Communication-Aware Stochastic Allocation and Scheduling Framework for Conditional Task Graphs in Multi-Processor Systems-on-Chip

Emiliano Dolif, Michele Lombardi, Martino Ruggiero, Michela Milano and Luca Benini
DEIS, University of Bologna
V.le Risorgimento 2
40136 Bologna, Italy

ABSTRACT

The increasing levels of system integration in Multi-Processor System-on-Chips (MPSoCs) emphasize the need for new design flows for efficient mapping of multi-task applications onto hardware platforms. Even though data-flow graphs are often used for pure data-streaming, many realistic applications can only be specified as conditional task graphs (CTG). The problem of allocating and scheduling conditional task graphs on processors in a distributed real-time system is NP-hard. The first contribution of this paper is a complete stochastic allocation and scheduling framework, where an MPSoC virtual platform is used to accurately derive input parameters, validate abstract models of system components and assess constraint satisfaction and objective function optimization. The optimizer implements an efficient and exact approach to allocation and scheduling based on problem decomposition. The original contributions of the approach appear both in the allocation and in the scheduling part of the optimizer. For the first, we propose an exact analytic formulation of the stochastic objective function based on the task graph analysis, while for the scheduling part we extend the timetable constraint for conditional activities. The second contribution of this paper is the introduction of a software library and API for the deployment of conditional task graph applications onto Multi-Processor System-on-Chips. With our library support, programmers can quickly develop multi-task applications which will run on a multi-core architecture and can easily apply the optimal solution found by our optimizer. The proposed programming support manages OS-level issues, such as task allocation and scheduling, as well as task-level issues, like inter-task communication and synchronization.

Categories and Subject Descriptors

D.1.0 [**Software**]: Programming Techniques; D.2.0 [**Software Engineering**]: [General]

EMSOFT'07, September 30–October 3, 2007, Salzburg, Austria.

General Terms

Algorithms, Performance, Design.

Keywords

Multimedia dataflow streaming, allocation, scheduling.

1. INTRODUCTION

As technology scales toward deep sub-micron, the integration of a complete system consisting of a very large number of IP blocks on the same silicon die is becoming technically feasible, so embedded system designs which include more than one processor are becoming more and more common [27, 30, 24]. Future multi-processor system-on-chip (MPSoCs) hosting a huge number of processors will guarantee high computational power thanks to their massive parallelism, but at the cost of a more complicated parallel programming paradigm. If we consider that software running on multiprocessor must be high performance, real-time, and low power, moreover that consumer applications are characterized by tight time-to-market constraints and extreme cost sensitivity, incoming MPsoC platforms will lead to several and interesting challenges in software development: there is a clear need for new deployment technologies which address multi processing issues in embedded systems.

Even though data-flow graphs are often used for pure data-streaming applications, many realistic applications can only be specified as conditional task graphs. The problem of allocating and scheduling conditional task graphs on processors in a distributed real-time system is NP-hard.

Moving from these considerations, in this paper we present a novel framework for developing, allocating and scheduling conditional multi-task graphs on multi-processor systems-on-chip. We target a general template for distributed memory embedded systems where the communication architecture is becoming a critical component. Interaction of multiple traffic patterns on the system bus causes congestion and hence unpredictable communication latencies. Neglecting this behaviour in high level optimization tools for allocation and scheduling might lead to unacceptable deviations of real performance metrics with respect to predicted ones and to the violation of real-time constraints.

Moreover, the abstraction gap between high level optimization tools and standard application programming models can introduce other unpredictable and undesired behaviours. In optimization tools many simplifying assumptions are generally considered and neglecting these assumptions can gen-

erate unpredictable system-level interactions of many concurrent execution flows. In the application developing phase, programmers must be conscious about simplified assumptions taken into account in optimization tools. For instance, a communication or synchronization sub-optimal task implementation leads to reduced throughput and/or latency and has also energy implications, due to the higher occupancy condition for system resources.
Our allocation and scheduling framework is based on problem decomposition and deploys techniques mutuated from the Artificial Intelligence and the Operations Research community: the allocation subproblem is solved through Integer Programming while the scheduling one through Constraint Programming. More interestingly, the two solvers can interact with each other by means of no-good generation, thus building an iterative procedure which has been proven to converge producing the optimal solution.
We propose two main contributions in this field: the first concerns both the allocation and scheduling components. The objective function we consider in the allocation component depends on the allocation variables. Clearly, having conditional tasks, the exact value of the communication cost cannot be computed. Therefore our objective function is the expected value of the communication cost. We propose here to identify an analytic approximation of this value. The approximation is based on the Conditional Task Graph analysis for identifying two data structures: the activation set of a node and the coexistence set of two nodes. The approximation turns out to be exact and polynomial. Concerning the scheduling, we propose an extension of the time-table constraint for cumulative resources, taking into account conditional activities. The propagation is polynomial if the task graph satisfies a condition called *Control Flow Uniqueness* which is quite common in many conditional task graphs for system design.
The other main contribution of this paper is the introduction of a new methodology for multi-task application development. We propose a software library and APIs for the deployment of conditional task graph applications onto Multi-Processor System-on-Chips. With our library support, programmers can quickly develop multi-task applications which will run on a multi-core architecture and can easily apply the optimal solution found by our optimizer. The proposed programming support manages OS-level issues, such as task allocation and scheduling, as well as task-level issues, like inter-task communication and synchronization. We carried out its implementation with both high flexibility and performance in mind. 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. In multi-processor systems, we believe this validation phase is critical in order to check modelling assumptions and make sure that second-order effects and/or modelling approximations impair optimizer-predicted performance (e.g., a required throughput) only marginally below 10%.

