Optimization Of Run-time Mapping On Heterogeneous CPU/FPGA Architectures

Omar Souissi, Rabie Ben Atitallah, Abdelhakim Artiba, Salah E. Elmaghraby

9^{th} *International Conference of Modeling, Optimization and Simulation - MOSIM'12*
June 6-8, 2012 - Bordeaux - France
"Performance, interoperability and safety for sustainable development"

Optimization Of Run-time Mapping On Heterogeneous CPU/FPGA Architectures

A. Omar Souissi, B. Rabie Ben Atitallah , C. Abdelhakim Artiba **D. Salah E. Elmaghraby**

University of Valenciennes
59313 Valenciennes cedex 9 - France
A. omar.souissi@meletu.univ-valenciennes.fr,
C. abdelhakim.artiba@univ-valenciennes.fr,
B. rabie.benatitallah@univ-valenciennes.fr

North Carolina State University
Raleigh, NC 27695 - USA
D. elmaghra@ncsu.edu

ABSTRACT: *This research investigates the problem of the optimisation of run-time task mapping on a real-time computing system CPU/FPGA (Central Processing Unit/Field-Programmable Gate Array) used to implement intimately coupled hardware and software models. The case study analyzed in this work is inspired by the avionic industry. Indeed, real-time computing systems are increasingly used in aerospace and avionic industries. In the face of power wall and real-time requirements, hardware designers are directed more and more towards reconfigurable computing with the use of heterogeneous CPU/FPGA systems. In such systems, multi-core processors (CPU) provide high computation rates while the reconfigurable logic (FPGA) offers high performance and adaptability to the application real-time constraints. However, there is a lack of CAD (Computer-Aided design) tools able to deal with the development of applications on such heterogeneous systems. This work includes the development and the comparison of efficient heuristics that focus on the static initial task mapping, the dynamic mapping of new applications at run-time, and the dynamic re-configuration to avoid the real-time constraint violation.*

KEYWORDS: *Heterogeneous computing; Static and Dynamic mapping; Run-Time mapping; Algorithm; MLPT heuristic; Greedy heuristic.*

1 Introduction

Real-time systems lead to the increasing demands on performance, power, and flexibility requirements. Today, the heterogeneous CPU/FPGA (Central Processing Unit/Field-Programmable Gate Array) architecture is one of the most promising solutions in this context leading to high performance reconfigurable computing. FPGA is a chip containing a large number of logic blocks, these blocs are connected together by a configurable routing matrix as shown in Figure 1, which allows the re-configuration of the component functionality as desired. FPGAs offer cheap and fast programmable silicon on some of the most advanced fabrication processes. As cited in (Afonso *et al*, 2011), with the management of the parallelism in applications, FPGA technology can offer better performance, up to 10x, compared to standard CPUs (Central Processing Unit). FPGAs have also the benefits of high speed and adaptability to the application constraints, with reduced performance per watt comparing to CPUs. Furthermore, FPGA technology enables today to implement massively parallel architectures due to the huge number of programmable logic fabrics available on the chip. Such architectures can be customized at runtime using Dynamic Partial Reconfiguration (DPR) feature offered by recent FPGA technologies. Today, system designers are directed more and more towards heterogeneous architectures that gather multi-core CPUs coupled with FPGAs in order to address specific application constraints (timing deadlines, power consumption, etc.).

In order to harvest the maximum benefits of this heterogeneous architecture, we must offer efficient methods that make profit from the performance of each part to propose optimized solutions for the static and dynamic task mapping. As highlighted in (Kim *et al*, 2007), an important research challenge is how to assign tasks to the available resources in order to maximize some performance criterion of the heterogeneous architecture. In this research, we focus first on the static initial mapping, then we study the dynamic remapping to prevent overloads and to deal with the arrival of new tasks at run-time in the environment. The strategy adopted in this work offers a *hierarchy of mapping options* to assign tasks in order to minimize the communication costs in the CPU/FPGA archi-

tecture. It also permits the anticipation of overloads in the CPU cores.

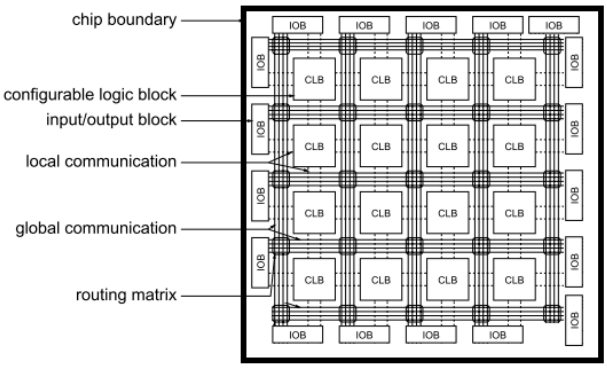

Figure 1: General structure of an FPGA

The main contributions of this paper are:

- The formulation of an exact method for the static initial mapping.
- The development of heuristic approach based on the LPT-Rule which we call the MPLT-Heuristic, also for the static initial mapping.
- The development of a fast greedy heuristic to improve the performance of the two methods mentioned above.
- The management of the dynamic reconfiguration to deal with an arrival of new tasks and overloads caused by the growth of task(s) in the CPU cores.

