Embedded System Synthesis under Memory Constraints

Jan Madsen
Department of Information Technology,
Technical University of Denmark
jan@it.dtu.dk

Peter Bjørn-Jørgensen
Nokia Mobile Phones A/S,
Copenhagen,
Peter.Bjoern-Joergensen@nmp.nokia.com

Abstract

This paper presents a genetic algorithm to solve the system synthesis problem of mapping a time constrained single-rate system specification onto a given heterogeneous architecture which may contain irregular interconnection structures. The synthesis is performed under memory constraints, that is, the algorithm takes into account the memory size of processors and the size of interface buffers of communication links, and in particular the complicated interplay of these. The presented algorithm is implemented as part of the LYCOS cosynthesis system.

1 Introduction

Embedded systems are usually implemented using a mixture of technologies including off-the-shelf components, such as general purpose microprocessors, and dedicated hardware, such as full- or semi-custom ASICs. This results in a heterogeneous architecture, in which also the communication links between the components use different technologies, e.g. point-to-point and busses with various bandwidths.

We present an approach based on the genetic algorithm paradigme to solve the problem of mapping a time constrained single-rate system specification onto a *given* heterogeneous architecture under multiple resource constraints which includes memory size of processors, buffer size of communication links, and area of ASICs. Hou et al. [5] presented a genetic algorithm for multiprocessor scheduling, but they did neither consider communication nor aspects of memory.

Memory is an important issue which is often neglected. In most approaches memory is assumed to be infinite! However, in embedded computer systems memory is usually a critical resource and the memory size is often very restricted. The approach by Prakash and Parker [12] is one of the few approach which tries to take memory into consideration during synthesis. They use MILP to synthesize optimal heterogeneous systems taking memory cost into account. However, they only consider the data size (i.e. dynamic memory) used inside a task. We consider both static and dynamic memory usage within a task and the dynamic memory usage due to communication.

Many approaches to solve the mapping problem are based on list scheduling, e.g. [2, 6, 9, 13, 14]. Where most approaches schedules tasks as well as communications [2, 6, 13], some assume a constant communication overhead [9, 14]. This, however, results in the unrealistic assumption that multiple communications can take place at the same time. In order to handle dynamic memory usage during communication, communication scheduling has to be handled properly. Many approaches assume a fully connected architecture where there is a direct connection between any two processors. Typically this is realized as a single bus system. However, embedded systems may use *irregular interconnection structures*, e.g. to avoid bus contentions. The approach by Sih and Lee [13] is able to handle these interconnection structures, but without the inclusion of memory.

Optimal methods such as ILP [1], MILP [11, 12], and constraint logic programming [8] have been used to solve the distributed system synthesis problem. These techniques produce an optimal hardware architecture for a given application. But, in practice [7] the architecture may be restricted by the company's/designers wish to *reuse* an existing design as part of the new design. I.e., products are often developed as part of a family of similar products.

We believe that it is important to keep the designer in control of the design process. Hence, we propose a system synthesis technique in which the designer specifies the architecture and then uses the technique to evaluate how well the system specification can be mapped onto it. In the following we will first present the models of the target architecture and the system specification. After having discussed how memory utilization is captured, we present our synthesis algorithm followed by some experimental results.

2 Target architecture

A target architecture is represented by a hyper-graph, $G_A = (V_A, E_A)$ in which each vertex describes a component and the edges describe interconnections among the components. Each component may be a *processing element* (PE), p, or an *interface*, i. A processing element represents an active component, i.e. a CPU or an ASIC, which is able to execute a task. An interface connects a processing element to a net. An edge, *n*, represents a *net* connecting two or more interfaces, i.e. a point-to-point connection or a bus, or connecting an interface to a PE. Figure 1 shows a target architecture containing 4 PEs, 5 interfaces, and 2 busses. Each processing

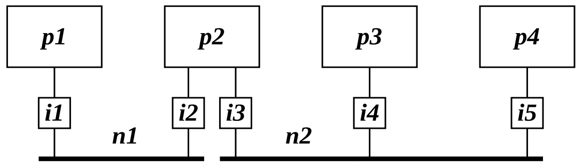


Figure 1: Example of a target architecture.

