Università degli Studi di Bologna
DEIS

Allocation and Scheduling for MPSoCs via decomposition and no-good generation

Luca Benini Davide Bertozzi Alessio Guerri
Michela Milano

March 24, 2005

DEIS Technical Report no. DEIS-LIA-001-05 *LIA Series no. 73*

Allocation and Scheduling for MPSoCs via decomposition and no-good generation

Luca Benini [1] Davide Bertozzi [1] Alessio Guerri [1] Michela Milano [1]

[1] *DEIS*
Università di Bologna
V.le Risorgimento, 2
40136 Bologna, Italy
{lbenini,dbertozzi,aguerri,mmilano}@deis.unibo.it

March 24, 2005

Abstract. This paper describes an efficient, complete approach for solving a complex allocation and scheduling problem for Multi-Processor System-on-Chip (MPSoC). Given a throughput constraint for a target application characterized as a task graph annotated with computation, communication and storage requirements, we compute an allocation and schedule which minimizes communication cost first, and then the makespan given the minimal communication cost. Our approach is based on problem decomposition where the allocation is solved through an Integer Programming solver, while the scheduling through a Constraint Programming solver. The two solvers are interleaved and their interaction regulated by no-good generation. Experimental results show significant speedups w.r.t. pure IP and CP solution strategies.

Contents

1 Introduction

This paper proposes a decomposition approach to the allocation and scheduling of a multi-task application on a multi-processor system-on-chip (MPSoCs) [**?**]. This is currently one of the most critical problems in electronic design automation for Very-Large Scale Integrated (VLSI) circuits. With the limits of chip integration reaching beyond one billion of elementary devices, current advanced integrated hardware platforms for high-end consumer application (e.g. multimedia-enabled phones) contain multiple processors and memories, as well as complex on-chip interconnects. The hardware resources in these MPSoCs need to be optimally allocated and scheduled under tight throughput constraints when executing a target software workload (e.g. a video decoder).

In a typical embedded system design scenario, the platform always runs the same application. Thus, extensive analysis and optimization can be performed at design time. In particular, allocation and scheduling can be pre-computed statically. The target application is pre-characterized and abstracted as a task graph. The task graph is annotated with computation (e.g., execution time), communication (e.g., number of bits to be communicated between tasks), storage (e.g., size of data and instruction memory required to execute the task) requirements. After solving the allocation and scheduling problem, the application can be loaded onto the target hardware platform, together with system software which orchestrates its execution according to the pre-computed solution.

The problem of allocating and scheduling tasks is NP-complete and the approach we propose is based on an hybrid Constraint Programming (CP) and Integer Programming (IP) approach. The solution scheme is based on a problem decomposition which interleaves (i) allocation of tasks to processors and required memory slot to storage devices and (ii) scheduling tasks in time. Since the two sub-problems are not independent, their interaction is regulated by no-good generation. The process is proved to converge producing the optimal solution. The method is inherited by Operations Research and is known with the name of Benders Decomposition [3]. This method partitions the problem variables in two sets x and y, assigns trial values to x by solving a master problem (containing only variables in x), so as to define a subproblem containing only the variables belonging to y. If the solution of the subproblem reveals that the trial values are not acceptable, a no-good is generated and new trial values are assigned according to the no-good. This method converges, hopefully after few steps, by providing the optimal solution.

In this paper, we present the problem and the proposed solution motivating our choice, and we give experimental evidence that our approach outperforms both the stand-alone IP and CP approaches.

2 Problem description

Recent advances in very large scale integration (VLSI) of digital electronic circuits have made it possible to integrate more than a billion of elementary devices onto a single chip, thereby enabling the development of low-power, low-cost, high-performance single-chip multiprocessors. These devices, called multi-processor systems-on-chip (MPSoCs), are finding widespread application in embedded systems (such as cellular phones, automotive control engines, etc.) where they are employed as special-purpose computing engines. In other words, once deployed in field, they always run the same application, in a well-characterized context. It is therefore possible to spend a large amount of time for finding an optimal allocation and scheduling off-line and then deploy it on the field. For this reason, many researchers in digital design automation have explored complete approaches for allocating and scheduling pre-characterized workloads on MPSoCs [?], instead of using on-line, dynamic (sub-optimal) schedulers [5, 4].

