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

Luca Benini[(1)], Davide Bertozzi[(2)], Alessio Guerri[(1)], and Michela Milano[(1)]

(1) DEIS, University of Bologna
V.le Risorgimento 2, 40136, Bologna, Italy
{lbenini, aguerri, mmilano}@deis.unibo.it

(2) Dipartimento di Ingegneria, University of Ferrara
V. Saragat 1, 41100, Ferrara, Italy
dbertozzi@ing.unife.it

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 speedups of orders of magnitude w.r.t. pure IP and CP solution strategies.

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) [1]. 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 and memories to MPSoCs is NP-complete. We propose here an hybrid Constraint Programming (CP) and Integer Programming (IP) approach. The solution scheme is based on 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. Eventually the process converges, producing the optimal solution. The method is inherited by Operations Research and it is known with the name of *Benders Decomposition* [2]. This method partitions the problem variables in two sets x and y, assigns trial values to x by solving the master problem (containing only variables in x) to optimality, 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. It is proved that this method converges, hopefully after few steps, by providing the optimal solution [2].

Benders Decomposition has been successfully applied in conjunction with Constraint Programming as we will extensively describe in section 6. For example, [3] and [4] face a similar problem using Benders Decomposition and found very promising results. Our main purpose in this paper is to show how a hybrid Constraint and Integer Programming approach can be used to solve a very complex optimization problem which has been traditionally approached with heuristic techniques or (for small instances) with complete Integer Programming approaches. We show that our method outperforms on one hand Integer Programming approaches which can be considered the state of the art complete approaches for this problem, and on the other hand Constraint Programming approaches that have been exploited much less frequently in this context.

2 Problem description

Advances in very large scale integration (VLSI) of digital electronic circuits have made it possible to develop multi-processor systems-on-chip (MPSoCs), which are finding widespread application in embedded systems (such as cellular phones, automotive control engines, etc.). Once deployed in field, these devices 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 [1], instead of using on-line, dynamic (sub-optimal) schedulers [5, 6].

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 scratch-

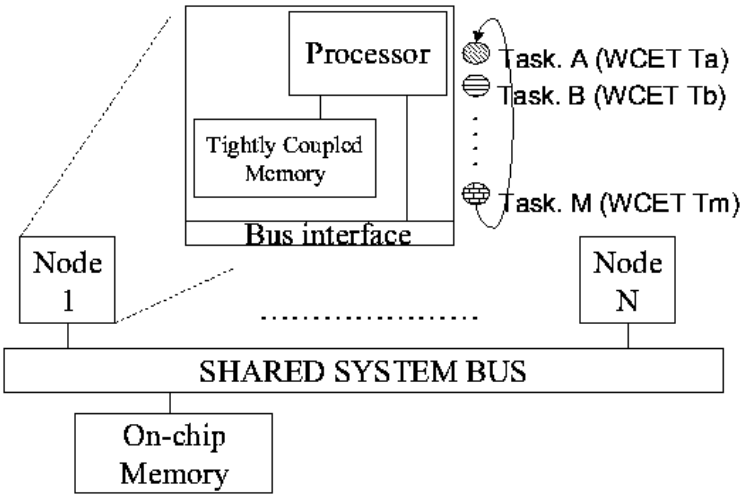

Fig. 1. Single chip multi-processor architecture.

pad memory is directly managed by the application, and not automatically in hardware as it 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 is 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.

The target application to be executed on top of the hardware platform is input to our methodology, and for this purpose it must be represented as a task graph. This latter consists of a graph pointing out the parallel structure of the program. The application workload is therefore partitioned into computation sub-units denoted as tasks, which

are the nodes of the graph. Graph edges connecting any two nodes indicate task dependencies. Computation, storage and communication requirements are annotated onto the graph. In detail, the worst case execution time (WCET) is specified for each node/task and plays a critical role whenever application real time constraints (expressed here in terms of minimum required throughput) are to be met. 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 node/task also has 3 kinds of associated 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, communication requirements of each task are automatically determined once computation data and internal state are physically allocated to scratchpad or remote memory, and obviously depend on the size of such data.

