A LOW COMPLEX SCHEDULING ALGORITHM FOR MULTI-PROCESSOR SYSTEM-ON-CHIP

Nicolas VENTROUX, Frédéric BLANC
Department or Laboratory
CEA-List DRT/DTSI/SARC
Image and Embedded Computers Laboratory
91191 Gif-Sur-Yvette, FRANCE
email: nicolas.ventroux@cea.fr

Dominique LAVENIER
IRISA / CNRS
Campus de Beaulieu
35042 Rennes cedex, FRANCE
email: lavenier@irisa.fr

Abstract

Multi-Processor System-on-Chip (MPSoC) represents today the main trend for future architectural designs. Nonetheless, the scheduling of tasks on these distributed systems is a major problem since it has a central impact on global performances. This problem is known to be NP-complete and only approximate methods can be used. In the past, to approach optimal results, many heuristics have been proposed. But their complexity continue to increase, without considering efficient HW implementations. The novel scheduling policy, introduced in this paper, finds an interesting trade off between performance and complexity. Our list scheduling heuristic, called LLD, can near-optimally compute non-malleable tasks on multiple processing elements to minimize the schedule length with a low complexity. The comparison study achieved with already proposed algorithms shows that the LLD scheduling algorithm significantly overcomes the previous approaches in terms of processing element occupation as well as overall execution time.

1 Introduction

The emergence of new media applications demands a steady increase in flexibility and efficiency. Typical applications, such as MPEG players, are usually computationally intensive, preventing them from being implemented on general-purpose processors. To achieve better performances, designers take an interest in a System-on-Chip (SoC) paradigm composed of multiple computation resources with a high efficiency network. This new trend in architecture design is named Multi-Processor SoC (MPSoC). Moreover, the execution of applications on such a multiprocessing system requires scheduling the computation between a set of processing elements, which can be either programmable processors or reconfigurable units.

Any application can be represented by a directed acyclic graph $G = (T, E)$, where T is a set of T_i tasks and E is a set of precedence constraints between tasks. Therefore, a task T_j can be scheduled only if all precedent tasks have completed their execution. Furthermore, since the scheduling of tasks on multiple resources is known to be NP-complete [7], a scheduling heuristic must be chosen.

The features of processing elements and tasks take an important part in the choice of the scheduling policy. In this paper, we assume that the number of PE and the duration of each task is known at compile time. If tasks can be executed on a dynamically variable number of processing elements (*malleable tasks*), the maximum completion time (*makespan*) can be decreased and a better PE occupation rate can be insured [6, 10, 19]. Nevertheless, we do not consider this scheduling feature, because malleable tasks need a new compilation or a multi-compiled code for each task, since the number of computation hosts is undefined at the compilation time. Therefore, it can hardly be implemented in a hardware task scheduling.

This paper considers the problem of generating a schedule for a set of *n* independent and non-malleable parallel tasks on a multiprocessor system consisting of *P* identical processing elements. A simple extension of this algorithm can be use to manage multiple heterogeneous processing elements. The aim of this scheduling is to find a non-preemptive schedule that minimizes the makespan. Tasks must be dispatched on one or several identical PE for computation. In addition, no dispatching overheads are considered. Our algorithm is called LLD (Level-by-level and Largest-task-first scheduling with Dynamic-resource-occupation). If this work is partially introduced in [13] with a more constrainted approach, our main novel paradigm remains in our processor allocation strategy, which considerably improves scheduling performances.

The issue of interconnection networks, shared resources or multiple synchronizations are not taken into account in this paper. We will only consider independent tasks without any interprocessing contention by using a hardware component. This manages all these constraints before the delivering of tasks. This architecture dedicated to the control and named *RAC* is presented in [20]. Moreover, we favor contiguous processor allocation for a same task implementation. It sounds more realistic for a hardware design that shared resources are close for better performances. The algorithm introduced in this paper could also be used with distant processor allocation, if the net-

work topology is adapted. Moreover, we do not consider only systems composed of $P = 2^m$ processing elements, with m $\in \mathbb{N}$, since it is more practical for a multiprocessing platform. But this neither causes constraints on the scheduling technique proposed in this paper nor reduces its performances.