This paper is structured as follows. Section 2 presents some representative related contributions. Sections 3 describes the system model and the case study which is based on real tests in the avionic industry. The exact method, MLPT-rule heuristic, and a fast greedy heuristic for the static initial mapping are presented in Section 4. Section 5 details the dynamic reconfiguration to deal with overloads and the arrival of new tasks in the run-time environment. In Section 6, the experimental results and the comparison of the performance of the different methods are exposed. Finally Section 7 summarize the contributions of this paper and presents the perspectives of the futur research.

2 Related Contributions

Several works in the field of task mapping in computing systems have been proposed, such as (Christos Koulamas and George J. Kyparisis, 2007), (Christos Koulamas and George J. Kyparisis, 2008), (Wun-Hwa Chen and Chin-Shien Lin, 1998), (S. Mounir Alaoui *et al*, 1999) and (Hans-ulrich Heiss, 1992), which target identical parallel machines. Recently, research efforts are increasingly oriented towards solving the mapping problem onto heterogeneous computing systems. The mapping problem consists of two significant parts; the *matching* that involves assigning tasks to the available resources, and the *scheduling* that considers the execution sequence of tasks. In the literature, the majority of the existing works refer to the term *scheduling* to mean *mapping*. For example, (Oscar H. *et al*, 1977) present heuristic algorithms for scheduling independent tasks on non identical processors. Already in 1988, (Thomas L. Casavant *et al*, 1988) presented a taxonomy of scheduling in general-purpose distributed computing systems. At the highest level, they distinguish between *local scheduling*, which we called scheduling, and global scheduling, which we called matching. The second level is the differentiation between two categories of mapping which are static and dynamic. A number of articles propose different approaches to address the problem of static mapping. As (Ali *et al*, 2007), (Hyunok Oh and Soonhoi Ha, 1996) and (Tracy D. Braun *et al*, 2001) in which a comparison of eleven static heuristics is presented for mapping a class of independent tasks onto heterogeneous distributed computing systems. Additionally several papers have investigated the dynamic mapping problem; see (Kim *et al*, 2007), (Mahmood *et al*, 2010), (Jeannot *et al*, 2011), (Muthucumaru Maheswaran and Howard Jay Siegel, 1998), (Bora Ucar *et al*, 2005) and (Hans-ulrich Heiss, 1992). The different approaches explored include the greedy heuristics in (Ping Luo *et al*, 2007), in which authors evaluate and compare 20 greedy heuristics for mapping a class of independent tasks onto heterogeneous computing systems. In (S. Mounir Alaoui *et al*, 1999), the authors expose a genetic algorithm; finally a hybrid heuristic is presented in (Wun-Hwa Chen and Chin-Shien Lin, 1998).

Our work differs from those cited by focusing especially on matching tasks into the heterogeneous CPU/FPGA system. In addition, we consider in this work the two phases of mapping. Indeed, we use a strategy taking into account not only resource allocation but also the dynamic re-allocation in order to anticipate the onset of overloads. In this work, we exploit the heterogeneous architecture CPU/FPGA by using several options for assigning tasks, including the partitioning of the task between a CPU core and the FPGA resources.

3 Problem statement

We consider the challenge of mapping a test or a simulation project containing several models (or tasks) onto a dynamically reconfigurable CPU/FPGA architecture at the initial phase and at run-time. We denote by C the set of cores and by F the set of FP-

GAs as illustrated in Figure 2. We define *CardC* as the cardinality of *C* and *CardF* as the cardinality of *F*.

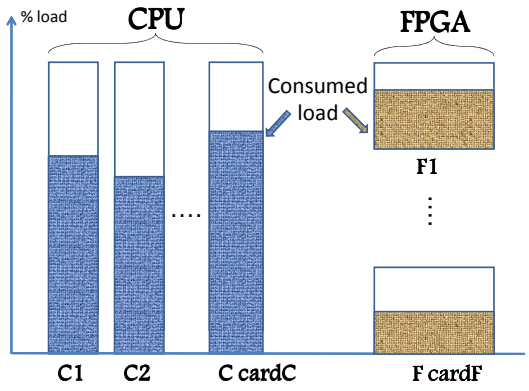

Figure 2: Occupancy of computing capacity at time t.

As described in Figure 3, we consider an example of a 16-tasks project mapped onto an architecture with 4-cores CPU and one FPGA. In a given project, models are sequentially executed respecting the task graph as shown in Figure 3. In the real-time environment, we consider that tasks are occupying at least their initial load during all the execution time of the project. In our case study C contains the set of cores $[C1, C2, C3, C4]$ and F contains only one element $[F1]$. Then $CardC = 4$ and $CardF = 1$.

