# A Constructive Algorithm for Memory-Aware Task Assignment and Scheduling

Radoslaw Szymanek
Dept. of Computer Science
Lund University, Sweden
e-mail: Radoslaw.Szymanek@cs.lth.se

Krzysztof Kuchcinski
Dept. of Computer Science
Lund University, Sweden
e-mail: Krzysztof.Kuchcinski@cs.lth.se

## ABSTRACT

This paper presents a constructive algorithm for memory-aware task assignment and scheduling, which is a part of the prototype system MATAS. The algorithm is well suited for image and video processing applications which have hard memory constraints as well as constraints on cost, execution time, and resource usage. Our algorithm takes into account code and data memory constraints together with the other constraints. It can create pipelined implementations. The algorithm finds a task assignment, a schedule, and data and code memory placement in memory. Infeasible solutions caused by memory fragmentation are avoided. The experiments show that our memory-aware algorithm reduces memory utilization comparing to greedy scheduling algorithm which has time minimization objective. Moreover, memory-aware algorithm is able to find task assignment and schedule when time minimization algorithm fails. MATAS can create pipelined implementations, therefore the design throughput is increased.

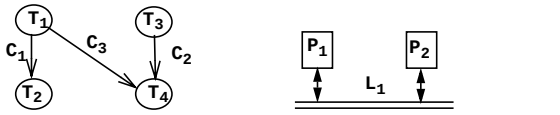
**keywords**: task scheduling, task assignment, memory constraints, constraint programming

## 1. INTRODUCTION

The data memory aspect in embedded systems is especially important for signal and image processing applications which process enormous amounts of data. A good utilization of memories is an essential issue in achieving low cost solutions with modest amount of memory components. Prior work on scheduling concentrated mostly on fulfilling deadlines without considering memory cost. Memory constraints are difficult to model, but neglecting them can produce considerable waste of memory capacity. In our approach, we consider memory constraints during task assignment and scheduling to reduce the amount of needed memory.

The algorithm, presented in this paper, is a part of the prototype design system **M**emory-**A**ware **T**ask **A**ssignment and **S**cheduling (MATAS). MATAS aims at helping a designer during the system level synthesis step. The decisions at this level have significant impact on the final product characteristics such as cost, performance, and time to market. A unified approach to the design synthesis step was presented in [3] and our work fits into task-level abstraction model defined there.

The complexity of task assignment and scheduling for heterogeneous architectures increases significantly when memory constraints are considered. These constraints define memory requirements for tasks and communications and therefore influence task assignment and scheduling decisions. New synthesis methods are required to cope with these constraints efficiently.




a) a task graph example b) an example of the system architecture

**Figure 1. Task data flow graph and target architecture**

The goal of the work presented in this paper is to develop an efficient algorithm for a system synthesis tool that accepts an input specification given as communicating tasks and system architecture. We assume that the input specification, described in C-like language, is compiled into an acyclic task graph annotated with estimates of execution time, code and data memory requirements. The task assignment and scheduling algorithm uses this task graph [8], generated from the original specification, as an input.

The rest of this paper is organized as follows. Section 2 motivates our work through an example. In section 3, we outline related work in this area. Section 4 defines our model of the architecture and application. MATAS system is briefly presented in section 5. Section 6 describes the flow of our algorithm. Section 7 presents experimental results. Finally, section 8 concludes the paper.

## 2. MOTIVATIONAL EXAMPLE

Consider, for example, a task graph depicted in Figure 1a consisting of four tasks $T_1$, $T_2$, $T_3$, and $T_4$. Data is transferred between these tasks by three communications $C_1$, $C_2$, and $C_3$. Each task requires 1 kb of code memory and 2 ms to execute on a processor. The data memory is used for storing incoming, outcoming data to/from the task as well as local data. Each task needs 4 kb of data

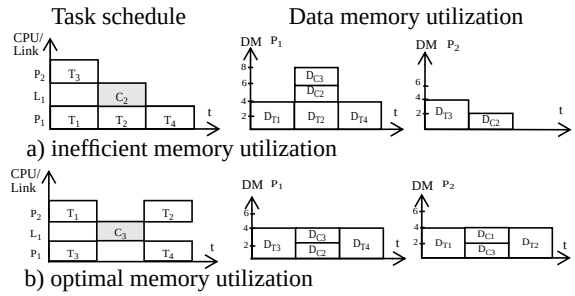



**Figure 2. Data memory requirements**

This work has been supported by the Foundation for Strategic Research, INTELECT program.

memory during its execution. Data transfers send 2 kb of data.

