Task Assignment and Scheduling under Memory Constraints

Radoslaw Szymanek and Krzysztof Kuchcinski

Dept. of Computer Science
Lund University
P.O. Box 118, SE 221 00, Lund, Sweden

e-mail: Radoslaw.Szymanek@cs.lth.se

Neither this paper nor any version close to it has been or is being offered elsewhere for publication of this paper. If accepted, the paper will be made available in Camera-ready forms by June 15th, 2000, and it will be personally presented at the EUROMICRO 2000 Conference by the author or one of the co-authors. The presenting author(s) will pre-register for EUROMICRO 2000 before the due date of the Camera-ready paper.

Task Assignment and Scheduling under Memory Constraints

Radoslaw Szymanek and Krzysztof Kuchcinski
Dept. of Computer Science
Lund University
P.O. Box 118, SE 221 00, Lund, Sweden
e-mail: Radoslaw.Szymanek@cs.lth.se

ABSTRACT

*Many DSP and image processing embedded systems have hard memory constraints which makes it difficult to find a good task assignment and scheduling which fulfill these constraints. This paper presents a new heuristic developed for task assignment and scheduling for such systems. These systems have also a large number of constraints of different nature, such as cost, execution time, memory capacity and limitations on resource usage. The heterogeneous constraints require new synthesis methods which will take them into account during searching for a valid solution. The heuristic presented in this paper is a part of the CLASS system (**C**onstraint **L**ogic Progr**a**mming based **S**ystem **S**ynthesis).*

Keywords

task scheduling, constraint programming, memory constraints.

1. INTRODUCTION

The complexity of the embedded systems grows constantly. This causes shifting to higher levels of abstractions of digital systems design, as well as need for the multiprocessor heterogeneous systems. These systems have a potential to achieve superior results in terms of cost and performance over homogenous systems since they can match the design requirements more closely. During the system level synthesis designer decides what are the main components of the system. The decisions at this level have enormous impact on the final product characteristics such as cost, performance, and time to market. Thus the responsibility of a designer is to perform fast exploration of different design trade-offs and choose an architecture which suits the problem best.

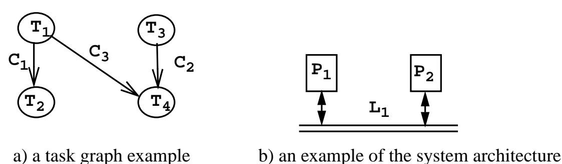
Embedded systems synthesis is usually decomposed into a chain of problems which must be solved to obtain a final solution. The first task for a designer is to decide what components should be used in the target architecture. Afterwards, the assignment of all tasks into the selected components is done. The last step is scheduling of the tasks on the processing units and the data transfers on the communication devices. The decisions taken during these steps are not independent making the process of finding (near) optimal design for industrial problems very difficult. The goodness of the final design is determined by all the decisions.

The complexity of the assignment and scheduling problem increases significantly when memory constraints are added. New synthesis methods capable of dealing with increased complexity are required. These methods should, in addition, provide good quality results and incorporate designer knowledge through interactions with synthesis methods.

Data memory aspect of the embedded systems is specially important in image processing and DSP applications which process enormous amounts of data. Memory constraints are difficult to model but neglecting them can produce considerable waste of memory components. A good utilization of memories is a key issue in achieving low cost solutions with modest amount of memory components. In our approach, we consider data memory constraints during task assignment and scheduling and therefore we are able to reduce the amount of needed memory.

The goal of this work is to develop system synthesis system which accepts an input specification given as communicating tasks. From this description the system will generate a final system architecture together with task assignment and scheduling. We assume that the input specification, described in C-like language, is compiled into a task graph which is synthesized. In this paper, we present the task assignment and scheduling heuristic. The input to this heuristic is an acyclic task graph [9] generated from the original specification.

Consider, for example, a task graph depicted in Figure 1a. There are four tasks T_1, T_2, T_3, and T_4. Data is transferred between those 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. All tasks need 4 kB of data memory which is denoted by D_t's rectangles in Figure 2. Data transfers send 2 kB of data. The data produced by the tasks need to be stored also in data memory which is denoted by



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

Figure 1. Task data flow graph and target architecture

D_c's rectangles in Figure 2. Note that during interprocessor communication both processors have to allocate memory for the transferred data.

Figure 2 depicts four distinct schedules. The last three are distributed ones implemented on the architecture presented in Figure 1b. The same schedule length has been obtained for distributed schedules but different memory usage is achieved. The distributed schedule depicted in Figure 2b has very unbalanced and inefficient memory utilization. The amount of needed data memory on processor P_1 and P_2 is 8kB and 4kB respectively. The utilization of code memory is unbalanced, 3kB and 1kB for processor P_1 and P_2 respectively. In Figure 2c the schedule with balanced code memory utilization but still inefficient data memory utilization is presented. The schedule depicted in Figure 2d has the same schedule length as previous ones while memory usage is balanced and optimal, 2kB for code memory and 4kB for data memory on both processors P_1 and P_2. The importance of memory consideration was also indicated in [10, 13, 14].

The main contribution of this paper is the development of a new constructive heuristic which takes into account memory constraints during task assignment and scheduling.

In this paper, we focus on the heuristic which solves the problem of task assignment and scheduling under memory constraints. In section 2, we outline related work in this area. Section 3 defines our model of the architecture and the system. Section 4 describes the flow of the heuristic. Section 5 presents experimental results. Finally, section 6 concludes the paper.

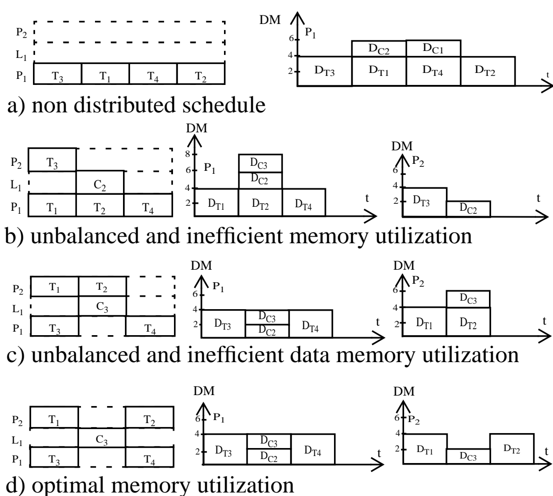


Figure 2. Data memory requirements

2. RELATED WORK

The synthesis problem encompasses a number of problems which can be studied separately. In our work, we try to integrate these problems. The research presented in [10] addresses similar problem of embedded system synthesis under memory constraints. It uses a genetic algorithm to solve the synthesis problem. This algorithm is iterative and stochastic. The authors assume that code and data memory 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 in our work and it will make the synthesis problem simpler. We believe that genetic algorithms can have problems giving a valid solution when the synthesis problem is very constrained and therefore we have developed a constructive heuristic. It seems that our heuristic has much shorter execution time than the genetic algorithm. An important difference is hardware sharing capability in our approach. We are able to model ASIC's hardware which is shared by different tasks.

