International Journal of Parallel Programming (© 2007)
DOI: 10.1007/s10766-007-0032-7

A Fast and Accurate Technique for Mapping Parallel Applications on Stream-Oriented MPSoC Platforms with Communication Awareness

Martino Ruggiero,[1] Alessio Guerri,[1] Davide Bertozzi,[2,3] Michela Milano[1] and Luca Benini[1]

Received: 22 November 2006 / Accepted: 17 January 2007

The problem of allocating and scheduling precedence-constrained tasks on the processors of a distributed real-time system is NP-hard. As such, it has been traditionally tackled by means of heuristics, which provide only approximate or near-optimal solutions. This paper proposes a complete allocation and scheduling framework, and deploys an MPSoC virtual platform to validate the accuracy of modelling assumptions. The optimizer implements an efficient and exact approach to the mapping problem based on a decomposition strategy. The allocation subproblem is solved through Integer Programming (IP) while the scheduling one through Constraint Programming (CP). The two solvers interact by means of an iterative procedure which has been proven to converge to the optimal solution. Experimental results show significant speed-ups w.r.t. pure IP and CP exact solution strategies as well as high accuracy with respect to cycle-accurate functional simulation. Two case studies further demonstrate the practical viability of our framework for real-life applications.

KEY WORDS: MPSoCs; allocation; scheduling; Integer Programming; Constraint Programming.

[1]University of Bologna, DEIS, Viale Risorgimento 2, Bologna 40136, Italy.
E-mails: {mruggiero; aguerri; mmilano; lbenini}@deis.unibo.it
[2]University of Ferrara, via Saragat 1, Ferrara 40132, Italy.
E-mail: dbertozzi@ing.unife.it
[3]To whom correspondence should be addressed. E-mail: dbertozzi@ing.unife.it

1. INTRODUCTION

Mapping and scheduling problems on multi-processor systems have been traditionally modelled as Integer Linear Programming (IP) problems.[31] 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, i.e. applications with a reduced task-level parallelism, because of their high computational cost. For this reason, heuristic approaches are widely used, such as genetic algorithms, simulated annealing and tabu search.[5] However, they do not provide any guarantees on the optimality of the final solution.

On the other hand, complete approaches, which compute the optimal solution at the cost of an increasing computational cost, can be attractive for statically scheduled systems, where the solution is computed once and applied throughout the entire lifetime of the system.

Static allocations and schedules are well suited for applications whose behaviour can be accurately predicted at design time, with minimum run-time fluctuations.[24] This is the case of signal processing applications such as baseband processing, data encryption or video graphics pipelines. Pipelining is a common workload allocation policy to increase throughput of such applications, and this explains why research efforts have been devoted to extending mapping and scheduling techniques to pipelined task graphs.[11]

The need to provide efficient solutions to the task-to-architecture mapping problem in reasonable time might lead to symplifying modelling assumptions that can make the problem more tractable. Negligible cache-miss penalties and inter-task communication times, contention-free communication or unbounded on-chip memory resources are examples thereof. Such assumptions however, jeopardize the liability of optimizer solutions, and might force the system to work in sub-optimal operating conditions.

In Multi-Processor Systems-on-Chip (MPSoCs) the main source of performance unpredictability stems from the interaction of many concurrent communication flows on the system bus, resulting in unpredictable bus access delays. This also stretches task execution times. Communication architectures should be therefore accurately modelled within task mapping frameworks, so that the correct amount of system-level communication for a given mapping solution can be correctly estimated and compared with the actual bandwidth the bus can deliver. A communication sub-optimal task mapping may lead to reduced throughput or increased latency due to the higher occupancy of system resources. This also has energy implications.

In this paper we present a novel framework for allocation and scheduling of pipelined task graphs on MPSoCs with communication awareness. We target a general template for distributed memory MPSoC architectures, where each processor has a local memory for fast and energy-efficient access to program data and where messaging support is implemented. A state-of-the-art shared bus is assumed as the system interconnect. Our framework is communication-aware in many senses.

First, we introduce a methodology that determines under which operating conditions system interconnect performance is predictable. In that regime, we derive an accurate high-level model for bus behaviour, which can be used by the optimizer to force a maximum level of bus utilization below which architecture-related uncertainties in system execution are negligible. The limit conditions for predictable bus behaviour are bus protocol-specific, and evolving communication protocols are extending the predictable operating region to higher levels of bus utilization. Our methodology allows system designers to precisely assess when delivered bus bandwidth is lower than the requirements and consequently decide whether to revert to a more advanced system interconnect or to tolerate a comunication-related degradation of system performance.

Second, our mapping strategy discriminates among allocation and scheduling solutions based on the communication cost, while meeting hardware/software constraints (e.g. memory capacity, application real-time requirements).

Our allocation and scheduling framework is based on problem decomposition and combines Artificial Intelligence and Operations Research techniques: the allocation subproblem is solved through IP, while scheduling through Constraint Programming (CP). However, the two solvers do not operate in isolation, but interact with each other by means of *no-goods* generation, resulting in an iterative procedure which has been proven to converge to the optimal solution. Experimental results show significant speed-ups w.r.t. pure IP and CP exact solution strategies.

Finally, we deploy an MPSoC virtual platform to validate the results of the optimization steps and to more accurately assess constraint satisfaction and objective function optimization. The practical viability of our framework for real-life systems and applications is shown by means of two demonstrators, namely GSM and Multiple-Input-Multiple-Output (MIMO) wireless communication.

The structure of this work is as follows. Section 2 illustrates related work. Section 3 presents the target architecture while application and system models are reported in Section 4. Highlights on CP and IP are illustrated in Section 5. Our combined solver for the mapping problem is described in Section 6, its computation efficiency in Section 7 and its

integration in a software optimization methodology for MPSoCs in 8. Section 9 finally shows experimental results.

2. RELATED WORK

System design methodologies have been investigated for more than a decade, so that now hardware/software codesign has such a rich literature, which is impossible to survey exhaustively in one article. This section addresses only the works that are more closely related to the problem and to the class of applications we target. A wider insight on the specific research themes addressed by the HW/SW codesign community over the last decade is reported in Ref. 37, while a very comprehensive update on the state of the art in system design can be found in Ref. 14.

Mapping and scheduling problems on multi-processor systems have been traditionally modelled as integer linear programming problems, and addressed by means of IP solvers. An early example is represented by the SOS system, (MILP)[31,32]. Partitioning with respect to timing constraints has been addressed in Ref. 25. A MILP model that allows to determine a mapping optimizing a trade-off function between execution time, processor and communication cost is reported in Ref. 7. An 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 Ref. 26.

Pipelining is a well known workload allocation policy in the signal processing domain. An overview of algorithms for scheduling pipelined task graphs is presented in Ref. 11. IP formulations as well as heuristic algorithms are traditionally employed. In Ref. 9 a retiming heuristic is used to implement pipelined scheduling, while simulated annealing is used in Ref. 28.

Pipelined execution of a set of periodic activities is also addressed in Ref. 17, for the case where tasks have deadlines larger than their periods.

The complexity of pure IP formulations for general task graphs has led to the deployment of heuristic approaches (refer to Ref. 24 for a comprehensive overview of early results). A comparative study of well-known heuristic search techniques (genetic algorithms, simulated annealing and tabu search) is reported in Ref. 5. Unfortunately, busses are implicit in the architecture. Simulated annealing and tabu search are also compared in Ref. 12 for hardware/software partitioning, and minimization of communication cost is adopted as an essential design objective. A scalability analysis of these algorithms for large real-time systems is introduced in Ref. 21. Many heuristic scheduling algorithms are variants and extensions of list scheduling.[13] In general, scheduling tables list all schedules for different

condition combinations in the task graph, and are therefore not suitable for control-intensive applications.