Consider a target architecture as depicted in Figure 1b. Two distinct schedules of a given task graph are depicted in Figure 2. The left most diagram in Figure 2a and b represents the task schedule while other diagrams represent data memory utilization. $D_T$'s rectangles represent the data memory used by tasks during their execution while $D_C$'s rectangles represent the data memory used to store data communicated between tasks. During interprocessor communication both processors have to allocate memory for the transferred data. For example, rectangle $D_{C2}$ in Figure 2a, which represents data transferred between tasks $T_3$ and $T_4$, appears in data memory usage for both processors. Please note that there is no $D_{C1}$ data rectangle in Figure 2a since the producer and consumer tasks are executed at the same processor one after the other.

The two schedules have the same length but use different amount of memory. The schedule depicted in Figure 2a has inefficient memory utilization. The amount of needed data memory (DM) on processor $P_1$ and $P_2$ is 8kb and 4kb respectively. The utilization of code memory is unbalanced. Processors $P_1$ and $P_2$ use 3kb and 1kb of code memory respectively (not shown in Figure 2). The schedule depicted in Figure 2b has the same length as the first one while memory usage is balanced and optimal. Both processors $P_1$ and $P_2$ use 2kb of code memory and 4kb of data memory. The importance of memory consideration was also indicated in [4, 9, 12, 13].

## 3. RELATED WORK

The work on memory aspects in system design has been studied by several authors. Their publications address different aspects of this research problem.

The research presented in [9] addresses the problem of embedded system synthesis under memory constraints. It uses a genetic algorithm to solve the synthesis problem. The authors assume that code and data memories are implemented using RAM memories. The code memory can be used as data memory and vice-versa. In our work, we made an assumption that ROM and RAM are used for implementing code memory and data memory respectively. The assumption of having RAM only implementation can be easily introduced into our work and it will make the synthesis problem simpler. In addition, we consider the actual placement of data in data memory and we avoid invalid solutions due to memory fragmentation problem. Genetic algorithms can have problems giving valid solutions when the synthesis problem is very constrained. Our constructive approach seems to tackle bigger problems even with additional extensions we have made.

Some work on task assignment and scheduling was also described in [12] and its previous version [11]. The former was extended by the inclusion of a simple memory model. This approach uses Mixed Integer Linear Programming which results in many inequalities and binary decision variables. Since the authors aimed at finding optimal solutions, the runtimes are prohibitively large even for the example consisting of nine tasks. This work does not address fragmentation problem and considers RAM memories only.

An other system synthesis approach which guarantees optimal solutions is described in [1]. The presented algorithm makes it possible to introduce multiple computations of the same task on several processing units in order to remove some communications from the buses. This approach imposes two restrictions on the target architecture. First, all buses have the same transmission rate. Second, all tasks assigned to ASIC's are executed sequentially. The global memory is used to store input and output data of the whole task graph only, the intermediate data produced and used by tasks cannot be stored there.

The research presented in [6] concentrates on storage estimation for data intensive applications. The estimators of data memory requirement are obtained through code analysis. This work can be used to optimize the data memory utilization within one task by changing the code of the task. This optimization can make it easier for our MATAS system to find valid task assignment and schedule for the whole task graph. This work is complementary to ours and targets a different abstraction level.

Our work concentrates on task assignment and scheduling for heterogeneous architectures with any type interconnection network. The data placement in data memory is also considered. In order to find a good solution we need to balance the utilization of the resources such as processing units, communication devices, code and data memory. Our work gives the possibility to create pipelined implementations. Up to our best knowledge there is no previous work which tackles all these aspects.

The work presented in this paper is based on the research presented in [7, 13]. Our approach uses Constraint Logic Programming (CLP) to represent the system synthesis problem by a set of finite domain variables and constraints imposed on these variables. The efficiency of CLP approach is compared with another approaches in favour of CLP [8]. Optimal solutions can be obtained for small problems, while large problems require use of heuristics. The system can minimize the design cost for a given execution time or vice versa.

