Journal of Universal Computer Science, vol. 10, no. 2 (2004), 120-144
submitted: 8/8/03, accepted: 24/2/04, appeared: 28/2/04 © J.UCS

A Rapid Prototyping Environment for Multi-DSP Systems based on Accurate Performance Prediction

Bernhard Rinner
(Graz University of Technology, Austria
rinner@iti.tugraz.at)

Martin Schmid
(Graz University of Technology, Austria
schmid@iti.tugraz.at)

Reinhold Weiss
(Graz University of Technology, Austria
rweiss@iti.tugraz.at)

Abstract: In this paper we present PEPSY, a novel **p**rototyping **e**nvironment for multi-DS**P** **sy**stems, with the primary goal to support the design and implementation of parallel digital signal processing (DSP) applications subject to various design constraints. Given a specification of the prototyping problem in the form of an application model, a hardware model and mapping constraints, PEPSY automatically maps and schedules the DSP application onto the multi-processor system and synthesizes the complete code for each processor.

A detailed performance model of the parallel application is an integral part of PEPSY. Important performance parameters such as computation and communication times as well as memory consumption can be estimated prior to the implementation. PEPSY not only solves the standard mapping and scheduling problem, but it is also able to explore various important design goals for embedded systems and DSP applications such as minimizing memory and power consumption and enforcing the timeliness of tasks. Two complex case studies demonstrate the feasibility of our prototyping environment.

Key Words: rapid prototyping; scheduling; performance prediction; multi-DSP; embedded systems; data flow
Category: B.8.2; D.2.2; C.3

1 Introduction

Parallel processing is a key technique in satisfying the steadily increasing performance requirements of many applications. Parallel processing is applied to boost the performance not only of general-purpose applications using supercomputers but also of embedded systems with dedicated hardware. Examples of such embedded systems include applications in the field of digital signal processing (DSP) and mobile computing. The design and implementation of such parallel applications, however, are tedious and more complex than a single-processor solution. In times of high market pressure and ever decreasing time-to-market, support for the design and implementation of parallel DSP applications is crucial.

In the conventional design process, specification, analysis and implementation are applied in a strict sequential order. DSP applications are commonly specified using block-oriented descriptions or data flow models [LP95]. Analysis is then performed only at the simulation level. While a simulation-based analysis may result in a detailed functional evaluation, the performance can only be roughly evaluated. A main problem of this design process is that specification faults are only detected late in the process and the development time is, therefore, increased. In a design process based on *rapid prototyping* [DK96], the application can be analyzed on the target platform before implementation. Simulation and/or emulation can be used to perform a functional as well as performance analysis in detail. Specification faults can, therefore, be detected at early stages in the design process.

The key steps in developing parallel DSP applications are *partitioning, mapping* and *scheduling*. The overall application has to be partitioned into smaller units (tasks); these tasks have to be mapped onto individual processing elements; and the execution order of all tasks has to be determined for each processing element. Given the large number of possible partitionings, there are numerous potential mappings and schedules and finding the optimal solution is NP-complete in the general case and in several restricted cases [Ull75]. In the implementation process, code is written (or synthesized), compiled and linked for each processor. This code includes the application tasks as well as communication and synchronization routines.

We have developed Pepsy, a **p**rototyping **e**nvironment for multi-DS**P sy**stems, with the primary goal to support the design and implementation of parallel DSP applications [MSS99, RRS01, MRS$^+$00, RSW03]. Pepsy automatically maps a DSP application onto a multi-processor system, generates a static schedule for each processor and synthesizes the complete multi-processor source code (1). In order to approximate an optimal mapping and scheduling, Pepsy accurately predicts the performance of the parallel application. The estimated computation and communication times as well as other cost factors such as the memory usage and task deadlines are used to evaluate the design goals of the parallel application prior to its implementation.

The contributions of this research include:

- Pepsy provides a complete framework for prototyping various applications. This framework consists of (i) a specification notation for the prototyping problem, (ii) an optimizer for approximating an optimal mapping and scheduling, (iii) a code synthesizer which automatically generates, compiles and links code for the target system and (iv) a convenient graphical user interface to control the prototyping process. With the Pepsy framework the prototyping time of multi-processor applications can be dramatically reduced.

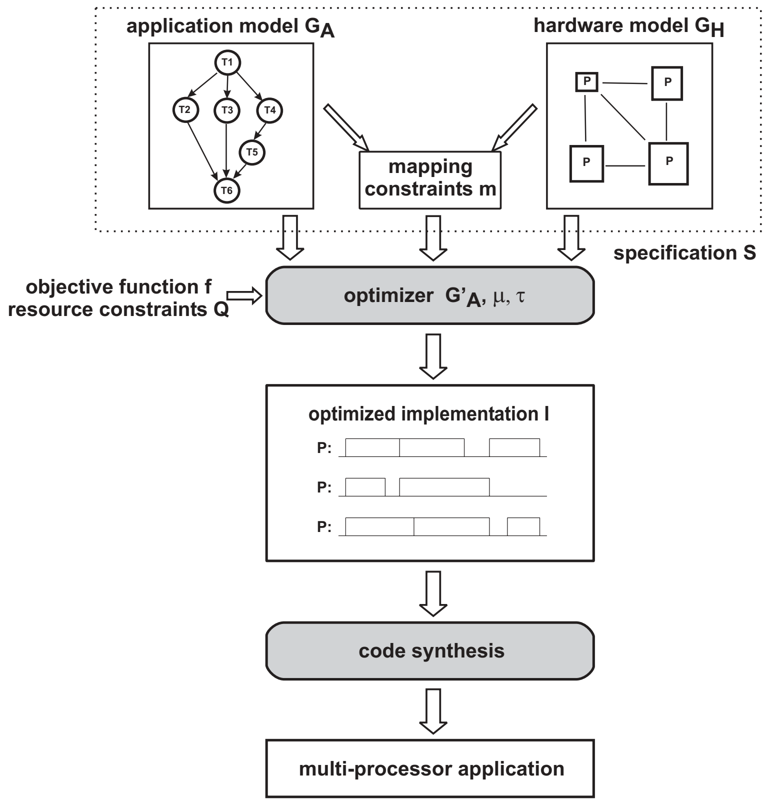

Figure 1: Overview of our prototyping framework Pepsy. Pepsy approximates an optimal implementation I given a specification S (consisting of an application model, a hardware model and mapping constraints) subject to an objective function and resource constraints. The optimizer extends the application model by introducing communication tasks and computes a mapping μ and schedule τ. The optimized implementation is then automatically synthesized.

- Pepsy casts the prototyping problem as approximating an optimal mapping and schedule for a given specification. Pepsy uses a generic optimizer based on simulated annealing [KJV83] for this optimization. Both Pepsy's specification notation as well as the optimizer are flexible and expressive. As a result, Pepsy not only solves the standard mapping and scheduling problem, but it is also able to explore various important design goals for embedded systems and DSP applications such as minimizing memory and power consumption and enforcing the timeliness of tasks.