element is characterized by the size of its local memory and, if an ASIC, its available area. Local memory is used by the PE to store data during execution of a task and is represented in units of data [1]. Memory used for data will be referred to as *dynamic* memory. If the PE is a CPU, the program will also have to be stored in the local memory. This memory contribution will be referred to as *static* memory. For off-the-shelf components like a general purpose CPU, the area will be zero, but if the PE represents an ASIC implementation, the area will reflect the available size for datapath and controller.

An interface component is characterized by the sizes of its transmit and receive buffers, which are FIFO buffers. Thus, the interface can store data and possibly free the processing element even though it does not have gained access to the bus. Furthermore, an interface declares the *package-size* and *transfer-rate* for both the connection between the processing element and the interface, and between the net and the interface. The package-size is represented in units of data, and the transfer-rate as the time taken to transfer a single unit of data.

Each net is characterized by a package-size and a transfer-rate which has to correspond to its connected interface components.

3 System specification

The behavior of an embedded system is described by a *task graph*, $G_T = (V_T, E_T)$, which is a partially-ordered set of *tasks* represented as a directed acyclic hyper-graph. Hence, each vertex, $\tau_i \in V_T$, in the task graph represents a task describing a single thread of execution which cannot be preempted. An edge, $e_{i,succ_i} \in E_T$, describes a data dependency between the task τ_i and the set of successor tasks of τ_i, i.e. $succ(\tau_i)$. Each edge is annotated with the amount of data, $d_{i,succ_i}$, which has to be transferred between the source task and its successors.

We assume that a characterization of each task has been done prior to the synthesis step [10]. A characterization of a task consists of, for each CPU, estimating the execution time, the code size and the data size, and for each ASIC estimating the execution time, the data size, and the area. Tasks are only characterized on PEs on which they can be implemented. As a task may have multiple characterizations, selecting among different implementations on the *same* processing element is possible, i.e. emulation of algorithmic choices.

When memory is taken into account, an important property is the sharing of code among different tasks executing on the same processing element. In order to handle this, we introduce the notion of *functions*. Hence, a task may use a set of functions when executing its behavior. This means that a characterization of a task on a processing element also includes a list of functions. Each function is characterized by its code size (if implemented in software) and area (if implemented in hardware). The time and data size of a function is captured in the characterization of the tasks using the function.

4 Evaluating Memory Utilization

To see how memory is taken into account during synthesis, consider the following example:

Example 1: Assume that we have to schedule the task graph in figure 2a on an architecture consisting of two PEs (p_1 and p_2) connected by a single bus as shown in figure 2b. Figure 2c shows a possible schedule. When task τ_1 on pro-

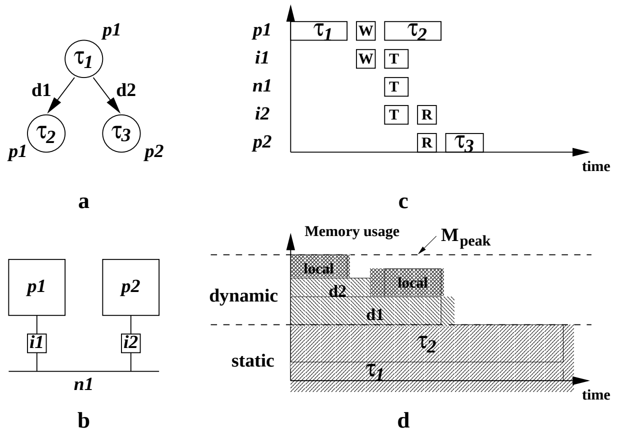


Figure 2: A simple example; a) task graph. b) architecture. c) schedule. d) memory utilization on p_1.

cessor p_1 has finished execution, the data d_2 to be send to task τ_3 on p_2 resides in local memory of p_1 where it is kept until it can be transferred. At the time the interface, i_1, and p_1 are ready, the data is written (W) to the transmit buffer in i_1. This process consumes time on both i_1 and p_1. When the net, n_1, is available, the data is transferred over the net and stored in the receive buffer of i_2. And at the time p_2 is ready, data can be read (R) from i_2 and stored in local memory of p_2. Then later on it can be used by task τ_3 executing on p_2.