The similar work was also described in [13] and its previous version [12]. The newer version 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. The architecture of the synthesized systems consists of processors, and buses or links only. Other components, such as ASIC's, are not considered.

The other system synthesis approach which guarantees optimal solutions is described in [3]. 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 which is an assumption that all buses have the same transmission rate, and that 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 complexity of synthesis of aperiodic and periodic tasks can be reduced by clustering. The COSYN system and its extension CASPER, described in [5] and [6], are using this method. The algorithm presented there can cope with

large industrial size problems but memory requirements are not considered. During the assignment of clusters, an architecture is gradually extended when task deadlines are not met.

In [1], the multiprocessor task assignment was modeled as a vector packing problem. The target architecture consists of an arbitrary number of heterogeneous processing units that communicate over one bus. Having only one bus can result in a bus bottleneck for data dominated applications. This bottleneck could make the best heuristic to produce solutions with the lowest bus utilization ratio. This work was extended in [2], where it solves also a configuration selection problem. The goal is to minimize the cost of the system while correct assignment and schedule can be found.

Our work is based on the research presented in [8]. 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 optimal solutions can be obtained for small problems, however large problems requires a heuristic. Our system can minimize the design cost for a given execution time or vice versa. The efficiency of CLP approach is compared with another approaches in favor of CLP [8].

Our approach removes limitations imposed on the target architecture in [8] and adds memory constraints. We consider processors, ASIC's, buses, links, and memories. Local memory is divided into code and data memory. The designer has a freedom to influence the final design by making decisions concerning the final architecture, task assignment, and task scheduling. He can also supply the synthesis system with 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 implemented optimization heuristic produces good results for large designs as presented in section 5.

3. SYSTEM MODELING

In our approach, CLP is used to model the system architecture and the design problem. Since CLP is relatively young programming framework, we will briefly present it here. The general introduction to CLP is presented in [11], for example. First, we will concentrate on finite domain variables (FDV's) and constraints over these variables since they are used in our work. A CLP program consists of constraints over FDV's and a search method. Each FDV is initially defined by a set of integer values which 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 a solution.

We use Constraint Handling in Prolog (CHIP) v. 5.1 system. The CHIP system implements basic and global constraints. The basic constraints are equality, disequality or conditional constraints. The CLP modeling with basic constraints only is similar to ILP modeling style but underlaying solvers are different. Both modeling approaches will result in explosion of constraints. Therefore these approaches are impractical for big problems. To address this problem CLP frameworks introduced powerful modeling constraints, called global constraints. The global constraints are based on extensive work conducted in operating research community. These constraints provide concise modeling, good time complexity bounds, and good constraint propagation. The global constraints encompasses particular modeling problems. For example, they can impose restrictions on cumulative use of resources, rectangle placement or partitioning of graphs [4]. Modeling the problem using global constraints gives a clean and understandable description of the problem.

3.1. 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 only data memory. 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.

The data memory stores data used by tasks. Each task requires a constant memory for data processing which is dynamically allocated during task execution. The amount of needed data memory on each processor changes therefore during the schedule as shown in Figure 2b. The processor can compute, send, or receive data concurrently. During the synthesis we have to assure that the maximal usage of data memory does not exceed the available memory size.

ASIC's consist of any number of parts. The ASIC parts operate independently making possible parallel execution of tasks. Since we fix 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 modeled.

There is no restriction on the nature of the interconnection structure. The designer has to specify the 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 processors.

3.2. Problem Definition

CLASS assumes that functional description of a system is given as an acyclic task graph, as presented in Figure 1a. Each task is characterized by the estimated execution time and memory requirements. The real-time DSP or image processing applications belong to the class of problems which can be modeled using our system. They are fairly deterministic, so the static schedule can be derived for them.

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

$$T=(\tau, \rho, \delta, \mu_c, \mu_d) \quad (3)$$

where τ denotes the start time of the task execution, ρ denotes the resource on which the task is executed, δ denotes the task duration, μ_c and μ_d denote the amount of code and data memory needed for the execution of the task.

The execution time and code memory required by the task depend on the processor. The tasks must be always scheduled on one of the processing units and they cannot be preempted. This is modeled by imposing constraints which define finite relations between FDV's of (3) representing different tasks [8].

The arcs in the task graph represent data transfers between tasks. Each arc is described by a tuple of FDV's:

$$C=(\tau, \rho, \delta, \alpha) \quad (4)$$

where τ denotes the start time of the communication, ρ denotes the resource which is used for transferring data, δ denotes the duration of the communication, and α denotes the amount of the transferred data.

Each arc in the task graph imposes a constraint on an execution order of two tasks. This constraint is defined by a conjuction of two following inequalities:

$$\tau_i + \delta_i \le \tau_c \wedge \tau_c + \delta_c \le \tau_j \quad (5)$$

where task T_i sends data to task T_j using communication c. All constraints derived from arcs create together a partial ordering of the tasks.

There are two possible scenarios for transferring data between two communicating tasks. First scenario takes place when tasks are executed on different processors. In this case the communication must be assigned to and scheduled on a communication device. In second scenario, both communicating tasks are executed on the same processor and they communicate using the processor's local memory. In this case, the previous constraints reduce to the following one $\tau_i + \delta_i \le \tau_j$, since δ_c equals 0.

The data memory has to be reserved for each task during its execution (depicted as D_t's rectangles in Figure 2). It needs to be reserved also for data transfers between tasks (depicted as D_c's rectangles). The data memory for a single data transfer can be reserved on more than one processing unit. First, data is stored on the processor executing producer task, from the producer task termination until the related communication termination. Second, the same data is stored on the processor executing consumer task, from the communication start until consumer task start. During communication the data memory is reserved on both processors.