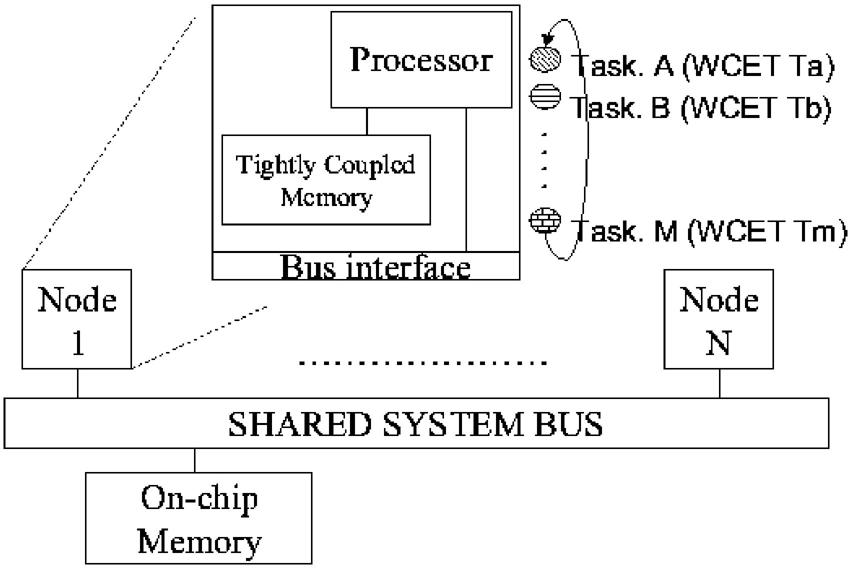

Figure 1. *Single chip multi-processor architecture.*

The resources we need to allocate and schedule in MPSoCs are heterogenous: they include processing (micro-processors, DSPs, hardware accelerators), storage (i.e. on-chip memories) and communication (i.e. on-chip buses and I/Os) elements. The multi-processor system we consider consists of a pre-defined number of distributed computation nodes, as depicted in Figure 1. All nodes are assumed to be homogeneous and made by a processing core and by a tightly coupled local memory. This latter is a low access cost scratchpad memory, which is commonly used both as hardware extension to support message passing and as a storage means for computation data and processor instructions which are frequently accessed. Data storage onto the scratchpad memory is directly managed by the application, and not automatically in hardware as is the case for processor caches.

Unfortunately, the scratchpad memory is of limited size, therefore data in excess must be stored externally in a remote on-chip memory, accessible via the bus. The bus for state-of-the-art MPSoCs is a shared communication resource, and serialization of bus access requests of the processors (the bus masters) is carried out by a centralized arbitration mechanism. The bus is re-arbitrated on a transaction basis (e.g., after single read/write transfers, or bursts of accesses of pre-defined length), based on several policies (fixed priority, round-robin, latency-driven, etc.). Modelling bus allocation at such a fine granularity would make the problem overly complex, therefore a more abstract bus model was devised, thus also bridging the gap with our high level task models, which express communication requirements of the tasks in terms of their required bus bandwidth for the duration of their execution. We will discuss this point in detail in section 4.

Whenever predictable performance is needed for applications, it is important to avoid high levels of congestion on the bus, since this makes completion time of bus transactions much less predictable. Moreover, under a low congestion regime, performance of state-of-the-art shared busses scales almost in the same way as that of advanced busses with topology and communication protocol enhancements. Finally, bus modelling turns out to be simpler under these working conditions. (e.g., additive models). Communication cost is therefore critical for determining overall system performance, and will be minimized in our task allocation framework.

Based on our methodology, the target application running on top of the hardware platform is pre-characterized and abstracted as a task graph, with specification of computation, storage and communication requirements. More in detail, the worst case execution time (WCET) is specified for each task and plays a critical role whenever application real time constraints (expressed here in terms of minimum required throughput) are to be met. In fact, tasks are scheduled on each processor based on a time-wheel. The sum of the WCETs of the tasks for one iteration of the time wheel must not exceed time period RT (i.e., the minimum task scheduling period ensuring that throughput constraints are met), which is the same for each processor since the minimum throughput is an application (not single processor) requirement.