The work in Ref. 23 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. Constraint (Logic) Programming is an alternative approach to IP for solving combinatorial optimization problems.[22] Both techniques can claim individual successes but practical experience indicates that neither approach dominates the other in terms of computational performance on problems similar to the one faced in this paper. The development of a hybrid CP-IP solver that captures the best features of both would appear to offer scope for improved overall performance. However, the issue of communication between different modelling paradigms arises. One method is inherited from the Operations Research and is known as Benders Decomposition:[8] it is an iterative solving strategy that has been proven to converge producing the optimal solution. Benders Decomposition has been extended, and called Logic-Based Benders Decomposition in Ref. 19, for dealing with any kind of solver, like a CP solver. There are a number of papers using Benders Decomposition in a CP setting.[15,18,20,35]

In this work, we take the Logic-Based Benders Decomposition approach, and come up with original design choices to effectively apply it to the context of MPSoCs. We opt for decomposing the mapping problem in two sub-problems: (i) mapping of tasks to processors and of data to memories and (ii) scheduling of tasks in time on their execution units. We tackle the mapping sub-problem with IP and the scheduling one with CP, and combine the two solvers in an iterative strategy which converges to the optimal solution.[19] Our problem formulation will be compared with the most widely used traditional approaches, namely CP and IP modelling of the entire mapping and scheduling problem as a whole, and the significant cut down on search time that we can achieve is proved. Moreover, in contrast to most previous work, the results of the optimization framework and its modelling assumptions are validated by means of cycle-accurate functional simulation on a virtual platform.

3. TARGET ARCHITECTURE

Our mapping strategy targets a general architectural template for a message-oriented distributed memory MPSoC. The distinctive features of this template include: (i) support for message exchange between parallel computation sub-systems, (ii) availability of local memory devices at each computation sub-system and of non-local (i.e. accessible through the system bus) memories to store program data exceeding local mem-

ory size. The remote storage can be provided by a unified memory with partitions associated with each processor or by a separate private memory for each processor core connected to the system bus. This assumption concerning the memory hierarchy reflects the typical trade-off between low access cost, low capacity local memory devices and high cost, high capacity memory devices at a higher level of the hierarchy. Several MPSoC platforms available on the market match our template, such as the Cell Processor,[16] the Silicon Hive Avispa-CH1 processor,[4] the Cradle CT3600 family of multiprocessor DSPs[10] or the ARM11 MPCore platform.[3]

The only restriction that we pose in the template concerns the communication queues, which are assumed to be single-token. Therefore, in a producer-consumer pair, each time a data unit is output by the producer, the consumer has to read it before the producer can run again, since it has its single-entry output queue occupied. The extension of our framework to multi-token queues is left for future work and can be seen as an incremental improvement of the optimization framework.

We modelled one instance of this architectural template in order to test our optimization framework (see Fig. 1). The computation subsystems are supposed to be homogeneous and consist of ARM7 cores (including instruction and data caches) and of tightly coupled software-controlled scratchpad memories for fast access to program operands and for storing input data. We used an AMBA AHB[2] bus as shared system interconnect.

In our implementation, hardware and software support for efficient messaging is provided. Messages can be directly moved between scratchpad memories. In order to send a message, a producer core writes in the message queue stored in its local scratchpad memory, without generating any traffic on the interconnect. After the message is ready, the consumer can transfer it to its own scratchpad or to a private memory space. Data can be transferred either by the processor itself or by a direct memory access controller, when available. In order to allow the consumer to read from the scratchpad memory of another processor, the scratchpad memories should be connected to the communication architecture also by means of slave ports, and their address space should be visible to the other processors.

As far as synchronization is concerned, when a producer intends to generate a message, it checks a local semaphore which indicates whether the queue is empty or not. When a message can be stored, its availability is signaled to the consumer by releasing its local semaphore through a single write operation that goes through the bus. Semaphores are therefore distributed among the processing tiles, resulting in two advantages: the

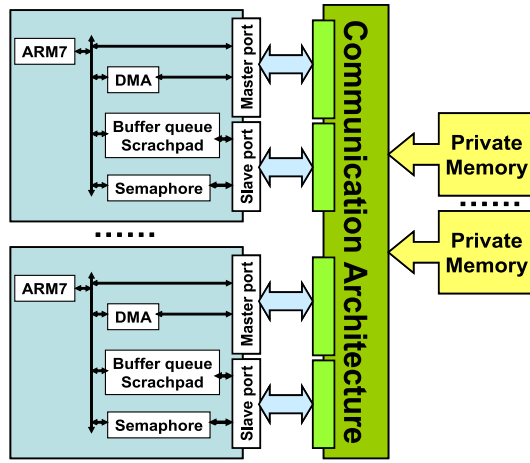

Fig. 1. Message-oriented distributed memory architecture.

read/write traffic to the semaphores is distributed and the producer (consumer) can locally poll whether space (a message) is available, thereby reducing bus traffic.

Furthermore, our semaphores may interrupt the local processor when released, providing an alternative mechanism to polling. In fact, if the semaphore is not available, the polling task registers itself on a list of tasks waiting for that semaphore and suspends itself. Other tasks on the processor can then execute. As soon as the semaphore is released, it generates an interrupt and the corresponding service routine reactivates all tasks on the waiting list.

A DMA engine is attached to each core, as presented in Ref. 29, allowing efficient data transfers between the local scratchpad and non-local memories reachable through the bus. The DMA control logic supports multi-channel programming, while the DMA transfer engine has a dedicated connection to the scratchpad memory allowing fast data transfers from or to it.

Finally, each processor core has a private memory, which can be accessed only by gaining bus ownership. This memory could be on-chip or off-chip depending on the specific platform instantiation. It has a higher access cost and can be used to store program operands that do not fit in scratchpad memory. Optimal memory allocation of task program data to the scratchpad versus the private memory is a specific goal of our optimization framework, dealing with the constraint of limited size of local memories in on-chip multi-processors.

The software support is provided by a real-time multi-processor operating system called RTEMS[34] and by a set of high-level APIs to support message passing on the considered distributed memory architecture. The communication and synchronization library abstracts low level architectural details to the programmer, such as memory maps or explicit management of hardware semaphores.[30]

Our implementation thus supports: (i) processor or DMA-initiated memory-to-memory transfers, (ii) polling-based or interrupt-based synchronization and (iii) flexible allocation of the consumer's message buffer to the local scratchpad or the non-local private memory.

4. HIGH-LEVEL APPLICATION AND SYSTEM MODELS

4.1. Task Model

Our mapping methodology requires to model the multi-task application to be mapped and executed on top of the target hardware platform as a Directed Acyclic Task Graph with precedence constraints. In particular, we focus on pipelined task graphs, representative of signal processing workloads. A real-time requirement is typically specified for this kind of applications, consisting for instance of a minimum required throughput for the pipeline of tasks. Tasks are the nodes of the graph and edges connecting any two node indicate task dependencies. Computation, storage and communication requirements should be annotated onto the graph as follows.

The task execution time is given in two cases: program data is stored entirely in scratchpad memory and local data is stored in remote private memory only. In this latter case, the impact of cache misses on execution time is taken into account.

Our application model associates three kinds of memory requirements to each task:

- **Program Data**: storage space is required for computation data and for processor instructions. They can be allocated by the optimizer either on the local scratchpad memory or on the remote private memory.
- **Internal State**: when needed, an internal state of the task can be stored either locally or remotely.
- **Communication queues**: the task needs communication queues to store outgoing as well as incoming messages to/from other tasks. For the sake of efficient messaging, we pose the constraint that such communication queues should be stored in local scratchpad memory only. So, allocation of these queues is not a degree of freedom for the optimizer.

We assume that application tasks initially check availability of input data and of space for writing computation results (i.e. the output queue must have been freed by the downstream task), in an SDF-like (synchronous dataflow) semantics. Actual input data transfer and task execution occur only when both conditions are met. These assumptions simply result in an atomic execution of the communication and computation phases of each task, thus avoiding the need to schedule communication as a separate task.

4.2. Bus Model

Whenever predictable performance is needed for time-critical applications, it is important to avoid high levels of congestion on the bus, since this makes completion time of bus transactions (and hence of task execution) much less predictable. Average or peak bus bandwidth utilization can be modulated by means of a proper communication-aware task mapping strategy.

When the bus is required to provide a cumulative bandwidth from concurrently executing tasks that does not exceed a certain threshold, its behaviour can be accurately abstracted by means of a very simple *additive model*. In other words, the bus delivers an overall bandwidth which is approximatively equal to the sum of the bandwidth requirements of the tasks that are concurrently making use of it.

This model, provided the working conditions under which it holds are carefully delimited, has some relevant advantages with respect to the scheduling problem model. First, it allows to model time at a coarse granularity. In fact, busses rely on the serialization of bus access requests by re-arbitrating on a transaction basis. Modelling bus allocation at such a fine granularity would make the scheduling problem overly complex since it should be modelled as a unary resource (i.e. a resource with capacity one). In this case, task execution should be modelled using the clock cycle as the unit of time and the resulting scheduling model would contain a huge number of variables. The additive model instead considers the bus as an additive resource, in the sense that more activities can share bus utilization using a different fraction of the total bus bandwidth. Fig. 2(a) illustrates this assumption. The figure represents the bus allocation and scheduling in a real processor, where the bus is assigned to different tasks at different times on a transaction-per-transaction basis. Each task, when owning the bus, uses its entire bandwidth.