The task assignment, task scheduling and communication scheduling influence the data memory utilization scheme, therefore the actual data memory requirements are known after performing these steps. However using CLP global constraints and our own techniques we are able to estimate the amount of needed data memory earlier in the design phase. The obtained estimations are vital for our heuristic. The constraints on data memory allocation are imposed using cumulative and conditional constraints. These constraints ensure that data memory requirements are not violated. The developed estimates are used by the assignment and scheduling heuristic increasing a chance for generation close to optimal solutions.

The code memory constraints are easier to model than data memory constraints. The code memory requirements does not change during execution and they can be expressed using a conditional sum constraints.

4. OPTIMIZATION HEURISTIC

The task assignment and scheduling are NP-complete problems and therefore heuristics are usually used to solve them. The inclusion of memory constraints requires the development of a new heuristic. This heuristic takes into account memory constraints and is a part of the developed system called CLASS [14] which supports interactive exploration of the design space.

4.1. Parameters Estimations

A solution to a synthesis problem is an assignment of each task to a selected processor and related schedule for this task. In addition, each communication task has to be assigned to and scheduled on a communication device. A number of parameters is estimated to guide our heuristic during the process of finding a good solution.

Generally, distributed execution of the task graph results in bigger data memory requirement than execution on the single resource as shown in section 1. Delaying the execution of not urgent tasks for the favor of the tasks which belong to the critical path decreases the schedule length but

$$V_{ij} = \begin{cases} \frac{C_{ij}}{LCM_{ij}} & \text{if } -1 < Ind < -L_1 \quad (1) \\ \frac{C_{ij}}{LCM_{ij}} + \frac{T_{ij}}{LTS_{ij}} \times (1 - |Ind|) & \text{if } -L_1 \le Ind < -L_2 \quad (2) \\ \frac{C_{ij}}{LCM_{ij}} + \frac{T_{ij}}{LTS_{ij}} & \text{if } -L_2 \le Ind \le L_2 \quad (3) \\ \frac{C_{ij}}{LCM_{ij}} \times (1 - |Ind|) + \frac{T_{ij}}{LTS_{ij}} & \text{if } L_2 < Ind \le L_1 \quad (4) \\ \frac{T_{ij}}{LTS_{ij}} & \text{if } L_1 < Ind < 1 \quad (5) \end{cases}$$

where $Ind = UTS - UCM$. The value *Ind* is in the range <-1, 1>. L_1 and L_2 are heuristic constants and are equal 0.16 and 0.08 respectively.

Figure 3. cost V_{ij} of implementing task T_j on the processor P_i

it usually increases data memory requirements. Decreasing the schedule length and decreasing the data memory requirements are two conflicting goals. Either the schedule length is decreased or data memory requirement is decreased. Our heuristic tries first to schedule tasks from the critical path until the estimation of data memory usage is below the memory size. When the estimation of future data memory usage exceeds memory size then our heuristic aims at choosing tasks which will decrease the estimated data memory usage. Since the actual data memory utilization depends on the future decisions regarding the schedule, it is difficult to know in advance how exactly the assignment of task T_j to processor P_i will influence the peak of future data memory requirements on all processors. To make this problem easier we use two estimations methods.

The first estimate of data memory utilization is computed as a sum of incoming communications of ready to execute tasks. These communications represent data that exist at a given stage of the heuristic execution and have to be stored somewhere before they can be used. This estimation is fast but not accurate because it does not take into account the actual schedule. The second estimation is used when the first one cannot guarantee that data memory will not be overused. It provides more precise estimation of the upper bound of the data memory requirement. This better estimation uses the lower bound of the schedule length denoted by E_m. The tightest possible schedule, defined by E_m, means the best possible parallelism. The best parallel execution can potentially result in the highest data memory requirement. In addition, we compute the latest possible start time for each task denoted by $\max(\tau_j)$. The latest possible start time of task T_j means that incoming data to this task will need to be stored for the longest possible time. Both E_m and $\max(\tau_j)$ are then used in the cumulative constraint to estimate data memory requirement.

Two kinds of measures, *UCM* (usage of code memory) and *UTS* (usage of processor time), are computed by our heuristic. They are used to select processor P_i for execution of task T_j. *UCM* and *UTS* use lower bounds for used code memory (*LCM*) and processor time (*LTS*).

$$UCM = \frac{LCM}{ACM}, \text{ where } ACM \text{ - available code memory}$$

$$UTS = \frac{LTS}{ATS}, \quad \text{where } ATS \text{ - available processor time}$$

Similar measures are computed when we assume that task T_j will be executed on processor P_i. These look-ahead measures are denoted by LCM_{ij} and LTS_{ij}.

These two kinds of measures are then used to compute cost V_{ij} of implementing task T_j on processor P_i as depicted in Figure 3. The cost function uses also C_{ij} which denotes the amount of code memory needed to execute task T_j on processor P_i and T_{ij} which represents the time needed to execute task T_j on processor P_i.

The intuitive meaning of cost V_{ij} is following. In case (1), when $Ind < -L_1$, the code memory is much more used than processor time and therefore only code memory contributes to cost V_{ij}. The heuristic should minimize further increase in code memory usage. On the other hand, when value *Ind* is greater than L_1, the processor time is more valuable resource and the heuristic should aim at minimizing further increase of processors utilization. When processor and code memory utilizations are balanced (3) then both factors are taken into consideration with the same weight. The remaining cases describe situations when one of the resources is slightly overused. To counteract this, the weight of the other resource is decreased.

Our heuristic not only looks ahead before committing to a decision but also adapts its own cost function in case when there is a relative shortage of one of the resources. The global constraints, provided by CLP engine, and our own estimation techniques give our heuristic very good look-ahead capabilities. Therefore our heuristic which is greedy in its nature successfully omits typical traps for greedy heuristics.

4.2. Heuristic's pseudo-code

In this section, we present the pseudo-code of our heuristic as depicted in Figure 4. In each iteration of the while loop, we first choose a next task to be schedule. The selection of the actual task depends on the data memory requirement estimate. In the next step, we assign chosen task to a processor which is selected according to the introduced cost function. After the assignment of the task to the processor, incoming communications are assigned to and scheduled on the communication devices and finally the task is scheduled.