The work presented in this paper removes the limitations imposed on the target architecture in [7] and adds memory constraints. We consider processors, ASIC's, buses, links, and memories. Local memory is divided into code and data memory. This research extends also work presented in [14] by considering actual data placement in data memory. Therefore we can avoid problems caused by memory fragmentation. The presented algorithm is a part of the prototype synthesis system MATAS. The system is interactive and a designer has freedom to influence the final design by making decisions concerning the final architecture, task assignment, and task scheduling. He can also provide a partial solution which will be used to obtain a final one. This makes it possible to guide the synthesis process in a clean manner and still use the full power of automatic synthesis methods. The prototype we developed produces good results for large designs as shown in section 7.

## 4. SYSTEM MODELING

In our approach, CLP is used to model the system architecture and the design problem. Since CLP is a relatively young programming framework, we will briefly present it here. A general introduction to CLP is given in [10], for example. A CLP program consists of constraints over finite domain variables (FDV's) and a search method. Each FDV is initially defined by a set of integer values that constitute its domain. Constraints specify relations among these variables. Constraint engines provide constraint consistency and propagation methods. Therefore restricting a domain of one FDV usually results in restricting domains of the other FDV's. Since CLP solvers over finite domain are not complete, they use search methods, such as branch-and-bound and heuristics, to find solutions.

We use the Constraint Handling in Prolog (CHIP) v. 5.2 sys-

tem. The CHIP system implements basic and global constraints. The basic constraints are equality, inequality or conditional constraints. The CLP modelling with basic constraints is similar to ILP modelling style, but the underlaying solvers are different. CLP frameworks introduced powerful modelling constraints, called global constraints, to limit number of constraints and provide advanced constraint consistency and propagation methods. The global constraints are based on extensive work conducted in operating research community. These constraints provide concise modelling, good time complexity bounds, and good constraint propagation. The global constraints encompass particular modelling problems. For example, they can impose restrictions on cumulative use of resources, rectangle placement or partitioning of graphs [2]. Modelling the problem using global constraints gives a clean and understandable description.

## 4.1. Architecture model

The target architecture, in our approach, consists of processing units, such as processors and ASIC's, and communication devices, such as buses and links. The processors have one local memory for code and one local memory for data. ASIC's have data memory only. The processing units can compute, send, or receive data concurrently. We will often refer to processor instead of processing units for both processors and ASIC's if it does not create confusions. The architecture is described by specifying processors, ASIC's, buses, links and their interconnection structure. Each bus or link is described by its cost and speed.

An ASIC consists of several parts. These parts operate independently making possible parallel execution of tasks. Since we define the maximum number of tasks which can be executed in parallel on the ASIC we can derive the ASIC's cost as needed for our synthesis tool. All tasks assigned to an ASIC have access to its local data memory.

The cost of the architecture is associated with processing units and communication devices. The cost of processing components includes the cost of their memory. This procedure of computing the cost suits the situation when a designer creates the architecture from off-the-shelf components which have all features fixed. Differentiating the cost of the processor with different amount or kind of memory can be modelled.

There is no restriction on the nature of the interconnection structure. The designer specifies possible connections between processing and communication devices. This specification is used to impose constraints on bus or link selection for transferring data between two communicating tasks, if they are executed on different processing units.

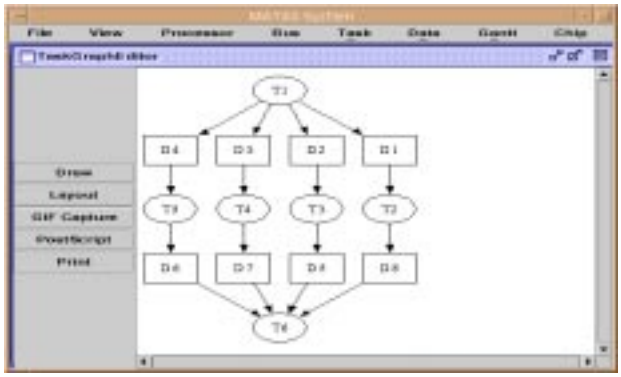


**Figure 3. Simple task graph example**

## 4.2. Problem definition

The functional description of a system is given as an acyclic task graph, as presented in Figure 3. Please note that this is different from a task graph presented in Figure 1, which was simplified for a purpose of motivational example.