Fig. 2(b), instead, represents how we model the bus, abstracting away the transaction-based allocation details. We assume that each task consumes a fraction of the bus bandwidth during its execution time. Note that

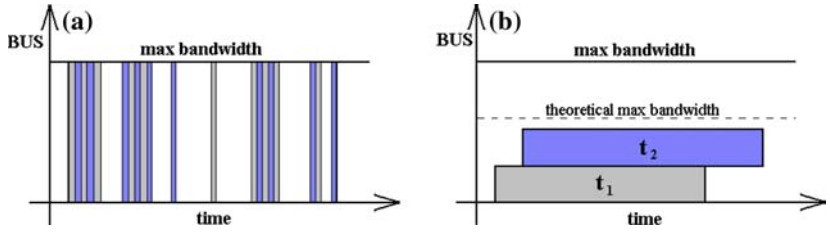

Fig. 2. (a) Bus allocation in a unary model; (b) Bus allocation in a coarse-grain additive model.

we have two thresholds: the maximum bandwidth that the bus is physically able to deliver, and the theoretical one beyond which the additive model fails to predict the interconnect behaviour because of the impact of contention. We will derive this latter threshold in the experimental section by means of extensive simulation runs.

In order to define the fraction of the bus bandwidth absorbed by each task, we consider the amount of data they have to access from their private memories and we spread it over its execution time. In this way we assume that the task is uniformly consuming a fraction of the bus bandwidth throughout its execution time. This assumption will be validated in presence of different traffic patterns in the experimental section.

Another important effect of the bus additive model is that task execution times will not be stretched as an effect of busy waiting on bus transaction completion. Once the execution time of a task is characterized in a congestion free regime, it will be only marginally affected by the presence of competing bus access patterns, in the domain where the additive model holds.

Mapping tasks in such a way that the bus utilization lies below the additive threshold forces the system to make efficient use of available bandwidth. However, our methodology can map tasks to the system while meeting any requirement on bus utilization. Therefore, if a given application cannot be mapped with the bus working in the additive regime, it is on burden of the designer to choose whether to increase maximum allowable peak bus utilization (at the cost of a lower degree of confidence in optimizer performance predictions) or to revert to a more advanced system interconnect. Even in the first case, our methodology helps designers to map their applications with minimum additive threshold crossing.

5. BACKGROUND ON OPTIMIZATION TECHNIQUES

In this section, we recall the basic concepts behind the method we use in this paper, namely the Logic Based Benders Decomposition, and the

two optimization techniques we use for solving each subproblem resulting from the decomposition, namely CP and IP.

5.1. Logic Based Benders Decomposition

The technique we use in this paper is derived from a method, known in Operations Research as Benders Decomposition,[8] and refined by Hooker and Ottosson[19] with the name of Logic-based Benders Decomposition. The classical Benders Decomposition method decomposes a problem into two loosely connected subproblems. It enumerates values for the connecting variables. For each set of enumerated values, it solves the subproblem that results from fixing the connecting variables to these values. The solution of the subproblem generates a Benders cut that the connecting variables must satisfy in all subsequent solutions enumerated. The process continues until the master problem and subproblem converge providing the same value. The classical Benders approach, however, requires that the subproblem be a continuous linear or nonlinear programming problem. Scheduling is a combinatorial problem that has no practical linear or nonlinear programming model. Therefore, the Benders decomposition idea can be extended to a logic-based form Logic Based Benders Decomposition (LBBD), that accommodates an arbitrary subproblem, such as a discrete scheduling problem. More formally, as introduced in Ref. 19, a problem can be written as

$$\min f(y), \tag{1}$$

$$s.t.\ p_i(y)\ i \in I_1 \text{ Master Problem Constraints}, \tag{2}$$

$$g_i(x)\ i \in I_2 \text{ Subproblem Constraints}, \tag{3}$$

$$q_i(y) \rightarrow h_i(x)\ i \in I_3 \text{ Conditional Constraints}, \tag{4}$$

$$y \in Y \text{ Master Problem Variables}, \tag{5}$$

$$x_j \in D_i \text{ Subproblem Variables}. \tag{6}$$

We have master problem constraints, subproblem constraints and conditional constraints linking the two models. If we solve the master problem to optimality, we obtain values for variables y in I_1, namely $\bar{y}$ and the remaining problem is a feasibility problem:

$$g_i(x)\ i \in I_2 \text{ Subproblem Constraints}, \tag{7}$$

$$q_i(\bar{y}) \rightarrow h_i(x)\ i \in I_3 \text{ Conditional Constraints}, \tag{8}$$

$$x_j \in D_i \text{ Subproblem Variables}. \tag{9}$$

We can add to this problem a secondary objective function, say $f_1(x)$ just to discriminate among feasible solutions. If the problem is infeasible, a Benders cut $B_y(y)$ is created constraining variables y. The master problem thus becomes

$$\min f(y), \tag{10}$$

$$s.t.\ p_i(y)\ i \in I_1 \text{ Master Problem Constraints}, \tag{11}$$

$$B_{y_i}(y)\ i \in 1..h \text{ Benders cuts}, \tag{12}$$

$$y \in Y \text{ Master Problem Variables}. \tag{13}$$

y_i is the solution found at iteration i of the master problem.

In practice, to avoid the generation of master problem solutions that are trivially infeasible for the subproblem, it is worth adding a relaxation of the subproblem to the master problem.

Deciding to use the LBBD to solve a combinatorial optimization problem implies a number of design choices that strongly affect the overall performance of the algorithm. Design choices are:

- how to decompose the problem, i.e. which constraints are part of the master problem and which instead are part of the subproblem. This influences the objective function and its dependency on master and subproblem variables;
- which solver to choose for each decomposition: not all problems are solved effectively by the same solver. We consider in this paper Constraint and Integer Linear programming that cover a variety of optimization problems effectively;
- which model to use for feeding each solver: given the problem and the solver we still need to design the problem model, i.e. variables, constraints and objective function. In combinatorial optimization, a wrong model results always in poor solver performances;
- which Benders cuts to use, establishing the interaction between the master and the subproblem;
- which relaxation to use so as to avoid the generation of trivially infeasible solutions in the master problem.

In the following we provide preliminaries on Constraint Programming and Integer Programming, while in Section 6 we detail the design choices performed for the mapping and scheduling problem at hand.

5.2. Constraint Programming

Constraint Programming has been recognized as a suitable modelling and solving tool to face combinatorial (optimization) problems. The

CP modeling and solving activity is highly influenced by the Artificial Intelligence area on Constraint Satisfaction Problems, CSPs (see, e.g. the book by Tsang[36]). A CSP is a triple $\langle V, D, C \rangle$ where V is a set of variables $X_1, \ldots, X_n$, D is a set of finite domains $D_1, \ldots, D_n$ representing the possible values that variables can assume, and C is a set of constraints $C_1, \ldots, C_k$. Each constraint involves a set of variables $V' \subseteq V$ and defines a subset of the cartesian product of the corresponding domains containing feasible tuples of values. Therefore, constraints limit the values that variables can simultaneously assume. A solution of a CSP is an assignment of values to variables which is consistent with constraints.