Figure 2d shows how the memory utilization of p_1 is calculated. The memory calculation consist of two contributions, a static and a dynamic. The static contribution is calculated as the summation of the code size for each task assigned to p_1. As outlined in figure 2d, the dynamic contribution consists of memory used for data during the execution of a task (local) and memory required to store data from the time it is produced until it is no longer needed. Figure 2d illustrates how data due to dependency d_1, which is produced

[1]A unit of data may be a bit, a byte, a frame, etc., as long as all data sizes in the system is expressed using the same unit.

by τ_1 on p_1 and used by τ_2 also on p_1, is kept alive until τ_2 has finished execution. Likewise, data d_2 is kept alive until it has been written to the transmit buffer, i_1.

Hence, at any point in time we can find the memory utilization by summation of the different memory contributions and thus, finding the peak memory requirement. It should be noted that we assume that the memory is always perfectly organized, i.e. no problems due to fragmentation.
□

5 Algorithm overview

Our synthesis algorithm is based on the genetic algorithm [5] which is an iterative and stochastic process that operates on a set of individuals (the population). Each individual represents a potential solution to the problem being solved, and is obtained by decoding the gene string of the individual. Initially, the population is randomly generated. Every individual in the population is assigned a fitness value which is a measure of its goodness with respect to the problem being considered. This value is the quantitative information the algorithm uses to guide the search for a feasible solution.

The basic genetic algorithm consists of three major stages: selection, reproduction, and replacement. During the selection stage, a temporary population is created in which the fittest individuals have a higher number of instances than those less fit. A new population is then created by performing crossover followed by mutation. Finally, individuals of the original population is substituted by the newly created individuals in such a way that the most fit individuals are kept deleting the worst ones. A thorough description of genetic algorithms may be found in [4].

There are two important issues which have to be addressed when formulating a problem to be solved by genetic algorithms; the encoding/decoding mechanism of the gene string of an individual, and the evaluation of the *fitness* of an individual.

5.1 Encoding/Decoding

For a task graph containing n tasks and m dependencies, the corresponding gene string consist of $n+m$ genes, n task genes and m dependency genes. For each task τ_i its gene contains two integers, $\mathtt{impl}_{\tau_i}$ and $\mathtt{prio}_{\tau_i}$. $\mathtt{impl}_{\tau_i}$ identifies an implementation, i.e. an allocation. If a task τ_i has k_i possible implementations (as identified from its characterization), then the actual implementation is found as $\mathtt{impl}_{\tau_i}$ modulus k_i. $\mathtt{prio}_{\tau_i}$ is a priority which is used when scheduling the task during the fitness evaluation.

A dependency gene contains two integers, $\mathtt{impl}_{d_{i,succ_i}}$ and $\mathtt{prio}_{d_{i,succ_i}}$. $\mathtt{impl}_{d_{i,succ_i}}$ identifies a path between the processing elements on which τ_i and its successors $succ(\tau_i)$ are allocated. I.e. it identifies one of the possible paths in the architecture which is able to fulfill the communication represented by the data dependency $\mathtt{impl}_{d_{i,succ_i}}$. This path is called a *message tree*. A message tree introduces a number of new tasks, called *communication tasks*. These tasks reflects the communication as described in example 1, i.e. a write, a transfer, and a read task.