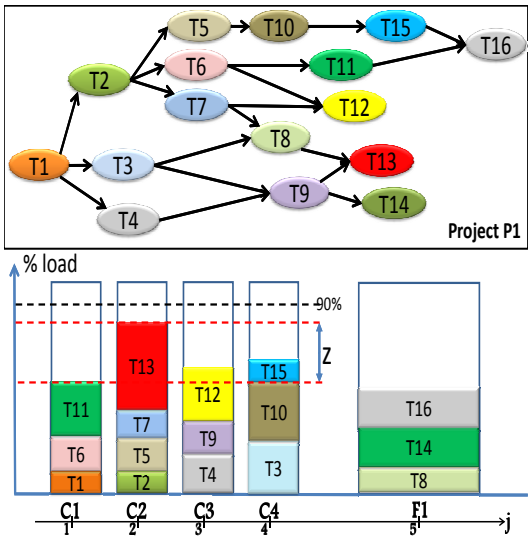

Figure 3: Example of task mapping onto a CPU/FPGA system

4 The static initial mapping

In real-time computing systems, the static initial mapping step is very important. At a given system workload, this step will guarantee that all real-time constraints will be satisfied just after the project is put into operation. In this section, we first present an exact method to get an optimal mapping of a given project, then we present two heuristic approaches in order to improve the speed performance. The first heuristic is based on the LPT-Rule[1], while the second is a greedy heuristic that we called The Check-greedy. For the static initial mapping on the heterogeneous CPU/FPGA architecture, we have two main goals. First, to avoid overloads, indeed in the context of our work (Avionic Testing) this requirement is very critical. Second, to minimize the cost. In this study we assume that assigning tasks to the FPGA is expensive due to the reconfiguration time and the communications cost between the CPU and the FPGA.

As shown in Figure 3, we define z as the difference between the maximum load and the minimum load of all available computing cores. To avoid overloads, we will balance the load assigned to each core in order to minimize z. Since we want to find a compromise between avoiding overloads and minimizing the costs of task allocation, we map the different tasks of a given project in the heterogeneous architecture according to three hierarchical mapping options:

- Option 1: Assign the entire task to the CPU.
- Option 2: Split the task between the CPU and the FPGA.
- Option 3: Assign the entire task on the FPGA.

4.1 Exact method

In order to model the exact approach, we let $p_o(i,j)$ be the fraction of the capacity of processor j that is occupied by task i under option o. $o = 1\ to\ 3, i = 1\ to\ n$ (for project P1, n=16), $j = 1\ to\ CardC \in C$, and $j = CardC + 1\ to\ CardC + CardF \in F$ (we consider in our model CardF= 1).

We define the binary variable x_o(i,j) as the decision variable to assign task i to processor j according to option o, then:

$$x_o(i,j) = \begin{cases} 1, \textit{ if task i is allocated to processor} \\ \quad j \textit{ in } C \cup F \textit{ under option } o \\ 0, \textit{ otherwise} \end{cases} \tag{1}$$

Now, we define Lc_j as the load of core j ∈ C, and Lf_j the load of hardware j ∈ F. This implies:

$$\begin{cases} Lc_j = \sum_{i=1}^{n} \sum_{o=1}^{2} x_o(i,j) * p_o(i,j), with j \in C. \\ Lf_j = \sum_{i=1}^{n} \sum_{o=2}^{3} x_o(i,j) * p_o(i,j), with j \in F. \end{cases} \tag{2}$$

[1]LPT-Rule: longest processing Time Rule is a well known scheduling heuristic

4.1.1 System constraints

- First, there are the processors capacity constraints:

$$Lc_j \leq 1 \ for\ each\ j \in C. \tag{3}$$

As our environment relies on the latest FPGA technology, we assume that the FPGA capacity is much larger than that of the CPU cores. Further, in this case study, we consider there is no constraint on FPGA's space occupation.

- Secondly, each task must be assigned to only one option:
$for\ i = 1\ to$ n.

$$\begin{cases} \sum_{o=1}^{2} \sum_{j=1}^{CardC} x_o(i,j) + x_3(i, CardC+1) = 1 \\ x_2(i, CardC+1) - \sum_{j=1}^{CardC} x_2(i,j) = 0 \} \end{cases} \tag{4}$$

These two sets of constraints ensure that each task is assigned according to one option. Further, if option $o = 2$ is selected, this implies a unique split of the task between a core processor and the FPGA.

- Finally, we define a constraint in order to anticipate the overloads. We then assign a high cost to a core when its load exceeds 90%. We define y_j, a binary variable, as follows:

$$y_j = \begin{cases} 1, if\ core\ j\ load\ is\ more\ than\ 90\% \\ 0, otherwise \end{cases} \tag{5}$$

Then, the constraint is:

$$\sum_{o=1}^{2} \sum_{i=1}^{n} (x_o(i,j) * p_o(i,j)) - 0.90 \leq y_j \ with\ j = 1\ to\ CardC \tag{6}$$

If one core load is higher than 90%, the left side of the previous equation shall be positive, so y_j will be equal to 1 as required by its definition. This coefficient is used in the objective function.

4.1.2 Objective variables definition

In order to define the gap between all available computing cores, we introduce the two variables u(j_1,j_2) and v(j_1,j_2) for every pair of processors j_1 and j_2 in C. Then, we have three options depending on the value of the expression between brackets in equation (7). As shown in Figure 4, if this expression is negative, it means that j_2's load is higher than j_1's. Then, v(j_1,j_2) will be positive, representing the (negative) gap between the two core loads. If it is less than zero, it means that j_1's load is higher than j_2's. Then u(j_1,j_2) will be positive, representing the (positive) gap between the two core loads.

$$[\sum_{i=1}^{n} \sum_{o=1}^{2} p_o(i,j_1) * x_o(i,j_1) - \sum_{i=1}^{n} \sum_{o=1}^{2} p_o(i,j_2) * x_o(i,j_2)] - u(j_1,j_2) + v(j_1,j_2) = 0 \quad \forall j_1, j_2 \in C \tag{7}$$

With $0 \leq u(j_1, j_2) \leq z$ *and* $0 \leq v(j_1, j_2) \leq z \quad \forall j_1, j_2 \in C$.