- Pepsy's optimizer is combined with a detailed performance model for the parallel application. A main focus of this model is the inter-processor communication on the target system. This performance prediction can be applied to check the fulfillment of the design goals prior to its implementation. Due to the high accuracy of the performance prediction, simulation-based prototyping is often sufficient for the performance analysis resulting in a further reduction of the development time.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents an overview of our prototyping framework and introduces a formal specification of the prototyping problem. Section 4 describes how the prototyping problem is approximated by simulated annealing. PEPSY's generic optimizer is also presented, and our model for estimating the performance is discussed in detail. Section 5 describes the code synthesis of PEPSY. Section 6 presents PEPSY's prototyping capabilities demonstrated on two case studies in the area of digital signal processing. Section 7 concludes the paper with a discussion.

2 Related Work

There is a vast body of work known in the literature on design automation of multi-processor systems. We divide the work related to our prototyping framework into *design automation* and *task scheduling*.

2.1 Design Automation

Recently much effort has been performed in developing methods and tools for the automation of (embedded) system design [Wol94, GV95, SDMH00, SVM01]. The design problem typically consists of finding an optimal implementation for a task-level specification on a heterogeneous target system comprised of software programmable components and dedicated hardware. A huge design space must be explored considering several and sometimes competing objectives such as cost, power dissipation and computing performance [ETZ00].

There are related prototyping systems for digital signal applications known in the literature. Madisetti [Mad96] presented the state of the art and identified future challenges in prototyping large DSP systems in the mid nineties. He has developed a *conceptual prototyping* method for embedded DSP systems. This method partitions the overall design flow into several stages in order to allow the designer to focus on the right level of detail during the design process. This program has been implemented as part of the *rapid prototyping application-specific signal processor (RASSP)* DARPA program [HPK97].

Fresse et al. [FAD00] have developed a prototyping environment for a multi-DSP system targeted for image processing applications. Their environment requires a functional description in the form of a data flow graph and generates a parallel implementation onto a heterogeneous target platform. The timing information for the individual tasks is determined from a single-processor implementation. A comparison with our approach is difficult because no details about the mapping and scheduling have been presented.

The Grape-II [LEAP95] environment uses synchronous and cyclo-static data flow graphs as application models. Grape-II maps and schedules the application tasks onto a heterogeneous target system comprised of digital signal processors and dedicated hardware (FPGA) [GR01]. Data transfer is also realized using communication buffers [CLP97]. Although Grape-II provides more functionality such as HW/SW codesign than Pepsy, it has only a limited performance model. The performance can, therefore, only be evaluated by emulation.

Ptolemy and its successor Ptolemy II [BML99, DHK$^+$01] are frameworks for modeling and design of heterogeneous embedded systems. A major emphasis of Ptolemy lies on the methodology for defining and producing embedded software. It supports various models of computation such as data flow models, discrete event models and asynchronous message passing.

2.2 Task Scheduling

Task scheduling is known to be NP-complete in its general form as well as several restricted cases [ERAL95]. Researchers have, therefore, studied methods for *efficiently* finding suboptimal solutions to the scheduling problem. These methods can be grouped into *heuristic approaches* and *approximation approaches*.

The basis of most heuristic scheduling algorithms is the classical list-scheduling approach [ACD74, Gra69]. In list scheduling, the scheduler assigns the tasks priorities and places the tasks in a ready list arranged in a descending order of priority. There are several tools that support the development of parallel applications using heuristic schedulers. These tools include CASCH [AKW00], Parallax [LER93] and Pyrros [YG92]. They basically differ in their underlying assumptions on the task set, the available scheduling heuristics and the generated output.

Another method used in finding (sub-)optimal solutions of the scheduling problem is realized by casting scheduling as an optimization problem. Particularly interesting in the scheduling context are approximation methods based on genetic algorithms (GA), tabu search and simulated annealing (SA). All these methods avoid trapping into local minima of the optimization problem by random choices during the approximation. These approximation methods are further domain independent, i.e., they do not require auxiliary information such as gradient-based techniques or greedy methods in order to work properly. Multiprocessor scheduling approaches based on simulated annealing include [BM91], [BS96], [NS00] and [TBW92]. These approaches have been applied to a wide range of parallel applications and on various target architectures.

3 The Prototyping Framework Pepsy

3.1 Overview

Figure 1 depicts the overall architecture of our prototyping framework Pepsy. Pepsy supports the prototyping of data-flow oriented applications onto heterogeneous multi-processor systems, i.e., it computes an optimized multi-processor implementation given a specification, resource constraints and an objective function.

The specification S of the prototyping problem is based on two models. The application model describes the overall application in the form of an extended data-flow graph G_A [BML99], in which the nodes of this graph represent (functional) tasks of the application and the arcs represent data dependencies between the tasks. The hardware model describes the multi-processor system onto which the application is mapped. Each processing element is represented by a node of this graph G_H. Physical point-to-point connections are described by the arcs. Restrictions on the mapping of tasks onto processing elements may be specified by mapping constraints m.

The optimizer computes an optimized multi-processor implementation I, i.e., it approximates an optimal mapping and scheduling for all tasks given the specification S subject to an objective function f and resource constraints Q. The mapping and scheduling generated by the optimizer consist of a task list for each processor. This task list includes the application tasks as well as sender and receiver tasks introduced for the purpose of inter-processor communication. For each task, start and end times are estimated by the optimizer using our communication model for buffered data transfer [MRS+00]. This communication model is the basis of Pepsy's performance estimation. Additional parameters such as each task's memory requirement and deadline, and each processor's memory capacity, may be specified and can be used to express design goals more complex than the standard multi-processor scheduling problem.

The final step in our prototyping environment is automatic code generation and synthesis. The goal of this step is to generate, compile and link the complete source code for the multi-processor system.

3.2 Problem Formulation

We refine the description of our prototyping framework by a formal specification of the input, output and main intermediate steps of the framework depicted in Figure 1.

We start our problem formulation by defining the application model as a graph.

Definition 1. The *application model* is a directed graph $G_A(V_A, E_A)$ where V_A is the set of (application) tasks $T_1, \ldots, T_n$ and $E_A \subseteq (V_A \times V_A)$ is a set of arcs. The arcs correspond to the data dependencies between the tasks.

Resource parameters, represented by the set R_A, may be optionally specified. Resource parameters can be assigned to nodes or arcs.

In our framework the application is represented as a *data dependency graph* [LP95] augmented by resource parameters. Resource parameters are useful in specifying additional information required for the optimization step in our framework. Examples for such parameters include the execution time of tasks and the amount of data to be transferred between tasks.

Definition 2. The *hardware model* is an undirected graph $G_H(V_H, E_H)$ where V_H is the set of processing elements $P_1, \ldots, P_m$ and $E_H \subseteq (V_H \times V_H)$ is the set of communication links between processing elements.

Resource parameters, represented by the set R_H, may be optionally specified. Resource parameters can be assigned to nodes or edges.

Similarly to the application, the (target) hardware is modeled as an augmented undirected graph. Examples of resource parameters in the hardware model are the clock frequency and memory capacity of the processing elements and the transfer rates of the communication links.