Each task also has 3 kinds of memory requirements:

- **Program Data**: storage locations are required for computation data and for processor instructions. They can be allocated either on the local scratchpad memory or on the remote on-chip memory.

- **Internal State**: when needed, an internal state of the task can be stored either locally or remotely.

- **Communication queues**: the task needs queues to transmit and receive messages to/from other tasks, eventually mapped on different processors. In the class of MPSoCs we are considering, such queues should be allocated only on local memories, in order to implement an efficient inter-processor communication mechanism.

Finally, the communication requirements of each task are automatically determined depending on the size of communication data and on the physical location of computation data in scratchpad or remote memory.

The methodology proposed in this paper has been applied to a synthetic signal processing pipeline. Functional pipelining is widely used in the domain of multimedia applications. Task parameters have been derived from a real video graphics pipeline processing pixels of a digital image. The proposed allocation and scheduling techniques can be easily extended to all applications using pipelining as workload allocation policy, and aim at providing system designers with an automated methodology to come up with effective solutions and cut down on design time.

3 Motivation for the approach

The problem described in the previous section has a very interesting structure. As a whole, the problem is a scheduling problem with alternative resources. In fact, each task should be allocated to one of the processors (Node i in Figure 1). In addition, each memory slot required for processing the task should be allocated to a memory device. Clearly, tasks should be scheduled in time subject to real time constraints, precedence constraints, and capacity constraints on all unary and cumulative resources. However, on a different perspective, the problem decomposes into two problems:

- the allocation of tasks to processors and the memory slots required by each task to the proper memory device;
- a scheduling problem with static resource allocation.

The objective function of the overall problem is the minimization of communication cost. This function involves only variables of the first problem. In particular, we have a communication cost each time two communicating tasks are allocated on different processors, and each time a memory slot is allocated on a remote memory device. The scheduling has a secondary objective which is the minimization of makespan given an optimal allocation.

The allocation problem is difficult to solve with Constraint Programming (CP). CP has a naive method for solving optimization problems: each time a solution is found, an additional constraint is added stating that each successive solution should be better than the best one found so far. If the objective function is strongly linked to decision variables, CP can be effective, otherwise it is hopeless to use CP to find the optimal solution. In case the objective function is related to a single variable, like for makespan in scheduling problems, CP works quite well. However, if the objective function is a sum of cost variables, CP is able to prune only few values, deep in the search tree since the connection between the objective function and the problem decision variables is weak. If the objective function relates to pairs of assignments the situation is even worse. This is the case of our application where the objective function relates alternative resources to couples of tasks. In fact, if communicating tasks are allocated to different processors, they should communicate on the bus and the cost increases. IP, instead, is extremely good to cope with these problems.

On the contrary, Integer Programming (IP) is weak in coping with time. Scheduling problems require to assign tasks to time slots, and each slot should be represented by an integer variable, and the number of variables increases enormously. CP, instead, is very effective to cope with time constraints.

Therefore, the first problem could be solved with IP effectively, while for the second CP is the technique of choice. The question is now: how do these problems interact?

We solve them separately, the allocation problem first (called master problem), and the scheduling problem (called subproblem) later. The master is solved to optimality and its solution passed to the subproblem solver. If the solution is feasible, then the overall problem is solved to optimality. If, instead, the master solution cannot be completed by the subproblem solver, a no-good is generated and added to the model of the master problem, roughly stating that the solution passed should not be recomputed again (it becomes infeasible), and a new optimal solution is found for the master problem respecting the (set of) no-good(s) generated so far. Being the allocation problem solver an IP solver, the no-good has the form of a linear constraint.

Now let us note the following: the assignment problem allocates tasks to processors, and memory requirements to storage devices minimizing communication costs. However, since real time constraints have not been taken into account yet, the assignment module tends to pack all tasks in the minimal number of processors. In other words, the only constraint that prevents to allocate all tasks to a single processors is the limited capacity of the tightly coupled memory devices. However, these trivial assignments do not consider throughput constraints which make them most probably infeasible for the overall problem. To avoid the generation of these (trivial) assignments, we should add to the master problem model a relaxation of the subproblem. In particular, we should state in the master problem that the sum of the durations of tasks allocated to a single processor does not exceed the real time requirement. In this case, the allocation is far more similar to the optimal one for the problem at hand.