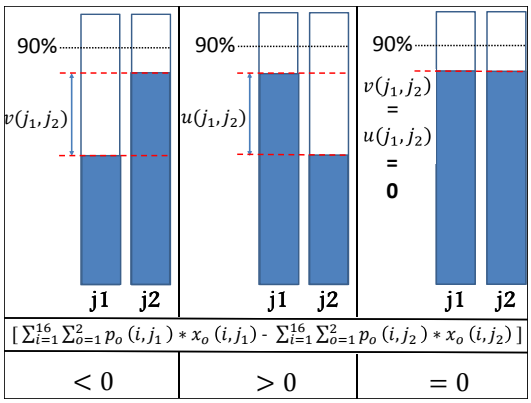

Figure 4: Objective variable cases

The variables u(j_1,j_2) and v(j_1,j_2) were introduced to secure a linear model by avoiding a *min* max operator in the objective function since minimizing the z parameter (for recall definition, see Figure 3) is equivalent to securing the *min* max(u(j_1,j_2),v(j_1,j_2)).

4.1.3 Objective function

We focus on avoiding any load exceeding 90% of the capacity of each core processor, and also ensuring load balancing among the core loads. Both objectives are realized by adopting the following objective function:

$$min[\mu 1 * z + \mu 2 * \sum_{i=1}^{n} x_2(i, CardC+1) + \mu 3 * \sum_{i=1}^{n} x_3(i, CardC+1) + \mu 4 * \sum_{j=1}^{CardC} y_j] \tag{8}$$

We define $\mu 1$, $\mu 2$, $\mu 3$, and $\mu 4$ as positive constants with $\mu 4 \rangle \mu 3 \rangle \mu 2 \rangle \mu 1 \rangle 0$ in order to get the best trade-off between avoiding overloads and minimizing the costs.

4.2 The MLPT-Rule Heuristic

LPT priority rule is one of the oldest methods in scheduling. Already in 1969, Graham (R. L. Graham, 1969) gave results about "the worst-case ratio bound of the LPT rule" for the parallel machine makespan[2]

[2]Makespan is the time difference between the start and finish of a sequence of jobs or tasks

minimization problem. But the use of the LPT rule and the MLPT (Modified version of the LPT) is still widespread. In 2008, Koulamas and Kyparisis (Christos Koulamas and George J. Kyparisis, 2008) demonstrated the robustness of this approach by implementing it for a two identical parallel machine scheduling problem.

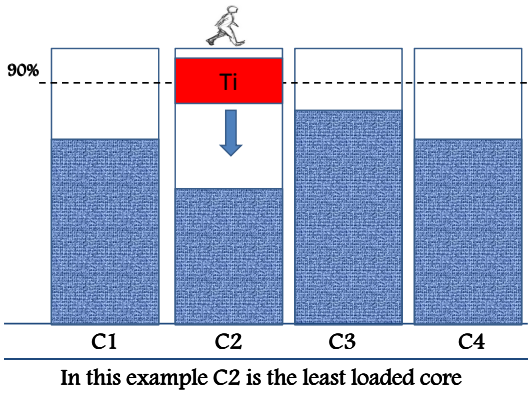

Figure 5: The MLPT Heuristic

In this paper, we use an MLPT Heuristic that implies the usage of the LPT-Rule on three steps, successively with the three options to keep the same strategy of mapping adopted for the exact method. Indeed, first we load with option 1, then with option 2 and finally with option 3. This heuristic requires a "preprocessing" step in which we rank the tasks according to the CPU occupancy rate p_1(i,j) in decreasing order. Then we use the LPT rule in order to load "the task of largest load first" on the "core with most available capacity" under option 1 respecting the load limit constraint (less than 90%) as described in Figure 5. We assume that n1 tasks could be loaded in STEP1. For the remaining tasks n-n1, we keep using LPT rule in STEP2 but now with option 2. Finally in STEP3, we assign the remaining tasks to the FPGA. The proposed algorithm is shown in Figure 6.

4.3 The Check-greedy Heuristic

A greedy algorithm, as defined in (Cormen *et al*, 1990), always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. In this work we introduce a greedy heuristic that we called "The Check-greedy", it is executed in three steps, first we load with Option 1 (the entire task is allocated to a certain CPU core), then with option 2 (split the task between a certain CPU core and the FPGA) and finally with option 3 (the entire task goes to FPGA). This heuristic requires a "preprocessing" step in which we rank the tasks in order of decreasing CPU occupancy rate p_1(i,j). Recall that Lc_j is the total load of core j.

(1) Initialize: T the list of tasks of project P1

(2) Preprocessing STEP: Rank the N tasks of " T " in order of decreasing CPU occupancy rate $p_1(i,j)$.