Constraints can be either mathematical or symbolic. Mathematical constraints have the form: $t_1 \, R \, t_2$ where t_1 and t_2 are finite terms, i.e. variables, finite domain objects and usual expressions, and R is one of the constraints defined on the domain of discourse (e.g. for integers we have the usual relations: $>, \geq, <, \leq, =, \neq$). For example, if two activities i and j characterized by starting times $Start_i$ and $Start_j$ and durations d_i and d_j are linked by a precedence constraint stating that activity i should be executed before activity j, the following mathematical constraint can be imposed, $Start_i + d_i \leq Start_j$. Symbolic constraints, called also global constraints, are predicates involving finite domain variables. They are expressive and powerful constraints (which can also be defined by the user) embedding constraint-dependent filtering algorithms. A typical global constraint is the

$$all\ different([X_1, \ldots, X_n]),$$

available in most CP solvers. Declaratively, the constraint *alldifferent* $([X_1, \ldots, X_n])$ holds iff all variables are assigned to a different value. Thus, it is declaratively equivalent to a set of $n^*(n-1)/2$ binary inequality constraints. However, its compact representation allows more concise models and embeds a specialized efficient graph-based filtering algorithm.[33] Many constraints have been devised for scheduling, which is the most successful application of Constraint Programming. In particular, many kinds of resource and temporal constraints have been devised so as to solve large problem instances, see.[6] As an example, let us consider the cumulative constraint used for modelling limited resource availability in scheduling problems. Its parameters are: a list of variables $[S_1, \ldots, S_n]$ representing the starting time of all activities sharing the resource, their duration $[D_1, \ldots, D_n]$, the resource consumption for each activity $[R_1, \ldots, R_n]$ and the available resource capacity C. Clearly this constraint holds if in any time step where at least one activity is running the sum of the required resource is less than or equal to the available capacity. The constraint

$cumulative([S_1, \ldots, S_n], [D_1, \ldots, D_n], [R_1, \ldots, R_n], C)$ holds iff

$$\forall j \sum_{S_j \leq i < S_j + D_j} R_i \leq C.$$

5.3. Integer Programming

Another solution technique, which is well known and widely used in the system design community is IP. IP is an older method, with roots that date back to the late 1950s. IP can be thought of as a restriction of CP. In fact, IP has only two types of variables: integer variables whose domain contain non-negative integers and continuous variables whose domain contain non-negative real values. In addition, IP allows only one type of constraint: linear inequalities. Finally, the objective function must be linear in the variables. It seems that these restrictions make integer programming much narrower than constraint programming. However, many problems can still be modelled effectively, and algorithms for integer programs can find optimal solutions quickly for many applications. The solving principle of IP is based on the solution of the *linear relaxation*, allowing arbitrary sets of linear constraints to be treated as a global constraint, providing a global view of the problem. The relaxation provides a bound enabling efficient pruning of the search tree and directing search towards promising regions.

The standard form of an IP is the following: let x be the vector of variables, $x = [x_1, x_2, \ldots, x_n]$. A set of these variables I are required to take on integer values, while the remaining variables can take on any real value. Each variable can have a range, represented by vectors l and u such that $l_i \leq x_i \leq u_i$. A *linear constraint* on the variables is a vector of coefficients $a = [a_1, \ldots, a_n]$ and a scalar right-hand-side b. The constraint is then the requirement that

$$\sum_j a_j x_j = b.$$

The "=" in the constraint can also be $\leq$ or $\geq$ (but not $<$ or $>$). The objective function is formed by a vector of coefficients $c = [c_1, c_2, \ldots, c_n]$, with the objective of minimizing (or maximizing) cx. An integer program consists of a single linear objective and a set of constraints. If we create a matrix $A = [a_{ij}]$, where a_{ij} is the coefficient for variable j in the ith constraint, then an integer program can be written:

$$\min cx, \tag{14}$$

$$s.t.\ Ax = b, \tag{15}$$

$$l \leq x \leq u, \tag{16}$$

$$x_j \text{ integer for all } j \in I. \tag{17}$$

For many applications, it is worth working within the limits of integer programming to achieve high performance.

6. MODEL DEFINITION

The two main approaches followed by the system design community when facing software mapping problems in MPSoCs are: (1) either modelling and solving the problem to optimality as an IP whatever the problem structure is or (2) using a special purpose heuristic algorithm requiring sophisticated debugging and tuning and achieving sub-optimal solutions. In this paper, we claim that:

- Whenever allocation and scheduling can be performed off-line due to the intrinsic features of the application (predictable workload), the correct approach is to solve these problems to optimality, since their solution is computed once for all at design time and applied during the entire lifetime of the system. Optimal solutions enable to achieve significant performance speed-ups.
- Analysing and exploiting the problem structure helps in choosing the best solving technique. Integer Programming is an effective solving framework but it is not always the best technique one can use. CP effectively deals with fine time granularities, temporal constraints, resource constraints, and different kind of activities. In general, the best solution strategy can be applied to each subproblem structure.

We have first tried to solve the overall problem (mapping and scheduling) to optimality using a single approach. We have tested both CP alone and IP alone on the problem without success. Therefore, we have switched to Logic Based Benders Decomposition. As shown in Section 5.1, a number of design choices should be addressed.

- **How to decompose the problem.** We split the overall mapping problem into two sub-problems: (1) the allocation of tasks to processors and memory requirements to storage devices, trying to minimize the communication cost and (2) the scheduling sub-problem, where the minimization of execution time (or makespan) can be chosen as secondary design objective.
  Given the critical role played by on-chip communication in determining performance predictability of highly integrated MPSoCs, we

select communication cost minimization as the objective function of the overall problem. 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 global schedule makespan.

Note that our decomposition choice is, to our knowledge, original. Other approaches to allocation and scheduling[18,20] cope with scheduling problems where tasks assigned to different machines are not linked by any constraint. Therefore, the subproblem is composed by a set of independent single machine scheduling problems. Different objective functions can be easily supported by our technique. Clearly, one should change the relaxation of the subproblem and the no-goods. The aim of this paper is not to prove the effectiveness of Logic-Based Benders Decomposition in general, but specifically for the problem at hand.

- **Which solver to choose for each decomposition.** There are no general guidelines for choosing the best solver for the problem at hand. Indeed, it is not always possible to choose the best solver for a given problem instance. For some problems, it is widely recognized that either IP or CP are the techniques of choice. IP is effective for coping with optimization problems, it has a global problem view due to the use of linear relaxations, but sometimes its models are too large and somewhat unnatural. On the other hand, CP has an effective way to cope with the so called *feasibility reasoning*, encapsulating efficient and incremental filtering algorithms into global constraints. However, CP has a naive way to cope with optimization problems by successively solving a set of constraint satisfaction problems with tighter bounds on the objective function.

  For the problem at hand, the allocation problem has been solved via IP. It better copes with objective functions based on the sum of assignment costs. For the scheduling problem, the solver is instead based on CP since it better copes with temporal resource constraints and finer time granularity.
- **Which model to use for feeding each solver.** This part will be described in detail in the next sections. In particular, the allocation problem model is described in Section 6.1 while the scheduling problem model is described in Section 6.2.
- **Which Benders cuts to use.** This aspect is essential for the interaction between the two solvers. We solve 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, since the main objective function depends only on master problem variables. 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.

- **Which relaxation to use.** 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 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 allocations 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. The use of a relaxation in the master problem is widely used in practice and helps in producing better solutions.

6.1. Allocation Problem Model

The allocation problem is the problem of allocating n tasks to m processors and memory requirements to storage devices. The objective function is the minimization of the amount of data transferred on the bus. We solve the allocation problem using an IP model. We consider four decision variables: T_{ij}, taking value 1 iff task i executes on processor j; Y_{ij}, taking value 1 iff task i allocates the program data on the scratchpad memory of processor j; Z_{ij}, taking value 1 iff task i allocates the internal state on the scratchpad memory of processor j; X_{ij}, taking value 1 iff tasks i and $i + 1$ execute on different processors, one of them being processor

j, therefore the two tasks communicate using the bus. Variables X_{ij} have only two indexes since we are considering a pipeline, where a task i communicates only with the task $i+1$. When modelling a general task graph these variables must have the form X_{ikj}, taking value 1 iff two communicating tasks i and k execute on different processors, one of them being processor j. The linear constraints introduced in the model are:

$$\sum_{j=1}^{m} T_{ij} = 1, \forall i \in 1 \ldots n, \tag{18}$$

$$T_{ij} + T_{i+1j} + X_{ij} - 2K_{ij} = 0, \quad \forall i \in 1 \ldots n, \forall j. \tag{19}$$

Constraints (18) state that each process can execute only on a processor, while constraints (19) 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. K_{ij} are integer binary variables forcing the sum $T_{ij} + T_{i+1j} + X_{ij}$ to be either 0 or 2 (in fact, X_{ij} is the `exor` of T_{ij} and T_{i+1j}). We also add to the model the constraints stating that if a task i does not execute on a processor j, it cannot allocate its program data or its internal state in the local scratchpad of processor j, i.e. $T_{ij} = 0 \Rightarrow Y_{ij} = 0, Z_{ij} = 0$. For each group of consecutive tasks whose execution times sum exceeds the RT requirement, we introduce in the model a constraint preventing the solver to allocate all the tasks in the group to the same processor. To generate these constraints, we find out all groups of consecutive tasks whose execution times sum exceeds RT. Constraints are the following:

$$\sum_{i \in S} Dur_i > \mathrm{RT} \Rightarrow \sum_{i \in S} T_{ij} \leq |S| - 1 \, \forall j. \tag{20}$$