The methodology proposed in this paper has been applied to a task graph extracted from a real video graphics application processing pixels of a digital image. Many real-life signal processing applications are subject to tight throughput constraints, therefore leverage a pipelined workload allocation policy. As a consequence, the input graph to our methodology consists of a pipeline of processing tasks, and can be easily extended to all pipelined applications.

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, from 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. Once we have optimally allocated tasks to resources, we can minimize the schedule makespan.

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 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, data transfer on the bus (and thus the objective function increase) occurs when two communicating tasks are allocated to different processors. Integer Programming (IP), instead, is extremely good to cope with these problems.

On the contrary, IP is weaker than CP in coping with time. Scheduling problems require to assign tasks to time slots, each slot being represented by an integer variable. The number of variables increases enormously if the granularity of the timeline is fine.

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 are not taken into account by the allocation module, the solution obtained tends to pack all tasks into 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 realtime requirement. In this case, the allocation is far more similar to the optimal one for the problem at hand. The use of a relaxation in the master problem is well known and widely used in practice and helps in producing better solutions.

A similar method is known in Operations Research as Benders Decomposition [2], where the overall problem can be decomposed in two parts connected by some variables. Indeed, in this method, the subproblem should be easy.

In [3], for example, Logic-Based Benders Decomposition is used to solve an allocation and scheduling problem where precedence constraints among tasks assigned to different resources are not considered; in this case we have a set of independent subproblems, for each facility. In our case, we can have precedence constraints between tasks allocated to different facilities and the subproblem is therefore an NP-complete problem, but CP is a very effective method to solve it.

4 Model definition

As described in section 3, the problem we are facing can be split into the resource allocation master problem and the scheduling sub-problem.

4.1 Allocation problem model

We start from the task graph presented in section 2. Each task should be allocated to a processor. In addition it needs a given amount of memory to store data. Data can be allocated either in the local memory of the processor running the task or in the remote one except for communication queues that are always mapped locally. The allocation problem is the problem of allocating n tasks to m processors, such that the total amount of memory allocated to the tasks, for each processor, does not exceed the maximum available.

We assume the remote on-chip memory to be of unlimited size since it is able to meet the memory requirement of the application we are facing (small granularity program data). The problem objective function is the minimization of the amount of data transferred on the bus. We model the problem as an integer program and we consider four decision variables in the model:

- 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 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) are not linear (X_{ij} is the `exor` of T_{ij} and T_{i+1j}), thus we cannot use them in a IP model. If we consider that the sum $Xij + T_{ij} + T_{i+1j}$ must always equal either to 0 or 2, constraints (2) 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 that enforce the sum $T_{ij} + T_{i+1j} + X_{ij}$ to be equal either to 0 or 2.

We add to the problem the constraints stating that $T_{ij} = 0 \Rightarrow Y_{ij} = 0, Z_{ij} = 0$ meaning that if a processor j is not assigned to a task i neither its program data nor the internal state can be stored in the local memory of processor j.

As explained in section 3, in order to prevent the master problem solver to produce trivially infeasible solutions, we need to add to the master problem model a relaxation of the subproblem. For this purpose, for each set of consecutive tasks whose execution times sum exceeds the real time requirement (RT), we impose constraints preventing the solver to allocate all the tasks in the group to the same processor.

To generate this constraints, we find out all groups of consecutive tasks sum of whose execution times (Dur_i) exceeds RT. Constraints are the following:

$$\sum_{i \in S} Dur_i > RT \Rightarrow \sum_{i \in S} T_{ij} \leq |S| - 1 \; \forall j \tag{4}$$