2. RELATED WORK

The synthesis of system architectures has been extensively studied in the past. Mapping and scheduling problems on multi-processor systems have been traditionally tackled by means of Integer Linear Programming (ILP). In general, even though ILP is used as a convenient modelling formalism, there is consensus on the fact that pure ILP formulations are suitable only for small problem instances, i.e. task graphs with a reduced number of nodes, because of their high computational cost. An early example is represented by the SOS system, which used mixed integer linear programming technique (MILP) [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 [4].
The complexity of pure ILP formulations for general task graphs has led to the deployment of heuristic approaches. Heuristic approaches provide no guarantees about the quality of the final solution, and many times the need to bound search times limits their applicability to moderately small task sets. In [8] a retiminig heuristic is used to implement pipelined scheduling, while simulated annealing is used in [23]. A comparative study of well-known heuristic search techniques (genetic algorithms, simulated annealing and tabu search) is reported in [3]. Unfortunately, busses are implicit in the architecture, unlike in [10]. A scalability analysis of these algorithms for large real-time systems is introduced in [17]. Many heuristic scheduling algorithms are variants and extensions of list scheduling [9]. In general, scheduling tables list all schedules for different condition combinations in the task graph, and are therefore not suitable for control-intensive applications.
Constraint Logic Programming (CP) is an alternative approach to Integer Programming for solving combinatorial optimization problems [19]. The work in [28] 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. Both ILP and CP 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. However, the issue of communication between different modelling paradigms arises. One method is inherited from the Operations Research and is known as Benders Decomposition [5]: it is has been proven to converge producing the optimal solution. There are a number of papers using Benders Decomposition in a CP setting [31] [11] [16] [15].
[26] presents an approach leverages a decomposition of the problem in the context of MPSoC systems. The authors tackle the mapping sub-problem with IP and the scheduling one with CP. The work considers only pipelined streaming applications and does not handle conditional task graphs. In order to solve the problem of allocating and scheduling a general conditional task graph onto a MPSoC, the introductions of more complex problem models and cost functions, such as more complex subproblem relaxations and Benders cuts are needed.
In the system design community, the problem of allocating and scheduling a conditional multi-task application is extremely important and many researchers have worked extensively on it, mainly with incomplete approaches: for instance in [33] a genetic algorithm is devised on the basis of a conditional scheduling table whose (exponential number of) columns represent the combination of conditions in the CTG and whose rows are the starting times of activities that appear in the scenario. The number of columns

is indeed reasonable in real applications. The same structure is used in [18], which is the only approach that uses Constraint Programming for modelling the allocation and scheduling problem. Indeed the solving algorithm used is complete only for small task graphs (up to 10 activities). Besides related literature for similar problems, the Operations Research community has extensively studied stochastic optimization in general. The main approaches are: sampling [2] consisting in approximating the expected value with its average value over a given sample; the *l-shaped* method [20] which faces two phase problems and is based on Benders Decomposition [5]. The master problem is a deterministic problem for computing the first phase decision variables. The subproblem is a stochastic problem that assigns the second phase decision variables minimizing the average value of the objective function. A different method is based on the branch and bound extended for dealing with stochastic variables, [22].
The CP community has recently faced stochastic problems: in [32] stochastic constraint programming is formally introduced and the concept of solution is replaced with the one of *policy*. In the same paper, two algorithms have been proposed based on backtrack search. This work has been extended in [29] where an algorithm based on the concept of scenarios is proposed. In particular, the paper shows how to reduce the number of scenarios, maintaining a good expressiveness.

3. TARGET ARCHITECTURE

Our mapping strategy targets a general template for a message-oriented distributed memory architecture. The specific platform instance, conforming to the template, only determines the annotated values in the application task graph (cost for communication and execution times), which is an input to our framework. Therefore, the allocation and scheduling methodology we propose is not affected by specific design choices (e.g., the kind of processing unit, the bus architecture). The characteristics of the architectural template targeted by our optimization framework include: (i) support for message exchange between the computation tiles, (ii) availability of local memory devices at the computation tiles and of remote (i.e., non-local to the tiles, accessible through the system bus) storage devices for those program data that cannot be stored in local memories. 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. The architecture of the Cell processor closely matches the template we are targeting, because of its synergistic processing elements with local storage and of its support for message-based stream processing[7].

We deployed the model of an instance of this architectural template in order to prove the viability of our approach (see Fig. 1). The computation tiles are supposed to be homogeneous and consist of ARM 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 bus as system interconnect. A DMA engine is attached to each

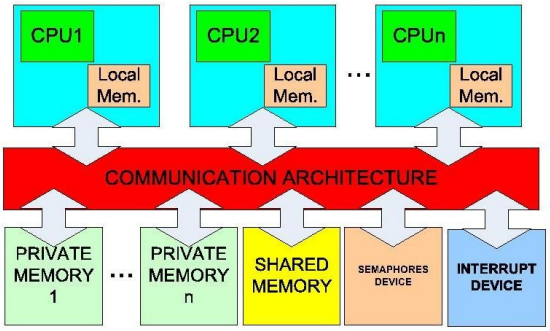

Figure 1: Message-oriented distributed memory architecture.

core, as presented in [13], allowing efficient data transfers between the local scratchpad and non-local memories reachable through the bus. The DMA control logic supports multichannel programming, while the DMA transfer engine has a dedicated connection to the scratch-pad memory allowing fast data transfers from or to it. In order to communicate each others, cores use non-cachable shared memory. For the synchronization among the processors, semaphore and interrupt facilities are used:

1. a core can send interrupt signals to each other using the hardware interrupt module mapped on in the global addressing space;

2. several cores can synchronize using the semaphore module that implements test-and-set operations.

Finally, each processor core has a private on-chip memory, which can be accessed only by gaining bus ownership. In principle, it could be also an off-chip memory. In any case, it has a higher access cost and can be used to store program operands that do not fit in scratch-pad memory. Optimal memory allocation of task program data to the scratch-pad 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 operating system called RTEMS [1]. Our implementation thus supports: (i) either processor or DMA-initiated memory-to-memory transfers, (ii) either polling-based or interrupt-based synchronization, and (iii) flexible allocation of the consumerŠs message buffer to the local scratchpad or the non-local private memory.