Example 2: Assume that we have to send the same data from p_1 to both p_3 and p_4 in figure 1, the transfer may be represented by the message tree as shown in figure 3. Notice

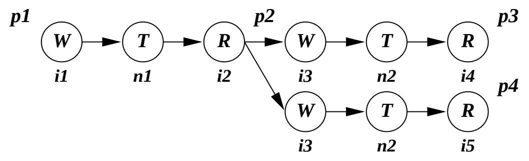


Figure 3: Example of a message tree.

that the data is first transferred to p_2 where it is stored in local memory. Then it is transferred independently from p_2 to p_3 and p_4, that is, p_3 and p_4 does not have to be ready at the same time.
□

$\mathtt{prio}_{d_{i,succ_i}}$ is used as priority for message scheduling during fitness evaluation.

5.2 Fitness evaluation

The fitness value is calculated as a cost summation of four contributions, the higher the cost is, the less fit is the individual. The four contributions reflects performance, area, local memory usage, and buffer memory usage.

$$C = C_T + C_A + C_M + C_B$$

In order to be able to compare and tradeoff the different contributions, we define a cost normalization function, $f_c(x)$ where x is the difference between the value of the constraint and that of the implementation. Figure 4 gives an outline of $f_c(x)$. An $x < 0$ means violation of the corresponding con-

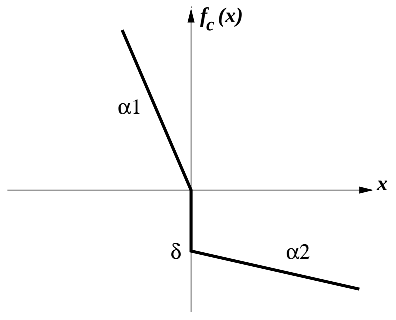


Figure 4: The cost normalization function.

straint and a high cost is associated with this situation. The actual cost is determined by the slope α_1. An $x \geq 0$ means meeting the constraint. δ determine how well this should be rewarded, in terms of a negative cost contribution, and the slope α_2 how well even better implementations should be rewarded.

Area

The simplest contribution is that of area,

$$C_A = \sum_{p_i \in E_A} f_c(A_{avail} - A_{used})$$

A_{used} depends on which tasks are allocated on the corresponding processing element and on the functions used by these tasks. I.e., the area used on processing element p_j is expressed as,

$$A_{used}(p_j) = \sum_{\tau_i \in p_j} A(\tau_i, p_j) + \sum_{f_k \in \bigcup_{\tau_i \in p_j} F(\tau_i)} A(f_k, p_j)$$

where f_k denotes a function (as explained in section 2.2), and $F(\tau_i)$ denotes all the functions used by τ_i. The first term is the area used by the tasks, whereas the second term is area used by the functions of the tasks, where each function is only implemented once.

Performance

Performance is calculated according to the deadline of the specification,

$$C_T = f_c(t_{deadline} - t_{sched})$$

The actual schedule, t_{sched}, is found by performing a list based scheduling of the tasks on their allocated processing elements, and of the communication tasks on the corresponding interfaces and nets.

List based scheduling relies on having a queue of ready tasks associated with each component. In our case we associate a *priority* queue with each processing element and use the priority $\mathtt{prio}_{\tau_i}$ when inserting task τ_i into the queue. For interfaces, we use a *FIFO* queue as the way to prioritize communication tasks, as this is the usual way to implement an interface[2].

The scheduling algorithm for a single individual (i.e. solution) is as follows:

1. Decode the gene string to obtain an allocation of the tasks and message trees for the dependencies. The decoding introduces a number of communication tasks to be inserted in the task graph as outlined in example 2. In the following a task may be an original task or a communication task.
2. Find all tasks τ_i which are ready to be scheduled, that is, tasks which has no predecessors. These tasks are inserted into the priority queues of their respective components (found from $\mathtt{impl}_{\tau_i}$) according to their priority ($\mathtt{prio}_{\tau_i}$).
3. Find the next point in time, t, where something happens in the schedule, i.e. the starting or ending of a task τ_i. If it is the end of a task, the end-point $t_{end}(\tau_i)$ is set to t, and the successors of τ_i, for which all of their predecessors already have been scheduled, are inserted into their respective queues If it is the start of a task, the start-point $t_{start}(\tau_i)$ is set to t.
4. If there are unscheduled tasks then goto step 3. Otherwise, the schedule is completed.