The proposed heuristic balances the usage of the code

while (R $\neq \emptyset$) {
// S- set of tasks which are already scheduled
// T - set of all tasks
// R- set of all ready tasks $\{x \mid x \in T - S \wedge \text{pred}(x) \subseteq S\}$

if ($\dfrac{\sum_{t_j \in R} DI_j}{ADM} < L_0$) {
// DO_j (DI_j) - the amount of transmitted data from (to) task T_j
// ADM - the available data memory
// L_0- the heuristic constant equals 0.4
// τ_j - the start time of task T_j
// Choose the task according to minimize_schedule_length criterion
find a task T_j among tasks in R with the smallest $\max(\tau_j)$.
(the second criterion is the smallest Δd_j.)
// $\Delta d_j = DO_j - DI_j$
else {
estimate data memory usage (EDMU) using cumulative constraint,
E_m estimation, and $\max(\tau_j)$ estimation for all tasks in R.
if (EDMU < ADM) {
// Choose the task according minimize_schedule_length criterion
find a task T_j among tasks in R with smallest $\max(\tau_j)$.
(the second criterion is the smallest Δd_j)}
else {
// Choose the task according to minimize_data_memory criterion
find a task T_j among tasks in R with smallest Δd_j.
(the second criterion is the smallest $\max(\tau_j)$)}}}
// The task (T_j) which we need to assign and scheduled is known
Compute V_{ij} for each processor which can execute task T_j. Assign task T_j to processor P_i with the smallest V_{ij}.
Schedule incoming communications of task T_j, in such a way, that the start time of task T_j - (τ_j) is minimal.
Schedule task T_j
// Task T_j and all incoming communications are assigned and scheduled }

Figure 4. Pseudo-code of the heuristic

memory and available time. The relation between the distributed execution and data memory requirements is also addressed in this heuristic during the selection of the next task for assignment and scheduling. Therefore our heuristic can cope with timing, code memory, and data memory constraints.

5. EXPERIMENTAL RESULTS

We used in our experiments complex image processing application based on CCITT recommendation H.261 presented in [3]. Since the example is relatively simple we constructed more complex task graphs by considering algorithmic pipelining. The pipelining was constrained by defining a latency of one iteration to be lower than 4000 clocks and an activation period to be 1000 clocks. We also assumed that establishing communication with environment take very small amount of time on the universal processor.

Table 1: Experimental results (random examples)

Exp.	Deadline	Avg. Execution Time	Code memory utilization [%]	Data memory utilization [%]	CPU utilization [%]	Bus/Link utilization [%]	Solutions #
1	120	104.4	86.0	85.5	86.3	72.0	20
2	110	101.2	87.7	86.5	87.0	75.0	19
3	100	95.4	88.8	86.7	89.6	76.8	17
4	100	93.5	89.7	80.8	90.6	80.5	18

Table 2: Experimental results (H.261 problem)

Exp. #	Pipeline Degree	Deadline	Average Execution Time	Σ Data Memory	ΔTime	ΔData Mem.
1	1	2965	2965	1385	-	-
2a	2	3965	1983	2290	-	-
2b	2	4002	2001	1673	1%	-27%
3a	3	4965	1655	2386	-	-
3b	3	5039	1680	1833	2%	-23%
4a	4	5965	1492	2386	-	-
4b	4	6172	1543	1737	3%	-27%

Experiments 2a, 3a, and 4a, shown in Table 2, assume no memory constraints, thus memory can be inefficiently used. In experiments 2b, 3b and 4b memory constraints were added. We obtained a small degradation of the average execution time while the utilization of data memory was decreased by ~25%. The experiments show that without taking memory into consideration task assignment and task scheduling could result in excessive memory usage.

Since example H.261 represents a relatively simple task graph we tried our heuristic on large randomly generated task graph examples. The results are presented in Table 1. Each of the experiments consists of the same set of 20 randomly generated task graphs. Each task graph consists of 100 computation tasks and 120 communications which need to be assigned and scheduled. The generated task graphs were constructed in such a way that the potential parallelism and the number of data were much higher then in H.261 example. The resources, code memory, data memory, and time, were quite restricted as utilization ratios show. In the experiments 2 and 3 we tighten the overall deadline comparing to experiment 1 thus obtaining better utilization ratios but for some task graphs we were not able to find a solution. In the last experiment, we increased data memory slightly obtaining lower data memory utilization ratio, higher CPU utilization, and one more solution comparing to the previous experiment. Our heuristic could find solutions for 74 out of 80 experiments even for very large task graph examples and tight deadline constraints. The heuristic execution time for these experiments was lower than 12 minutes on 50MHz Sparc workstation.

6. CONCLUSIONS

This paper presented a new heuristic for the task assignment and scheduling under memory constraints. The proposed constructive heuristic gives a valid task assignment and schedule fulfilling all constraints including memory constraints. Data memory usage peak varies very much and is influenced by task assignment, task schedule, and communication schedule. Incorporating timing constraints with memory constraints results in better decisions during task assignment and task scheduling. The experimental results show that data memory should be taken into account during system level synthesis to avoid inefficient, costly designs. The presented heuristic provides good results for large randomly generated task graph examples, as well as for a real example. The runtimes for presented real example were lower than one minute in each experiment.

REFERENCES

[1] J. Beck and D. P. Siewiorek, Modeling Multicomputer Task Allocation as a Vector Packing Problem, *International Symposium on System Synthesis*, 1996

[2] J. E. Beck and D. P. Siewiorek, Automatic Configuration of Embedded Multicomputer Systems, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, feb, 2, pp. 84-95, volume 17, 1998

[3] A. Bender, Design of an Optimal Loosely coupled Heterogeneous Multiprocessor System, *European Design and Testing Conference,* 1996

[4] COSYTEC, CHIP, System Documentation, 1996

[5] B. P. Dave, G. Lakshminarayana and N. K. Jha, COSYN: Hardware-Software Co-Synthesis of Embedded Systems, *DAC*, 1997

[6] B. P. Dave and N. K. Jha, CASPER: Concurrent Hardware-Software Co-Synthesis of Hard Real-Time Aperiodic and Periodic Specifications of Embedded System Architecture, *DATE Conference*, 1998

[7] B. P. Dave and N. K. Jha, COHRA: Hardware-Software Cosynthesis of Hierarchical Heterogeneous Distributed Embedded Systems, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October, vol. 17, 1998

[8] K. Kuchcinski, Embedded System Synthesis by Timing Constraint Solving, *ISSS* 1997

[9] K. Kuchcinski, Integrated Resource Assignment and Scheduling of Task Graphs Using Finite Domain Constraints, *DATE Conference*, 1999

[10] J. Madsen, P. Bjørn-Jørgensen, Embedded System Synthesis under Memory Constraints, CODES'99, Rome, Italy

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

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

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

[14] R. Szymanek, K. Kuchcinski, Design Space Exploration under Memory Constraints. 25th EuroMicro, 8-10 September, 1999

Task Assignment and Scheduling under Memory Constraints

Radoslaw Szymanek and Krzysztof Kuchcinski

Dept. of Computer Science
Lund University
P.O. Box 118, SE 221 00, Lund, Sweden

e-mail: Radoslaw.Szymanek@cs.lth.se

Neither this paper nor any version close to it has been or is being offered elsewhere for publication of this paper. If accepted, the paper will be made available in Camera-ready forms by June 15th, 2000, and it will be personally presented at the EUROMICRO 2000 Conference by the author or one of the co-authors. The presenting author(s) will pre-register for EUROMICRO 2000 before the due date of the Camera-ready paper.

