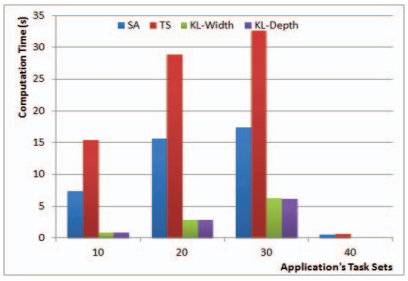


Figure 12 – Computation Time – Test-Set 2

TS consumes more CPU time than any other method in this comparison due to its list based method, and SA requires vastly more CPU time than KL* (a migration method) do. However, both KL* versions did not produce any stable partition over 40 tasks. In terms of energy consumption, TS and SA produce better results in almost all test cases. On the other hand, both approaches of KL* produce more load balanced partitions.

VI. CONCLUSIONS

Partitioning algorithms can be compared on their difficulty of implementation, their performance on common partitioning problem, and their runtimes. The partitioning methods have become more sophisticated over time as computation time has become more affordable. This work explores the application of a variety of algorithms capable of performing application's tasks partitioning, aiming better results related to energy consumption minimization mainly on MPSoC. KL* is a migration method that enables to achieve good load balanced partitions with low computation effort.

The objective of this research is to provide an evaluation of a set of partitioning algorithms that are considered well-suited for application's tasks partitioning on NoC based MPSoC, especially those with heterogeneous processing elements.

REFERENCES

[1] SINGH, A. K.; JIGANG, W.; PRAKASH, A.; SRIKANTHAN, T.: 'Mapping Algorithms for NoC-based Heterogeneous MPSoC Platforms', *IEEE Comput.*, 2009, pp. 133-140

[2] OGRAS, U.Y.; HU, J.; MARCULESCU, R.: 'Key research problems in NoC design: a holistic perspective', *CODES+ISSS'05*, 2005, pp. 69–74

[3] ALPERT, C. J.; KAHNG, A. B.: 'Recent directions in netlist partitioning: a survey', *INTEGRATION, the VLSI Journal,* 1995, 1-81.

[4] MARCON, C. et al. 'Comparison of network-on-chip mapping algorithms targeting low energy consumption', *IET Comput. Digit. Tech*., 2008, Vol. 2, No. 6, pp. 471-482

[5] KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P.: 'Optimization by simulated annealing', *Science*, 1983, 220, pp. 671–680.

[6] BEATY, S. J.: 'Genetic Algorithms versus Tabu Search for Instruction Scheduling', *Intern. Conference on Neural Networks and Genetic Algorithms*, 1993, pp. 496-501.

[7] WIANGTONG, T.; CHEUNG, P. Y. K.; LUK, W.; 'Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware-Software Codesign', *Design Automation for Embedded Systems,* 2002, pp. 425-449.

[8] KERNIGHAN, B.; LIN, S. 'An efficient heuristic procedure for partitioning graphs', *Bell Systems Technical Journal*, 1970, pp. 291–307.

[9] SCHWEIKERT, D. G.; KERNIGHAN, B.: 'A Proper Model for the Partitioning of Electrical Circuits', *Proceedings of the 9th Design Automation Workshop,* 1972, 57-62.

[10] FIDUCCIA, C. M.; MATTHEYSES, R. M.: 'A Linear-Time Heuristic for Improving Network Partitions', *Proceedings of the 19th Design Automation Conference,* 1982, pp. 175-181.