4. HIGH-LEVEL APPLICATION

Our methodology requires to model the conditional multitask application to be mapped and executed on top of the target hardware platform as a task graph with precedence constraints. A real-time requirement is specified, consisting of a minimum required throughput for the overall application. The target application to be executed on top of the hardware platform is the input to our algorithm. It is represented as a Conditional Task Graph. A CTG is a triple $\langle T, A, C \rangle$, where T is the set of nodes modeling generic tasks

(e.g. elementary operations, subprograms, ...), A the set of arcs modeling precedence constraints (e.g. due to data communication), and C is a set of conditions, each one associated to an arc, modeling what should be true in order to choose that branch during execution (e.g. the condition of a if-then-else construct). A node with more than one outgoing arc is said to be a *branch* if all arcs are conditional, a *fork* if all arcs are not conditional; mixed nodes are not allowed. A node with more than one ingoing arc is an *or-node* if all arcs are mutually exclusive, it is instead an *and-node* if all arcs are not mutually exclusive; mixed nodes are not allowed.
Since the truth or the falsity of conditions is not known in advance, the model is stochastic. In particular, we can associate to each branch a stochastic variable $\mathcal{B}$ with probability space $\langle C, \mathcal{A}, p \rangle$, where C is the set of possible branch exit conditions c, $\mathcal{A}$ the set of events (one for each condition) and p the branch probability distribution (in particular $p(c)$ is the probability that condition c is true).
We can associate to each node and arc an activation function, expressed as a composition of conditions by means of the logical operators $\wedge$ and $\vee$. We call it $f_i(X(\omega))$, where X is the stochastic variable associated to the composite experiment $\mathcal{B}_0 \times \mathcal{B}_1 \times ... \times \mathcal{B}_b$ (b = number of branches) and $\omega \in \mathcal{D}(\mathcal{B}_0) \times \mathcal{D}(\mathcal{B}_1) \times ... \times \mathcal{D}(\mathcal{B}_b)$ (i.e. ω is a scenario).
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 deadlines) are to be met.
Each node/task also has three kinds of associated memory requirements: **Program Data**: storage locations are required for computation data and for processor instructions; **Internal State**; **Communication queues**: the task needs queues to transmit and receive messages to/from other tasks, eventually mapped on different processors. Each of these memory requirement can be allocated either locally in the scratchpad memory or remotely in the on-chip memory.
Finally, the communication to be minimized counts two contributions: one related to single tasks, once computation data and internal state are physically allocated to the scratchpad or remote memory, and obviously depending on the size of such data; the second related to pairs of communicating tasks in the task graph, depending on the amount of data the two tasks should exchange.

5. MODEL DEFINITION

As already presented and motivated in [6], the problem is split into the resource allocation master problem and the scheduling sub-problem. The intuition behind our approach is to decompose the problem and exploit the structure of each component to chose the best algorithm to solve it.

5.1 Allocation Problem Model

The allocation problem can be stated as the one of assigning processing elements to tasks and storage devices to their memory requirements. First, we state the stochastic allocation model, then we show how this model can be transformed into a deterministic model through the use of existence and co-existence probabilities of tasks. To compute these probabilities, we propose two polynomial time algorithms exploiting the CTG structure. For the lack of space we do not explain these algorithms here, but they can be found in [21].

5.1.1 Stochastic integer linear model

Suppose n is the number of tasks, p the number of processors, and n_a the number of arcs. We introduce for each task and each PE a variable T_{ij} such that $T_{ij} = 1$ iff task i is assigned to processor j. We also define variables M_{ij} such that $M_{ij} = 1$ iff task i allocates its program data locally, $M_{ij} = 0$ otherwise. Similarly we introduce variables S_{ij} for task i internal state requirements and C_{rj} for arc r communication queue. X is the stochastic variable associated to the scenario ω. The allocation model, where the objective function is the minimization of bus traffic expected value, is defined as follows:

$$\min z = E(busTraffic(M, S, C, X(\omega)))$$

$$s.t. \quad \sum_{j=0}^{p-1} T_{ij} = 1 \qquad \forall i = 0, .., n-1 \tag{1}$$

$$S_{ij} \leq T_{ij} \qquad \forall i = 0, .., n-1, j = 0, .., p-1 \tag{2}$$

$$M_{ij} \leq T_{ij} \qquad \forall i = 0, .., n-1, j = 0, .., p-1 \tag{3}$$

$$C_{rj} \leq T_{ij} \qquad \forall arc_r = (t_i, t_k), r = 0, .., n_a - 1, j = 0, .., p-1 \tag{4}$$

$$C_{rj} \leq T_{kj} \qquad \forall arc_r = (t_i, t_k), r = 0, .., n_a - 1, j = 0, .., p-1 \tag{5}$$

$$\sum_{i=0}^{n-1} [s_i S_{ij} + m_i M_{ij}] + \sum_{r=0}^{n_a-1} c_r C_{rj} \leq Cap_j \qquad \forall j = 0, .., p-1 \tag{6}$$

Constraints (1) force each task to be assigned to a single processor. Constraints (3) and (4) state that program data and internal state can be locally allocated on the PE j only if task i runs on it. Constraints (5) and (6) enforce that the communication queue of arc r can be locally allocated only if both the source and the destination tasks run on processor j. Finally, constraints (7) ensure that the sum of locally allocated internal state (s_i), program data (m_i) and communication (c_r) memory cannot exceed the scratchpad device capacity (Cap_j). All tasks have to be considered here, regardless they execute or not at runtime, since a scratchpad memory is, by definition, statically allocated. Some symmetries breaking constraints have been added to the model.
The bus traffic expression is composed by two contributions: one depending on single tasks and one due to the communication between pairs of tasks.