Definition 3. The *mapping constraints* are represented by a function $m : V_A \times V_H \mapsto \{0, 1\}$ where $m(T_i, P_j) = 0$ indicates that no mapping of task T_i is allowed on processing element P_j. Conversely, $m(T_i, P_j) = 1$ indicates that a mapping is possible.

These constraints specify restrictions for the mapping of tasks onto processing elements. Given Definitions 1 to 3 we can now define a *specification* as it is used in our framework.

Definition 4. The *specification* of the prototyping problem is the tuple $S = (G_A, G_H, m)$.

With this specification, we can define a mapping of tasks onto processing elements.

Definition 5. A *mapping* of a specification S is a function $\mu : V_A \times V_H \mapsto \{0, 1\}$ such that $\forall_{T_i \in V_A} \exists_{P_j \in V_H}\ \mu(T_i, P_j) = 1$. The mapping function $\mu(T_i, P_j) = 1$ has the natural interpretation that task T_i is mapped onto processing element P_j. A mapping is *feasible* iff $\forall_{i,j}\ \mu(T_i, P_j) = 1 \Rightarrow m(T_i, P_j) = 1$, i.e., μ does not violate any mapping constraint.[1]

[1] Note that this definition allows to map the same task onto multiple processing elements. We can, therefore, introduce *replica tasks* into the overall schedule.

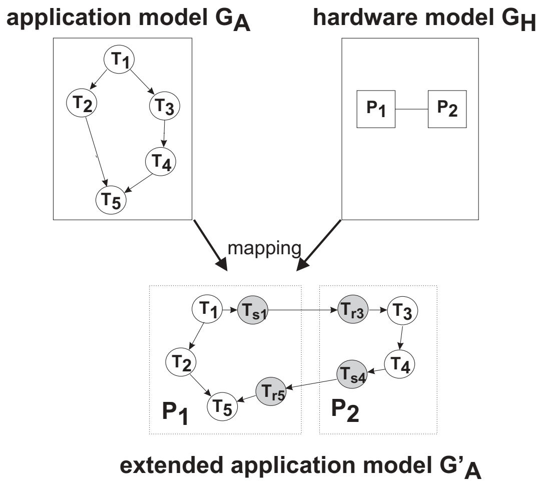

Figure 2: Extended application model G'_A. Every arc between tasks mapped onto different processing elements is replaced by a pair of communication tasks.

Definition 6. For a given specification S and a mapping μ a schedule is a function $\tau : V_A \mapsto \mathbb{R}$ that satisfies $\forall_{(T_i,T_j)\in E_A}\ \tau(T_j) \geq \tau(T_i) + \delta(T_i, \mu)$. $\tau(T_i)$ represents the start time of a task T_i and $\delta(T_i, \mu)$ represents the execution time of task T_i on a given mapping μ.

This definition of a schedule based on the start and execution times of all tasks allows only the orderings of tasks in time that do not violate the data dependencies given in G_A. Time for transferring data between tasks is not considered in Definition 6. This may be feasible for scheduling on a single processor but not for scheduling on a multi-processor system where communication times cannot be neglected. We account for inter-processor communication by extending our application model.

Definition 7. Given a specification S and a mapping μ, we replace every arc between tasks mapped onto different processing elements by a pair of communication tasks T_s and T_r and the corresponding arcs. Thus, all arcs

$$e = (T_i, T_j) \in E_A \ :\ \mu(T_i, P_k) = 1 \wedge \mu(T_j, P_l) = 1 \wedge k \neq l$$

are replaced by the tasks T_{si} and T_{rj} and the arcs (T_i, T_{si}), (T_{si}, T_{rj}) and (T_{rj}, T_j). The communication tasks are mapped onto the corresponding processing elements, i.e., $\mu(T_{si}, P_k) = 1$ and $\mu(T_{rj}, P_l) = 1$.

This *extended application model* is referred to as $G'_A = (V'_A, E'_A)$. The tasks from the original application model $T_i \in V_A$ are referred to as *application* tasks; T_{si} and T_{rj} are referred to as *communication* tasks.

Now, the time required for inter-processor communication can be expressed as components $\delta(T_s, \mu)$ and $\delta(T_r, \mu)$. Introducing communication tasks for inter-processor communication is also motivated from a practical point of view. In most multi-processor systems, inter-processor communication is realized by dedicated transfer functions. These functions can, therefore, be directly mapped onto communication tasks in our framework. The extended application model is depicted in Figure 2.

In general, resource constraints can be expressed as a set of functions Q : $q_i(R_A, R_H, \mu) \geq 0$. These constraints are typically used to set limits on resources such as memory consumption, completion times of tasks and other costs.

We define a *feasible* schedule τ of a specification S as a schedule that satisfies all resource constraints Q in addition to the mapping constraints and dependencies. This is also referred to as an *implementation* of a prototyping problem. Our goal can, therefore, be formulated as finding the *optimal* implementation.

Definition 8. Given a specification S and the resource constraints Q a (valid) *implementation* I is the tuple (μ, τ) where μ is a feasible mapping and τ is a feasible schedule.

Definition 9. Finding the optimal implementation is the following optimization problem:

minimize $f(\mu, \tau)$
subject to
  μ is a feasible mapping
  τ is a feasible schedule

In order to easily focus on different design goals, the objective function is specified as a weighted sum of several components in our prototyping framework. These components include parameters such as memory usage and completion times (cp. Section 4.1.4).

Given this specification notation, the remaining problem is how to compute the mapping μ and the schedule τ consistent with the resource constraints Q. We describe this computation in the following section.

4 Optimization and Performance Prediction

The key component of our prototyping framework is the optimizer which approximates an optimal mapping and scheduling. This approximation is based on simulated annealing and an accurate prediction of the performance on the target platform.

Simulated annealing (SA) is an optimization technique which belongs to the class of stochastic local search algorithms [KJV83, Ĉer85]. It avoids trapping into local minima during search by adding a stochastic component to the process of neighborhood state selection and acceptance [LA89].

The simulated annealing method is easy to adapt to different optimization problems because of its domain-independent structure. The adaption to a specific optimization problem requires the specification of four SA-elements: (i) a state description as manifestation of a problem solution, (ii) a mechanism to generate neighborhood states, (iii) a strategy to explore the neighborhood and (iv) a cost function to control the optimization process.

4.1 Casting Mapping and Scheduling as a SA-Problem

In order to cast the prototyping problem defined in Definition 9 as a SA-problem we have to specify the four SA-elements.

4.1.1 State Description

In our SA-framework, a state is defined as a schedule of the application tasks onto the processing elements. The state represents, therefore, the mapping of application tasks onto processing elements as well as their temporal ordering. There are a large number of possible states for a given problem specification (cp. Definition 4), and many of these states are infeasible due to the violation of some constraints. To reduce the number of states during optimization, we consider only states that do not violate the data dependency constraints as given in the application model G_A. The generation of new states is combined with a list scheduling algorithm. Therefore, states violating data dependencies are excluded from further processing and no expensive evaluation of the cost function is required.