Task Assignment and Scheduling under Memory Constraints

Radoslaw Szymanek and Krzysztof Kuchcinski
Dept. of Computer Science
Lund University
P.O. Box 118, SE 221 00, Lund, Sweden
e-mail: Radoslaw.Szymanek@cs.lth.se

ABSTRACT

*Many DSP and image processing embedded systems have hard memory constraints which makes it difficult to find a good task assignment and scheduling which fulfill these constraints. This paper presents a new heuristic developed for task assignment and scheduling for such systems. These systems have also a large number of constraints of different nature, such as cost, execution time, memory capacity and limitations on resource usage. The heterogeneous constraints require new synthesis methods which will take them into account during searching for a valid solution. The heuristic presented in this paper is a part of the CLASS system (**C**onstraint **L**ogic Progr**a**mming based **S**ystem **S**ynthesis).*

Keywords

task scheduling, constraint programming, memory constraints.

1. INTRODUCTION

The complexity of the embedded systems grows constantly. This causes shifting to higher levels of abstractions of digital systems design, as well as need for the multiprocessor heterogeneous systems. These systems have a potential to achieve superior results in terms of cost and performance over homogenous systems since they can match the design requirements more closely. During the system level synthesis designer decides what are the main components of the system. The decisions at this level have enormous impact on the final product characteristics such as cost, performance, and time to market. Thus the responsibility of a designer is to perform fast exploration of different design trade-offs and choose an architecture which suits the problem best.

Embedded systems synthesis is usually decomposed into a chain of problems which must be solved to obtain a final solution. The first task for a designer is to decide what components should be used in the target architecture. Afterwards, the assignment of all tasks into the selected components is done. The last step is scheduling of the tasks on the processing units and the data transfers on the communication devices. The decisions taken during these steps are not independent making the process of finding (near) optimal design for industrial problems very difficult. The goodness of the final design is determined by all the decisions.

The complexity of the assignment and scheduling problem increases significantly when memory constraints are added. New synthesis methods capable of dealing with increased complexity are required. These methods should, in addition, provide good quality results and incorporate designer knowledge through interactions with synthesis methods.

Data memory aspect of the embedded systems is specially important in image processing and DSP applications which process enormous amounts of data. Memory constraints are difficult to model but neglecting them can produce considerable waste of memory components. A good utilization of memories is a key issue in achieving low cost solutions with modest amount of memory components. In our approach, we consider data memory constraints during task assignment and scheduling and therefore we are able to reduce the amount of needed memory.

The goal of this work is to develop system synthesis system which accepts an input specification given as communicating tasks. From this description the system will generate a final system architecture together with task assignment and scheduling. We assume that the input specification, described in C-like language, is compiled into a task graph which is synthesized. In this paper, we present the task assignment and scheduling heuristic. The input to this heuristic is an acyclic task graph [9] generated from the original specification.

Consider, for example, a task graph depicted in Figure 1a. There are four tasks T_1, T_2, T_3, and T_4. Data is transferred between those 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. All tasks need 4 kB of data memory which is denoted by D_t's rectangles in Figure 2. Data transfers send 2 kB of data. The data produced by the tasks need to be stored also in data memory which is denoted by

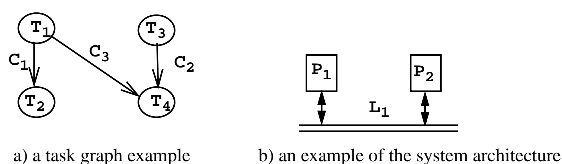


Figure 1. Task data flow graph and target architecture

D_c's rectangles in Figure 2. Note that during interprocessor communication both processors have to allocate memory for the transferred data.

Figure 2 depicts four distinct schedules. The last three are distributed ones implemented on the architecture presented in Figure 1b. The same schedule length has been obtained for distributed schedules but different memory usage is achieved. The distributed schedule depicted in Figure 2b has very unbalanced and inefficient memory utilization. The amount of needed data memory on processor P_1 and P_2 is 8kB and 4kB respectively. The utilization of code memory is unbalanced, 3kB and 1kB for processor P_1 and P_2 respectively. In Figure 2c the schedule with balanced code memory utilization but still inefficient data memory utilization is presented. The schedule depicted in Figure 2d has the same schedule length as previous ones while memory usage is balanced and optimal, 2kB for code memory and 4kB for data memory on both processors P_1 and P_2. The importance of memory consideration was also indicated in [10, 13, 14].

The main contribution of this paper is the development of a new constructive heuristic which takes into account memory constraints during task assignment and scheduling.

In this paper, we focus on the heuristic which solves the problem of task assignment and scheduling under memory constraints. In section 2, we outline related work in this area. Section 3 defines our model of the architecture and the system. Section 4 describes the flow of the heuristic. Section 5 presents experimental results. Finally, section 6 concludes the paper.

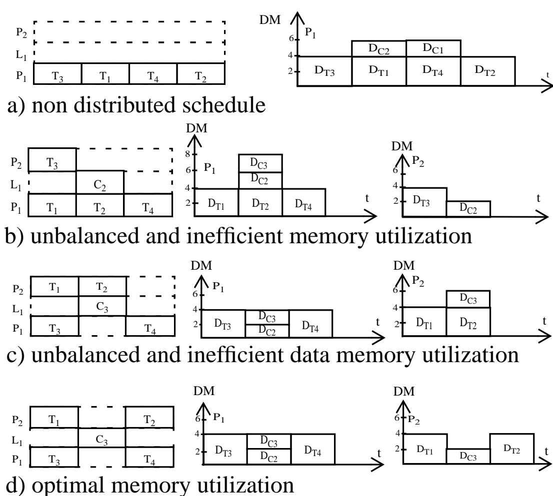


Figure 2. Data memory requirements

2. RELATED WORK

The synthesis problem encompasses a number of problems which can be studied separately. In our work, we try to integrate these problems. The research presented in [10] addresses similar problem of embedded system synthesis under memory constraints. It uses a genetic algorithm to solve the synthesis problem. This algorithm is iterative and stochastic. The authors assume that code and data memory 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 in our work and it will make the synthesis problem simpler. We believe that genetic algorithms can have problems giving a valid solution when the synthesis problem is very constrained and therefore we have developed a constructive heuristic. It seems that our heuristic has much shorter execution time than the genetic algorithm. An important difference is hardware sharing capability in our approach. We are able to model ASIC's hardware which is shared by different tasks.