A similar method is known in Operations Research as Benders Decomposition [3], where the overall problem can be decomposed in two parts connected by some variables. Indeed, in this method, the subproblem should be easy. In our case, the subproblem is an NP-complete problem, but CP is a very effective method to solve it.

We will show that this method is extremely effective compared to the approaches considering the problem as a whole.

4 Model definition

As described in section 3, the problem we are facing can be split into 2 problems: the allocation master problem and the scheduling sub-problem. In the following we will describe the models used to solve these problems.

4.1 Allocation problem model

The allocation problem is the problem of allocating n tasks to m processors, such that every cumulative resource constraint on the local memories is statically satisfied. We assume the remote on-chip memory to be of unlimited size. The problem objective function is the minimization of the amount of data transferred on the bus. Since in this problem we do not consider temporal constraints, we can model the problem as an IP model.

In the IP model we consider 4 decision variables:

- T_{ij}, taking value 1 if task i executes on processor j, 0 otherwise,
- Y_{ij}, taking value 1 if task i allocates the program data on the scratchpad memory of processor j, 0 otherwise,
- Z_{ij}, taking value 1 if task i allocates the internal state on the scratchpad memory of processor j, 0 otherwise,
- X_{ij}, taking value 1 if task i executes on processor j and task $i+1$ does not, 0 otherwise.

The linear constraints we introduced in the model are:

$$\sum_{j=1}^{m} T_{ij} = 1, \forall i \in 1 \dots n \tag{1}$$

$$X_{ij} = |(T_{ij} - T_{i+1j})|, \forall i \in 1 \dots n, \forall j \in 1 \dots m \tag{2}$$

Constraints (1) state that each process can execute only on a processor, while constraints (2) state that X_{ij} can be equal to 1 iff $T_{ij} \neq T_{i+1j}$, that is, iff task i and task $i+1$ execute on different processors. Constraints (2), for efficiency reasons, can be rewritten as:

$$T_{ij} + T_{i+1j} + X_{ij} - 2K_{ij} = 0 \ , \forall i \ , \forall j \tag{3}$$

where K_{ij} are integer binary variables.

We also add to the problem the constraints stating that $T_{ij} = 0 \Rightarrow Y_{ij} = 0, Z_{ij} = 0$, and, for each group of consecutive tasks whose execution times sum exceeds the real time requirement, we introduce in the model a constraint preventing the solver to allocate all the tasks in the group, and in particular the first and the last, to the same processor. This is a relaxation of the subproblem, added to the master problem to prevent the generation of trivially infeasible solutions for the overall problem, as described in Section 3.

The objective function is the minimization of the total amount of data transferred on the bus for each pipeline. This amount consists of 3 contributions: when a task allocates its program data in the remote memory, it reads these data throughout the execution time; when a task allocates the internal state in the remote memory, it reads these data at the beginning of its execution and updates them at the end; if 2 consecutive tasks execute on different processors, their communication messages must be transferred through the bus from the communication queue of one processor to the other. Using the decision variables described above, we have a contribution respectively when: $T_{ij} = 1, Y_{ij} = 0$; $T_{ij} = 1, Z_{ij} = 0$; $X_{ij} = 1$. Therefore, the objective function is to minimize:

$$Mem_i(T_{ij} - Y_{ij}) + State_i(T_{ij} - Z_{ij}) + (Data_i X_{ij})/2, \tag{4}$$

where Mem_i, $State_i$ and $Data_i$ are the amount of data used by task i to store respectively the program data, the internal state and the communication queue.

4.2 Scheduling problem model

Once tasks have been allocated to the processors, we need to schedule process execution. Since we are considering a pipeline of tasks, we need to analyze the system behavior at working rate, that is when all processes are running or ready to run. To do that, we need to consider several instantiations of the same process; to achieve a working rate configuration, the number of repetitions of each task must be at least equal to the number of tasks n; in fact, after n iterations, the pipeline is at working rate. So, to solve the scheduling problem, we must consider at least n^2 tasks (n iterations for each process), see Figure 2.