$$busTraffic = \sum_{i=0}^{n-1} taskBusTraffic_i + \sum_{arc_r=(t_i,t_k)} commBusTraffic_r$$

where

$$taskBusTraffic_i = f_i(X(\omega)) \left[m_i(1 - \sum_{j=0}^{p-1} M_{ij}) + s_i(1 - \sum_{j=0}^{p-1} S_{ij}) \right]$$

$$commBusTraffic_r = f_i(X(\omega)) f_k(X(\omega)) \left[c_r(1 - \sum_{j=0}^{p-1} C_{rj}) \right]$$

In the *taskBusTraffic* expression, if task i executes (thus $f_i(X(\omega)) = 1$), then $(1 - \sum_{j=0}^{p-1} M_{ij})$ is 1 iff the task i program data is remotely allocated. The same holds for the internal state. In the *commBusTraffic* expression we have a contribution if both the source and the destination task execute ($f_i(X(\omega)) = f_k(X(\omega)) = 1$) and the queue is remotely allocated ($1 - \sum_{j=0}^{p-1} C_{rj} = 1$).

5.1.2 Transformation in a deterministic model

In most cases, the minimization of a stochastic functional, such as the expected value, is a very complex operation (even more than exponential), since it often requires to repeatedly solve a deterministic subproblem [20]. The cost of such a procedure is not affordable for hardware design purposes since the deterministic subproblem is by itself NP-hard. One of the main contributions of this paper is the way to reduce the bus traffic expected value to a deterministic expression. Since all tasks have to be assigned before running the application, the allocation is a stochastic *one phase* problem: thus, for a given task-PE assignment, the expected value depends only on the stochastic variables. Intuitively, if we properly weight the bus traffic contributions according to task probabilities we should be able to get an analytic expression for the expected value.
Now, since both the expected value operator and the bus traffic expression are linear, the objective function can be decomposed into task related and arc related blocks:

$$E(busTraffic) = \sum_{i=0}^{n-1} E(taskBusTraffic_i) + \sum_{arc_r=(t_i,t_k)} E(commBusTraffic_r)$$

Since for a given allocation the objective function depends only on the stochastic variables, the contributions of decision variables are constants: we call them $KT_i = \left[m_i(1 - \sum_{j=0}^{p-1} M_{ij}) + s_i(1 - \sum_{j=0}^{p-1} S_{ij})\right]$, and $KC_r = \left[c_r(1 - \sum_{j=0}^{p-1} C_{rj})\right]$. Let us call $p(\omega)$ the probability of scenario ω.
The expected value of each contribution to the objective function is a weighted sum on all scenarios. Weights are scenario probabilities.

$$E(taskBusTraffic_i) = \sum_{\omega\in\Omega} p(\omega) f_i(X(\omega)) KT_i = KT_i \sum_{\omega\in\Omega_i} p(\omega)$$

$$E(commBusTraffic_r) = \sum_{\omega\in\Omega} p(\omega) f_i(X(\omega)) f_k(X(\omega)) KC_r = KC_r \sum_{\omega\in\Omega_i\cap\Omega_k} p(\omega)$$

where r is the index of arc (t_i, t_j) and $\Omega_i = \{\omega \mid \text{task } i \text{ executes}\}$ is the set of all scenarios where task i executes. Now every stochastic dependence is removed and the expected value is reduced to a deterministic expression. Note that $\sum_{\omega\in\Omega_i} p(\omega)$ is simply the existence probability of node/task i while $\sum_{\omega\in\Omega_i\cap\Omega_k} p(\omega)$ is the coexistence probability of nodes i and k. To apply the transformation we need both those probabilities; moreover, to achieve an effective overall complexity reduction, they have to be computed in a reasonable time. We developed two polynomial cost algorithms to compute these probabilities.

5.2 Scheduling Problem Model

The scheduling subproblem has been solved by means of Constraint Programming. Since the objective function depends only on the allocation of tasks and memory requirements, scheduling is just a feasibility problem. Therefore we decided to provide a unique worst case schedule, forcing each task to execute after all its predecessors in any scenario. Tasks using the same resources can overlap if they are on alternative paths (under two mutually exclusive conditions). Tasks have a five phases behavior: they read all communication queues (INPUT), eventually read their internal state (RS), execute (EXEC), write their states (WS) and finally write all the communications queues (OUTPUT). Each task is modeled as a group of not breakable activities; the adopted schema and precedence relations vary with the type of the corresponding node (or/and, branch/fork). For the lack of space we do not explain these relations here, but they can be found in [21].
Each activity duration is an input parameter and can vary depending on the allocation of internal state and program data. The processing elements are unary resources: we modeled them defining a simple disjunctive constraint proposed in [18].
The bus, as in [26], is modeled as a cumulative resource, according with the so called "additive model", which allows an error less than 10% until bandwidth usage is under 60% of the real capacity. Computing the bus usage in presence of alternative activities is not trivial, since the bus usage varies in a not linear way and every activity can have its own bus view (see fig 2).