(3) STEP1: While (Lc_1 < 90%) | |(Lc_2 < 90%) | |(Lc_3< 90%) | |(Lc_4 <90%)
// reminder " Lcj " is the total load of " core j "
for (i=0 ; i<N ; i++) {
if (p1 (T_{N-i})> 90 %) { store T_{N-i} on new list " T_S1 " ; N1 ++ }
else { assign T_{N-i} to the least loaded core under option 1 } }
store the remaining tasks on the list "T_S1" // N1 is the cardinality of T_S1

(4) STEP2: While (Lc_1 < 90%) | |(Lc_2 < 90%) | |(Lc_3 < 90%) | |(Lc_4 < 90%)
for (i=0 ; i<N1 ; i++) {
split T_{N1-i} between the least loaded core and the FPGA under option 2 }
store the remaining tasks on the list "T_S2" // N2 is the cardinality of T_S2

(5) STEP3: store the remaining N2 tasks on the FPGA under option 3

Figure 6: Algorithm MLPT

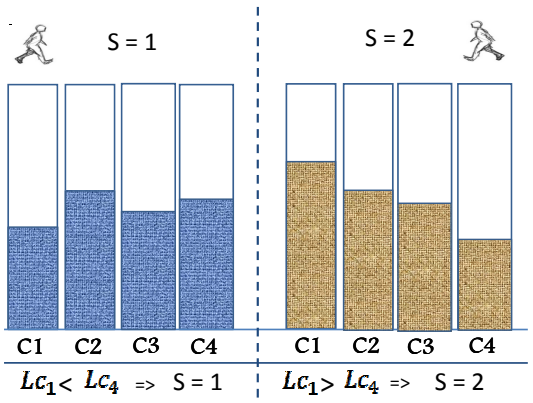

Figure 7: The Check-greedy Heuristic

It is important to note that these heuristics use the difference "Lc_1 - Lc_4 " value to guide the heuristic search. Moreover, let S be a binary variable defined as

$$\begin{cases} S = 1 \; if \; \text{Lc}_1 \rangle \; Lc_4 \\ S = 2 \; if \; \text{Lc}_4 \rangle \; Lc_1 \end{cases} \tag{9}$$

- STEP1:

  First we put all the tasks which have a CPU occupation rate larger than 90% in a list called STEP1; clearly, these tasks cannot be assigned to option 1. We will use the "Largest Load First" rule to assign tasks in the CPU cores which require the preprocessing step mentioned before. Then the first set of four tasks will be allocated to cores C1 to C4, the second set of four tasks will be allocated to cores C4 to C1. After the eight first tasks our strategy depends on the value of S. If S = 1 we assign a set of four tasks to cores C1 to C4; otherwise (S = 2) we assign a set of four tasks in the CPU cores from C4 to C1 as described Figure 7. In each iteration if we cannot assign a certain task to a certain core, we will try to do it with the next middle core in this

(1) Initialize: " T " the list of tasks of project P1

(2) Preprocessing STEP: Rank the N tasks of " T " in order of decreasing CPU occupancy rate μ_1 (1.1) .

(3) STEP1:
- Store all the tasks of load above 90% on a new list "TS1"
- Assign one task to each core
- While (Lc_1< 90%) | |(Lc_2< 90%) | | … | |(Lc_4 <90%)
  if (S=1) { Assign the next 4 tasks respectively to cores [C1, C2, C3, C4]}
  if (S=2) { Assign the next 4 tasks respectively to cores [C4, C3, C2, C1]}
- Store the remaining tasks on the list "TS1"

(4) STEP2: While (Lc_1< 90%) | |(Lc_2< 90%) | | … | |(Lc_4< 90%)
if (S=1) {split the next 4 tasks respectively between one of the cores [C1, C2, C3, C4] and the FPGA under option 2 }
if (S=1) {split the next 4 tasks respectively between one of the cores [C4, C3, C2, C1] and the FPGA under option 2 }
- Store the remaining tasks on the list "TS2" // N2 is the cardinality of "TS2"

(5) STEP3: store the remaining N2 tasks on the FPGA under option 3.

Figure 8: The Check-greedy Algorithm

sequence. For instance, if S=1 and one cannot assign task 13 to core C1, one tries successively with the middle cores C2 and C3; if still without success task 13 will be stored in TS1. One continues with task 14, and tries to assign it to core 4. The iterations continue until all the tasks are either allocated to the CPU cores or stored in the list TS1.

- STEP2:

  In the second STEP we switch to option 2 but retain the same scheduling strategy. Indeed the list stored in TS1 is already ordered in a decreasing way, then we will choose the sense of allocation by evaluating S (equal to 1 or 2). If it is possible to use option 2, we split the task between a core and the FPGA; otherwise we store the task in the list TS2.

- STEP3:

  In the third step we switch to option 3, the strategy then is to allocate all the tasks stored in the list TS2 to the FPGA.

5 The Dynamic mapping

By *run-time mapping* we mean that the decisions are made during the execution of the application. For example in avionic industry a change in environmental parameters (weather, geographical factors, etc.) will generate changes of the application inputs. Hence run-time mapping is suitable for realizing applications that involve dynamic reconfiguration. Moreover, the duration of an avionic test can last up to 20 hours or more, and the onset of an overload in such an application will generate just the failure of the test and all the extra costs that may ensue. All these elements make the runtime mapping necessary in order to anticipate the onset of overloads and minimize losses.

In order to assign a new arrival task we proceed as in the static mapping. This results, for example, with the MLPT-Rule Heuristic on assigning this task to the least loaded core taking into account the initial load.