This paper is organized into four sections. In the next section a taxonomy of task-scheduling algorithms is provided. In section 3, the LLD algorithm is detailed and in section 4, a comparison analysis with other already proposed algorithms is presented. Finally, section 5 concludes this paper.

2 Previous Work

The problem of non-malleable task assignment has been widely studied. Solutions can be classified into several different categories such as guided random search, clustering, duplication-based or list-scheduling algorithms.

Genetic algorithms (GA) are the most extensively investigated guided random search methods for task scheduling. They are expected to reach good performances, but their execution time and their hardware complexity are significantly higher than the other alternatives [21, 22]. Moreover, their results never reach more than 10% over classical list-scheduling techniques [17].

Conversely, clustering algorithms are two-phase methods of scheduling. Before task scheduling, a task clustering determines the optimal number of PE on which to schedule tasks according to their granularities. The generated clusters are then merged in order to be executed on a fixed number of PE [14]. These methods have good scheduling properties, but finding a clustering of a task graph to minimize the overall execution time is difficult and expensive. In addition, duplication-based algorithms can inherently produce optimal solutions but cannot be implemented due to their high complexity [15].

The simplicity and the rapidity of list-scheduling algorithms make them well-adapted for simple hardware implementations. Even if their results may be less efficient in simulation due to lack of physical considerations, favoring simple and fast scheduling prevents from spending time to schedule tasks and therefore decreases the makespan. Firstly, an ordered list of tasks is constructed according to a predetermined policy (*Longest/Shortest-Task-First*, *First/Last-In-First-Out*, etc.). Finally, tasks are selected in the order and scheduled to one or more PE.

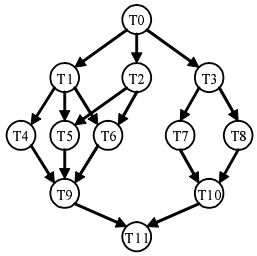
Since our problem is quite similar to two-dimensional bin-packing, the literature is composed of many different heuristics dedicated to particular scheduling features. Common simplifying assumptions include availability of unlimited numbers of processors, uniform task execution, no precedence constrained, non-contiguous allocation, one processor per task, etc. [11, 12]. These last years, more and more complex heuristics have been developed to obtain better approximations, without considering possible and efficient hardware implementations.

For instance, Blazewicz et al. investigate the problem of finding exact solutions in the case where all the tasks have the same execution time [4]. Some of these heuristics, not far from our study, reach good performance ratios but require complex algorithms [2]. Jansen et al. consider the scheduling of *n* independent tasks to minimize the maximum completion time [9]. They assume that each task can be executed with only one PE, and propose a fully polynomial approximation. They also envisage minimizing both the makespan and a global cost incured by each task. Even if the algorithm complexity is low, the proposed solution remains difficult to be efficiently implemented in a hardware solution. In [1], different sorted rules are presented like the *Greatest number of immediate successors first* or the *Maximum of the sum of the processing times of all successors first*. Nevertheless, if scheduling tasks in considering precedence constraints is well-adapted for acyclic graphs, these algorithms increase the execution time. In [18], Topcuoglu et al. propose two algorithms : the Heterogeneous Earliest-Finish-Time (HEFT) and the Critical-Path-On-a-Processor (CPOP) algorithm. In the HEFT algorithm, the task priority depends on the remaining time of its execution in order to minimize the completion time of each PE. The CPOP algorithm allows us to minimize a critical cost associated with each task. These approaches could not be used in an asynchronous circuit since timing is unknown. Moreover, the current time execution of each task is unnecessary for non-real-time task scheduling. In addition, a dynamic scheduling increases the energy consumption due to constant updating of memories.