This is a relaxation of the scheduling problem, added to the master problem to prevent the generation of trivially infeasible solutions. The objective function is the minimization of the communication cost, i.e. the total amount of data transferred on the bus for each pipeline iteration. Contributions to the communication cost arise when a task allocates its program data and/or internal state to the remote memory, and when two consecutive tasks execute on different processors, and their communication messages must be transferred through the bus from the communication queue of one processor to that of the other one. Using the decision variables described above, we have a contribution, respectively, when: $T_{ij} = 1, Y_{ij} = 0,$

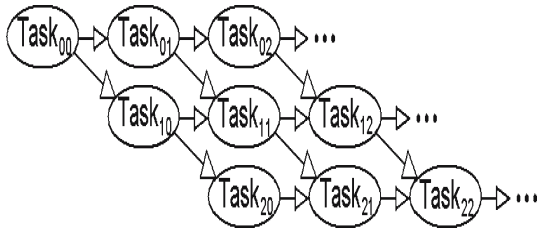

Fig. 3. Precedence constraints among the activities.

or $T_{ij} = 1, Z_{ij} = 0$, or $X_{ij} = 1$. Therefore, the objective function is to minimize:

$$\sum_{j=1}^{m}\sum_{i=1}^{n} \left(\mathrm{Mem}_i(T_{ij} - Y_{ij}) + 2 \times \mathrm{State}_i(T_{ij} - Z_{ij}) + (\mathrm{Data}_i X_{ij})/2)\right), \tag{21}$$

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.

6.2. 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 analyse the system behaviour at working rate, that is when all processes are running or ready to run. To do that, we 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 Fig. 3.

In the scheduling problem model, for each task Task_{ij} (the jth iteration of the ith process) we introduce a variable A_{ij}, representing the computation activity of the task. 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 from 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 $i = 0$. For this purpose, 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 written after the execution: we therefore introduce in the model the activities RS_{ij} and WS_{ij} for each process i requiring an internal state. The durations of all these activities depend on whether data are stored in the local or the remote memory but, after the allocation, these times can be statically estimated. Figure 3 depicts the precedence constraints among tasks. The horizontal arcs (between Task_{ij} and $\mathrm{Task}_{i,j+1}$) represent just precedence constraints, while the diagonal arcs (between Task_{ij} and $\mathrm{Task}_{i+1,j}$) represent precedences due to communication and are labelled with the amount of data to communicate. Each task Task_{ij} is composed by 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 are:

$$A_{i,j-1} \prec In_{ij}, \quad \forall i, j \tag{22}$$

$$In_{ij} \prec A_{ij}, \quad \forall i, j \tag{23}$$

$$A_{i-1,j} \prec In_{ij}, \quad \forall i, j \tag{24}$$

$$RS_{ij} \preceq A_{ij}, \quad \forall i, j \tag{25}$$

$$A_{ij} \preceq WS_{ij}, \quad \forall i, j \tag{26}$$

$$In_{i+1,j-1} \prec A_{ij}, \quad \forall i, j \tag{27}$$

$$A_{i,j-1} \prec A_{ij}, \quad \forall i, j, \tag{28}$$

where the symbol $\prec$ means that the activity on the right should follow the activity on the left, and the symbol $\preceq$ means that the activity on the right must start as soon as the execution of the activity on the left completes: i.e. $A \prec B$ means $\mathrm{Start}_A + \mathrm{Dur}_A \leq \mathrm{Start}_B$ and $A \preceq B$ means $\mathrm{Start}_A + \mathrm{Dur}_A = \mathrm{Start}_B$. Constraints (22) state that each process iteration can start reading the communication queue only after the end of its previous iteration: a task needs to access the data stored in the communication queue during its whole execution, so the memory storing these data can only be freed when the computation activity A_{ij} ends. Constraints (23) state that each task can start computing only when it has read the input data, while constraints (24) state that each task can read the input data only when the previous task has generated them. Constraints (25) and (26) state that each task must read the internal state just before the execution and write it just afterwards. Constraints (27) 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 data stored in its communication queue has been read by the target process. Constraints (28)

state that the iterations of each task must execute in order. We also introduced the real-time requirement constraints $\text{Start}(A_{ij}) - \text{Start}(A_{i,j-1}) \leq \text{RT}$, $\forall i, j$, whose relaxation is used in the allocation problem model. The time elapsing between two consecutive executions of the same task can be at most RT. Processors are modelled as unary resources, stating that only one activity at a time can execute on each processor, while the bus is modelled as a shared resource (see Section 4.2): several activities can share the bus, each of them consuming a fraction of the total bandwidth; a *cumulative* constraint is introduced ensuring that the total bus bandwidth consumption (or a lower threshold) is never exceeded.

7. COMPUTATIONAL EFFICIENCY

To test the computational efficiency 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 to optimality. Actually, since the first experiments showed that both CP and IP approaches are not able to find even the first 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 ten instances for each task or processor number. All problems considered have a solution. Experiments were performed on a 2 GHz Pentium 4 with 512 Mb RAM and leveraged state-of-the-art professional solving tools, namely ILOG CPLEX 8.1, ILOG Solver 5.3 and ILOG Scheduler 5.3.

In Fig. 4 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.

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 and the problem instance is very easy to be solved, and Hybrid incurs the communication overhead between two models. As soon as the number of tasks and/or processors grows, IP and CP performance worsen and their search times become

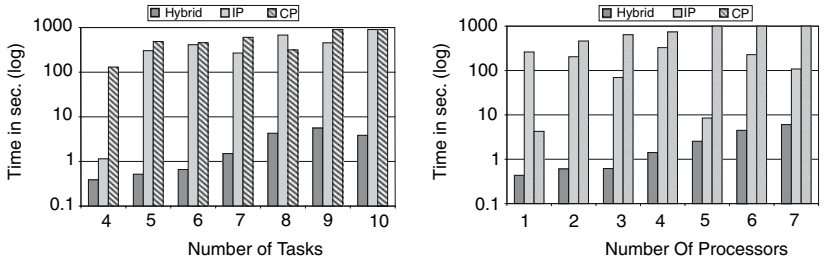

Fig. 4. Comparison between algorithms search times for different task number (left) and for different processor number (right).

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 six tasks or three processors and more, IP and CP can find the solution only in the 50% or less of the cases, while Hybrid can solve 100% of the instances. We can see in addition, that Hybrid search time scales up linearly in the logarithmic scale.

We also measured the number of times the CP and IP solvers iterate. We found that, due to the limited size of the scratchpad and to the relaxation of the sub-problem added to the master, the solver iterates always one or two 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 although very simple.

8. VALIDATION METHODOLOGY

In this section we explain how to deploy our optimization framework in the context of a real system-level design flow. Our approach consists of using a virtual platform to pre-characterize the input task set, to simulate the allocation and scheduling solutions provided by the optimizer and to detect deviations of measured performance metrics with respect to predicted ones.

For each task in the input graph we need to provide the following information: bus bandwidth requirement for reading input data in case the producer runs on a different processor, time for reading input data if the producer runs on the same processor, task execution time with program data in scratchpad memory, task execution overhead due to cache misses when program data resides in remote private memory. For each pipelined task graph, this information can be collected with $2 + N$

simulation runs on the MPARM simulator,[1] where N is the number of tasks. Recall that this is done once for all. We model task communication and computation separately to better account for their requirement on bus utilization, although from a practical viewpoint they are part of the same atomic task. The initial communication phase consumes a bus bandwidth which is determined by the hardware support for data transfer (DMA engines or not) and by the bus protocol efficiency (latency for a read transaction). The computation part of the task instead consumes an average bandwidth defined by the ratio of program data size (in case of remote mapping) and execution time. A less accurate characterization framework can be used to model the task set, though potentially incurring more uncertainty with respect to optimizer's solutions. We use the virtual platform also to calibrate the bus additive model, specifying the range where this model holds. For an AMBA AHB bus, we found that tasks should not concurrently ask for more than 50% of the theoretical bandwidth the bus can provide (400 MByte/sec with one wait state memories), otherwise congestion causes a bandwidth delivery which does not keep up with the requirements.

The input task parameters are then fed to the optimization framework, which provides optimal allocation of tasks and memory locations to processor and storage devices, respectively, and a feasible schedule for the tasks meeting the real-time requirements of the application. Two options are feasible at this point. First, the optimizer uses the conservative maximum bus bandwidth indicated by the virtual platform, and the derived solutions are guaranteed to be accurate (see Section 9). Second, the optimizer uses a higher bandwidth than specified, in order to improve bus utilization, and the virtual platform must then be used to assess the accuracy of the optimization step (e.g. constraint satisfaction, validation of execution and data transfer times). If the accuracy is not satisfactory, a new iteration of the procedure will allow to progressively decrease the maximum bandwidth until the desired level of accuracy is reached with the simulator.