The task graph nodes denoted as ellipses represent computational tasks. Each task is described by the following FDV's tuple:

$$T=(\tau, \rho, \delta, \mu_c, \mu_d, \pi) \quad (1)$$

where $\tau$ denotes the start time of the task execution, $\rho$ denotes the resource on which the task is executed, $\delta$ denotes the task execution time, $\mu_c$ and $\mu_d$ denote the amount of code and data memory needed for the execution of the task, and finally $\pi$ denotes the exact placement of data in the data memory. For example, task $T_1$ from Figure 3 can be modelled in the following way $\tau_{T1} :: 0..20$, $\rho_{T1} :: 1..2$, $\delta_{T1} :: 1..2$, $\mu_{cT1} :: 1..2$, $\mu_{dT1} :: 3..4$, $\pi :: 0..100$.

The execution time, code memory and data memory required by the task depends on the processor. The tasks must be always scheduled on one of the processing units. This is modelled by imposing constraints which define finite relations between FDV's of (1) representing different tasks [7].

The task graph nodes denoted as rectangles represent data transfers between tasks. Each data is described by a tuple of FDV's:

$$C=(\tau_d, \rho_d, \delta_d, \alpha, \pi_p, \pi_c) \quad (2)$$

where $\tau_d$ denotes the start time of the communication, $\rho_d$ denotes the resource which is used for transferring data, $\delta_d$ denotes the duration of the communication, $\alpha$ denotes the amount of the transferred data, $\pi_p$ and $\pi_c$ denotes the exact placement of data in memory of the processing units which execute producer and consumer tasks.

### *4.2.1. Basic constraints*

The FDV's of (1) and (2) are constrained. In most cases, the constraints imposed on these FDV's are basic constraints since they involve small number of FDV's and lack the global view of the problem. The basic constraints ensure that the FDV's are consistent. The basic constraint presented below ensures that the consumer task does not start before the end of the data communication to this task.

Each data node in a task graph imposes a constraint on an execution order of its producer and consumer tasks. There are two possible scenarios for transferring data between two communicating tasks. In the first scenario, tasks are executed on different processors. In this case, the communication must be assigned to and scheduled on a communication device. This case is defined by a conjunction of two following inequality constraints:

$$\tau_p + \delta_p \le \tau_d \quad \wedge \quad \tau_d + \delta_d \le \tau_c \quad (3)$$

where task $T_p$ sends data to task $T_c$ using communication $C_d$.

In the second scenario, both communicating tasks are executed on the same processor and they communicate using the local memory of the processing unit. In this case, the previous constraints reduce to the following one $\tau_p + \delta_p \le \tau_c$, since $\delta_d$ equals zero.

### *4.2.2. Global constraints*

The important shortcoming of basic constraint is their local perspective. The global constraints express a global view of the problem. We use global constraints mostly for ensuring that resources like time, code memory, and data memory are not overused. The main constraint used in this framework is diffn/1 [2, 7].

The diffn/1 constraint takes as an argument a list of n-dimensional rectangles and assures that for each pair of *i*, *j* (*i*≠*j*) of n-dimensional rectangles, there exist at least one dimension *k* where *i* is after *j* or *j* is after *i*. The n-dimensional rectangle is defined by a tuple [$O_1$, ..., $O_n$, $L_1$, ..., $L_n$], where $O_i$ and $L_i$ are respectively called the origin and the length of the n-dimensional rectangle in *i-th* dimension. Obviously the diffn/1 constraint can be used to express requirements for packing and placement problems but in our approach we will use it for defining constraints for scheduling and resource binding.

Two dimensional rectangles are used in our approach to represent tasks in processor/time space as well as memory placement in memory address/time space. These constraints represent data memory placement and Gantt diagrams for tasks and communications see Figure 6. Using global constraints we are able to estimate the amount of needed data memory early in the design phase. The estimates developed for our algorithm use constraints and increase the chance of finding a solution for which the maximal usage of data memory does not exceed the available memory size.

# 5. MATAS

The MATAS prototype system was implemented in Java and Constraint programming language CHIP v5.2. CHIP is used for modelling and constraint solving. The user interface was developed in Java. During synthesis the designer can express different restrictions and constraints on the final design. The system allows to constrain all FDV which constitute the model. Therefore timing constrains for task $T_i$ can be expressed by restricting $\tau_i$. The constrains on assignment of task $T_i$ can be expressed by restricting $\rho_i$.