The similar work was also described in [13] and its previous version [12]. The newer version 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. The architecture of the synthesized systems consists of processors, and buses or links only. Other components, such as ASIC's, are not considered.

The other system synthesis approach which guarantees optimal solutions is described in [3]. 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 which is an assumption that all buses have the same transmission rate, and that 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 complexity of synthesis of aperiodic and periodic tasks can be reduced by clustering. The COSYN system and its extension CASPER, described in [5] and [6], are using this method. The algorithm presented there can cope with

large industrial size problems but memory requirements are not considered. During the assignment of clusters, an architecture is gradually extended when task deadlines are not met.

In [1], the multiprocessor task assignment was modeled as a vector packing problem. The target architecture consists of an arbitrary number of heterogeneous processing units that communicate over one bus. Having only one bus can result in a bus bottleneck for data dominated applications. This bottleneck could make the best heuristic to produce solutions with the lowest bus utilization ratio. This work was extended in [2], where it solves also a configuration selection problem. The goal is to minimize the cost of the system while correct assignment and schedule can be found.

Our work is based on the research presented in [8]. 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 optimal solutions can be obtained for small problems, however large problems requires a heuristic. Our system can minimize the design cost for a given execution time or vice versa. The efficiency of CLP approach is compared with another approaches in favor of CLP [8].

Our approach removes limitations imposed on the target architecture in [8] and adds memory constraints. We consider processors, ASIC's, buses, links, and memories. Local memory is divided into code and data memory. The designer has a freedom to influence the final design by making decisions concerning the final architecture, task assignment, and task scheduling. He can also supply the synthesis system with 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 implemented optimization heuristic produces good results for large designs as presented in section 5.

3. SYSTEM MODELING

In our approach, CLP is used to model the system architecture and the design problem. Since CLP is relatively young programming framework, we will briefly present it here. The general introduction to CLP is presented in [11], for example. First, we will concentrate on finite domain variables (FDV's) and constraints over these variables since they are used in our work. A CLP program consists of constraints over FDV's and a search method. Each FDV is initially defined by a set of integer values which 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 a solution.

We use Constraint Handling in Prolog (CHIP) v. 5.1 system. The CHIP system implements basic and global constraints. The basic constraints are equality, disequality or conditional constraints. The CLP modeling with basic constraints only is similar to ILP modeling style but underlaying solvers are different. Both modeling approaches will result in explosion of constraints. Therefore these approaches are impractical for big problems. To address this problem CLP frameworks introduced powerful modeling constraints, called global constraints. The global constraints are based on extensive work conducted in operating research community. These constraints provide concise modeling, good time complexity bounds, and good constraint propagation. The global constraints encompasses particular modeling problems. For example, they can impose restrictions on cumulative use of resources, rectangle placement or partitioning of graphs [4]. Modeling the problem using global constraints gives a clean and understandable description of the problem.

3.1. 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 only data memory. 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.

The data memory stores data used by tasks. Each task requires a constant memory for data processing which is dynamically allocated during task execution. The amount of needed data memory on each processor changes therefore during the schedule as shown in Figure 2b. The processor can compute, send, or receive data concurrently. During the synthesis we have to assure that the maximal usage of data memory does not exceed the available memory size.

ASIC's consist of any number of parts. The ASIC parts operate independently making possible parallel execution of tasks. Since we fix 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 modeled.

There is no restriction on the nature of the interconnection structure. The designer has to specify the 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 processors.

3.2. Problem Definition

CLASS assumes that functional description of a system is given as an acyclic task graph, as presented in Figure 1a. Each task is characterized by the estimated execution time and memory requirements. The real-time DSP or image processing applications belong to the class of problems which can be modeled using our system. They are fairly deterministic, so the static schedule can be derived for them.

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

$$T=(\tau, \rho, \delta, \mu_c, \mu_d) \quad (3)$$

where τ denotes the start time of the task execution, ρ denotes the resource on which the task is executed, δ denotes the task duration, μ_c and μ_d denote the amount of code and data memory needed for the execution of the task.

The execution time and code memory required by the task depend on the processor. The tasks must be always scheduled on one of the processing units and they cannot be preempted. This is modeled by imposing constraints which define finite relations between FDV's of (3) representing different tasks [8].

The arcs in the task graph represent data transfers between tasks. Each arc is described by a tuple of FDV's:

$$C=(\tau, \rho, \delta, \alpha) \quad (4)$$

where τ denotes the start time of the communication, ρ denotes the resource which is used for transferring data, δ denotes the duration of the communication, and α denotes the amount of the transferred data.

Each arc in the task graph imposes a constraint on an execution order of two tasks. This constraint is defined by a conjuction of two following inequalities:

$$\tau_i + \delta_i \le \tau_c \wedge \tau_c + \delta_c \le \tau_j \quad (5)$$

where task T_i sends data to task T_j using communication c. All constraints derived from arcs create together a partial ordering of the tasks.

There are two possible scenarios for transferring data between two communicating tasks. First scenario takes place when tasks are executed on different processors. In this case the communication must be assigned to and scheduled on a communication device. In second scenario, both communicating tasks are executed on the same processor and they communicate using the processor's local memory. In this case, the previous constraints reduce to the following one $\tau_i + \delta_i \le \tau_j$, since δ_c equals 0.

The data memory has to be reserved for each task during its execution (depicted as D_t's rectangles in Figure 2). It needs to be reserved also for data transfers between tasks (depicted as D_c's rectangles). The data memory for a single data transfer can be reserved on more than one processing unit. First, data is stored on the processor executing producer task, from the producer task termination until the related communication termination. Second, the same data is stored on the processor executing consumer task, from the communication start until consumer task start. During communication the data memory is reserved on both processors.