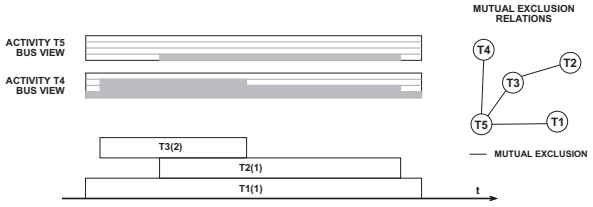

Figure 2: Activity bus view

Suppose for instance we have the five tasks of figure 2; activities **T1**, **T2**, **T3** have already been scheduled: the bus usage for each of them is reported between round brackets, while all the mutual exclusion relations are showed on the right. Let's consider activity **T4**, which is not mutually exclusive with any of the scheduled tasks. As long as only **T1** is present, the bus usage is 1. It becomes 1 + 2 = 3 when also activity **T3** starts, but when both **T1**, **T2** and **T3** execute the bus usage remains 3, since **T2** and **T3** are alternative. Thus the bus usage at a given time is always the maximum among all the combinations of not alternative running tasks. Furthermore, let's consider activity **T5**: since it is mutually exclusive with all tasks but **T2**, it only sees the bus usage due to that task. Therefore the bus view at a given time depends on the activity we are considering.
We modeled the bus creating a new global timetable constraint for cumulative resources and conditional tasks in the not preemptive case. The global constraint keeps a list of all known entry and exit points of activities: given an activity A, if $lst(A) \leq eet(A)$ then the entry point of A is $lst(A)$ and $eet(A)$ is its exit point (where lst stands for latest start time and so on).
Let A be the target activity: algorithm A3 (see below) scans the interval $[est(A), finish)$ checking the bus usage at all entry points (as long as $good = true$). If it finds an entry point with not enough bandwidth left it starts to scan all exit points ($good = false$) in order to determine a new possible starting time for activity A. If such an instant is found its value is stored ($lastGoodTime$) and the finish line is updated (step 4.2.2.2), then A3 restarts to scan other entry points, and so on. When the finish line is reached the algorithm updates $est(A)$ or fails. A3 has $O(a(c + b))$ complexity, where a is the number of activities, b the one of branches, c the number of conditions. The algorithm

can be easily extended to update also $let(A)$: we tried to do it, but the added filtering is not enough to justify the increased propagation time.

algorithm: Propagation of the cumulative resource constraint with alternative activities (A3)

1. $time = est(a), finish = eet(a)$
2. $latestGoodTime = time$
3. $good = true$
4. While $\neg\,[(good = false \wedge time > lst(a)) \vee (good = true \wedge time >= finish)]$:
   - 4.1. if $busreq(a) + usedBandwith > busBandwidth$:
     - 4.1.1. $time =$ next exit point
     - 4.1.2. $good = false$
   - 4.2. else:
     - 4.2.1. $time =$ next entry point
     - 4.2.2. if $good = false$:
       - 4.2.2.1. $lastGoodTime = time$
       - 4.2.2.2. $finish = max(finish, time + mindur(a))$
       - 4.2.2.3. $good = true$
5. if $good = true$: $est(a) = lastGoodTime$
6. else: $fail$

end

A3 is able to compute the bandwidth usage seen from each activity in $O(b+c)$ by taking advantage of a particular data structure we introduced, named Branch Fork Graph (BFG). For lack of space we suggest the reader look at [21]. The BFG makes it possible to compute bus usage in a very efficient way, by making direct use of the graph structure: if we only took into account the exclusion relations it would be an NP-hard problem. To have a polynomial time algorithm however the graph should satisfy a particular condition (called "Control Flow Uniqueness") which states that each "and" node must have a main ingoing arc, whose activation implies the activation of the other ingoing arcs. This is not a very restrictive condition since it is satisfied by every graph resulting from the natural parsing of programs written in a language such C++ or Java.

5.3 Benders Cuts and Subproblem Relaxation

Each time the master problem solution is not feasible for the scheduling subproblem a cut is generated which forbids that solution. Moreover, all solutions obtained by permutation of PEs are forbidden, too. Unfortunately, this kind of cut, although sufficient, is weak; this is why we decided to introduce another cut type, generated as follows: (1) solve to feasibility a single machine scheduling model with only one PE and tasks running on it; (2) if there is no solution the tasks considered cannot be allocated to any other PE. The cut is very effective, but we need to solve an NP-hard problem to generate it; however, in practice, the problem can be quickly solved. With the objective to limit iteration number (which strongly influences the solution method efficiency) we also inserted in the master problem a relaxation of the subproblem. This forbids the allocator to store in a single processor a set of non mutually exclusive tasks whose duration exceeds the time limit, and to assign memory devices in such a way that the total length of a track is greater than the deadline.

5.4 Computational Efficiency

We tested the method on two set of instances with a time limit for the solution process was 30 minutes: instances of the first group are only slightly structured, i.e. they have very short tracks and quite often contain singleton nodes; a second group of instances is completely structured (one head, one tail, long tracks).
The results of the tests on the first group are summarized in table 1. Instances are grouped according to the number of activities (acts); beside this, the table reports also the number of processing elements (PEs), the number of instances in the group (inst.), the instances which were proven to be infeasible (inf.), the mean overall time (in seconds). The solution times are of the same order of the deterministic case (scheduling of Task Graphs), which is a very good result, since we are working on conditional task graphs and thus dealing with a stochastic problem.
For a limited number of instances the overall solving time was high: the last column in the table shows the number of instances for which this happened, mainly due to the master problem (A), the scheduling problem (S) or the number of iterations (I). The solution time of these instances was not counted in the mean; in general it was greater than than thirty minutes.

acts	PEs	inst.	inf.	time	A/S/I
10-12	2	6	0	0.0337	0/0/0
13-15	2	8	1	0.5251	0/0/0
16-18	2-3	12	0	0.1091	0/0/0
19-21	2-3	14	1	0.1216	0/0/0
22-24	2-3	23	4	0.2336	0/0/0
25-27	2-3	16	3	1.7849	0/0/0
28-30	2-3	13	2	0.3331	0/1/0
31-33	3-4	4	2	0.3008	0/0/0
34-36	3-4	13	4	0.6840	0/0/0
37-39	3-4	7	0	1.5670	0/0/0
40-42	3-4	6	3	2.9162	0/0/0
43-45	3-4	6	1	5.3670	0/0/0
46-48	4-5	11	0	3.2719	1/2/0
49-51	4-5	11	1	1.9950	1/1/0
52-54	5-6	6	0	8.0000	1/1/0
55-67	6	8	0	2.2810	1/4/0

Table 1: Results of the tests on the first group of instances (slightly structured)

Although this extremely high solution time occurs with increasing frequency as the number of activities grows, it seems it is not completely determinated by that parameter: sometimes even a very small change of the deadline or of some branch probability makes the computation time explode.
In some cases, when the scheduler is the cause of inefficiency, this happens because of search heuristic: for some input graph topologies and parameter configurations the heuristic does not make the right choices and the solution time dramatically grows. Perhaps this could be avoided by randomizing the solution method and by using restart strategies [14].