It is important to bring appropriate solutions even if they turn out longer makespan. In addition, the scheduling complexity must be distributed with the allocation process in order to dispatch the algorithm complexity. First, a simple sorting like *Longest-Task-First* (LTF) or *LArgest-Task-First* (LATF), can be chosen. Furthermore, Belhale et al. give an approximation algorithm with polynomial running time for the multiprocessor scheduling problem, under the additional constraint that work done by tasks is non-decreasing in the number of processors (LATF) [3]. In addition, Kequin Li et al. make a probabilistic analysis for the LATF scheduling and show that the suboptimality bounds on the makespan is not worse than 2 [13]. According to these results, we decided to use the LATF scheduling for its quite efficient average-case performance ratio. In addition, a large variety of different heuristics have been proposed, but few PE allotment techniques exist. Yet, their efficiency takes an important part in task scheduling [8]. The main proposition consists in parallelizing tasks (malleable tasks), but dynamic resource allocation can also bring significant improvements. In addition to a static scheduling, a dynamic and on-line resource allocation can dynamically assign tasks according to the current availability of system resources.

In the following section, we present our static scheduling algorithm. This heuristic is merged with a novel dynamic resource allocation strategy.

Task	Number of PE $\delta(T_i)$	Computation time (μs) $\tau(T_i)$
T0	1	100
T1	4	100
T2	1	200
T3	2	600
T4	1	200
T5	1	400
T6	1	300
T7	3	400
T8	4	100
T9	1	300
T10	4	100
T11	1	100



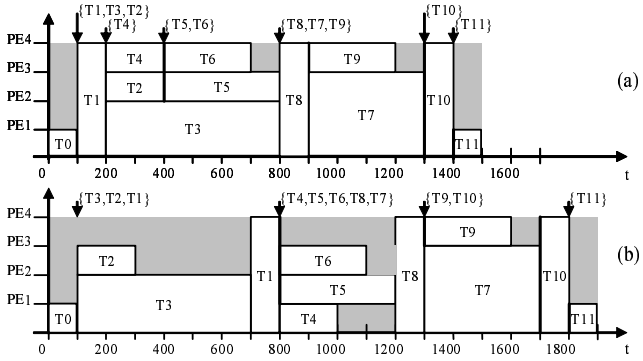


Figure 1. A directed acyclic graph scheduling

3 The LLD Scheduling

The *RAC* is a hardware component supporting the implementation of Control Flow Graph (CFG). It is a self-reconfigurable and asynchronous architecture, which supports a high level of control parallelism. The management of exclusion and synchronization mechanisms is done automatically and the *RAC* only delivers independent tasks without inter-processing contentions [20]. Due to these interesting features, we associate this component to our simple heuristic named LLD to carry out an efficient scheduling.

In a directed acyclic graph, precedence constraints must be respected. For instance in figure 1, tasks T_1, T_2 and T_3 can start only after the execution of task T_0. This set of tasks constitute a *level* of execution. The structure of the *RAC* can explicitely provide these *levels* of execution in respecting synchronizations between tasks, accesses to shared memories or even disponibilities of the communication network. As soon as all the tasks of the current level have been assigned, the *RAC* will build another *level* with every task whose precedence tasks have been executed. Consequently, the execution of an application graph consists in processing *level-by-level* independent tasks.

Let $\delta(T_i)$ be the number of PE needed to execute the task T_i and $\tau(T_i)$ its execution time. First, our scheduling algorithm sorts out tasks of an execution level according to a non-increasing order of PE needed for their execution. Consider a sorted list C of n independent tasks such that $C = \{T_1, T_2, ...T_n\}$. Then, we assume that $\delta(T_1) \geq \delta(T_2) ... \geq \delta(T_n)$. Moreover, if the number of PE for the execution of a task is equal to an other task, they are sorted out into non-increasing order of their computation time. For instance, if $\delta(T_i) = \delta(T_j)$ and $\tau(T_i) \geq \tau(T_j)$ then T_i is to be executed before T_j.

Once the list is sorted, the allocation of tasks begins. The number of free PE is checked. According to this number, the most priority task is loaded on allocated resources. Then, the number of free PE is updated and this process is repeated until the assignation of all the tasks of the current level is done.

This algorithm, called Level-by-level Largest-task-first with Dynamic-resource-occupation (LLD), has an $O(n \times P)$ complexity, but can be efficiently used as a hardware scheduling mechanism.