The generation of the (initial) states consists of four steps. First, a global task priority list is generated that includes all application tasks. The tasks' priorities correspond to the partial task order specified in G_A. Second, the application tasks are mapped randomly onto the processing elements. As a consequence of this mapping, the global task priority list is then partitioned into local priority lists, one for each processing element. Next, the extended application model G'_A is generated by introducing communication tasks between adjacent tasks mapped onto different processing nodes. These communication tasks are also inserted into the priority lists. Finally, the cost (discussed in Section 4.1.4) of the initial state is computed based on PEPSY's performance estimation.

Note that the schedule can be directly derived from the priority lists by ordering the tasks with non-increasing priorities. Tasks with the same priority are randomly ordered.

4.1.2 Neighborhood Generation

During optimization, rearrangements of the system are applied by choosing neighbor states of the current state. Neighbor states are simply generated by moving an application task from one processing element to another. The task lists of both processing elements as well as the extended application model must then be updated. This may result in the deletion of old communication tasks and the insertion of new ones.

4.1.3 Neighborhood Exploration

Our SA optimizer explores only states that do not violate the data dependency constraints during optimization. However, the optimizer may generate non-feasible solutions (states) violating resource and mapping constraints. Such states are not rejected from the optimization process a priori. This helps in improving the convergence rate of the optimizer. Nevertheless, to avoid getting trapped in such a non-feasible state, these states are assigned with higher costs.

4.1.4 Cost Function

The cost function E corresponds to the objective function f of the optimizer and is computed for each state during the optimization process. In our SA-framework the cost function is a sum of weighted partial costs E_i:

$$E = E_{et}k_{et} + E_{ct}k_{ct} + E_{pm}k_{pm} + E_{mo}k_{mo} + E_{wm}k_{wm} + E_{wr}k_{wr} \tag{1}$$

The cost can be divided into components related to the solution quality, the resource constraints and the mapping constraints. Cost components related to the solution quality are (i) the overall completion time of the multi-processor application E_{et}, (ii) the total time required for inter-processor communication E_{ct} and (iii) the number of processors E_{pm} with at least one task mapped. A component related to resource constraints is the total memory overflow E_{mo}, i.e., the amount by which the required memory exceeds the amount of available memory on all processing elements. Partial costs related to mapping constraints are (i) the number of tasks mapped onto excluded processing elements E_{wm} and (ii) the number of replica tasks mapped onto identical processing elements E_{wr}.

Different optimization objectives are easily obtained by modifying the weighting factors k_i. To achieve only solutions that do not violate the mapping and resource constraints it is recommended to make the weights k_{mo}, k_{wm} and k_{wr} significantly larger than the other weights. The cost function can be easily extended to consider further resource constraints.

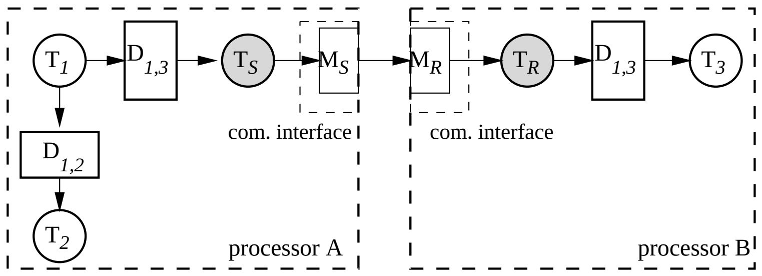

Figure 3: Realizing buffered intra- and inter-processor communication by introducing buffers (D_{ij}) and communication tasks (T_S and T_R). Inter-processor communication is realized by point-to-point links with hardware buffers of limited size (M_S and M_R) and may result in the synchronization of T_S and T_R.

4.2 Performance Prediction

The computation of the individual cost components of Equation 1 is based on a performance prediction of each state during optimization. Each feasible state corresponds to an implementation I of the prototyping problem. Pepsy's performance prediction includes the estimation of the memory utilization and the execution time. The performance prediction is based on the communication model applied in our prototyping framework.

4.2.1 Communication Model

Data transfer in Pepsy's optimized multi-processor applications is based on buffered communication (Figure 3). In this communication model each task writes its output data into a communication buffer. The task(s) receiving this data read(s) from that buffer. As a result of this model, asynchronous communication is guaranteed and both sender and receiver tasks are decoupled when the buffer size is larger than the amount of data transferred. To realize buffered communication, the optimizer must allocate communication buffers between tasks transferring data. If both tasks T_i and T_j are mapped onto the same processing element, a buffer of size D_{ij} is allocated. If the tasks are mapped onto different processing elements, a buffer of size D_{ij} is required on both processing elements. In this case, the optimizer additionally introduces a sender task T_S on one processor and a receiver tasks T_R on the other processor (see Figure 3).[2] T_S reads data from the buffer D_{ij} and writes it to the corresponding hardware buffer (M_S) of the communication interface, which is of fixed size. A similar transfer but in reverse order applies for task T_R whose hardware buffer is M_R.

[2] Introducing additional communication tasks corresponds to the extended application model G'_A.

symbol	*description*	*unit*
K_{Ai}	maximum execution time of each $T_i \in G_A$	cycles
M_{Ai}	memory (code and data) required for each $T_i \in G_A$	bytes
D_{ij}	size of transferred data between each $(T_i, T_j) \in E_A$	bytes
K_{Pi}	execution speed of each $P_i \in G_H$	cycles/sec.
M_{Pi}	amount of local memory on each $P_i \in G_H$	bytes
C_{ij}	transfer mode of each link $(P_i, P_j) \in E_H$	uni-/bidirec.
L_{Sij}	data send initialization time for each $(P_i, P_j) \in E_H$	cycles
L_{Rij}	data receive initialization for each $(P_i, P_j) \in E_H$	cycles
K_{Sij}	data transfer rate (send) for each $(P_i, P_j) \in E_H$	bytes/cycle
K_{Rij}	data transfer rate (receive) for each $(P_i, P_j) \in E_H$	bytes/cycle
M_{Sij}	size of the output buffer for each $(P_i, P_j) \in E_H$	bytes
M_{Rij}	size of the input buffer for each $(P_i, P_j) \in E_H$	bytes

Table 1: Resource parameters for our performance prediction. The first three parameters are included in R_A, the others are included in R_H.

To reduce the number of communication buffers, the optimizer allocates only a single buffer among tasks receiving the same input data from an individual task. These tasks are identified in the application model. Communication buffers can be allocated either statically or dynamically. Statically allocated buffers result in a faster execution. Dynamically allocated buffers are more memory efficient, since buffers can be released after the last receiving task has read the buffered data. Dynamically allocated buffers are useful in target systems with tight memory limitations such as embedded systems.

4.2.2 Resource Parameters and Mapping Constraints

Resource parameters (R_A and R_H) are useful in specifying additional information about the application and the target hardware, respectively. Table 1 presents the various resource parameters of the application and the hardware model that are used with our performance prediction.

There are two reasons for including mapping constraints in our framework. First, tasks with special resource requirements such as I/O may only be mapped onto special processing elements. Second, replica tasks must not be mapped onto the same processors for fault tolerance.