The objective function is the minimization of the total amount of data transferred on the bus for each pipeline. This amount consists of three 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 two 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:

$$\sum_{i=1}^{n} \sum_{j=1}^{m} (mem_i(T_{ij} - Y_{ij}) + 2 \times state_i(T_{ij} - Z_{ij}) + (data_i X_{ij})/2) \tag{5}$$

where mem_i, $state_i$ and $data_i$ are coefficients representing 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 considered an activity A_{ij}, ($i = [0 \dots n-1]$, $j = [0 \dots n-1]$), representing the computation 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 consider the activities In_{ij}. If a process requires an internal state, the state must be read before the execution and written after the execution: we therefore consider 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 through the bus needs more time than the transfer of the same amount of data using the local memory) but, after the allocation, these durations can be statically calculated. These activity are introduced in the model using variables $Start\_A_{ij}$, $Start\_In_{ij}$, $Start\_RS_{ij}$ and $Start\_WS_{ij}$, representing the starting time of the corresponding activity. We also use the values Dur_A_{ij}, Dur_In_{ij}, Dur_RS_{ij} and Dur_WS_{ij} to represent the execution times of the corresponding activities.

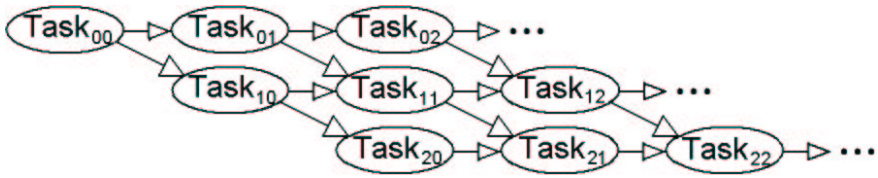

Fig. 2. Precedence constraints among the activities

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

The precedence constraints among the activities introduced in the model are:

$$A_{i,j-1} \prec In_{ij} \ , \ \forall \, i,j \tag{6}$$
$$In_{ij} \prec A_{ij} \ , \ \forall \, i,j \tag{7}$$
$$A_{i-1,j} \prec In_{ij} \ , \ \forall \, i,j \tag{8}$$
$$RS_{ij} \preceq A_{ij} \ , \ \forall \, i,j \tag{9}$$
$$A_{ij} \preceq WS_{ij} \ , \ \forall \, i,j \tag{10}$$
$$In_{i+1,j-1} \prec A_{ij} \ , \ \forall \, i,j \tag{11}$$
$$A_{i,j-1} \prec A_{ij} \ , \ \forall \, i,j \tag{12}$$

where the symbol $\prec$ means that the activity on the left should precede the activity on the right, and the symbol $\preceq$ means that the activity on the right must start as soon as the execution of the activity on the left ends: i.e., $In_{ij} \prec A_{ij}$ means $Start\_In_{ij} + Dur\_In_{ij} \leq Start\_A_{ij}$, and $RS_{ij} \preceq A_{ij}$ means $Start\_RS_{ij} + Dur\_RS_{ij} = Start\_A_{ij}$.

Constraints (6) state that each task iteration can start reading the communication queue only after the end of its previous iteration. Constraints (7) state that each task can start only when it has read the communication queue, while constraints (8) state that each ask can read the data in the communication queue only when the previous task has generated them. Constraints (9) and (10) state that each task must read the internal state just before the execution and write it just after. Constraints (11) state that each task 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 for storing the communication queue has been freed. Constraints (12) state that the iterations of each task must execute in order.

Furthermore, we introduced the real time requirement constraints (13), whose relaxation is used in the allocation problem model. Each task must execute at most each time period RT.

$$Start(A_{ij}) - Start(A_{i,j-1}) \leq RT \ , \ \forall \ i, j \tag{13}$$

Each processor is modelled as a unary resource, that is a resource with capacity one. As far as the bus is concerned, as explained in section 2, we make a simplification: a real bus is a unary resource but, if we model a bus as a unary resource, we should describe the problem at a finer grain with respect to the one we use, i.e., we have to model task execution using the clock cycle as unit of time. The resulting scheduling model would contain a huge number of variables. We therefore consider the bus as an additive resource, in the sense that more activities can share the bus resource using only a fraction of the total bandwidth available.

Figure 3 depicts this assumption. The leftmost figure represents the bus allocation in a real processor, where the bus is assigned to different tasks at different times. Each task, when owning the bus, uses its total bandwidth. The rightmost figure, instead, represents how we model the bus. The bus arbitration mechanism will then transform the bus allocation into the interleaving of fine granularity bus transactions on the real platform.