In the scheduling problem model, for each task $Task_{ij}$ we introduce a variable A_{ij}, ($i = [0 \dots n-1], j = [0 \dots n-1]$), representing the computation activity of the task. A_{ij} is the j-th iteration of the i-th process. Once the allocation problem is solved, we statically know if a task needs to use the bus to communicate with another task, or to read/write computation data and internal state in the remote memory. In particular, each activity A_{ij} must read the communication queue from the activity A_{i-1j}, or from the pipeline input if $i = 0$. To schedule these phases, we introduce in the model the activities In_{ij}. If a process requires an internal state, the state must be read before the execution and wrote after the execution: we therefore introduce in the model the activities RS_{ij} and WS_{ij} for each task i requiring an internal state. The duration of these activities depends on whether the data are stored in the local or the remote memory (data transfer trough the bus needs more time than the transfer of the same amount of data using the local memory) but, after the allocation, these times can be statically calculated.

The precedence constraints among the activities are:

$$A_{i,j-1} \prec In_{ij} \;,\; \forall\, i,j \tag{5}$$
$$In_{ij} \prec A_{ij} \;,\; \forall\, i,j \tag{6}$$
$$A_{i-1,j} \prec In_{ij} \;,\; \forall\, i,j \tag{7}$$
$$RS_{ij} \preceq A_{ij} \;,\; \forall\, i,j \tag{8}$$
$$A_{ij} \preceq WS_{ij} \;,\; \forall\, i,j \tag{9}$$
$$In_{i+1,j-1} \prec A_{ij} \;,\; \forall\, i,j \tag{10}$$
$$A_{i,j-1} \prec A_{ij} \;,\; \forall\, i,j \tag{11}$$
$$Start(A_{ij}) - Start(A_{i,j-1}) \leq RT \;,\; \forall\, i,j \tag{12}$$

where the symbol $\prec$ means that the process on the right can execute only after the execution of the activity on the left.

Constraints (5) state that each task iteration can start reading the communication queue only after the end of its previous iteration. Constraints (6) state that each task can start only when it has read the communication queue, while constraints (7) state that each ask can read the data in the communication queue only when the previous task has generated them. Constraints (8) and (9) state that each task must read the internal state just before the execution and write it just after. Constraints (10) state that each ask can execute only if the previous iteration of the following task has read the input data; in other words, it can start only when the memory allocated to the process to store the communication queue has been freed. Constraints (11) state that the iterations of each task must execute in order. Finally, constraints (12) are the real time requirement, whose relaxation is used in the allocation problem model.

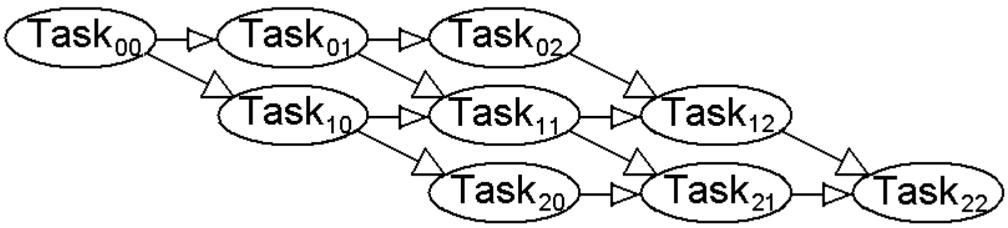

Figure 2. *Precedence constraints among the activities*

Figure 2 depicts the precedence constraints among the tasks. Each task $Task_{ij}$ represents the activity A_{ij} eventually preceded by the internal state reading activity RS_{ij}, and input data reading activity In_{ij}, and eventually followed by the internal state writing activity WS_{ij}.

Each processor is modelled as a unary resource, that is a resource with capacity one. As far as the bus is concerned, we make a simplification. In fact, we consider the bus as an additive resource, in the sense that more activities can share the resource until the resource maximum availability is reached. We model the communication requirements of each task (the amount of computation data stored in the remote memory) in such a way that they are spread over the WCET of the related task, thus consuming a fraction of the overall bus bandwidth for the duration of the task. The sum of these fractions must not exceed the maximum bus bandwidth. Each task starts by reading its input data and its internal state, and completes by updating/writing its internal state to memory. Here, for modelling purposes we assume that such reads can rely on $1/N$-th of the total bus bandwidth, where N is the number of processors. We therefore derive an approximation of the duration of such transfers (and of their relative bus bandwidth occupancy) in order to derive a model of bus activity.