In figure 1, a scheduling example is presented. The scheduling named *(a)* is the result obtained after the LLD algorithm and the scheduling *(b)* after the First-Come-first-Serve (FCFS) algorithm. After the execution of the first task T_0, the list C of ready tasks, which must be scheduled, is composed of T_1, T_2, T_3. Then, the LLD algorithm will schedule the task of this list C by their non-increasing order of PE needed, i.e. T_1, T_3 and T_2. At the end of the assignation of these tasks, a new list is created composed of ready tasks (here T_4 because T_1 has finished its execution). Since there is free resources, T_4 is assigned immediately. Finally, the scheduling of all the application is done in following the same rules. In the FCFS scheduling, tasks are scheduled by their date of arrival without any policy, and can generate a less efficient scheduling as represented Figure 1 (b).

In the next section, a detailed comparison study is presented in order to compare our scheduling algorithm with other list-scheduling heuristics. Again, the acceleration and the PE occupation rate are considered. Actually, these two criteria mainly represent the quality of a scheduling related to the maximum completion time minimization.

4 Implementation and Evaluation

Amongst many different heuristics, we choose to compare our algorithm with others, which have an equivalent hardware implementation complexity and computation time. These algorithms are the *First-Come-First-Serve* (FCFS), the *Longest/Shortest-Task-First* (LTF-STF), the *LATF*, the *SLEAT*, the *Next-Fit-Decreasing-Height* (NFDH), the *First-Fit-Decreasing-Height* (FFDH) and the *First-Fit-Increasing-Height* (FFIH) scheduling algorithms. We have also used our dynamic resource allocation method with some of these schemes in order to underline its efficiency. These algorithms are marked with the *DRO* extension. However, the performance comparison would not be fair unless we also compare these heuristics to the upper