Finding the next point in time where something happens, is the most complicated task of the scheduling algorithm, and will be explained in more details in the following.

Let τ_{c_i} denote the last task on a component c_i (i.e. a processing element or an interface), that is, the task currently active on c_i or the last active task on c_i. The next task to be selected is the one which has the earliest time point, t_e, for its event, that being the starting or ending of its execution. This is determined as,

$$\min_{c_i \in G_A} (t_e(c_i))$$

where the earliest time point for an event on a component c_i is given as,

$$t_e(c_i) = \begin{cases} t_{end}(\tau_{c_i}) & \text{if } \tau_{c_i} \text{ is active} \\ \max(t_{end}(\tau_{c_i}), t_{est}(\tau_j, c_i)) & \text{otherwise} \end{cases}$$

that is, if a task is already active on c_i then the first event will be the ending of this task. If no task is currently active, the next event will be the earliest start time (t_{est}) of the next task τ_j on c_i, i.e. if c_i is a processing element, then it is the first task in the priority queue of c_i, else if c_i is an interface, it is the first task of the FIFO queue. This task is found as the maximum end-time of all predecessors of τ_j and the end time of the last active task on c_i.

Local Memory

Local memory is calculated according to the peak memory usage,

$$C_M = \sum_{p_i \in E_A} f_c(M_{avail}(p_i) - M_{peak}(p_i))$$

where the peak memory usage is calculated as described in example 1.

Buffer Memory

Buffer memory is also calculated according to the peak memory usage,

$$C_B = \sum_{i_i \in E_A} f_c(B_{avail}(i_i) - B_{peak}(i_i))$$

where the peak buffer memory usage only has a dynamic contribution. This contribution is calculated in the same way as for the local memory.

6 Experimental Results

The presented algorithm is implemented in Java and is integrated within the LYCOS [10] hardware/software cosynthesis system. All experiments in this section are carried out on a 166MHz Pentium MMX running JDK1.1 under Linux, and execution times are given in seconds.

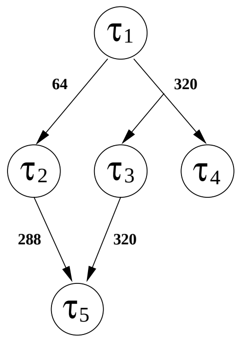
The first experiment is that of figure 5 using a deadline of 400. Assume that we have an architecture corresponding to figure 1, where the nets and interfaces are characterized as shown in table 1. In this experiment we assume that no task can execute on p_1 and that the processors p_2, p_3, and p_4 each have a local memory of 1024 units of data.

The task graph is first mapped to the architecture considering performance as the only cost. This results in a schedule, where τ_4 is allocated on p_2, τ_1 and τ_2 on p_3, and τ_3 and τ_5 on p_4. We get a solution with a schedule length of 323, which is much shorter than the 400 required, however,

[2]We are currently working on supporting other types of interfaces.

memory calculation shows, that p_3 and p_4 uses 22% and 29% more memory than available.

If all constraints are considered we get a schedule, where τ_2 and τ_4 are allocated on p_2, τ_1 and τ_3 on p_3, and τ_5 on p_4. Here all memory constraints are met and the schedule length of 380 is within the deadline.



		p_2	p_3	p_4
τ_1	time	80	60	80
	code	260	230	200
	data	90	90	110
	func	f_1	f_1	f_1
τ_2	time	100	80	80
	code	280	250	230
	data	110	1300	150
	func	f_2	f_2	f_2
τ_3	time	130	120	110
	code	220	220	250
	data	120	130	120
	func	f_1	f_1	f_1
τ_4	time	140	160	180
	code	180	180	200
	data	160	190	140
	func	f_2	f_2	f_2
τ_5	time	150	140	120
	code	310	280	320
	data	240	270	260
f_1	code	210	180	190
f_2	code	210	240	200

Figure 5: Task graph and task characterization.