In case of an overload of a task in a specific core, there are two possible scenarios:

- The first scenario, as described in Fig. 9, is an overload occurring in the least loaded core. In this case, we transfer the task which causes the overload to the FPGA.

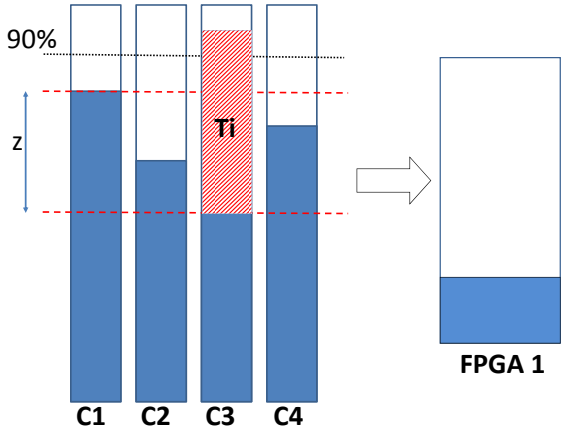

Figure 9: An overload occurs in the least loaded core, scenario 1.

- The second scenario described in Figure 10 involves an overload not occurring in the least loaded core. In order to study this scenario, we suppose that task Ti in core C4 is the cause of the overload. We denote by L3, the load of the core least loaded (Core C3 in this example), and we denote by L4 and Lo4 respectively the initial load and the overload of the core overloaded, here C4. Finally, and for illustrative purposes, we assume that the initial load of task " Ti " is 20%. Then we will proceed as follows :
  If "L3 + Lo4 + 20% $\leq$ 90%" we assign task "Ti" to core "C3". Else we assign "Ti" to the FPGA component.

6 Experimental and simulation results

6.1 Experimental environment setup

In order to compare the performances of the MLPT Heuristic and the exact method, we used a machine equipped with a 3GHz XEON 5160 processor and 16 GB RAM Memory. To solve the integer linear program described in Section 4.1, we used the IBM ILOG CPLEX optimization tool. We assigned the following values to the coefficients of the objective function: $\mu 1 = 50$; $\mu 2 = 100$; $\mu 3 = 200$; $\mu 4 = 400$. In order to

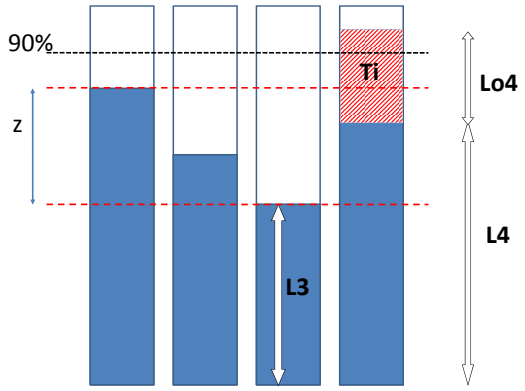

Figure 10: An overload does not occur in the least loaded core, scenario 2.

evaluate the performance of our model, we assigned different projects to the heterogeneous architecture containing CardC = 4 cores and CardF = 1 FPGA. Each project contains n tasks, with n varying from 5 to 16 tasks.

Task	Option	Occupancy		Task	Option	Occupancy	
		J= (1) ... (4)	J=5			J= (1) ... (4)	J=5
T1	O1	0,321	0,000	T9	O1	0,955	0,000
	O2	0,161	0,080		O2	0,478	0,239
	O3	0,000	0,107		O3	0,000	0,318
T2	O1	0,232	0,000	T10	O1	0,239	0,000
	O2	0,116	0,058		O2	0,120	0,060
	O3	0,000	0,077		O3	0,000	0,080
T3	O1	0,122	0,000	T11	O1	0,321	0,000
	O2	0,061	0,031		O2	0,161	0,080
	O3	0,000	0,041		O3	0,000	0,107
T4	O1	0,955	0,000	T12	O1	0,955	0,000
	O2	0,478	0,239		O2	0,478	0,239
	O3	0,000	0,318		O3	0,000	0,318
T5	O1	0,239	0,000	T13	O1	0,239	0,000
	O2	0,120	0,060		O2	0,120	0,060
	O3	0,000	0,080		O3	0,000	0,080
T6	O1	0,122	0,000	T14	O1	0,321	0,000
	O2	0,061	0,031		O2	0,161	0,080
	O3	0,000	0,041		O3	0,000	0,107
T7	O1	0,232	0,000	T15	O1	0,122	0,000
	O2	0,116	0,058		O2	0,061	0,031
	O3	0,000	0,077		O3	0,000	0,041
T8	O1	0,321	0,000	T16	O1	0,232	0,000
	O2	0,161	0,080		O2	0,116	0,058
	O3	0,000	0,107		O3	0,000	0,077

Figure 11: Projects data

As described in the introduction of Section 4, each task could be assigned to the heterogeneous architecture according to one of three options. For this case study, we consider a proportionality between the used capacity ratio of a processor and the chosen option as follows: let p_1(i,j) be the capacity ratio of CPU processor j (j ∈ C) that is occupied by task i under option 1. Then for option 2 we consider:

$$\begin{cases} p_2(i,j)/j \in C = (1/2) * p_1(i,j) \\ p_2(i,5) = (1/2) * p_2(i,j)/j \in C \end{cases} \quad (10)$$