4.2.3 Memory Estimation

The memory requirement M_k of a processing element is the sum of the memory M_{Ai} required for T_i mapped onto P_k, and the memory required for all communication buffers on this processing element. Thus, the total memory required on processing element P_k is given as:

$$M_k = \sum_{\mu(T_i,P_k)=1} M_{Ai} + \sum_{e_{ij}:\mu(T_i,P_k)=1 \vee \mu(T_j,P_k)=1} D_{ij} \tag{2}$$

Note that a communication buffer is required for both intra- as well as inter-processor communication. Equation 2 derives an upper bound for the memory requirement since it does not consider allocating a single buffer among tasks receiving the same input or allow for dynamic buffer allocation. Both techniques may reduce the memory requirements but the reduction is strongly dependent on G_A.

4.2.4 Time Estimation

The execution time on each processing element is comprised of the execution times of the tasks mapped onto this processing element and the communication times between these tasks. The task execution times are specified in the resource parameters; no further estimation is therefore required. There are two kinds of communication: intra-processor communication and inter-processor communication. In our model, we assume that the intra-processor communication time is included in the task execution time. This assumption is reasonable since intra-processor communication is guaranteed not to be synchronized (or blocked). The intra-processor communication time is determined by writing or reading to or from communication buffers. These buffer access times can be easily included in the task execution times. As a result, only the inter-processor communication times have to be computed during the approximation process.

For simplicity in the following analysis, we omit the indices for identifying the sender and receiver tasks as well as for the resource parameters. Parameters associated with the sender and receiver tasks are labeled by S and R, respectively.

Due to the limited size of the input and output hardware buffers (M_R and M_S) of the processing elements, synchronization between sender and receiver tasks may occur in inter-processor communication. We model buffered inter-processor communication to determine the execution times of sender and receiver tasks transferring D data words over a buffered communication link of size $B = M_R + M_S$. K_S and K_R represent the transfer rates for writing to and reading from the buffers, respectively.

Figure 4 presents the timing diagram of the inter-processor data transfer between a sender and a receiver task. Three important time points can be identified for the sender as well as the receiver. At t_{S1} and t_{R1}, the sender and receiver tasks are initiated. After some initialization (L_S for the sender and L_R for the receiver), the sender task starts writing data words into the communication buffer at t_{S2}, and the receiver task starts reading data words from the communication

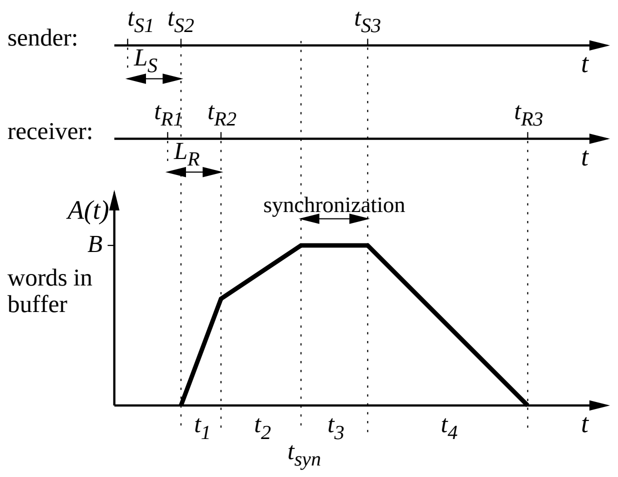

Figure 4: Timing diagram of a synchronized inter-processor communication.

buffer at t_{R2}. Writing and reading data to and from the buffer is finished at t_{S3} and t_{R3}, respectively.

Data transfer over a buffered communication link can be separated into four phases. If we know the duration for each phase, the execution times for the sender and receiver tasks can be determined. In phase 1, only the sender writes data into the buffer. The duration of this phase is given as $t_1 = t_{R1} + L_R - (t_{S1} + L_S)$. At the end of the first phase (t_{R2}), $A(t_{R2}) = \min(B, D, t_1 K_S)$ data words are stored in the buffer. In phase 2, both sender and receiver write/read data to/from the buffer asynchronously.[3] During this phase, the amount of data in the buffer is given by

$$A(t) = A(t_{R2}) + (K_S - K_R)(t - t_{R2}). \tag{3}$$

Phase 2 ends when synchronization between sender and receiver is enforced.

If we consider that the sender is faster than the receiver ($K_S > K_R$), synchronization is enforced when the buffer is completely filled ($A(t) = B$). Thus, by combining the synchronization condition with Equation 3, the synchronization time point can be determined as

$$t_{syn} = \frac{B - A(t_{R2})}{K_S - K_R} + t_{R2}. \tag{4}$$

Synchronization does not occur if too little data is transmitted to fill the buffer ($D \leq K_S(t_{syn} - t_{S2})$). In this case, phase 3 is skipped and the duration of

[3] Phase 2 is only entered if sufficient data has to be transmitted ($D > A(t_{R2})$).

phase 2 is given as $t_2 = \frac{D-A(t_{r2})}{K_S}$. If sufficient data is transmitted, the duration of phase 2 is $t_2 = t_{syn} - t_{R2}$. During phase 3 the sender and the receiver tasks are synchronized. Data is written to and read from the buffer at the slower transfer rate. In the case considered, the remaining data is written to the buffer at the speed of the receiver. Thus, the duration of phase 3 is given by $t_3 = \frac{D-K_S(t_{syn}-t_{S2})}{K_R}$. In the final phase, the receiver only reads data from the buffer. The duration of phase 4 is given as $t_4 = \frac{A(t_{S3})}{K_R}$.

For the other cases, i.e., if the sender is at the same speed or slower than the receiver ($K_S \leq K_R$), the durations of phases 2 to 4 can be determined similarly. To summarize, the total execution time for the sender task T_S to write D words into the communication buffer is $t_S = L_S + t_1 + t_2 + t_3$, and the execution time for the receiver task T_R to read D words from the communication buffer is $t_R = L_R + t_2 + t_3 + t_4$.

With these equations, the time for inter-processor communication can be estimated resulting in the computation of the total execution times of each processing element. The individual costs as specified in Section 4.1.4 can be determined given the memory requirements M_k as well as the execution and communication times (K_{Ai}, t_{Rij} and t_{Sij}) for all tasks mapped onto processing element P_k.

5 Code Synthesis

Central to Pepsy's code synthesis is the generation of an *executive* for each processing element. The executive corresponds to the top-level function of each processing element and is responsible for the memory allocation and the execution of all application and communication tasks in the order given by the optimized static schedule. The complete multi-processor application, i.e., executable code for all processing elements, is finally synthesized using commercial compiler and linker tools.

In order to synthesize the complete multi-processor application, we need the code for the application tasks, the communication and synchronization routines, the memory allocation and the executive. Pepsy's code synthesis requires the user-specified code for the application tasks and the hardware-specific code for communication and synchronization as input. The executive is automatically generated by Pepsy's code generator.[4]

In the following, we briefly describe the main code modules.

Application Tasks. The user has to provide the code for all application tasks in the form of a source code library which must follow the buffered communication model, i.e., the input and output to a task is given by (pointers to) communication buffers. In the source code library, each application task

[4] In the current implementation, Pepsy generates ANSI-C source code.