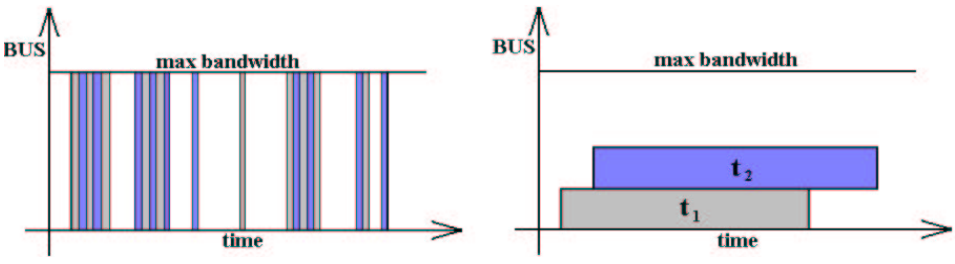

Fig. 3. Bus allocation in a real processor (left) and in our model (right)

In particular, to define the communication requirements of each task (the amount of computation data stored in the remote memory) we consider the amount of data they have to communicate and we spread it over its WCET. In this way we consume only a fraction of the overall bus bandwidth for the duration of the task. In the 2 graphs in figure 3 light grey and dark grey areas are equal.

When an allocation is provided, the minimal makespan schedule is computed if it exists. On the contrary, if no feasible schedule exists, we have to generate a no-good and pass it to the allocation module. The no-good should prevent the allocation to be the same of the previous iteration. Since the allocation module is an Integer Programming solver, the no-good should have the form of a linear constraint. In particular, we select all the resources that provoke a failure, e.g., either resources whose capacity is violated, or resources that lead to a violation of real time constraints. We call them *conflicting resources*, CR. Then, we impose that for each resource in $R \in CR$ the set of tasks ST_R allocated to R should not be reassigned to the same resource in the next iteration. For example if a conflicting resource R is a processor and ST_R the set of tasks previously allocated to it, the resulting no-good is:

$$\sum_{i \in ST_R} T_{iR} \leq |ST_R| - 1$$

In the same way, we have constraints for preventing failures in storage device.

These are the simplest kind of no-goods that can be added to the master problem since they state that the current solution must not be computed again. Even if they can be improved, as shown in [7], we will show in the next section that they are very effective.

5 Experimental results

To validate the strength of our approach, we now compare the results obtained using this model (**Hybrid** 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 approaches are 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 generated 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 4 and 5 we compare the algorithms search time for problems with a different number of tasks and processors respectively. Times are expressed in seconds and the y-axis has a logarithmic scale.

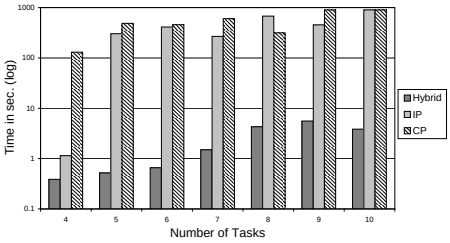

Fig. 4. Comparison between algorithms search times for different task number

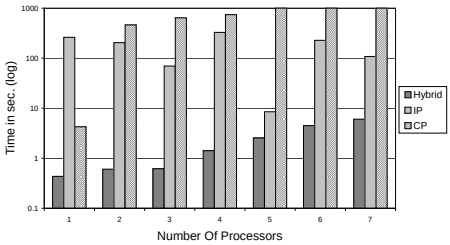

Fig. 5. Comparison between algorithms search times for different processor number

Although CP and IP deal with a simpler problem model, we can see that these algorithms are not comparable with Hybrid, 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 Hybrid 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 orders of magnitude higher w.r.t. Hybrid. 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. On the contrary, we can see that Hybrid search time scales up linearly (in the logarithmic scale) for all the case.