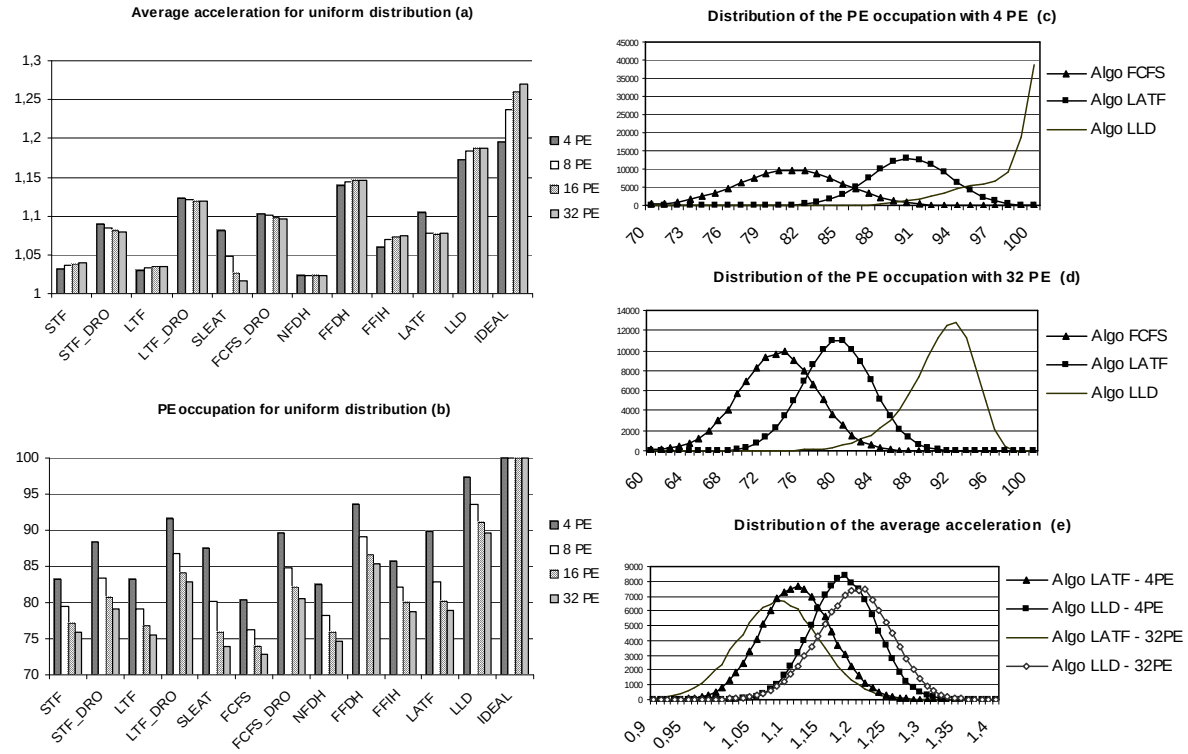


Figure 2. Results and comparisons

limit that we could reach. For this reason, we have implemented an *IDEAL* and an *OPTIMAL* algorithm.

FCFS scheduling This algorithm schedules tasks without sorting them according to their arrival date. It constitutes our scheduling reference.

LTF/STF/LATF scheduling The LTF sequentially packs the remaining tasks having the longest running time into the schedule. Let w be the makespan of a given task system and w^* be the makespan of its optimal scheduling, then $w \leq 2w^* + h_{max}$, where h_{max} is the maximum execution time [19]. The STF algorithm consists in sorting tasks by their non-decreasing time of computation. Moreover, the LATF algorithm sorts out tasks according to a non-increasing order of PE needed for their execution.

SLEAT scheduling First, this algorithm schedules iteratively all the tasks requiring more than $P/2$ PE. Then, the remaining tasks are sorted out into a non-increasing time execution order, and two groups of $P/2$ contiguous processors are carried out. Finally, tasks are scheduled either on the first or on the second processor group depending on which has the current smaller makespan. It has a suboptimality bound of 2.5 [16].

NFDH/FFDH/FFIH scheduling These algorithms were developed by Coffman et al. [5]. The NFDH algorithm packs the next task, left justified, on the current level if its fits. Otherwise the level is *closed*, a new current level is created (as a vertical line on the top of the longest task packed on the current level), and the task is packed on it, left justified. On the contrary, the FFDH algorithm does not close any level. It was shown that $FFDH(C) \leq 1.7 \times OPT(C) + 1$. Finally, the FFIH algorithm is very similar to the FFDH algorithm. The only difference remains in the request-list, which is sorted in a non-decreasing order. This decreases the scheduling efficiency but reduces the average waiting time for task execution, which can be interesting for interactive usage.

IDEAL/OPTIMAL scheduling All these algorithms respect precedence constraints imposed by application graphs. The IDEAL algorithm schedules tasks in compacting them as much as possible without respecting the parallelism and the number of PE imposed by each task. Actually, this algorithm does not consider HW feasibility, but it represents the upper limit that any scheduling heuristic could reach. The OPTIMAL algorithm is not a heuristic. This algorithm tests all scheduling possibilities and keeps the solution that owns the shortest makespan. Of course, since this problem is NP-complete, this algorithm needs long computation and only few cases have been evaluated.

In order to evaluate the quality of each algorithm, a large amount of independent task lists have been generated with a uniform probability distribution to form execution *levels*. We assume that $\tau(T_i) \in [1..100]$, $\delta(T_i) \in [1..P]$ with $P = \{4, 8, 16, 32\}$ and the number of independent task of each scheduling list is about 30. A uniform distribution analysis can demonstrate the generic efficiency of an algorithm. Moreover, a power of two of PE is chosen to

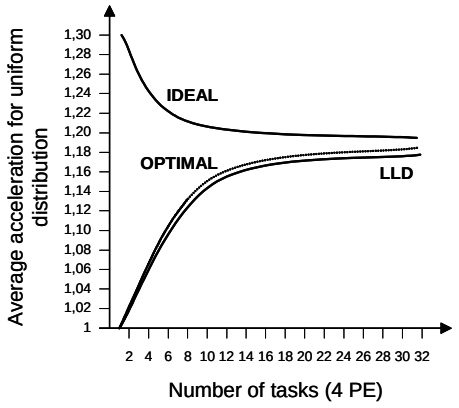


Figure 3. Average acceleration comparison with the IDEAL and OPTIMAL algorithms

carry out a comparison with other limited algorithms. The acceleration $\lambda(H)$ of a heuristic H, in comparison with no particular scheduling policy (FCFS) for the scheduling of a list C is:

$$\lambda(H) = \frac{FCFS(C) - H(C)}{FCFS(C)}.$$

Thus for instance, the acceleration $\lambda(LLD)$ of the example depicted Figure 1 approximates 21% and the PE occupation rate is about 86.7% compared to 68.4% with a FCFS scheduling.