T_i is realized as an individual function with a well-defined interface (function prototype). Each function has a unique name which corresponds to the task identifier specified in the application model. Thus, the interface of the function `taski` that implements task T_i looks like:

$$\texttt{void taski}(inb_1, \ldots, inb_m, outb_1, \ldots, outb_n).$$

Formal parameters of this function are the references (pointers) to all input buffers inb_k where $k = 1 \ldots m$, and all output buffers $outb_l$ where $l = 1 \ldots n$. A function implementing a task must not return a value.

Communication Routines. Inter-processor communication is realized by introducing dedicated communication tasks that transfer data from a buffer to a different processor or vice versa. Buffered inter-processor communication requires that two communication routines be available on each processing element: a sender function `send` and a receiver function `receive`. These functions are hardware-specific and their code must be provided for each target system. Formal parameters of both functions are the reference to the output or input buffer, respectively, and the identifier of the destination processor.

Memory Allocation. In our buffered communication model, a communication buffer b_k is required for each arc in our extended application model. The size s_k of this communication buffer is determined by the amount of transferred data which is specified in the application model using resource parameters. For the code synthesis, buffers with sufficient size must be provided by either *static* or *dynamic* allocation.

Executive. The main steps in the automatic generation of the executive source code are as follows: First, unique names for the communication buffers are generated. Second, code for the memory allocation is inserted at the beginning of the executive file. Third, the call to the executive function is inserted. Finally, the function calls for the application and communication tasks are generated in the order given by the schedule. The formal parameters are replaced by the actual buffer names and the processor identifiers.

6 Experimental Results

Over the last few years the complete prototyping framework for multi-DSP systems, PEPSY has been developed. A graphical user interface implemented in Java combines the individual tools of PEPSY. PEPSY's SA-optimizer and the code synthesis tool have been written in C and Java, respectively. The data transfer between the individual components as well as their configuration is realized by XML files.

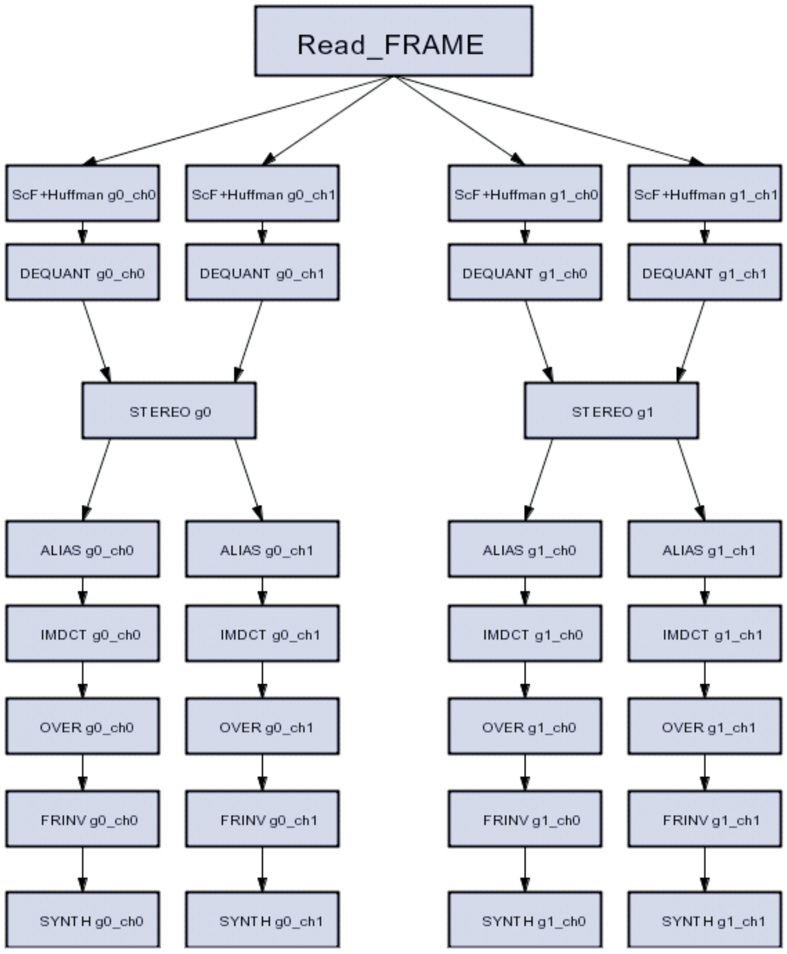

Figure 5: Screen shot of the task graph of the MP3 decoder.

6.1 Parallel Implementation of an MP3 Decoder

We demonstrate the automated design of a multi-processor application with Pepsy by parallelizing an MPEG-1 layer 3 (MP3) decoder. Figure 5 shows the task graph of this example generated with Pepsy's graph editor. MP3 decoding is decomposed into 31 tasks.

The target hardware platform consists of 4 TMS320C40 TIM modules on a TRANSTECH ISA daughterboard each processor running at $50MHz$; each of these processors has at least one direct link to every other processor. No special mapping constraints or resource constraints were taken into account for this sample design.

In this design example the primary goal was to minimize the overall execution time. Thus, the weighting factor were given as $k_{et} = 1$, and all other cost components were not considered, i.e., their weighting factors were specified as zero. The achieved implementation resulted in a final schedule with a total of 28 communication tasks introduced to realize inter-processor communication. The speedup of this multi-processor application is estimated as 3.911.

6.2 Optimization Performance

We evaluate our SA-based optimizer by comparing its performance with an exhaustive search over all mappings and schedules. The exhaustive search finds the optimal solution of the parallelization problem. However, it is only feasible for small problem sizes.

Three different problem classes have been selected for this comparison; all of them are scalable in the sense that the number of tasks can be varied. The target platform is fixed for all problem classes. It consists of a homogeneous four-processor multi-DSP system connected in a ring structure. The problem class "independent" consists of a set of independent tasks with different task execution times. The problem class "recursive" corresponds to an application with two recursive procedure calls per level. In the upper part of the task graph, each task is connected to two child tasks doubling the number of tasks at each level. In the lower part, each task has two predecessors converging to a single task at the bottom level. The problem class "chained" corresponds to an application with parallel threads that start and terminate at a single task. All tasks in the classes "recursive" and "chained" have the same execution times, and only a single word is transferred between adjacent tasks.

Table 2 compares the (optimal) costs and computation times achieved by exhaustive search with the costs and computation times achieved by our SA-optimizer. These results demonstrate that our optimizer finds solutions very close to the optimum within a reasonable time. This is true also for complex problems that are infeasible for the exhaustive search. For all problem classes, the deviation from the optimal solution has never exceeded 2%.