The results of the second group of instances (completely structured) are reported in table 2. In this case the higher number of arcs (and thus of precedence constraints) reduces the time windows and makes the scheduling problem much more stable: no instance solution time exploded due to the scheduling problem. On the other hand the increased number of arcs makes the allocation more complex and the scheduling problem approximation less strict, thus increasing the number of iterations and their duration. In two cases we go beyond the time limit. We also ran a set of tests to verify the effectiveness of the cuts we proposed in section 5.3

acts	PEs	inst.	inf.	time	A/S/I
20-29	2	7	2	0.5227	0/0/0
30-39	2-3	6	0	1.7625	0/0/0
40-49	3	3	0	0.4380	0/0/0
50-59	3-4	7	0	1.1403	0/0/1
60-69	4-5	4	0	10.1598	0/0/0
70-79	4-5	4	0	88.9650	0/0/0
80-90	4-6	7	0	202.4655	0/0/1

Table 2: Result of the tests on the second group of instances (completely structured)

with respect to the basic cuts removing only the solution just found: table 3 reports results for a 34 activities instance repeatedly solved with a decreasing deadline values, until the problem becomes infeasible. The iteration number greatly reduces. Also, despite the mean time to generate a cut grows by a factor of ten, the overall solving time per instance is definitely advantageous with the tighter cuts.

mean time to gen. a cut					
basic case:				0.0074	
with relaxation based cuts (RBC):				0.0499	
	n. of iter.		**exec. time**		
deadline	**basic**	**RBC**	**case**	**RBC**	**result**
8557573	2	3	1.18	0.609	opt. found
625918	1	1	0.771	0.765	opt. found
590846	1	1	0.562	0.592	opt. found
473108	19	6	6.169	1.186	opt. found
464512	190	14	201.124	9.032	opt. found
454268	195	24	331.449	10.189	opt. found
444444	78	15	60.747	6.144	opt. found
433330	9	4	4.396	1.657	opt. found
430835	5	3	3.347	1.046	opt. found
430490	5	3	3.896	1.703	opt. found
427251	3	2	2.153	0.188	inf.

Table 3: Number of iterations without and with scheduling relaxation based cuts

Finally, to estimate the quality of the chosen objective function (bus traffic expected value), we tested it against an easier, heuristic technique of deterministic reduction. The chosen heuristic simply optimizes bus traffic for the scenario when each branch is assigned the most likely outcome; despite its simplicity, this is a particularly relevant technique, since it is widely used in modern compilers ([12]).
We ran tests on three instances: we solved them with our method and the heuristic one (obtaining two different allocations) and we computed the bus traffic for each scenario with both the allocations. The results are shown in table 4, where for each instance are reported the mean, minimum and maximum quality improvement against the heuristic method. Note that on the average our method always improves the heuristic solution; moreover, our solution seems to be never much worse then the other, while it is often considerably better.

			quality improvement		
instance	**activities**	**scenarios**	**mean**	**min**	**max**
1	53	10	4.72%	-0.88%	13.08%
2	57	10	2.59%	-0.11%	8.82%
3	54	24	12.65%	-0.72%	39.22%

Table 4: Comparison with heuristic deterministic reduction

6. EFFICIENT APPLICATION DEVELOPMENT SUPPORT

As already pointed out in previous sections, in optimization tools many simplifying assumptions are generally considered and the neglecting of these assumptions in software implementation can generate unpredictable and not desired system-level interactions and make the overall system error-prone. In this section we describe our new application development support. We propose an entire innovative framework to help programmers in software implementation. It is mainly composed by a generic customizable application template and a set of high-level APIs, which handle all the main possible issues in embedded parallel programming. Our programming support's facilities tackle both OS-level issues, such as task allocation and scheduling, as well as task-level issues, like inter-task communication and synchronization. The main goal of our development framework is the exact and reliable application' execution after the optimization step, and at the same time guarantees about high performance and constraint satisfaction.

6.1 Customizable Application Template

One of the main features of our development framework is the generic customizable application template. Thanks to this template, software developers can easily and quickly build their application infrastructure starting from a high level task and data flow graph. Once programmer has defined the high level features of the target application (in terms of tasks, tasks' dependencies and data communication flows) he can intuitively translate this representation into C-code using our facilities and library. More in details, users can specify the number of tasks included in the target application, their nature (e.g. branch, fork, or-node, and-node) and their precedence constraints (e.g. due to data communication), thus quickly drawing its CTG. Once programmer has build this application skeleton, he can focus onto the functionalities of the tasks, thus giving the main effort of his work only to the more specific and critic sections of the application.

6.2 OS-level and Task-level APIs

We implemented a series of APIs by which users can easily reproduce optimizer solutions, thus indirectly neglecting optimizer's abstractions and, at the same time, obtaining the needed application constraint satisfaction.
Once the target application has been implemented using our generic customizable template, programmer can allocate both tasks, program data and queues to the right hardware resources. Task can be quickly associated to the right core, program data and queues stored into the right memory resource, only easily configuring the init task of our template which at the booting time of the application will allocate and launch all these activities.
In order to reproduce the exact scheduling from the optimizer, we implemented a scheduling support APIs. Using this facility, programmer has only to specify the desired scheduling for every core, subsequently our middleware will handle it, using the system calls offered by the OS.
After the boot of the application, our framework sets to active only the first task in scheduling list, while the other ones are set to sleep state. In this way, we guarantee not desired task' preemption by the OS scheduler which can introduce more latency and errors in the reproduction of

the optimal scheduling order. After task has finished its execution, the active task is set to sleep releasing thus the cpu, while the subsequent task in the scheduling list is waked up changing its state to active.