Note that the scheduler of the RTEMS operating system allows to implement all the scheduling solutions provided by the optimizer. For the case we are considering (stream-oriented processing with single token communication among the pipeline stages) it can be proven that all schedules are periodic. The interested reader can read the proof in Appendix 1. Our framework assumes that no preemption nor time-slicing is implemented by the OS. Most schedules generated by the optimizer can be implemented by means of priority-based scheduling, but not all of them. For those remaining cases, RTEMS provides scheduling APIs with which one task

can decide which task to activate next. In this way, all possible schedules can be implemented.

9. EXPERIMENTAL RESULTS

We have performed three kinds of experiments, namely (i) validation and calibration of the bus additive model, (ii) measurement of deviations of simulated throughput from the one computed by the optimizer on a large number of problem instances, (iii) experiments devoted to show the viability of the proposed approach by means of two demonstrators.

9.1. Validation of the Bus Additive Model

The behaviour of the bus additive model is illustrated by the experiment of Fig. 5. An increasing number of AMBA-compliant uniform traffic generators, consuming each 10% of the maximum theoretical bandwidth (400 MByte/sec), have been connected to the bus, and the resulting real bandwidth provided by the bus measured in the virtual platform. It can be clearly observed that the delivered bandwidth keeps up with the requested one until the sum of the requirements amounts to 60% of the maximum theoretical bandwidth. This defines the actual maximum bandwidth, notified to the optimizer, under which the bus works in a predictable way. If the communication requirements exceed the threshold, as a side effect we observe an increase of the execution times of running tasks with respect to those measured without bus contention, as depicted in Fig. 6. For this experiment, synthetic tasks running on each processor have been

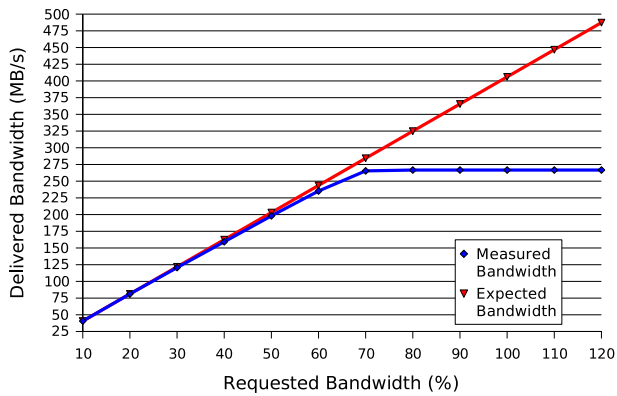

Fig. 5. Implications of the bus additive model.

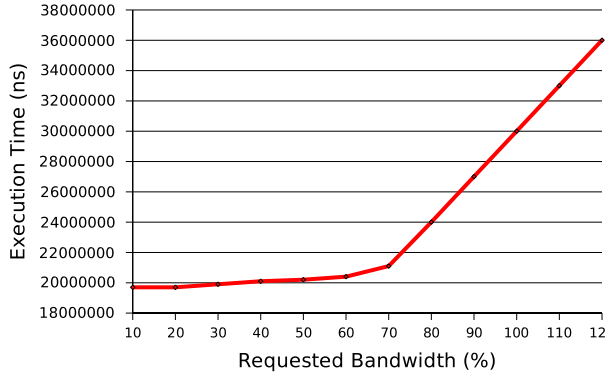

Fig. 6. Execution time variation.

employed. The 60% bandwidth threshold value corresponds to an execution time variation of about 2% due to longer bus transactions.

However, the threshold value also depends on the ratio of bandwidth requirements of the tasks concurrently trying to access the bus. Contrarily to Fig. 5, where each processor consumes the same fraction of bus bandwidth, Fig. 7 shows the deviations of offered versus required bandwidth for competing tasks with different bus bandwidth requirements. Configurations with different number of processors are explored, and numbers on the x-axis show the percentage of maximum theoretical bandwidth required by each task. It can be observed that the most significant deviations arise when one task starts draining most of the bandwidth, thus creating a strong interference with all other access patterns. The presence of such communication hotspots suggests that the maximum cumulative bandwidth requirement which still stimulates an additive behaviour of the bus is lower than the one computed before, and amounts to about 50% of the theoretical maximum bandwidth. We also tried to reproduce Fig. 7 varying the burstiness of the generated traffic. Till now, the traffic generators have used single bus transactions to stimulate bus traffic. We then generated burst transactions of fixed length (four beat bursts, corresponding to a cache line refill of an ARM7 processor) but with varying inter-burst periods. Results are not reported here since the measured upper thresholds for the additive model are more conservative than those obtained with single transfers. Therefore, frequent single transfers and unbalanced bus utilization frequencies of the concurrent tasks running on different processors represent the worst case scenario for the accuracy of the bus additive model.

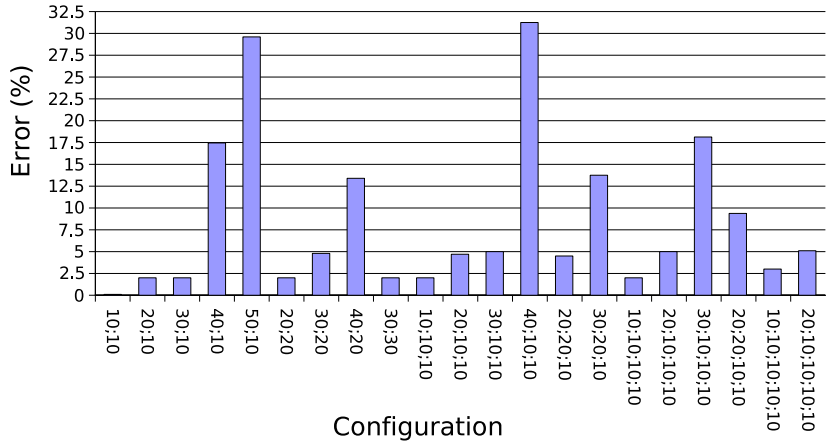

Fig. 7. Bus additive model for different ratios of bandwidth requirements among competing tasks for bus access.

9.2. Validation of Allocation and Scheduling Solutions

We have deployed the virtual platform to implement the allocations and schedules generated by the optimizer, and we have measured deviations of the simulated throughput from the predicted one for 50 problem instances. A synthetic benchmark has been used for this experiment, allowing to change system and application parameters (local memory size, execution times, data size, etc.). We want to make sure that modelling approximations are not such to significantly impact the accuracy of optimizer results with respect to real-life systems. The results of the validation phase are reported in Fig. 8, which shows the probability for throughput differences between optimizer and simulator results. The average difference between measured and predicted values is 0.76%, with 0.79 standard deviation. This confirms the high level of accuracy achieved by the developed optimization framework, thanks to the calibration of system model parameters against functional timing-accurate simulation and to the control of system working conditions.

In general, knowing the accuracy of the optimizer with respect to functional simulation is not enough, since the relative sign of the error decides whether real-time requirements will be met or not in cases where there is only very little slack time. Figure 9 tries to answer this question by reporting the distribution of the sign of prediction vs measurement errors. A negative error indicates that the optimizer has been conservative, therefore the real throughput is higher than the predicted one. The contrary holds in case of positive errors. This latter case is the most critical,

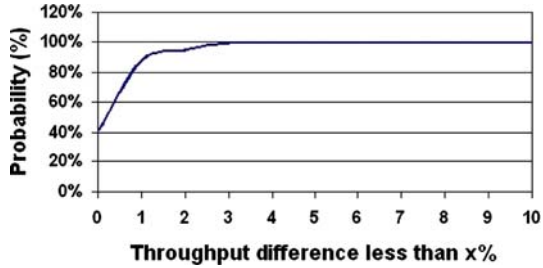

Fig. 8. Probability of throughput differences.