Finally for option 3, we consider

$$p_3(i,j) = (1/3) * p_1(i,j) \quad (11)$$

6.2 MLPT heuristic VS Exact method

Figure 12 and Figure 13 compare the exact method with the MLPT heuristic. They show the initial mapping time and the z parameter depending on the number of tasks. We observe that for projects containing a small number of tasks, both methods are very fast while we lost in optimality for the z parameter. But once we start allocating projects with a higher number of tasks the z parameter is almost the same and decreases in the both methods, becoming very small when the load is high. For the execution time, it remains less than 0.032 seconds with the MLPT heuristic, while it increases very quickly with the exact method (1.68 sec for 16 tasks), indeed, the problem studied is NP-complete. As shown in Figure 12, even though the trends of the red and blue curves are negative, there is a great deal variation; indeed, the workload of the tasks assigned influences the z parameter. Finally, the variations of the gray curve representing execution time for the exact method, are caused by whether or not we can minimize the z parameter according to the possibilities of load-balancing.

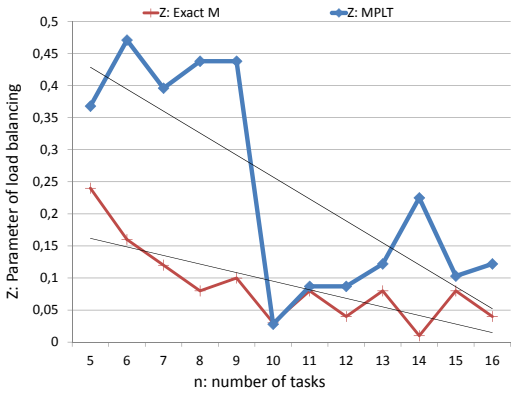

Figure 12: Z parameter: "exact method" VS "MLPT Heuristic"

6.3 The Check-greedy Heuristic VS The MLPT Heuristic

In order to compare the performances of the Greedy heuristic and the MLPT Heuristic, we used the same data as in the comparison between the MLPT Heuristic and the exact method.

Figure 14 and Figure 15 compare the MLPT Heuristic with the Check-greedy Heuristic. They show the initial mapping time and the z parameter depending on

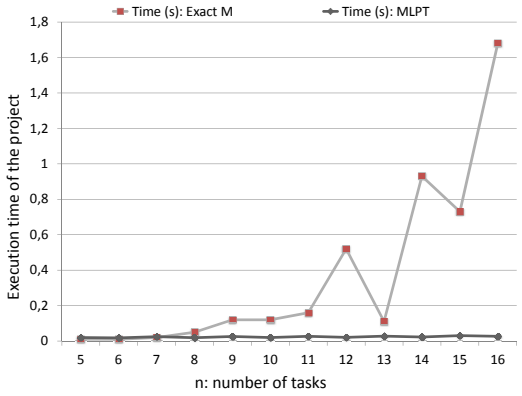

Figure 13: Comparison "exact method" VS "MLPT Heuristic"

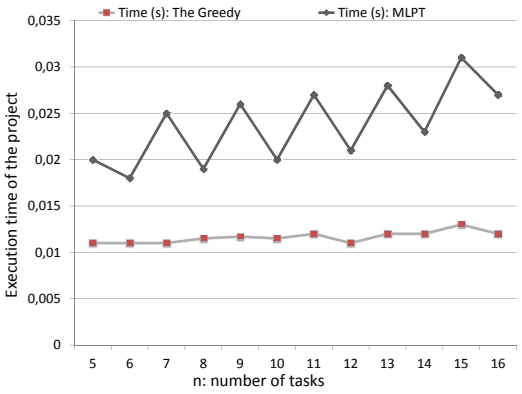

Figure 15: Comparison "MLPT Heuristic" VS "The Check-greedy Heuristic"

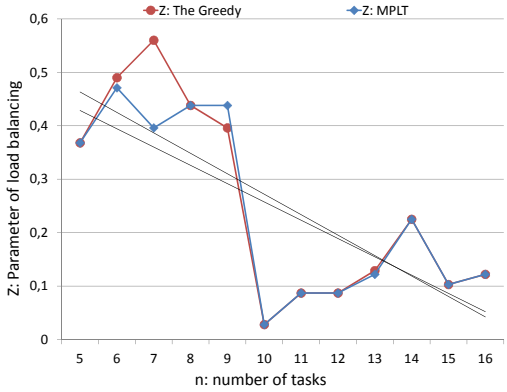

Figure 14: Z parameter: "MLPT Heuristic" VS "The Check-greedy Heuristic"

the number of tasks. We observe in Figure 15 that the Check-greedy Heuristic is faster than the MLPT rule heuristic. Indeed, the execution time never exceeds 0,015 seconds with Check-greedy Heuristic while it is never below 0,015 with the MLPT Heuristic. The two curves representing the Z parameters for both methods are almost identical. Which is logical given the distribution of data we took in our case study. Indeed with the panel of data studied, applying the MLPT rule Heuristic amounts to cover the cores from C1 to C4 then from C4 to C1, assigning each time a task to a core from the list established in the preprocessing step, which is the Greedy, by definition of the latter as explored in this research. Note that we avoided to put the exact method on the comparison "MLPT VS Check-greedy" in order to see the difference between the two heuristics in a smaller scale.