According to figure 2 (a,b), the quality of the STF or the LTF scheduling are quite similar and better than the NFDH. In addition, it is important to note that the LATF algorithm has effectively the best result compared to STF, LTF or SLEAT heuristics. It was expected that the PE occupation rate would be better since the aim of this scheduling is to favor the resource occupation, but its average acceleration shows that its behaviour remains particularly efficient whatever the number of PE is. The SLEAT algorithm reaches good performances but they decrease with the number of PE. In fact, this heuristic has an important drawback since each task, which uses more than $P/2$ PE, is sequentially executed without considering free resources. For instance, a task using 17 PE on a 32 PE platform will allocate only 53% of its resources during all the task execution. This algorithm can reach good performances if most of tasks use less than $P/2$ PE or if few PE are envisaged.

Finally, the LLD algorithm introduced in this paper reaches the best results. It surpasses every tested heuristics in maximum, minimum and average acceleration as well as PE occupation. Moreover, scheduling results obtained with the IDEAL algorithm are finally not far from our results. Thus, any heuristics with better performances than the LLD would have at most only an increase of 7%. Moreover, the necessary effort to reach this low improvement increases drastically the complexity of a HW implementation. In addition, the increase of performance would be lost by the latence generated by the complexity of these algorithms or the time spent by HW computations. In figure 2 (c,d,e), the frequency of results obtained by the LLD are shown. As expected, gaussian curves are generally obtained, but the LLD algorithm with 4 PE has an exponential behaviour. The performance of our dynamic allocation improves significantly the PE occupation or the average acceleration of the LLD compared to the LATF. Furthermore, only algorithms with a specific assignation policy can reach good performances. It has a significant impact on the quality of a task scheduling.

Results obtained by the IDEAL algorithm present an unreachable limit. Even the best heuristic could not obtain these results, since the IDEAL algorithm does not take into account HW constraints. On the other hand, the OPTIMAL algorithm represents a reachable limit that best heuristics can approach and according to the figure 3, we can estimate that its results are between those obtained by the IDEAL and the LLD algorithms. Consequently, we can consider that possible improvements brought by any other heuristics for our problem would have a low interest.

Nonetheless, the performances of our algorithm decrease when little value of $\delta(T_i)$ are favored. Actually, if we consider for instance an exponential distribution with a mean $\mu = P/4$ (figure 4), we demonstrate that even if the LLD still keeps a high level of quality, when tasks need less parallelism for their execution, the efficiency of our heuristic decreases. However, compared to its average result the LLD scheduling algorithm remains particularly effective and constant.

5 Summary

In this paper, we have introduced a new algorithm, called LLD, for scheduling application graphs onto a distributed processing system, such as MPSoC. Based on a comparison study using a large set of random task lists, the LLD heuristic significantly outperformed the other algorithms with an equivalent complexity, in term of acceleration compared to the FCFS algorithm (about 20% of improvements) and PE occupation. The scheduling of precedence constraints parallel task with this algorithm is particularly efficient and constant whatever the application is. Its dynamic allocation mechanism improves the PE occupation and consequently reduces the maximum completion time. Besides, it can be

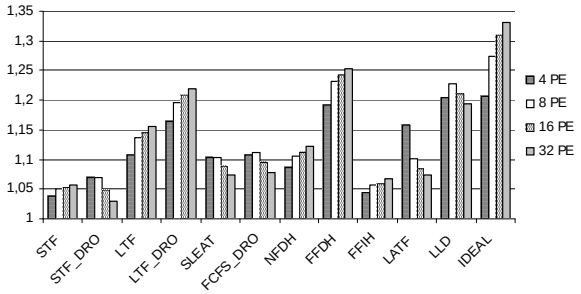


Figure 4. Average acceleration for exponential distribution ($\mu = P/4$)

easily added to other heuristics and efficiently implemented for distributed control dispatching. Moreover, the management of heterogeneous resources just consists in considering for each task only specific free resources. Finally, the main advantage of this list scheduling is its low complexity, that makes it very attractive for a possible HW implementation.