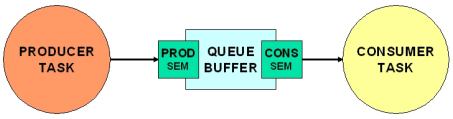

Figure 3: The structure of a queue.

Software support for efficient messaging is also provided by our set of high-level APIs. The communication and synchronization library abstracts low level architectural details to the programmer, such as memory maps or explicit management of hardware semaphores or interrupt signaling. Messages can be directly moved between scratch-pad memories. The structure of queue is shown in Fig. 3. A queue for the communication between a producer task and a consumer one is composed by a data queue and two semaphores. In order to send a message, a producer core writes in the message queue stored in its local scratch-pad 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 by the other processors.
As far as synchronization is concerned, when a producer intends to generate a message, it locally checks an integer semaphore which contains the number of free messages in the queue. If enough space is available, it decrements the semaphore and stores the message in its scratch-pad. Completion of the write transaction and availability of the message is signaled to the consumer by remotely incrementing its local semaphore. This single write operation goes through the bus. Semaphores are therefore distributed among the processing elements, resulting in two advantages: the 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 interrupt routine reactivates all tasks on the wait list.

If one task has got more than one input or output queue, our optimizer can specify the optimal reading/writing sequence from/to them. We aimed our framework in order to address this issue. This is a very important feature, since an optimal queue-usage ordering can boost performance and parallelism. Let's see for example Fig. 4 to better understand this statement. Fig. 4 shows a case in which six tasks are allocated to two different cores. Task T1 has to communicate with both T2 and T3, allocated to its own core, and with T4 allocated to a different processor. At starting time on CPU1 will be scheduled T1 and on CPU2 T4. While T1 will immediately start its execution, T4 has to wait data from T1 thus keeping CPU2 stalled. The T4's waiting time depends on the queue-fill ordering of T1: it will be shorter if T1 will give a high priority to queue C3. These options and optimizations can be selected by means of our high-level APIs.

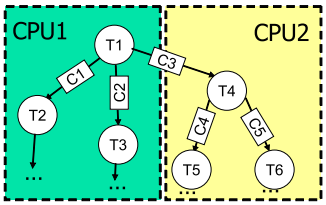

Figure 4: Optimal queue usage ordering: example.

7. METHODOLOGY

In this section we explain how to deploy our optimization framework in the context of a real system-level design flow. Fig. 5 shows a pictural overview of the overall application development methodology flow proposed. 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. The target application is pre-characterized and abstracted as a Conditional Task Graph. The task graph is annotated with computation time, amount of communication and storage requirements. However, not all tasks will run on the target platform: in fact, the application contains conditional branches (like if-then-else control structures) which will prevent the execution of some of them. Therefore, an accurate application profiling step is needed, from which we have a probability distribution on each conditional branch that intuitively gives the probability of choosing that branch during real future execution.
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

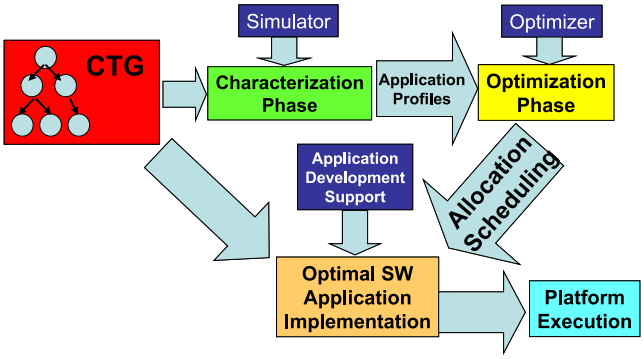

Figure 5: Application development methodology.

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 optimizerŠs solutions.
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.
After the optimization phase, we can build the optimal implementation of our target application using both the optimizer solution for the hardware platform (i.e. optimal allocation and scheduling) and the application development support (i.e. Customizable Application Template and OS-level and Task-level APIs).

8. EXPERIMENTAL RESULTS

We have performed two kinds of experiments, namely (i) comparison of simulated throughput with optimizer-derived values, and (ii) prove of viability of the proposed approach for real-life demonstrators (GSM, Software Radio).

8.1 Validation of optmizer 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 30 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.6

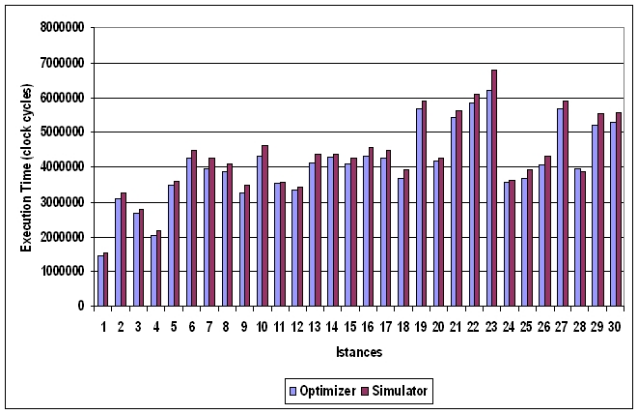

Figure 6: Difference in execution time

and Fig.7. Fig.6 shows the differences in execution time between the predicted one by the optimizer and the real one by the cycle accurate simulator. It can be noticed that the differences are marginal and we can point out that all the

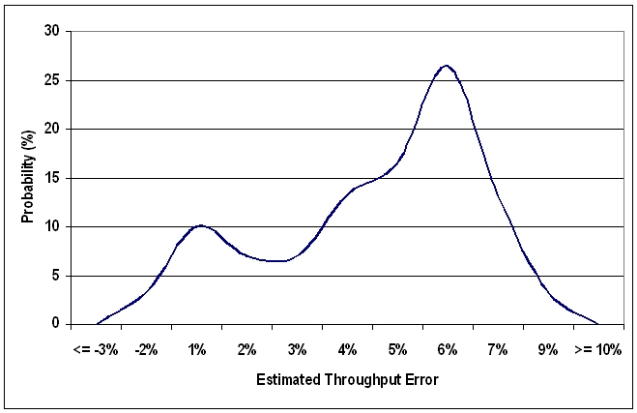

Figure 7: Probability for throughput differences

deadline constraints are satisfied. Fig.7 shows the probability for throughput differences between optimizer and simulator results. The average difference between measured and predicted values is 4.8%, with 2.41 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.