6.3 Automatic Parallelization of a Complex Audio Application

In this case study, we demonstrate the performance of PEPSY on the parallelization of a complex audio application, i.e., a simulator of the human peripheral auditory system is automatically mapped and scheduled onto a multi-DSP system. Based on a functional model of the human ear, this simulator generates the excitation pattern for the auditory nerve given an audio signal as input [MWB98]. The model of the human ear consists of various (non-linear) filters

problem class	*# tasks*	*exhaustive search*		*simulated annealing*	
		cost	*computation time*	*cost*	*computation time*
independent	10	140	7	140	4
	11	170	24	170	5
	12	200	88	200	2
	13	230	408	230	3
	14	270	1708	270	2
	22	n.a.	10^{8*}	640	4
	46	n.a.	$3 \cdot 10^{22*}$	2710	136
	94	n.a.	$2 \cdot 10^{51*}$	11170	1243
recursive	10	504	86	504	17
	22	n.a.	10^{9*}	810	180
	46	n.a.	$4 \cdot 10^{23*}$	1421	584
	94	n.a.	$3 \cdot 10^{52*}$	2656	5309
chained	10	405	128	407	25
	14	505	32768^{*}	512	57
	22	705	$2 \cdot 10^{9*}$	714	135
	46	1505	$6 \cdot 10^{23*}$	1531	1051
	94	2505	$5 \cdot 10^{52*}$	2555	3719

Table 2: Comparison of exhaustive search and our SA-based optimizer with regard to cost and computation times (in s) for three different problem classes. Results marked with '*' have been analytically extrapolated.

and transformation functions. A single-processor implementation of the simulator written in C and assembler serves as the starting point for our evaluation.

To derive an application model, we have partitioned the simulator into 93 tasks. This partitioning has been motivated by the functional structure of the simulator. In most cases, the individual tasks correspond to transformations and filtering steps such as Fourier transformations and gamma tone filters. Figure 6 depicts an abstracted data flow graph of the simulator. Tasks which require the same input data are combined to blocks in this graph. The execution times as well as the memory consumption of all 93 tasks have been measured using the simulator running on a single processor.

The PPDS from Texas Instruments is used as the target system. This multi-DSP platform consists of four TMS320C40 processors running at 32 MHz; each of these processors has at least one direct link to every other processor. The resource parameters R_H for this platform have been determined by several experiments. They are given as $K_{Pi} = 1, M_{Pi} = 64000\ bytes, L_S = L_R = 0\ cycles, K_S = K_R = 0.05\ bytes/cycle$ and $M_S = M_R = 8\ bytes$. The simulation

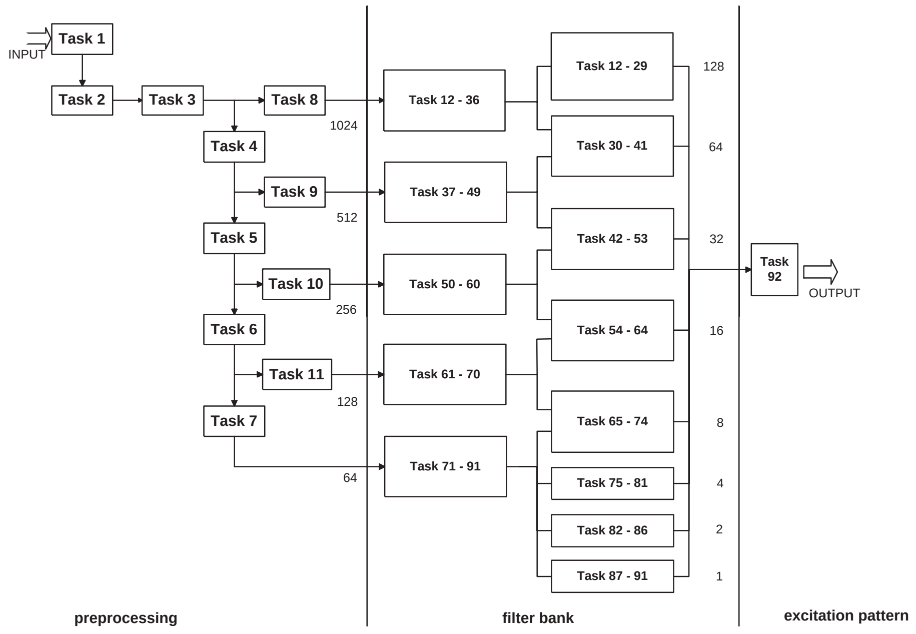

Figure 6: Task graph of the simulator of the human peripheral auditory system. The task required for sampling is not depicted in this figure.

Proc.	*AT*	*CT*	*Data*	t_c [ms]
A	22	68	5385	76.4
B	18	23	5304	75.5
C	24	31	3364	75.5
D	29	34	5805	75.5
total	93	156	19858	76.4

Table 3: Optimization result. The optimizer maps the application tasks *(AT)* and introduces communication tasks *(CT)* onto each processor. For each processor, the number of transferred data and the completion time (t_c) are shown in the last two columns.

of a block of 1024 data samples requires 251.7 ms on a single processor and serves as reference for this evaluation.

Table 3 summarizes the results of the optimization step for the parallelization of the simulator onto 4 processors labeled A to D. The optimizer introduces a total of 156 communication tasks to implement the data transfer among all 4 processors. A total of 19858 bytes is transferred between the processor and thus

	performance prediction		*measurement*	
Proc.	t_{comp}	t_{comm}	t_{comp}	t_{comm}
A	60.0	8.4	59.9	7.5
B	63.0	7.3	63.4	7.2
C	62.3	4.8	62.5	4.4
D	66.8	8.0	69.9	7.3

Table 4: Comparison of the overall computation and communication time t_{comp} and t_{comm} estimated by the optimizer and measured on the multi-DSP system for each processor. All times are given in ms.

represents the total memory required for inter-processor communication. The last column shows the completion time on each processor, i.e., the time when the last task in the processor schedule terminates. Processor A has the longest completion time because the last task of the overall data flow graph is mapped onto this processor. Thus, the optimizer estimates the overall execution time for the four processor solution as 76.4 ms.

The complete C-code for all executives has been automatically generated by our prototyping environment. The application tasks have been synthesized from the single-processor code with only minor modifications. Parameterized functions implementing individual tasks have been wrapped by an additional function to realize the task interface corresponding to our convention. Code for the static allocation of all necessary communication buffers has been introduced. The communication routines (`send` and `receive`) have been provided for the target system.

The parallel implementation results in an overall execution time of 75.9 ms which is almost identical to the estimated execution time. The overall speedup of the parallel implementation is, therefore, given as 3.3. Table 4 compares the computation and communication times derived by Pepsy's performance estimation with the times measured on the four-processor implementation automatically generated by the synthesis step of our prototyping environment. The estimated times are very close to the measured times on the implementation. The maximum deviation for the computation time is 4 % and 10 % for the communication time, respectively.

7 Conclusion

In this paper we have presented Pepsy, a novel prototyping environment for the automated design and implementation of parallel DSP applications subject to various design constraints. Our environment is grounded on a formal specification of the prototyping problem, an accurate modeling of the performance on all processors and a generic optimizer based on simulated annealing. Our SA optimizer results in a solution of the prototyping problem that is close to the optimum.