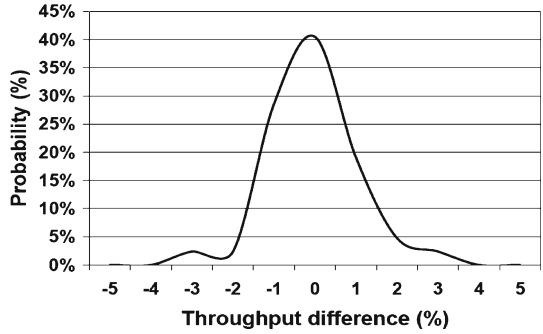

Fig. 9. Probability of throughput differences in variable realtime study.

since it corresponds to the case where the optimizer has been optimistic. However, we clearly see that the error margin is very small (within 5%). Moreover, since the scheduling step of the optimization framework targets makespan minimization, the optimizer usually provides a schedule which results in throughput values that are far more conservative than those that were required to the optimizer. As a consequence, even if the real throughput is 5% worse, the margins with respect to the timing constraints are typically much larger.

The scalability of our approach with the number of tasks and processors has already been showed in Section 7, and compared with state-of-the-art solving techniques. In contrast, the case studies that follow aim at proving the applicability of our approach to real-life applications and MPSoC systems. Most applications are natively coded in imperative sequential C language, and their efficient parallelization goes beyond the scope of this work. We therefore manually decomposed the GSM and MIMO benchmarks in a reasonable number of tasks and tested our mapping methodology with them.

	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6
Computation Time (ns)	281639	437038	317032	308899	306213	306470
Remote Data Overhead (ns)	3978	1620	1099	2243	1916	1707
Local Communication (ns)		4754	6675	5810	6020	5810
Remote Communication (ns)		8621	12266	10773	10609	10576
Program Data (Byte)	420	420	560	560	560	560
Communication Data In – Out (Byte)	0 - 340	340 – 444	444 - 444	444 - 444	444 - 444	444 – 0
Processor	1	1	2	2	3	3
Data Location	Local	Local	Remote	Remote	Remote	Local
4 Processors with 2048 Byte ScratchPad Memory						

Fig. 10. GSM case study.

9.3. Application to GSM

Most state-of-the-art cell-phone chip-sets include dual-processor architectures. GSM encoding and decoding have been among the first target applications to be mapped onto parallel multi-processor architectures. Therefore, we first proved the viability of our approach with a GSM encoder application. The source code has been parallelized into six pipeline stages, and each task has been pre-characterized by the virtual platform to provide parameters of task models to the optimizer. Such information, together with the results of the optimization run, are reported in Fig. 10. Note that the optimizer makes use of 3 of the 4 available processors, since it tries to minimize the cost of communication while meeting hardware and software constraints. The throughput required to the optimizer in this case was 1 frame/10ms, compliant with the GSM minimum requirements. The obtained throughput was 1.35 frames/ms, far more conservative. The simulation on the virtual platform provided an application throughput within 4.1% of the predicted one. The table also shows that program data has been allocated in scratchpad memory for Tasks 1,2 and 6 since they have smaller communication queues. Schedules for this problem instance are trivial. The time taken by the optimizer to come to a solution was 0.1 seconds.

9.4. MIMO Processing

One major technological breakthrough that will make an increase in data rate possible in wireless communication is the use of multiple antennas at the transmitters and receivers (Multiple-input Multiple-output systems). MIMO technology is expected to be a cornerstone of many next-generation wireless communication systems. The scalable computation power provided by MPSoCs is progressively making the implementation of MIMO systems and associated signal processing algorithms feasible,

	Task 1	Task 2	Task 3	Task 4	Task 5
Computation Time (ns)	526737	1633286	66385	324883	5253632
Remote Data Overhead (ns)	8683	13734	749	2279	62899
Local Communication (ns)		3639	12052	5373	10215
Remote Communication (ns)		6037	17605	10615	16960
Program Data (Byte)	676	2500	256	4	3136
Communication Data In – Out (Byte)	0 - 256	256 – 784	784 – 400	400 – 784	784 – 0
Processor	1	1	1	1	2
Data Location	Remote	Remote	Local	Local	Local
6 Processors with 4K Byte ScratchPad Memory					

Fig. 11. MIMO processing results.

therefore we applied our optimization framework to spatial multiplexing-based MIMO processing.[27]

The MIMO computation kernel was partitioned into five pipeline stages. Optimal allocation and scheduling results for a system of 6 ARM7 processors are reported in Fig. 11. The reported mapping configuration is referred to the case where the tightest feasible real-time constraint was applied to the system (about 1.26 Mbit/sec). Obviously, further improvements of the throughput can be obtained by replacing the ARM7 cores with more computation-efficient processor cores. In this benchmark, Task 5 has the heaviest computation requirements, and requires a large amount of program data for its computation. In order to meet the timing requirements and to be able to allocate program data locally, this task has been allocated on a separate processor.

As can be observed, the optimizer has not mapped each remaining task on a different processor, since this would have been a waste of resources providing sub-optimal results. In other words, the throughput would have been guaranteed just at the same, but at a higher communication cost. Instead, Tasks 1–4 have been mapped to the same processor. Interestingly, the sum of the local memory requirements related to communication queues leaves a very small remaining space in scratchpad memory, which allows the optimizer to map locally only the small program data of Tasks 3 and 4. The overall mapping solution was therefore not trivial to devise without the support of the combined CP-IP solver, which provides the optimal allocation and scheduling in about 600 ms. The derived configuration was then simulated onto the virtual platform, and throughput accuracy was found to be (conservatively) within 1%.

10. CONCLUSIONS

We target allocation and scheduling of pipelined stream-oriented applications on top of distributed memory architectures with messaging support. We tackle the complexity of the problem by means of

decomposition and no-good generation, and prove the increased computational efficiency of this approach with respect to traditional ones. Moreover, we deploy a virtual platform to validate the results of the optimization framework and to check modelling assumptions, showing a very high level of accuracy. Finally, we show the viability of our approach by means of two demonstrators: GSM and MIMO processing. Our methodology contributes to the advance in the field of software optimization tools for highly integrated on-chip multiprocessors, and can be applied to all pipelined applications with design-time predictable workloads. The extension to generic task graphs does not present theoretical hindrances and is ongoing work.

Appendix 1: PROOF OF SCHEDULE PERIODICITY

In this section we prove that despite our algorithm considers an unbounded number j of iterations of a pipeline with n tasks $Task_{ij}$, $i = 1..n$, our final schedule is always periodic. The proof assumes single token communication queues (i.e. length one queues), but it can be easily extended to any finite length.

Tasks are partitioned by the allocation module on p processors. So let us consider p partitions: $\text{Task}_{ij}\ i \in Sp_k \forall j$, where $k = 1..p$ and Sp_k is the set of tasks assigned to processor k. Our aim is to show that our (time discrete) scheduling algorithm that minimizes the makespan produces a periodic solution even if we have a (theoretical) infinite number of pipeline iterations.

The proof is based on the following idea: if we identify in the solution a state of the system that assumes a finite number of configurations, than the solution is periodic. In fact, after a given state S the algorithm performs optimal choices; as soon as we encounter S again, the same choices are performed.

For each iteration j, the state we consider is the following: the slack of each task in S_k to its deadline. The state of the system is the following: For each processor $k = 1..p$ we have $\langle Slack^k_{1j}, \ldots, Slack^k_{lj} \rangle$, where $Slack^k_{ij}$ is the difference between the deadline of Task_{ij} running on processor k and its completion time. Therefore, if we prove that the number of possible state configurations is finite (i.e. it does not depend on the iteration number j), being the transitions between two states deterministic, even if we have an infinite number of repetition of the pipeline, the solution is periodic.

After the pipeline starts up, the deadline of each task Task_{ij} is defined by the first iteration of task i. i.e. Task_{i1}. In fact, the real-time

(throughput) constraint states that every P time points each task should be repeated. Therefore, if the first iteration of a task i is performed at time t_i, the second iteration of i should be performed at time $t_i + P$, and the j-th iteration at time $t_i + (j-1) * P - \text{diration}(\text{Task}_{ij})$.

Now, let us consider two cases:

- if the tasks in S_k are consecutive in the pipeline, then their repetition cannot change. For example, if tasks T_{1j}, T_{2j} and T_{3j} are allocated to the same processor (for all j), having length one queues, they can be repeated only in this order. Indeed, one can repeat T_{1j} after T_{2j}, but minimizing the makespan it is not the right decision.
- if instead the tasks in S_k are not consecutive, then there could be repetitions in between that could break the periodicity. Therefore, we should concentrate on this case.