MATAS prototype system gives the choice to create pipelined implementation of the task graph. The designer has to specify the number of pipeline stages, the maximal latency of a pipeline stage as well as initiation rate. In functional pipelining, we assume that the same tasks from different pipeline stages execute on the same processor. Thus the code memory does not increase.

# 6. TASK ASSIGNMENT AND SCHEDULING ALGORITHM

The task assignment and scheduling algorithm presented in this paper takes into account memory constraints. The general flow of the algorithm is depicted in Figure 4.

The proposed constructive algorithm balances the usage of the code memory, data memory, and time. It copes with code memory, data memory, and timing constraints. Our algorithm tries first to schedule tasks from the critical path until the estimates of data memory usage is below the memory size. When the estimate of future utilization of data memory exceeds memory size, our algorithm chooses tasks which will decrease the estimated data memory usage. The actual data memory utilization depends on future decisions regarding the schedule. We use two estimation methods to estimate the peak of data memory requirement on all processors.

The algorithm selects in each iteration a single task $T_i$ from tasks with all preceding tasks already scheduled. At this stage we choose a task from the critical path to reduce the schedule length.

$$c_{data} = \begin{cases} \dfrac{\Delta d_i}{ProcAM_n} & if \quad \Delta d_i > 0 \wedge \dfrac{ProcAM_n}{ProcM} < \dfrac{2}{3} \\ 0 & if \quad otherwise \end{cases} \tag{4}$$

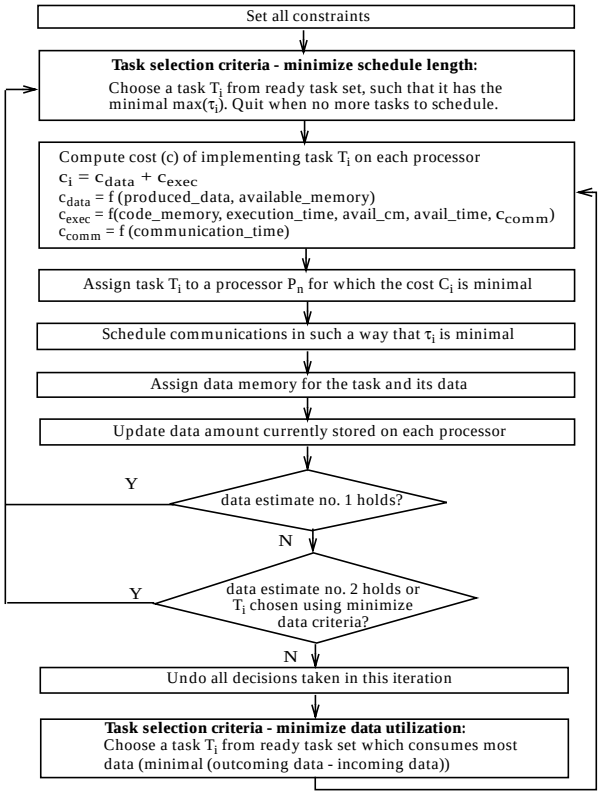



**Figure 4. Algorithm for task assignment and scheduling**

This decision is justified by the fact that, in general, delaying the execution of not urgent tasks for the favour of the tasks which belong to the critical path, decreases the schedule length. However, it usually increases data memory requirements as a side effect.

The second step computes the cost of implementing task $T_i$ on each processor. The implementation cost $c_i$ consists of two terms. The $c_{data}$ term, depicted in (4), depends on $\Delta d_i$, which is computed as a difference between data produced and consumed by task $T_i$. The $c_{data}$ cost depends also on the amount of currently available data memory on the processor ($ProcAM_n$), and the size of processor memory (ProcM). The communication cost $c_{comm}$, depicted in (5), represents the average time of interprocessor communication needed. We use minimum, $min(\delta_i)$, and maximum communication duration time, $max(\delta_i)$, since the incoming communication $C_i$ is not assigned yet and actual communication time depends on the assigned communication device.

$$c_{comm} = \sum_{c_i} \frac{min(\delta_i) + max(\delta_i)}{2} \tag{5}$$

Finally, The implementation cost $c_{exec}$ requires the following measures to be computed.

Two kinds of measures, UCM (utilization of code memory) and UTS (utilization of processor time slots) are general and describe the current utilization of the whole architecture. UCM and UTS use two lower bounds: the used amount of code memory (LCM) and the processor time units (LTS).