The resource requirement for the scheduling problem states that each activity A_{ij}, In_{ij},

RS_{ij} and WS_{ij} requires the processor where process i is allocated by the master problem and, eventually, the bus.

5 Experimental results

To validate the strength of our approach, we now compare the results obtained using this model (MP-Asm in the following) with results obtained using only a CP or IP model to solve the overall problem. Actually, since the first experiments showed that both CP and IP were not able to find a solution, except for the easiest instances, within 15 minutes, we simplified these models removing some variables and constraints. In CP, we fixed the activities execution time not considering the execution time variability due to remote memory accesses, therefore we do not consider the In_{ij}, RS_{ij} and WS_{ij} activities, including them statically in the activities A_{ij}. In IP, we do not consider all the variables and constraints involving the bus: we do not model the bus resource and we therefore suppose that each activity can access data whenever it is necessary.

We generate a large variety of problems, varying both the number of tasks and processors. All the results presented are the mean over a set of 10 problems for each task or processor number. All problems considered have a solution. Experiments were performed on a 2GHz Pentium 4 with 512 Mb RAM. We used ILOG CPLEX 8.1 and ILOG Solver 5.3 as solving tools.

In figures 3 and 4 we compare the algorithms search time for problems with a different number of, respectively, tasks and processors. Times are expressed in seconds and the y-axis has a logarithmic scale.

Although CP and IP deal with a simpler problem model, we can see that these algorithms are not comparable with MP-Asm, except when the number of tasks and processors is low; this is due to the fact that the problem instance is very easy to be solved, and MP-Asm loses time creating and solving two models, the allocation and the scheduling. As soon as the number of tasks and/or processors grows, IP and CP performances worsen and their search times become order of magnitude higher w.r.t. MP-Asm. Furthermore, we considered in the figures only instances where the algorithms are able to find the optimal solution within 15 minutes, and, for problems with 6 tasks or 3 processors and more, IP and CP can find the solution only in the 50% or less of the cases.

6 Related work

There are a number of papers using Benders Decomposition in a CP setting. [16] proposes the branch and check framework using Benders Decomposition (BD). [8] embed BD in the CP environment ECLiPSe and show that it can be useful in practice. [9] applied Benders decomposition to minimum cost planning and scheduling problems; in this work the objective function involves only master problem variables, while the subproblem is simply a feasibil-