References

[1] J. L. Baer. A Survey of Some Theoretical Aspects of Multiprocessing. *ACM Computing Surveys*, 5(1):31–80, Mar. 1973.

[2] B. S. Baker, D. J. Brown, and H. P. Katseff. A 5/4 Algorithm for Two-Dimensional Packing. *Journal of Algorithms*, 2(4):348–368, June 1981.

[3] K. Belkhale and P. Banerjee. Approximate Scheduling Algorithms for the Partitionable Independent Task Scheduling Problem. In *the 1990 International Conference of Parallel Processing*, Aug. 1990.

[4] J. Blazewicz, M. Drabowski, and Weglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, 35(5):389–393, May 1986.

[5] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM J. Computing*, 9(4):808–826, Nov. 1980.

[6] P.-F. Dutot and D. Trystram. Scheduling on hierarchical clusters using Malleable Tasks. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, Crete, Greece, July 2001.

[7] M. R. Garey and D. S. Jonhson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W.H. Freeman, June 1979.

[8] T. Ibaraki and N. Katoh. *Resource allocation Problems: Algorithmic Approaches*. The MIT Press, Cambridge, Massachusetts, 1988.

[9] K. Jansen and L. Porkolab. Improved Approximation Schemes for Scheduling Unrelated Parallel Machines. In *Annual ACM Symposium on Theory of Computing (STOC)*, Atlanta, USA, May 1999.

[10] K. Jansen and L. Porkolab. Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks. *Algorithmica*, 32(3):507–520, 2002.

[11] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, Dec. 1999.

[12] C. Y. Lee, L. Lei, and M. Pinedo. Current trends in deterministic scheduling. *Annals of Operations Research*, 70:1–41, 1997.

[13] K. Li and Y. Pan. Probabilistic Analysis of Scheduling Precedence Constrained Parallel Tasks on Multicomputers with Contiguous Processor Allocation. *IEEE Transactions on Computer*, 49(10):1021–1030, Oct. 2000.

[14] J. Liou and M. A. Palis. A Comparison of General Approaches to Multiprocessor Scheduling. In *International Parallel Processing Symposium (IPPS)*, Geneva, Switzerland, Apr. 1997.

[15] S. Ranaweera and D. Agrawal. A Scalable Task Duplication Based Scheduling Algorithm for Heterogeneous Systems. In *International Conference on Parallel Processing*, pages 383–390, Toronto, Canada, Aug. 2000.

[16] D. Sleator. A 2.5 Times Optimal Algorithm for Packing in Two Dimensions. *Information Processing Letters*, 10(1):37–40, Feb. 1980.

[17] T.D. Braun et al. A Comparison Study of Static Mapping Heuristics for a Class of Meta-tasks on Heterogeneous Computing Systems. In *Heterogeneous Computing Workshop*, San Juan, Puerto Rico, Apr. 1999.

[18] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar. 2002.

[19] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, San Diego, USA, June 1992.

[20] N. Ventroux, S. Chevobbe, F. Blanc, and T. Collette. An Auto-Adaptative Reconfigurable Architecture for the Control. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 72–87, Beijing, China, Sept. 2004. Springer-Verlag LNCS 3189.

[21] L. Wang, H. Siegel, V. Roychowdhury, and A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):1–15, Nov. 1997.

[22] F. Xu. Integration, Simulation and Implementation of a hardware-based Genetic Optimizer to Adjust Smart Antenna Receiver. Master's thesis, Friedrich-Alexander-University Erlangen-Nuremberg and Fraunhofer IIS Erlangen, Aug. 2003.