Interface	package-size	trans. rate
i_1, i_2	16	1
i_3, i_4, i_5	32	1
Net		
n_1	16	6
n_2	32	8

Table 1: Interface and net parameters.

Finally, we illustrate how the algorithm can be used to solve larger mapping problems, which can not be solved in resonable time with exact methods like MILP [12]. We use the task graph generator TGFF proposed in [3] to generate a taskgraph with 47 tasks. The call to the generator is: tgff -n1 -e2:2 -N4:0 -c100:80 -T '100:80:t:exec 400:320:t:code 400:320:t:area 200:160:t:data'.

The properties of the task graph are the following. The average execution time is 100, code size is 400, data size is 200, area is 400, and the dependency data size is 100. The specification uses 23 different functions and the deadline is 2700. The architecture is that of figure 1, the nets and interfaces are characterized as shown in table 1, and the available memory on p_2, p_3 and p_4 are 5000. p_1 is characterized as an asic with the available memory of 1000, and an area of 4000.

The genetic algorithm optimize the mapping problem in 22 minutes (50 generations), but a solution which meets all the constraints is found after 26 generations. The optimized solution has a schedule length of 2632. The memory usage is shown in table 2. The area usage of p_1 is 3347.

7 Conclusion

We have presented a genetic algorithm which solves the problem of mapping a system specification onto a given architecture. The main advantage over previous approaches is that we handle the constraints of memory and buffer sizes which are typically found in embedded computer systems.

PE	available	static	dynamic	peak
p_1	1000	–	681	681
p_2	5000	3834	1144	4978
p_3	5000	3262	1104	4366
p_4	5000	3042	761	3803

Table 2: Optimized memory usage for the tgff taskgraph.

We are currently working on extending our approach to handle conditionals and system-level pipelining, as well as handling several interface types. We are also working on including passive components such as global memory and display units. Finally, we are working on improving the execution time of the genetic algorithm.

Acknowledgements

This work is supported by the Danish National Center for IT Research under grant no. CIT 149.

References

[1] A. Bender. Design of an Optimal Loosely Coupled Heterogeneous Multiprocessor System. In *European Design and Test Conference*, 1996.

[2] P. Bjørn-Jørgensen and J. Madsen. Critical Path Driven Cosynthesis for Heterogeneous Target Architectures. In *5th International Workshop on Hardware/Software Codesign*, pages 15 – 19, 1997.

[3] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task Graphs for Free. In *6th International Workshop on Hardware/Software Codesign, Codes'98*, pages 97 – 101, 1998.

[4] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley, 1989.

[5] E. S. H. Hou, N. Ansari, and H. Ren. A Genetic Algorithm for Multiprocessor Scheduling. *IEEE Transaction on Parallel and Distributed Systems*, 5(2):113 – 120, 1994.

[6] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM Journal of Computing*, 18(2):244 – 257, 1989.

[7] B. Keinhuis, E. Depretter, K. Vissers, and P. van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In *11th Int. Conference of Applications-specific Systems, Architectures and Processors*, pages 338 – 349, 1997.

[8] K. Kuchcinski. Embedded System Synthesis by Timing Constraints Solving. In *10th International Symposium on System Synthesis*, pages 50 – 57, 1997.

[9] Y.-K. Kwok and I. Ahmad. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Transaction on Parallel and Distributed Systems*, 7(5):506 – 521, 1996.

[10] J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen. LYCOS: the Lyngby Co-Synthesis System. *Design Automation for Embedded Systems*, 2(2):195 – 235, 1997.

[11] S. Prakash and A. C. Parker. SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 16:338 – 351, 1992.

[12] S. Prakash and A. C. Parker. Synthesis of Application-Specific Multiprocessor Systems Including Memory Components. *Journal of VLSI Signal Processing*, 8:97 – 116, 1994.

[13] G. C. Sih and E. A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transaction on Parallel and Distributed Systems*, 4(2):175 – 187, 1993.

[14] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transaction on Parallel and Distributed Systems*, 5(9):951 – 967, 1994.