UCM = LCM/PCM, where PCM - processor code memory (6)

UTS = LTS/PTS, where PTS - processor time units (7)

Ind = UTS - UCM (8)

We also compute the $ACM_{in}$ and $ATS_{in}$ measures. They denote the available code memory and available time units on processor $P_n$, after assigning task $T_i$ to processor $P_n$. These two kinds of measures are used when computing the cost $c_{exec}$ of implementing task $T_i$ on processor $P_n$. The cost function uses, in addition, the amount of code memory ($C_{in}$) needed to execute task $T_i$ on processor $P_n$ and $D_{in}$ which represents the time needed to execute task $T_i$ on processor $P_n$ plus cost $c_{comm}$.

The execution cost formula is depicted in Figure 5. In case (10), when $Ind < -L_1$, the code memory is used much more than the processor time and, therefore, only the code memory contributes to cost $c_{exec}$. The algorithm should minimize further increase in code memory utilization. On the other hand, when value Ind is greater than $L_1$, the algorithm aims at minimizing further increase of processors utilization. When the utilization of processor time and code memory is balanced, (12), both terms are taken into consideration with the same weight. The remaining cases, (11) and (13), describe situation when one of the resources is slightly overused. To counteract this, the weight of the other resource is decreased. The algorithm selects the processor $P_n$ for implementation of task $T_i$ if the related implementation cost, $c_i$, is lowest.

After assigning task $T_i$ to processor $P_n$, the task is scheduled. The algorithm performs branch-and-bound search to find earliest possible start time of task $T_i$. During this search, the communications are also scheduled. Finally, data memory for incoming data as well as data memory for task execution is assigned.

The next algorithm step updates the amount of currently stored data on the processor. This takes into account data which were produced by all already scheduled tasks, but not yet consumed. This is our first estimate of used data memory. The estimate is depicted in (9), where $S_n$ is a set of tasks scheduled on processor

$$\varepsilon_1(P_n) = \sum_{\forall T_i \in S_n, T_j \notin S} d_{ij} \quad (9)$$

$P_n$, S is a set of already scheduled tasks, tasks $T_j$ are direct successors of task $T_i$, and data $d_{ij}$ is amount of data communicated between $T_i$ and $T_j$. This estimate is very fast to compute but inaccurate, since it does not take into account the data memory needed for task execution and additional data needed during interprocessor communication. However, if the amount of needed data memory on each processor is less than a certain limit we can proceed and start next iteration of the algorithm.

When this simple estimate does not hold, a more accurate and time consuming estimate is computed. It uses the global constraint diffn/1 to assess the required data memory. During the algorithm execution, only some of the rectangles in diffn/1 constraint have fixed size and placement. For “data rectangles” which have not fixed placement and size we assume the biggest possible size. If the global constraint does not fail immediately with this pessimistic assumption, we can expect that there exist data memory placement. The Gantt diagram on the right in Figure 6 is a diffn/1 constraint viewer and it shows the data memory requirements after the algorithm has finished. Therefore all rectangles have fixed size and exact amount of needed data memory as well as data memory placement is known.

$$c_{exec} = \begin{cases} \frac{C_{in}}{ACM_{in}} & \text{if} \quad -1 < Ind < -L_1 \quad (10) \\ \frac{C_{in}}{ACM_{in}} + \frac{D_{in}}{ATS_{in}} \times (1 - |Ind|) & \text{if} \quad -L_1 \le Ind < -L_2 \quad (11) \\ \frac{C_{in}}{ACM_{in}} + \frac{D_{in}}{ATS_{in}} & \text{if} \quad -L_2 \le Ind \le L_2 \quad (12) \\ \frac{C_{in}}{ACM_{in}} \times (1 - |Ind|) + \frac{D_{in}}{ATS_{in}} & \text{if} \quad L_2 < Ind \le L_1 \quad (13) \\ \frac{D_{in}}{ATS_{in}} & \text{if} \quad L_1 < Ind < 1 \quad (14) \end{cases}$$