We also measured the number of times the solver iterates between the master and the sub-problem. We found that, due to the limited size of the local memories and to the relaxation of the sub-problem added to the master, the solver iterates 1 or 2 times. Removing the relaxation, it iterates up to 15 times. This result gives evidence that, in a Benders decomposition based approach, it is very important to introduce a relaxation of the sub-problem in the master, and that the relaxation we use is very effective.

6 Related work

The synthesis of distributed system architectures has been extensively studied in the past. The mapping and scheduling problems on multi-processor systems have been traditionally modelled as integer linear programming problems. An early example is represented by the SOS system, which used mixed integer linear programming (MILP) model [8]. SOS considers processor nodes with local memory, connected through direct point-to-point channels. The algorithm does not consider real-time constraints. Partitioning under timing constraints has been addressed in [9]. A MILP model that allows to determine a mapping optimizing a trade-off function between execution time, processor and communication cost is reported in [10].

Extensions of the IP formulation have also been used to account for memory allocation requirements, besides communication and computation ones. A hardware/software co-synthesis algorithm of distributed real-time systems that optimizes the memory hierarchy (caches) along with the rest of the architecture is reported in [11]. An integer

linear programming model is used in [12] to obtain an optimal distributed shared memory architecture minimizing the global cost to access shared data in the application, and the memory cost.

The above techniques lead to static allocations and schedules that are well suited for applications whose behaviour can be accurately predicted at design time, with minimum run-time fluctuations. This is the case of signal processing and multimedia applications. Pipelining is one common workload allocation policy for increasing throughput of such applications, and this explains why research efforts have been devoted to extending mapping and scheduling techniques to pipelined task graphs. An overview of these techniques is presented in [13]. IP formulations as well as heuristic algorithms have been traditionally employed. In [14] a retiming heuristic is used to implement pipelined scheduling, that optimizes the initiation interval, the number of pipeline stages and memory requirements of a particular design alternative. Pipelined execution of a set of periodic activities is also addressed in [15], for the case where tasks have deadlines larger than their periods. Palazzari et al. [16], focus on scheduling to sustain the throughput of a given periodic task set and to serve aperiodic requests associated with hard real-time constraints. Mapping of tasks to processors, pipelining of system specification and scheduling of each pipeline stage have been addressed in [17], aiming at satisfying throughput constraints at minimal hardware cost.

In general, even though IP is used as a convenient modelling formalism, there is consensus on the fact that pure IP formulations are suitable only for small problem instances (task graphs with a reduced number of nodes) because of their high computational cost. For this reason, heuristic approaches are widely used. A comparative study of well-known heuristic search techniques (genetic algorithms, simulated annealing and tabu search) is reported in [18]. Eles et al. [19] compare the use of simulated annealing and tabu search for partitioning a graph into hardware and software parts while trying to reduce communication and synchronization between parts. More scalable versions of these algorithms for large real-time systems are introduced in [20]. Many heuristic scheduling algorithms are variants and extensions of list scheduling [21].

Heuristic approaches provide no guarantees about the quality of the final solution. On the other hand, complete approaches which compute the optimum solution (possibly, with a high computational cost), can be attractive for statically scheduled systems, where the solution is computed once and applied throughout the entire lifetime of the system.

Constraint Programming (CP) is an alternative approach to Integer Programming (IP) for solving combinatorial optimization problems. The work in [22] is based on Constraint Logic Programming to represent system synthesis problem, and leverages a set of finite domain variables and constraints imposed on these variables. Optimal solutions can be obtained for small problems, while large problems require use of heuristics. The proposed framework is able to create pipelined implementations in order to increase the design throughput. In [23] the embedded system is represented by a set of finite domain constraints defining different requirements on process timing, system resources and interprocess communication. The assignment of processes to processors and interprocess communications to buses as well as their scheduling are then defined as an optimization problem tackled by means of constraint solving techniques.

Both CP and IP techniques can claim individual successes but practical experience indicates that neither approach dominates the other in terms of computational performance. The development of a hybrid CP-IP solver that captures the best features of both would appear to offer scope for improved overall performance [24]. However, the issue of communication between different modelling paradigms arises. One method is inherited from the Operations Research and is known as Benders Decomposition [2]: it is proved to converge producing the optimal solution. Benders Decomposition (BD) technique has been extensively used to solve a large variety of problems.