The task assignment, task scheduling and communication scheduling influence the data memory utilization scheme, therefore the actual data memory requirements are known after performing these steps. However using CLP global constraints and our own techniques we are able to estimate the amount of needed data memory earlier in the design phase. The obtained estimations are vital for our heuristic. The constraints on data memory allocation are imposed using cumulative and conditional constraints. These constraints ensure that data memory requirements are not violated. The developed estimates are used by the assignment and scheduling heuristic increasing a chance for generation close to optimal solutions.

The code memory constraints are easier to model than data memory constraints. The code memory requirements does not change during execution and they can be expressed using a conditional sum constraints.

4. OPTIMIZATION HEURISTIC

The task assignment and scheduling are NP-complete problems and therefore heuristics are usually used to solve them. The inclusion of memory constraints requires the development of a new heuristic. This heuristic takes into account memory constraints and is a part of the developed system called CLASS [14] which supports interactive exploration of the design space.

4.1. Parameters Estimations

A solution to a synthesis problem is an assignment of each task to a selected processor and related schedule for this task. In addition, each communication task has to be assigned to and scheduled on a communication device. A number of parameters is estimated to guide our heuristic during the process of finding a good solution.

Generally, distributed execution of the task graph results in bigger data memory requirement than execution on the single resource as shown in section 1. Delaying the execution of not urgent tasks for the favor of the tasks which belong to the critical path decreases the schedule length but

$$V_{ij} = \begin{cases} \dfrac{C_{ij}}{LCM_{ij}} & \text{if } -1 < Ind < -L_1 \quad (1) \\ \dfrac{C_{ij}}{LCM_{ij}} + \dfrac{T_{ij}}{LTS_{ij}} \times (1 - |Ind|) & \text{if } -L_1 \le Ind < -L_2 \quad (2) \\ \dfrac{C_{ij}}{LCM_{ij}} + \dfrac{T_{ij}}{LTS_{ij}} & \text{if } -L_2 \le Ind \le L_2 \quad (3) \\ \dfrac{C_{ij}}{LCM_{ij}} \times (1 - |Ind|) + \dfrac{T_{ij}}{LTS_{ij}} & \text{if } L_2 < Ind \le L_1 \quad (4) \\ \dfrac{T_{ij}}{LTS_{ij}} & \text{if } L_1 < Ind < 1 \quad (5) \end{cases}$$

where $Ind = UTS - UCM$. The value *Ind* is in the range <-1, 1>. L_1 and L_2 are heuristic constants and are equal 0.16 and 0.08 respectively.

Figure 3. cost V_{ij} of implementing task T_j on the processor P_i

it usually increases data memory requirements. Decreasing the schedule length and decreasing the data memory requirements are two conflicting goals. Either the schedule length is decreased or data memory requirement is decreased. Our heuristic tries first to schedule tasks from the critical path until the estimation of data memory usage is below the memory size. When the estimation of future data memory usage exceeds memory size then our heuristic aims at choosing tasks which will decrease the estimated data memory usage. Since the actual data memory utilization depends on the future decisions regarding the schedule, it is difficult to know in advance how exactly the assignment of task T_j to processor P_i will influence the peak of future data memory requirements on all processors. To make this problem easier we use two estimations methods.

The first estimate of data memory utilization is computed as a sum of incoming communications of ready to execute tasks. These communications represent data that exist at a given stage of the heuristic execution and have to be stored somewhere before they can be used. This estimation is fast but not accurate because it does not take into account the actual schedule. The second estimation is used when the first one cannot guarantee that data memory will not be overused. It provides more precise estimation of the upper bound of the data memory requirement. This better estimation uses the lower bound of the schedule length denoted by E_m. The tightest possible schedule, defined by E_m, means the best possible parallelism. The best parallel execution can potentially result in the highest data memory requirement. In addition, we compute the latest possible start time for each task denoted by $\max(\tau_j)$. The latest possible start time of task T_j means that incoming data to this task will need to be stored for the longest possible time. Both E_m and $\max(\tau_j)$ are then used in the cumulative constraint to estimate data memory requirement.

Two kinds of measures, *UCM* (usage of code memory) and *UTS* (usage of processor time), are computed by our heuristic. They are used to select processor P_i for execution of task T_j. *UCM* and *UTS* use lower bounds for used code memory (*LCM*) and processor time (*LTS*).

$$UCM = \frac{LCM}{ACM}, \text{ where } ACM \text{ - available code memory}$$

$$UTS = \frac{LTS}{ATS}, \quad \text{where } ATS \text{ - available processor time}$$

Similar measures are computed when we assume that task T_j will be executed on processor P_i. These look-ahead measures are denoted by LCM_{ij} and LTS_{ij}.

These two kinds of measures are then used to compute cost V_{ij} of implementing task T_j on processor P_i as depicted in Figure 3. The cost function uses also C_{ij} which denotes the amount of code memory needed to execute task T_j on processor P_i and T_{ij} which represents the time needed to execute task T_j on processor P_i.

The intuitive meaning of cost V_{ij} is following. In case (1), when $Ind < -L_1$, the code memory is much more used than processor time and therefore only code memory contributes to cost V_{ij}. The heuristic should minimize further increase in code memory usage. On the other hand, when value *Ind* is greater than L_1, the processor time is more valuable resource and the heuristic should aim at minimizing further increase of processors utilization. When processor and code memory utilizations are balanced (3) then both factors are taken into consideration with the same weight. The remaining cases describe situations when one of the resources is slightly overused. To counteract this, the weight of the other resource is decreased.

Our heuristic not only looks ahead before committing to a decision but also adapts its own cost function in case when there is a relative shortage of one of the resources. The global constraints, provided by CLP engine, and our own estimation techniques give our heuristic very good look-ahead capabilities. Therefore our heuristic which is greedy in its nature successfully omits typical traps for greedy heuristics.

4.2. Heuristic's pseudo-code

In this section, we present the pseudo-code of our heuristic as depicted in Figure 4. In each iteration of the while loop, we first choose a next task to be schedule. The selection of the actual task depends on the data memory requirement estimate. In the next step, we assign chosen task to a processor which is selected according to the introduced cost function. After the assignment of the task to the processor, incoming communications are assigned to and scheduled on the communication devices and finally the task is scheduled.