where $L_1$ and $L_2$ are constants and are equal 0.16 and 0.08 respectively

**Figure 5. Execution cost**

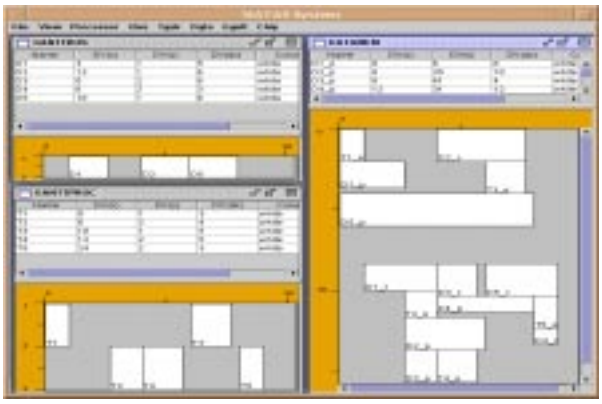


**Figure 6. Gantt diagrams**

If the second pessimistic estimate fails we have to roll back all decisions taken in this iteration and choose a different task $T_i$. Task $T_i$ is chosen now according to a different criteria. A task which consumes most data and, thus, reduces the data memory usage is selected. In most cases, this task will not belong to the critical path, so the schedule length will increase. With the new task $T_i$ the algorithm proceeds and finds an appropriate task assignment and schedule. The algorithm continues then by selecting tasks from the critical path.

# 7. EXPERIMENTAL RESULTS

We compare our memory-aware algorithm, which is multiobjective, to a greedy scheduling algorithm. The greedy algorithm always chooses a task from the critical path and assigns it to a processor which can finish its execution earliest. Therefore, this algorithm does not care about data and code memory usage. The greedy algorithm is one objective algorithm. Our greedy algorithm is very good in minimizing the deadline since it benefits from the use of global constraints available in the CHIP solver.

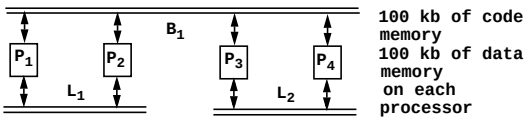



**Figure 7. Architecture used in random experiments**

First, we evaluated our system using five sets of experiments of random tasks graphs generated by TGFF v. 2.0 [5]. The graphs were scheduled on the architecture presented in Figure 7. To be able to repeat experiments we provide the options used for graph generation in Figure 8. An experiment set consists of 22 graphs which were generated using seed from 1 to 22. In the first experiment, we used the greedy algorithm. The values presented in Table 1 are averages over all graphs. The greedy scheduling algorithm was able to give short execution time due to unlimited amount of resources. In addition, it yields more unbalanced proc-

**Table 1: Experimental results (random task graphs)**

| Experiment setup | | | | Results | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Each CPU | | | Data Mem. | | Code Mem. | | |
| Exp. | algorithm | data memory size [kB] | code memory size [kB] | Deadline [ms] | utilization [kB] | peak [kB] | utilization [kB] | peak [kB] | # solutions |
| 1 | greedy | 100 | 100 | 171 | 243 | 78 | 319 | 98 | 22 |
| 2 | greedy | 75 | 100 | 182 | 233 | 68 | 310 | 96 | 8 |
| 3 | greedy | 100 | 75 | 169 | 257 | 79 | 294 | 75 | 4 |
| 4 | greedy | 75 | 75 | - | - | - | - | - | 0 |
| 5 | memory | 75 | 75 | 295 | 222 | 64 | 247 | 65 | 22 |

essor usage. The data memory peek was 18% higher and code memory peek was 33% higher than the figures obtained with our approach (exp. 5). In the second experiment, we decreased the amount of available data memory. The greedy algorithm fails to find a solution in 14 cases. The third experiment is done for an architecture with lower amount of code memory. For this setting, the greedy algorithm fails to give a solution in 18 cases. The comparison between greedy scheduling algorithm and memory-aware algorithm is more fair in the case where the resources code and data memory are constrained. The fourth experiment shows that the greedy scheduling is not able to find any solutions when both the code and data memory size is decreased. The memory-aware algorithm finds solutions to all graphs for reduced memory size as presented in the fifth experiment. It is much more robust in the case when we do not have an unlimited amount of memory.

We also applied our memory algorithm to a complex image processing application based on CCITT recommendation H.261 presented in [1]. For this example, we present results obtained when using algorithmic pipelining in Table 2. The pipeline implementation was constrained by defining the latency of one application iteration to be at most 4000 clocks and an activation period to be 1000 clocks. We assumed that the computation of tasks require 50 data memory units for execution. We also assumed that establishing communication with the environment takes little time on the universal processor. The experiment no. 3 shows that using our memory-aware algorithm we decrease the memory size by 16% and get only 1% deadline increase.