In [25] BD is applied to a numeric algorithm in order to solve the problem of verifying logic circuits: results show that, for some kind of circuits, the technique is an order of magnitude faster w.r.t. other state of the art algorithms. In [26], BD is embedded in the CP environment ECLiPSe and is shown that it can be useful in practice. There are a number of papers using Benders Decomposition in a CP setting. In [27] BD is applied to an allocation and scheduling problem; the master problem (allocation) is based on CP and the sub-problem (scheduling) is solved using a real-time scheduler with fixed task priority. In [28] the branch and check framework is proposed using Benders Decomposition. This technique is applied to the problem of scheduling orders on dissimilar parallel machines, where a set of tasks, linked by precedence constraints, must be performed on a set of parallel machines minimizing the total cost of the process. The machines are dissimilar, so the same task can be executed on a different machine with a different cost and processing time. In [4], BD is applied to minimum cost planning and scheduling problems in a scenario similar to the one described in this paper, considering also release and due date constraints. Here costs depend only on the assignment of tasks to machines, differently from our problem, where contributes to the objective function depend on pairs of assignments. In [3] and [7], Logic-Based BD (a variant of BD introduced in [29], where the sub-problem should not necessarily be an IP problem) is used for Planning and Scheduling problems. Here different objective functions are considered: total cost minimization, makespan, tardiness, and number of late jobs. Precedence constraints among tasks assigned to different resources are not considered[1]: after the allocation phase, the scheduling can be done solving a separate scheduling problem for each facility. Our work addresses therefore an harder problem, being the schedules on different facilities all interconnected.

Although a lot of work has been done applying BD to allocation and scheduling problems, we believe that our approach is not directly comparable with them, mainly because we take in consideration a real application where data must be exchanged between tasks and each task must read/write data (and thus must use the bus resource) during its execution.

7 Conclusion and future works

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

[1] In [7] the author considers precedence constraints among tasks allocated to the same facility

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.

Currently, we are investigating the executability of the solutions found using a MPSoCs platform simulator. We are also extending our tool to an allocation and scheduling problem in a platform where processors can scale their voltage. An optimal solution must therefore not only allocate tasks to processors and memory slot to storage devices, but also associate a voltage and a clock frequency to each task execution, minimizing the total power consumption.

Acknowledgments

This work has been partially supported by ARTIST2 Network of Excellence on Embedded Systems Design (IST-004527).

References

1. Wolf, W.: The future of multiprocessor systems-on-chips. In: In Procs. of the 41st Design and Automation Conference - DAC 2004, San Diego, CA, USA, ACM (2004) 681–685
2. Benders, J.F.: Partitioning procedures for solving mixed-variables programming problems. Numerische Mathematik **4** (1962) 238–252
3. Hooker, J.N.: A hybrid method for planning and scheduling. In: Procs. of the 10th Intern. Conference on Principles and Practice of Constraint Programming - CP 2004, Toronto, Canada, Springer (2004) 305–316
4. Grossmann, I.E., Jain, V.: Algorithms for hybrid milp/cp models for a class of optimization problems. INFORMS Journal on Computing **13** (2001) 258–276
5. Culler, D.A., Singh, J.P.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann (1999)
6. Compton, K., Hauck, S.: Reconfigurable computing: A survey of systems and software. ACM Computing Surveys **34** (1999) 171–210
7. Hooker, J.N.: Planning and scheduling by logic-based benders decomposition. Technical report (2004) http://web.tepper.cmu.edu/jnh/planning.pdf.
8. Prakash, S., Parker, A.: Sos: Synthesis of application-specific heterogeneous multiprocessor systems. Journal of Parallel and Distributed Computing **16** (1992) 338–351
9. Lee, C., Potkonjak, M., Wolf, W.: System-level synthesis of application-specific systems using A* search and generalized force-directed heuristics, San Diego, California (1996) 2–7
10. Bender, A.: Milp based task mapping for heterogeneous multiprocessor systems. In: EURO-DAC '96/EURO-VHDL '96: Procs. of the conference on European design automation, Geneva, Switzerland, IEEE (1996) 190–197
11. Li, Y., Wolf, W.H.: Hardware/software co-synthesis with memory hierarchies. **18** (1999) 1405–1417