The proposed heuristic balances the usage of the code

while ($R \neq \emptyset$) {
// S- set of tasks which are already scheduled
// T - set of all tasks
// R- set of all ready tasks $\{x \mid x \in T - S \wedge pred(x) \subseteq S\}$

if ($\frac{\sum_{t_j \in R} DI_j}{ADM} < L_0$) {
  // DO_j (DI_j) - the amount of transmitted data from (to) task T_j
  // ADM - the available data memory
  // L_0- the heuristic constant equals 0.4
  // τ_j - the start time of task T_j
  // Choose the task according to minimize_schedule_length criterion
    find a task T_j among tasks in R with the smallest max(τ_j).
    (the second criterion is the smallest Δd_j.)
  // $\Delta d_j = DO_j - DI_j$
else {
  estimate data memory usage (EDMU) using cumulative constraint,
    E_m estimation, and max(τ_j) estimation for all tasks in R.
  if (EDMU < ADM) {
    // Choose the task according minimize_schedule_length criterion
    find a task T_j among tasks in R with smallest max(τ_j).
    (the second criterion is the smallest Δd_j)}
  else {
    // Choose the task according to minimize_data_memory criterion
    find a task T_j among tasks in R with smallest Δd_j.
    (the second criterion is the smallest max(τ_j))}}}
// The task (T_j) which we need to assign and scheduled is known
Compute V_{ij} for each processor which can execute task T_j. Assign task T_j to processor P_i with the smallest V_{ij}.
Schedule incoming communications of task T_j, in such a way, that the start time of task T_j - (τ_j) is minimal.
Schedule task T_j
// Task T_j and all incoming communications are assigned and scheduled }

Figure 4. Pseudo-code of the heuristic

memory and available time. The relation between the distributed execution and data memory requirements is also addressed in this heuristic during the selection of the next task for assignment and scheduling. Therefore our heuristic can cope with timing, code memory, and data memory constraints.

5. EXPERIMENTAL RESULTS

We used in our experiments complex image processing

Table 1: Experimental results (random examples)

Exp.	Deadline	Avg. Execution Time	Code memory utilization [%]	Data memory utilization [%]	CPU utilization [%]	Bus/Link utilization [%]	Solutions #
1	120	104.4	86.0	85.5	86.3	72.0	20
2	110	101.2	87.7	86.5	87.0	75.0	19
3	100	95.4	88.8	86.7	89.6	76.8	17
4	100	93.5	89.7	80.8	90.6	80.5	18

Table 2: Experimental results (H.261 problem)

Exp. #	Pipeline Degree	Deadline	Average Execution Time	Σ Data Memory	ΔTime	ΔData Mem.
1	1	2965	2965	1385	-	-
2a	2	3965	1983	2290	-	-
2b	2	4002	2001	1673	1%	-27%
3a	3	4965	1655	2386	-	-
3b	3	5039	1680	1833	2%	-23%
4a	4	5965	1492	2386	-	-
4b	4	6172	1543	1737	3%	-27%

application based on CCITT recommendation H.261 presented in [3]. Since the example is relatively simple we constructed more complex task graphs by considering algorithmic pipelining. The pipelining was constrained by defining a latency of one iteration to be lower than 4000 clocks and an activation period to be 1000 clocks. We also assumed that establishing communication with environment take very small amount of time on the universal processor.

Experiments 2a, 3a, and 4a, shown in Table 2, assume no memory constraints, thus memory can be inefficiently used. In experiments 2b, 3b and 4b memory constraints were added. We obtained a small degradation of the average execution time while the utilization of data memory was decreased by ~25%. The experiments show that without taking memory into consideration task assignment and task scheduling could result in excessive memory usage.

Since example H.261 represents a relatively simple task graph we tried our heuristic on large randomly generated task graph examples. The results are presented in Table 1. Each of the experiments consists of the same set of 20 randomly generated task graphs. Each task graph consists of 100 computation tasks and 120 communications which need to be assigned and scheduled. The generated task graphs were constructed in such a way that the potential parallelism and the number of data were much higher then in H.261 example. The resources, code memory, data memory, and time, were quite restricted as utilization ratios show. In the experiments 2 and 3 we tighten the overall deadline comparing to experiment 1 thus obtaining better utilization ratios but for some task graphs we were not able to find a solution. In the last experiment, we increased data memory slightly obtaining lower data memory utilization ratio, higher CPU utilization, and one more solution comparing to the previous experiment. Our heuristic could find solutions for 74 out of 80 experiments even for very large task graph examples and tight deadline constraints. The heuristic execution time for these experiments was lower than 12 minutes on 50MHz Sparc workstation.

6. CONCLUSIONS

This paper presented a new heuristic for the task assignment and scheduling under memory constraints. The proposed constructive heuristic gives a valid task assignment and schedule fulfilling all constraints including memory constraints. Data memory usage peak varies very much and is influenced by task assignment, task schedule, and communication schedule. Incorporating timing constraints with memory constraints results in better decisions during task assignment and task scheduling. The experimental results show that data memory should be taken into account during system level synthesis to avoid inefficient, costly designs. The presented heuristic provides good results for large randomly generated task graph examples, as well as for a real example. The runtimes for presented real example were lower than one minute in each experiment.

REFERENCES

[1] J. Beck and D. P. Siewiorek, Modeling Multicomputer Task Allocation as a Vector Packing Problem, *International Symposium on System Synthesis*, 1996

[2] J. E. Beck and D. P. Siewiorek, Automatic Configuration of Embedded Multicomputer Systems, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, feb, 2, pp. 84-95, volume 17, 1998

[3] A. Bender, Design of an Optimal Loosely coupled Heterogeneous Multiprocessor System, *European Design and Testing Conference,* 1996

[4] COSYTEC, CHIP, System Documentation, 1996

[5] B. P. Dave, G. Lakshminarayana and N. K. Jha, COSYN: Hardware-Software Co-Synthesis of Embedded Systems, *DAC*, 1997

[6] B. P. Dave and N. K. Jha, CASPER: Concurrent Hardware-Software Co-Synthesis of Hard Real-Time Aperiodic and Periodic Specifications of Embedded System Architecture, *DATE Conference*, 1998

[7] B. P. Dave and N. K. Jha, COHRA: Hardware-Software Cosynthesis of Hierarchical Heterogeneous Distributed Embedded Systems, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October, vol. 17, 1998

[8] K. Kuchcinski, Embedded System Synthesis by Timing Constraint Solving, *ISSS* 1997

[9] K. Kuchcinski, Integrated Resource Assignment and Scheduling of Task Graphs Using Finite Domain Constraints, *DATE Conference*, 1999

[10] J. Madsen, P. Bjørn-Jørgensen, Embedded System Synthesis under Memory Constraints, CODES'99, Rome, Italy

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

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

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

[14] R. Szymanek, K. Kuchcinski, Design Space Exploration under Memory Constraints. 25th EuroMicro, 8-10 September, 1999