# 8. CONCLUSIONS

This paper presents a constructive algorithm for a task assignment and scheduling under memory constraints. The algorithm gives valid task assignments and schedules fulfilling all constraints, including memory constraints. Since the peak of data memory usage depends on task assignment, task schedule, and communication schedule, incorporating timing constraints with memory constraints results in better resource utilization. The experimental results show that data memory should be taken into account during system level synthesis to avoid inefficient and costly designs. Our algorithm provides good results for large, randomly generated task graph examples and also for a real-life example.

```
tg_cnt 1 seed ? task_degree 2 2 task_cnt 80 1 task_unique true
period_mul 1, 1
for tasks exec_time 4 2 1, code 4 2 1, data 3 1 1
for data data_amount 5 2 1
```

**Figure 8. TGFF options**

**Table 2: Experimental results (H.261 algorithm)**

| Exp. # | algorithm | Pipeline Degree | Deadline | Average Execution Time | Σ Data Memory | Δ Time | ΔData Memory |
|---|---|---|---|---|---|---|---|
| 1 | both | 1 | 2871 | 2871 | 2683 | - | - |
| 2 | greedy | 4 | 6743 | 1686 | 3812 | 0 | 0 |
| 3 | memory | 4 | 6781 | 1696 | 3259 | 1% | -16% |

# 9. REFERENCES


[1] Bender, A., "Design of an optimal loosely coupled heterogeneous multiprocessor system", *European Design and Test Conference, 1996. ED&TC; Proceedings*, 1996, Page(s): 275 -281

[2] COSYTEC, CHIP, System Documentation, 1996

[3] Catthoor, F.; Verkest, D.; Brockmeyer, E., "Proposal for unified system design meta flow in task-level and instruction-level design technology research for multimedia applications", *System Synthesis, 1998. Proceedings. 11th International Symposium on*, 1998, Page(s): 89 -95

[4] Danckaert, K.; Catthoor, F.; De Man, H., "System level memory optimization for hardware-software co-design", *Hardware/Software Codesign, 1997. (CODES/CASHE '97), Proceedings of the Fifth International Workshop on*, 1997, Page(s): 55 -59

[5] Dick, R.P.; Rhodes, D.L.; Wolf, W., "TGFF: task graphs for free", *Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on* , 1998, Page(s): 97 -101

[6] Kjeldsberg, P.G.; Catthoor, F.; Aas, E.J., "Storage requirement estimation for data intensive applications with partially fixed execution ordering", *Hardware/Software Codesign, CODES 2000. Proceedings of the Eighth International Workshop on*, Page(s): 56 -60

[7] Kuchcinski, K., "Embedded system synthesis by timing constraints solving", *System Synthesis, 1997. Proceedings., Tenth International Symposium on*, 1997, Page(s): 50 -57

[8] Kuchcinski, K., "Integrated resource assignment and scheduling of task graphs using finite domain constraints", *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, 1999, Page(s): 772 -773

[9] Madsen, J.; Bjorn-Jorgensen, P., Embedded system synthesis under memory constraints, *Hardware/Software Codesign, CODES'99, Proceedings of the Seventh International Workshop on*, 1999, Page(s): 188 -192

[10] Marriot K. and Stuckey P. J., *Programming with Constraints - An Introduction*, The MIT Press, ISBN 0-262-13341-5

[11] Prakash S. and A. C. Parker A. C., SOS: *Synthesis of Application-Specific Heterogeneous Multiprocessor Systems*, Parallel and Distributed Computing, pp. 338-351, 1992

[12] Prakash S. and Parker A. C., *Synthesis of Application-Specific Multiprocessor Systems Including Memory Components*, VLSI Signal Processing, 1994

[13] Szymanek, R.; Kuchcinski, K., "Design space exploration in system level synthesis under memory constraints", *EUROMICRO Conference, 1999. Proceedings. 25th, Volume: 1*, 1999, Page(s): 29 -36 vol.1

[14] Szymanek, R.; Kuchcinski, K., "Task assignment and scheduling under memory constraints", *Euromicro Conference, 2000. Proceedings of the 26th, Volume: 1*, 2000, Page(s): 84 -90 vol.1