7 Conclusion and perspectives

The matching problem into the heterogeneous architecture CPU/FPGA is NP-complete. Indeed the research effort in this study shows that it is impossible to use the exact method as a solution for the static initial mapping because the execution time will increase exponentially according to the number of tasks, while the MPLT-Rule Heuristic offers a good compromise between efficiency and optimality. Further, the greedy heuristic offers a good performance but, as noted in Section 6.3, optimality depends on the data; indeed, if there are breaks in the panel data of models, it could affect the optimality of the z parameter. In future work, we look to improve the Greedy heuristic in order to obtain a better compromise between efficiency and the optimality of the solution, dealing with the case of breaks in the panel data. Furthermore in order to enrich this work, we will include the communications times between the models as a direct parameter, which will permit to study the problem in terms of minimization of the makespan. Then the extra costs in the objective function due to the choice of options 2 and 3 will be replaced by the minimization of the makespan.

Concerning the dynamic mapping which is briefly discussed in this paper, the solution adopted is very fast but we can explore in future works the use of option 2, as well as some heuristics similar to those adopted in the static part in order to improve the optimality of our solution without losing much in performance, which is essential in the run-time phase.

REFERENCES

Bora Ucar, Cevdet Aykanat, Kamer Kaya and Murat Ikinci, 2005. Task assignment in heterogeneous computing systems. *Journal of Parallel and Distributed Computing (January 2006)*, Vol. 66, No. 1., pp. 32-46.

Christos Koulamas and George J. Kyparisis, 2007. An improved delayed-start LPT algorithm for a partition problem on two identical parallel machines. *European Journal of Operational Re-*

search (June 2007), Vol. 187, pp. 660-666.

Christos Koulamas and George J. Kyparisis, 2008. A modified LPT algorithm for the two uniform parallel machine makespan minimization problem. *European Journal of Operational Research (July 2009)*, Vol. 196, pp. 61-68.

Emmanuel Jeannot, Erik Saule and Denis Trystram, 2011. Optimizing performance and reliability on heterogeneous parallel systems: Approximation algorithms and heuristics. *Journal of Parallel and Distributed Computing (February 2012)*, Vol. 72, pp. 268â280.

George Afonso, Rabie Ben Atitallah, Alexandre Loyer, Nicolas Belanger, Martial Rubio and Jean-Luc Dekeyser, 2011. A prototyping environment for high performance reconfigurable computing. *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, Montpellier, France.

Hans-ulrich Heiss, 1992. Mapping tasks to processors at run-time. *International Symposium on Computer and Information Sciences.*

Hyunok Oh and Soonhoi Ha, 1996. A static Scheduling Heuristic for Heterogenous Processors. *EURO-PAR 96 PARALLEL PROCESSING, Lecture Notes in Computer Science (1996)*, Vol. 1124/1996, pp. 573-577.

Jong-Kook Kim, Sameer Shivle, Howard Jay Siegel, Anthony A.Maciejewski, Tracy D. Braunb, Myron Schneider, Sonja Tideman, Ramakrishna Chitta, Raheleh B.Dilmaghani, Rohit Joshi, Aditya Kaul, Ashish Sharma, Siddhartha Sripadab, Praveen Vangari and Siva Sankar Yellampalli, 2007. Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment. *Journal of Parallel and Distributed Computing (February 2007)*, Vol. 67, pp. 154-169.

Mahmood Fazlali, Mojtaba Sabeghi, Ali Zakerolhosseini and Koen Bertels, 2010. Efficient task scheduling for runtime reconfigurable systems. *Journal of Systems Architecture (November 2010)*, Vol. 56, pp. 623-632.

Muthucumaru Maheswaran and Howard Jay Siegel, 1998. A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems. *In Seventh Heterogeneous Computing Workshop. IEEE Computer*, pp. 57-69.

Oscar H. Ibarra and Chul E. Kim, 1977. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *Journal of the Association for Computing Machinery (April 1977)*, Vol. 24, pp. 280-289.

Ping Luo, Kevin Lu and Zhongzhi Shi, 2007. A revisit of fast greedy heuristics for mapping a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing (June 2007)*, Vol. 67, pp. 695-714.

R. L. Graham, 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics.*

S. Mounir Alaoui, O. Frieder and T. El-Ghazawi, 1999. A Parallel Genetic Algorithm for task mapping on parallel machines *Proceedings of the 11 IPPS/SPDP 99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pp. 201-209.

Shoukat Ali, Jong-Kook Kim, Howard Jay Siegel, Anthony A. Maciejewski, Yang Yu, Shriram B. Gundala, Sethavidth Gertphol and Viktor Prasanna, 2007. Greedy Heuristics for Resource Allocation in Dynamic Distributed Real-Time Heterogeneous Computing Systems. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Vol. 2, pp. 519-530.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to algorithms.

Thomas L. Casavant, Jon and G. Kuhl, 1988. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering (February 1988)*, Vol. 14, pp. 141-154.

Tracy D. Braun, Howard Jay Siegel and Noah Beck, 2001. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing (June 2001)*, Vol. 61, pp. 810–837.

Wun-Hwa Chen and Chin-Shien Lin, 1998. A hybrid heuristic to solve a task allocation problem. *Computers Operations Research (March 2000)*, Vol. 27, pp. 287-303.