8.2 Demonstrators

The GSM application has been used to prove the viability of our approach. The source code has been parallelized into 10 task (see Fig.8), and each task has been pre-characterized by the virtual platform to provide parameters of task models to the optimizer. The time taken by the optimizer to come to a solution was 0.2 seconds. The validation process of the solution on the virtual platform running two cores showed an accuracy by 5.1% on throughput requirement.

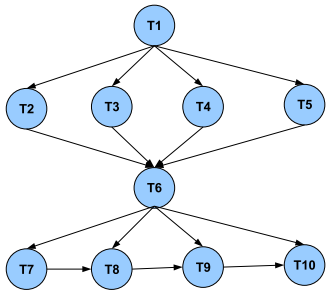

Figure 8: GSM encoder case study.

Our optimization framework was then applied to a Software Radio application. Fig.9 shows the obtained task graph. The target application computation kernel was partitioned into 10 stages, and the accuracy on throughput estimation was 6.33% with a solution found in 0.25 seconds.

9. CONCLUSIONS

We target allocation and scheduling of conditional multitask applications on top of distributed memory architectures with messaging support. We tackle the complexity of the

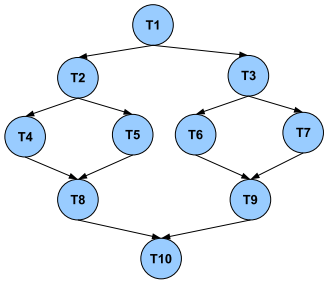

Figure 9: Software Radio application case study.

problem by means of decomposition and no-good generation, and introduce a software library and API for the reliable software deployment. Moreover, we propose an entire innovative framework to help programmers in software implementation and deploy a virtual platform to validate the results of the development framework and to check modelling assumptions of optimizer, showing a very high level of accuracy. Our methodology can potentially contribute to the advance in the field of software optimization and development tools for highly integrated on-chip multiprocessors.

10. REFERENCES

[1] Rtems home page, http://www.rtems.com.

[2] S. Ahmed and A. Shapiro. The sample average approximation method for stochastic programs with integer recourse. *In: Optimization on line.*, 2002.

[3] J. Axelsson. Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search strategies. In *CODES '97*.

[4] A. Bender. Milp based task mapping for heterogeneous multiprocessor systems. In *EURO-DAC '96/EURO-VHDL '96*.

[5] J. Benders. Partitioning procedures for solving mixed-variables programming problems. *Computational Management Science*, 2005.

[6] L. Benini, D. Bertozzi, A. Guerri, and M. Milano. Allocation and scheduling for mpsocs via decomposition and no-good generation. In *Proc. of the Int.l Conference in Principles and Practice of Constraint Programming. (2005)*.

[7] B.Flachs. A streaming processor unit for the cell processor. In *pp.134-135, ISSCC 2005*.

[8] K. S. Chatha and R. Vemuri. Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Trans. Very Large Scale Integr. Syst.*

[9] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *DATE '98*.

[10] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. In *Journal on Design Automation for Embedded Systems, 1997*.

[11] A. Eremin and M. Wallace. Hybrid benders decomposition algorithms in constraint logic programming. In *CP '01*.

[12] P. Faraboschi, J. Fisher, and C. Young. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, pages 1638–1659, 2001.

[13] P. Francesco, P. Antonio, and P. Marchal. Flexible hardware/software support for message passing on a distributed shared memory architecture. In *DATE '05*.

[14] C. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. *AIPS*, pages 208–213, 1998.

[15] J. N. Hooker. A hybrid method for planning and scheduling. In *CP*.

[16] V. Jain and I. E. Grossmann. Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS J. on Computing*.

[17] S. Kodase, S. Wang, Z. Gu, and K. G. Shin. Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers. In *RTAS '03*.

[18] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*

[19] K. Kuchcinski. Embedded system synthesis by timing constraint solving. In *IEEE Transactions on CAD*.

[20] G. Laporte and F. Louveaux. The integer l-shaped method for stochastic integer programs with complete recourse. *In: Operations Research Letters*, 1993.

[21] M. Lombardi and M. Milano. Stochastic allocation and scheduling for conditional task graphs in mpsocs. In *Technical Report 77 LIA-003-06*.

[22] V. I. Norkin, G. C. Pflug, and A. Ruszczyński. A branch and bound method for stochastic global optimization. *Math. Program.*, 1998.

[23] 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. *ipdps*.

[24] D. Pham and et al. The design and implementation of a first-generation cell processor.

[25] S. Prakash and A. C. Parker. Sos: synthesis of application-specific heterogeneous multiprocessor systems.

[26] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *DATE '06*.

[27] A. Semiconductor. Arm11 mpcore multiprocessor. In *http://arm.convergencepromotions.com/catalog/753.htm*.

[28] R. Szymanek and K. Kuchcinski. A constructive algorithm for memory-aware task assignment and scheduling. In *CODES '01*.

[29] S. A. Tarim, S. Manandhar, and T. Walsh. Stochastic constraint programming: A scenario-based approach. *Constraints, 2006*.

[30] C. Technologies. The multi-core dsp advantage for multimedia. In *Available at http://www.cradle.com/*.

[31] E. S. Thorsteinsson. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. In *CP '01*.

[32] T. Walsh. Stochastic constraint programming. *Proc. of ECAI*, 2002.

[33] D. Wu, B. Al-Hashimi, and P. Eles. Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems. In *Computers and Digital Techniques, IEE Proceedings. Volume 150 (5). (2003)*.