For the sake of readibility we now omit the index representing the iteration since we concentrate on the maximum slack a task can assume. Let us consider two non consecutive tasks $T_A \in S_k$ and $T_B \in S_k$. Suppose that between T_A and T_B there are m tasks allocated on other processors different from k. Let us call them $T_{A1}, T_{A2}, \ldots, T_{Am}$ ordered by precedence constraints. If we have communication queues of length one, between T_A and T_B there are AT MOST m iterations of T_A. In fact, T_A can be repeated as soon as T_{A1} starts on another processor. Also, it can be repeated as soon as another iteration of T_{A1} starts, that can happen as soon as T_{A2} starts and so on. Clearly, m iterations are possible only if

$$m^* \text{duration}(T_A) \leq \sum_{i=1}^{m} \text{duration}(T_{Ai})$$

but if this relation does not hold, there can be only less iterations of T_A. Therefore, m is an upper bound on the number of iterations of T_A between the first T_A and T_B. If t_A is the time where the first repetition of T_A is performed, the mth iteration of T_A has a deadline of $t_A + (m-1) * P$. Its slack is clearly bounded to the maximum deadline minus its duration, $t_A + (m-1) * P - \text{duration}(T_A)$.

The upper bound for m is $n-2$. In fact, in a pipeline of n tasks the maximum number of repetitions of a task happen if only the first and the last task are allocated on the same processor. They have $n-2$ tasks in between allocated on different processors. Therefore, the maximum number of repetitions of T_1 between T_1 and T_n is $n-2$.

Therefore, if the first iteration of T_1 is executed at time t_1 its $(n-2)th$ iteration has a max deadline $t_1 + (n-3) * P - \text{duration}(T_1)$.

Being the max deadline of a task finite, also its max slack is finite despite the number of iteration of the pipeline.

Therefore, whatever the state is, each task belonging to the state has a finite slack. The combination of slacks are finite, and therefore, after a finite number of repetition, the system finds a state already found and becomes periodic.

REFERENCES

1. F. Angiolini, L. Benini, D. Bertozzi, M. Loghi, and R. Zafalon, Analyzing On-Chip Communication in a MPSoC environment, in *Proceedings of the IEEE Design and Test in Europe Conference (DATE)*, Paris, France, pp. 752–757 (2004).
2. ARM Ltd., Sheffield, UK, AMBA 2.0 Specification. http://www.arm.com/armtech/AMBA
3. ARM11 MPCore, http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html
4. Avispa-CH1 Communications Signal Processor, http://www.silicon-hive.com/t.php?assetname=text&id=131
5. J. Axelsson, Architecture Synthesis and Partitioning of Real-Time Synthesis: A Comparison of 3 Heuristic Search Strategies, in *Proceedings of the 5th International Workshop on Hardware/Software Codesign (CODES/CASHE97)*, Braunschweig, Germany, pp. 161–166 (1997).
6. P. Baptiste, C. Le Pape, and W. Nuijten, Constraint-Based Scheduling, in *Proceedings of the International Series in Operations Research and Management Science*, Vol. 39, Springer, New York, USA (2001).
7. A. Bender, MILP based Task Mapping for Heterogeneous Multiprocessor Systems, EURO-DAC '96/EURO-VHDL '96, in *Proceedings of the conference on European design automation*, Geneva, Switzerland, pp. 190–197 (1996).
8. J. F. Benders, Partitioning Procedures for Solving Mixed-Variables Programming Problems, *Numerische Mathematik*, **4**:238–252 (1962).
9. K. S. Chatha and R.Vemuri, Hardware-Software Partitioning and Pipelined Scheduling of Transformative Applications, in *Transactions on Very Large Scale Integration Systems*, **10**(3):193–208 (2002).
10. CT3600 Family of Multi-core DSPs, http://www.cradle.com/products/sil_3600_family.shtml
11. G. De. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw Hill, New York (1994).
12. P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu search, *Journal on Design Automation for Embedded Systems*, **2**:5–32 (1997).
13. P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, and P. Pop, Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems, in *Proceedings of the conference on Design, automation and test in Europe*, Paris, France, pp. 132–139 (1998).
14. Embedded microelectronic systems: status and trends, in *IEE Proceedings—Computers and Digital Techniques—March 2005* – Vol. 152, Issue 2.
15. A. Eremin and M. Wallace, Hybrid Benders Decomposition Algorithms in Constraint Logic Programming, in *Proc. of the 7th International Conference on Principles and Practice of Constraint Programming - CP 2001*, Paphos, Cyprus, pp. 1–15 (2001).

16. B. Flachs *et al.*, A Streaming Processor Unit for the CELL Processor, in *Proceedings of the ISSCC*, San Francisco, USA, pp. 134–135 (2005).
17. G. Fohler and K. Ramamritham, Static Scheduling of Pipelined Periodic Tasks in Distributed Real-Time Systems, in *Proc. of the 9th EUROMICRO Workshop on Real-Time Systems - EUROMICRO-RTS '97*, Toledo, Spain, pp. 128–135 (1997).
18. I. E. Grossmann and V. Jain, Algorithms for Hybrid MILP/CP Models for a Class of Optimization Problems, *INFORMS Journal on Computing*, **13**(4):258–276 (2001).
19. J. N. Hooker and G. Ottosson, Logic-Based Benders Decomposition, *Mathematical Programming*, **96**:33–60 (2003).
20. J. N. Hooker, A Hybrid Method for Planning and Scheduling, in *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming - CP 2004*, Toronto, Canada, pp. 305–316 (2004).
21. 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 *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003)*, Washington, USA, pp. 181–188 (2003).
22. K. Kuchcinski, Embedded System Synthesis by Timing Constraint Solving, in *Proceedings of the 10th International Symposium on System Synthesis,* Antwerp, Belgium, pp. 50–57 (1997).
23. K. Kuchcinski and R.Szymanek, A Constructive Algorithm for Memory-Aware Task Assignment and Scheduling, in *Proceedings of the 9th International Symposium on Hardware/Software Codesign - CODES 2001*, Copenhagen, Denmark, pp. 147–152 (2001).
24. Y. Kwok, and I. Ahmad, Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors, *ACM Computing Surveys*, **31**(4):406–471 (1999).
25. C. Lee, M. Potkonjak, and W. Wolf, System-Level Synthesis of Application-Specific Systems Using A* Search and Generalized Force-Directed Heuristics, in *Proceedings of the 9th International Symposium on System Synthesis—ISSS '96*, San Diego, USA, pp. 2–7 (1996).
26. Y. Li and W. H. Wolf, Hardware/Software Co-Synthesis with Memory Hierarchies, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1405–1417 (1999).
27. D. Novo, W. Moffat, V. Derudder, and B. Bougard, Mapping a Multiple Antenna SDM-OFDM Receiver on the ADRES Coarse-Grained Reconfigurable Processor, in *Proceedings of the IEEE Workshop on Signal Processing Systems Design and Implementation,* Athens, Greece, pp. 473–478 (2–4 Nov. 2005).
28. P. Palazzari, L. Baldini, and M. Coli, Synthesis of Pipelined Systems for the Contemporaneous Execution of Periodic and Aperiodic Tasks with Hard Real-Time Constraints, in *Proceedings of the 18th International Parallel and Distributed Processing Symposium - IPDPS'04*, Santa Fe, USA, pp. 121–128 (2004).
29. F. Poletti, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias, An Integrated Hardware/Software Approach For Run-Time Scratchpad Management, in *Proceedings of the DAC 2004*, San Diego, USA, pp. 238–243 (2004).
30. F. Poletti, A. Poggiali, and P. Marchal, Flexible Hardware/Software Support for Message Passing on a Distributed Shared Memory Architecture, in *Design And Test Europe Conference 2005 Proceedings*, Munich, Germany, pp. 736–741 (2005).
31. S. Prakash and A. Parker, SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems, Journal of Parallel and Distributed Computing, 338–351 (1992).
32. S. Prakash and A. C. Parker, Synthesis of Application-Specific Multiprocessor Systems Including Memory Components, in *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, USA (1992).

33. J. C. Régin, A filtering algorithm for constraints of difference in CSPs, in *Proceedings of the 12th National Conference on Artificial Intelligence—AAAI94*, Seattle, USA, pp. 362–367 (1994).
34. RTEMS Home Page, http://www.rtems.com
35. E. S.Thorsteinsson, A Hybrid Framework Integrating Mixed Integer Programming and Constraint Programming, in *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming - CP 2001*, Paphos, Cyprus, pp. 16–30 (2001).
36. E. P. K. Tsang, *Foundation of Constraint Satisfaction*, Academic Press, New York (1993).
37. W. Wolf, A Decade of Hardware/Software Codesign, *IEEE Computer*, **36**(4):38–43 (2003).