12. Meftali, S., Gharsalli, F., Jerraya, A.A., Rousseau, F.: An optimal memory allocation for application-specific multiprocessor system-on-chip. In: Proceedings of the 14th international symposium on Systems synthesis - ISSS '01, ACM Press (2001) 19–24
13. De Micheli, G.: Synthesis and optimization of digital circuits. McGraw Hill (1994)
14. Chatha, K.S., Vemuri, R.: Hardware-software partitioning and pipelined scheduling of transformative applications. **10** (2002) 193–208
15. Fohler, G., Ramamritham, K.: Static scheduling of pipelined periodic tasks in distributed real-time systems. In: Procs. of the 9th EUROMICRO Workshop on Real-Time Systems - EUROMICRO-RTS '97, Toledo, Spain, IEEE (1997) 128–135
16. Palazzari, P., Baldini, L., Coli, M.: Synthesis of pipelined systems for the contemporaneous execution of periodic and aperiodic tasks with hard real-time constraints. In: 18th International Parallel and Distributed Processing Symposium - IPDPS'04. (2004) 121–128
17. Bakshi, S., Gajski, D.D.: A scheduling and pipelining algorithm for hardware/software systems. In: Proceedings of the 10th international symposium on System synthesis - ISSS '97, Washington, DC, USA, IEEE Computer Society (1997) 113–118
18. Axelsson, J.: Architecture synthesis and partitioning of real-time synthesis: a comparison of 3 heuristic search strategies. In: Procs. of the 5th Intern. Workshop on Hardware/Software Codesign (CODES/CASHE97), Braunschweig, Germany, IEEE (1997) 161–166
19. Eles, P., Peng, Z., Kuchcinski, K., Doboli, A.: System level hardware/software partitioning based on simulated annealing and tabu search. Design Automation for Embedded Systems **2** (1997) 5–32
20. Kodase, S., Wang, S., Gu, Z., Shin, K.: 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), Toronto, Canada, IEEE (2003) 181–188
21. Eles, P., Peng, Z., Kuchcinski, K., Doboli, A., Pop, P.: Scheduling of conditional process graphs for the synthesis of embedded systems, Paris, France (1998) 132–139
22. Kuchcinski, K., Szymanek, R.: A constructive algorithm for memory-aware task assignment and scheduling. In: Procs of the Ninth International Symposium on Hardware/Software Codesign - CODES 2001, Copenhagen, Denmark, ACM Press (2001) 147–152
23. Kuchcinski, K.: Embedded system synthesis by timing constraint solving. IEEE Transactions on CAD **13** (1994) 537–551
24. Milano, M.: Constraint and Integer Programming: toward a unified methodolody. Kluwer Academic Publisher (2004)
25. Hooker, J.N., Yan, H.: Logic circuit verification by benders decomposition. In: Principles and Practice of Constraint Programming: The Newport Papers, Cambridge, MA, MIT Press (1995) 267–288
26. Eremin, A., Wallace, M.: Hybrid benders decomposition algorithms in constraint logic programming. In: Procs. of the 7th Intern. Conference on Principles and Practice of Constraint Programming - CP 2001, Paphos, Cyprus, Springer (2001) 1–15
27. Cambazard, H., Déplanche, A.M., Hladik, P.E., Jussien, N., Trinquet, Y.: Decomposition and learning for a hard real time task allocation problem. In: Procs. of the 10th Intern. Conference on Principles and Practice of Constraint Programming - CP 2004, Toronto, Canada, Springer (2004) 153–167
28. Thorsteinsson, E.S.: A hybrid framework integrating mixed integer programming and constraint programming, Paphos, Cyprus (2001) 16–30
29. Hooker, J.N., Ottosson, G.: Logic-based benders decomposition. Mathematical Programming **96** (2003) 33–60