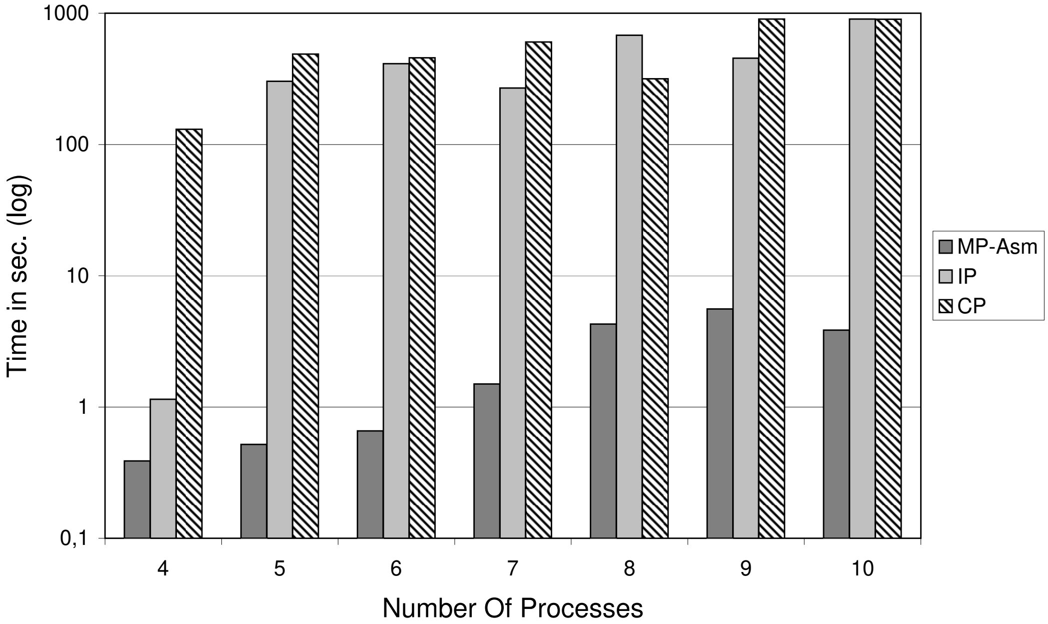

Figure 3. *Comparison between algorithms search times for different task number*

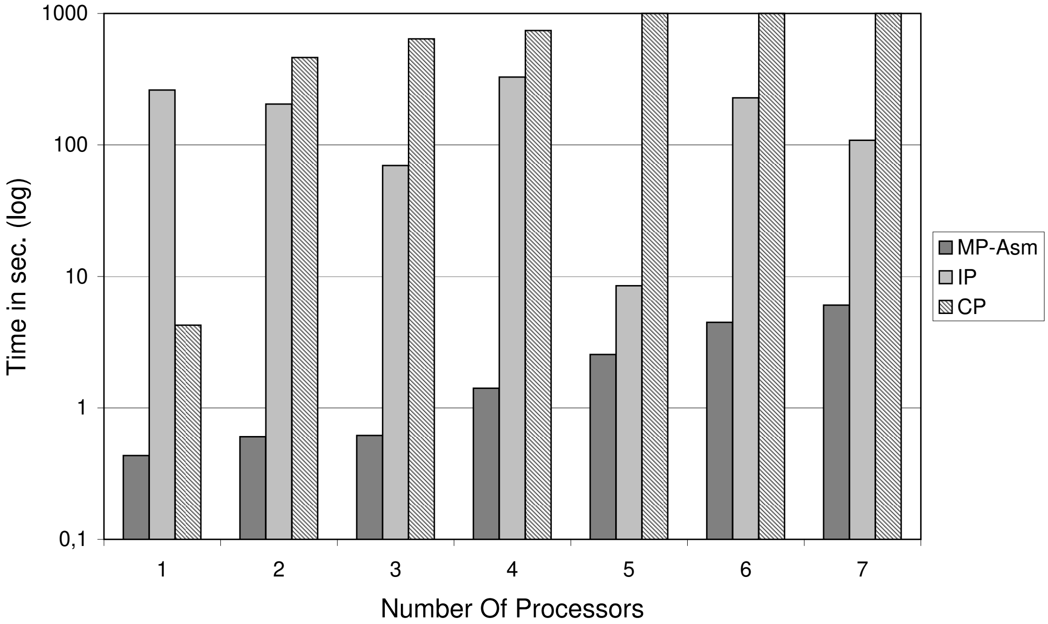

Figure 4. *Comparison between algorithms search times for different processor number*

ity problem. [11] used Benders decomposition for Planning and Scheduling problems; here the objective function involves both master problem and subproblem variables, so also the subproblem is an optimization problem. In the present work we consider as subproblem a scheduling problem where only the alternative resource constraints are fixed by the master problem. The subproblem is therefore NP-complete.

The synthesis of system architectures has been studied extensively in the past. Many scheduling problems have been modelled as integer linear programming problems, and addressed by means of IP solvers. An early example is represented by the SOS system, which used mixed integer linear programming technique (MILP) [15]. That framework considers processor nodes with local memory and connected through direct point-to-point channels. The algorithm does not consider real-time constraints. Partitioning with respect to timing constraints has been addressed in [14]. A MILP model that allows to determine a mapping optimizing a trade-off function between execution time, processor and communication cost is reported in [2]. In practice, optimal approaches are suitable only for small task graphs because of their time complexity. Heuristic approaches are by far the most widely used approaches. A comparative study of well-known heuristic search techniques (genetic algorithms, simulated annealing and tabu search) is reported in [1]. Unfortunately, busses are implicit in the architecture. [6] compare the use of SA and TS for partitioning a graph into hardware and software parts while trying to reduce communication and synchronization between parts. A better scalability of these algorithms for large real-time systems is introduce in [12]. Many heuristic scheduling algorithms are variants and extensions of list scheduling [7]. In general, scheduling tables list all schedules for different condition combinations in the task graph, and therefore not suitable for control-intensive applications. Another approach consists of employing CP techniques to schedule conditional task graphs. This is done in [13], wherein commercial constraint solvers are used to find the solution. A recent co-synthesis tool that takes into account the impact of different custom ASIC implementations of tasks on system performance and cost in the co-synthesis process is proposed in [17], and the corresponding allocation and scheduling algorithm is described in [18]. Finally, an overview of algorithms for scheduling pipelined circuits is presented in[10]. ILP formulations as well as heuristic algorithms are described.