As demonstrated in a complex audio application, PEPSY's performance estimation is also very close to the measured performance on the implemented multi-processor application. This is due to the following reasons. First, the optimizer uses measured instead of simulated or estimated task execution times. The tasks in this audio application have almost no data dependency and, therefore, almost no variation of the execution times. Second, the optimizer uses an accurate communication model to estimate the (inter-processor) communication times. This model accounts also for the blocking of a communication task as well as synchronization between sender and receiver.

Our prototyping environment dramatically reduces the development time of multi-processor applications. Typically, the most time-consuming procedure in the development is the generation of the application model. When the code for the application tasks is already available, the development time is basically determined by the execution time of the optimizer. As a result, a parallelization onto a different number of processors and/or a different target system can be realized within minutes.

PEPSY has already been extended by a reconfiguration environment in order to enable a modification of software tasks while the multi-DSP application is running [RSW03]. Directions for future work include (i) extending this framework to various target platforms, (ii) integrating additional different design constraints for embedded systems such as minimizing the energy consumption, and (iii) investigating the feasibility of this framework to hardware/software codesign.

References

[ACD74] T.L. Adam, K.M. Chandy, and J. Dickson. A Comparison of List Scheduling for Parallel Processing Systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[AKW00] Ishfaq Ahmad, Yu-Kwong Kwok, and Min-You Wu. CASCH: A Tool for Computer-Aided Scheduling. *IEEE Concurrency*, 8(4):21–33, 2000.

[BM91] S. Wayne Bollinger and Scott F. Midkiff. Heuristic Technique for Processor and Link Assignment in Multicomputers. *IEEE Transactions on Computers*, 40(3):325–333, March 1991.

[BML99] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2), 1999.

[BS96] James E. Beck and Daniel P. Siewiorek. Simulated Annealing Applied to Multicomputer Task Allocation and Processor Specification. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 232–239, 1996.

[Ĉer85] V. Ĉerny. Thermodynamical Approach to the Traveling Salesman Problem: an Efficient Simulation Algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[CLP97] L. De Coster, R. Lauwereins, and J.A. Peperstraete. Data routing in dataflow graphs. In *Proceedings of the 8th Interantional Workshop on rapid System Prototyping (RSP '97)*, pages 100–106, 1997.

[DHK+01] John Davis, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong. Heterogeneous Concurrent Modeling and Design in Java. Technical Report Memorandum UCB/ERL M01/12, Department of Electrical Engineering and Computer Science University of California at Berkeley, 2001.

[DK96] Apostolos Dollas and Nick Kanopoulos. Reducing the Time to Market Through Rapid Prototyping. *IEEE Computer*, 28(2):14–15, 1996.

[ERAL95] Hesham El-Rewini, Hesham H. Ali, and Ted Lewis. Task Scheduling in Multiprocessing Systems. *IEEE Computer*, 28(12):27–37, December 1995.

[ETZ00] Michael Eisenring, Lothar Thiele, and Eckart Zitzler. Conflicting criteria in embedded system design. *IEEE Design & Test of Computers*, 17(2):51–59, 2000.

[FAD00] Virginie Fresse, Mustapha Assouil, and Olivier Deforges. Rapid prototyping of image processing applications onto a multiprocessor architecture. In *International Conference on Acoustics, Speech, and Signal Processing ICASSP2000*, May 2000.

[GR01] Varghese George and Jan M. Rabaey. *Low-Energy FPGAs - Architecture and Design*. Kluwer Academic Publishers, 2001.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[GV95] Daniel D. Gajski and Frank Vahid. Specification and Design of Embedded Hardware-Software. *IEEE Design & Test of Computers*, 12(1):53–67, 1995.

[HPK97] C. Hein, J. Pridgen, and W. Klein. *RASSP Virtual Prototyping of DSP Systems*. ACM, Anaheim, CA, 1997.

[KJV83] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[LA89] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers, 1989.

[LEAP95] Rudy Lauwereins, Marc Engels, Marleen Ade, and J.A. Peperstraete. Grape-II:A System-Level Prototyping Environment for DSP Applications. *IEEE Computer*, 28(2):35–43, February 1995.

[LER93] Ted Lewis and Hesham El-Rewini. Parallax: A Tool for Parallel Program Scheduling. *IEEE Parallel & Distributed Technology*, 1(2):62–72, May 1993.

[LP95] Edward A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[Mad96] Vijay K. Madisetti. Rapid Digital System Prototyping: Current Practice, Future Challenges. *IEEE Design & Test of Computers*, 13(3):12–22, 1996.

[MRS+00] Claudia Mathis, Bernhard Rinner, Martin Schmid, Reinhard Schneider, and Reinhold Weiss. A New Approach to Model Communication for Mapping and Scheduling DSP-Applications. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing ICASSP 2000*, pages 3354–3357, Istanbul, Turkey, June 2000. IEEE.

[MSS99] Claudia Mathis, Martin Schmid, and Reinhard Schneider. A Flexible Tool for Mapping and Scheduling Real-Time Applications onto Parallel Systems. In R. Wyrzykowski, B. Mochnacki, H. Piech, and J. Szopa, editors, *Proceedings of the Third International Conference on Parallel Processing & Applied Mathematics*, pages 437–444, Kazimierz Dolny, Poland, September 1999. Institute of Mathematics and Computer Science, Technical University of Czestochowa.

[MWB98] Claudia Mathis, Reinhold Weiss, and Rainer Buechel. Design and Experimental Evaluation of a Multi-DSP based Simulation of the Human Peripheral Auditory System. In *Proceedings of the International Conference on*

Signal Processing Applications & Technologies (ICSPAT), volume 2, pages 1098–1102. Miller Freeman, September 13-16 1998.

[NS00] Marco Di Natale and John A. Stankovic. Scheduling Distributed Real-Time Tasks with Minimum Jitter. *IEEE Transactions on Computers*, 49(4):303–316, 2000.

[RRS01] Bernhard Rinner, Bernd Ruprechter, and Martin Schmid. Rapid Prototyping of Multi-DSP Systems Based on Accurate Performance Estimation. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing ICASSP 2001*, Salt Lake City, U.S.A., May 2001. IEEE.

[RSW03] Bernhard Rinner, Martin Schmid, and Reinhold Weiss. Rapid Prototyping of flexible Embedded Systems on multi-DSP Architectures. In *Proceedings of the Design, Automation & Test in Europe Conference (DATE 03)*, pages 204–209, Munich, Germany, March 2003.

[SDMH00] Frank Slomka, Matthias Dorfel, Ralf Munzenberger, and Richard Hofmann. Hardware/Software Codesign and Rapid Prototyping of Embedded Systems. *IEEE Design & Test of Computers*, 17(2):28–38, 2000.

[SVM01] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design & Test of Computers*, 18(6):23–33, 2001.

[TBW92] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy. *The Journal of Real-Time Systems*, 4(2):145–165, 1992.

[Ull75] J. Ullmann. NP-Complete Scheduling Problems. *Journal on Computers and System Sciences*, 10:384–393, 1975.

[Wol94] Wayne Wolf. Hardware-Software Co-Design for Embedded Systems. *IEEE Proceedings*, 82(7):967–989, July 1994.

[YG92] T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, pages 428–443, New York, 1992. ACM.