7 Conclusion

In this paper, we have faced a challenging problem arising in the field of multi-processor systems-on-chip (MPSoCs). The structure of the problem suggests a decomposition approach based on the interaction of two problem solvers: one allocating tasks to alternative resources and memory requirement to storage devices; the second scheduling tasks subject to temporal and resource constraints. The first problem solver exploits mathematical programming techniques, while the second is based on CP. The interaction between these problem solvers is regulated by no-good generation.

We provide experimental evidence that our approach outperforms the one considering the problem as a whole and using a single technique (CP or IP) separately. The work in progress is aimed at generalizing the problem for introducing message queues on the shared memories so as to decouple the computation and communication through non blocking synchronization.

References

[1] J. Axelsson. Architecture synthesis and partitioning of real-time synthesis: a comparison of three heuristic search strategies. In *Procs. of the 5th Intern. Workshop on Hardware/Software Codesign (CODES/CASHE 1997)*, pages 161–166, Braunschweig, Germany, Mar. 1997. IEEE.

[2] A. Bender. Milp based task mapping for heterogeneous multiprocessor systems. In *EURO-DAC '96/EURO-VHDL '96: Procs. of the conference on European design automation*, pages 190–197, Geneva, Switzerland, Sept. 1996. IEEE.

[3] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

[4] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, 1999.

[5] D. A. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[6] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems*, 2(1):5–32, 1997.

[7] P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. pages 132–139, Paris, France, 1998.

[8] A. Eremin and M. Wallace. Hybrid benders decomposition algorithms in constraint logic programming. In *Procs. of the 7th Intern. Conference on Principles and Practice of Constraint Programming - CP 2001*, pages 1–15, Paphos, Cyprus, Nov. 2001. Springer.

[9] I. E. Grossmann and V. Jain. Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS Journal on Computing*, 13:258–276, 2001.

[10] G. De Micheli. *Synthesis and optimization of digital circuits*. McGraw Hill, 1994.

[11] J. N. Hooker. A hybrid method for planning and scheduling. In *Procs. of the 10th Intern. Conference on Principles and Practice of Constraint Programming - CP 2004*, pages 305–316, Toronto, Canada, Sept. 2004. Springer.

[12] S. Kodase, S. Wang, Z. Gu, and K. Shin. Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers. In *Procs. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003)*, pages 181–188, Toronto, Canada, May 2003. IEEE.

[13] K. Kuchcinski. Embedded system synthesis by timing constraint solving. *IEEE Transactions on CAD*, 13(5):537–551, 1994.

[14] C. Lee, M. Potkonjak, and W. Wolf. System-level synthesis of application-specific systems using A* search and generalized force-directed heuristics. pages 2–7, San Diego, California, Nov. 1996.

[15] S. Prakash and A. Parker. Sos: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16(4):338–351, 1992.

[16] E. S. Thorsteinsson. A hybrid framework integrating mixed integer programming and constraint programming. *Lecture Notes in Computer Science*, 2239:16–30, 2001.

[17] Y. Xie and W. Wolf. Co-synthesis with custom asic. In *Procs. of ASP-DAC 2000, Asia and South Pacific Design Automation Conference 2000*, pages 129–134, Yokohama, Japan, Jan. 2000. ACM.

[18] Y. Xie and W. Wolf. Allocation and scheduling of conditional task graph in hardware/software co-synthesis. In *2001 Design, Automation and Test in Europe Conference and Exposition (DATE 2001)*, pages 620–625, Munich, Germany, Mar. 2001. ACM.