RAPID PROTOTYPING OF MULTI-DSP SYSTEMS BASED ON ACCURATE PERFORMANCE ESTIMATION

Bernhard Rinner, Bernd Ruprechter and Martin Schmid

Institute for Technical Informatics
Graz University of Technology, AUSTRIA
http://www.iti.tu-graz.ac.at

ABSTRACT

The development of parallel applications is tedious and more complex than a single-processor solution. We have developed PEPSY, a **p**rototyping **e**nvironment for multi-DS**P** **sy**stems, with the primary goal to automate the design and implementation of parallel DSP applications. Given an extended data flow graph of the DSP application and a description of the target multi-processor system, PEPSY automatically maps and schedules the DSP application onto the multi-processor system and generates complete code for each processor.

PEPSY excels in an accurate performance estimation. The design goals of the parallel application can, therefore, be verified prior to its implementation. With PEPSY, parallelization of a DSP application onto various processors can be realized within minutes.

keywords: multi-DSP; rapid prototyping; performance estimation; automatic code generation; PEPSY

1. INTRODUCTION

Parallel processing is a key technique to satisfy the steadily increasing performance requirements of applications in the field of digital signal processing (DSP). 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, automization of the design and implementation process for parallel DSP applications is crucial.

The design process of parallel DSP applications typically consists of *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 myriads of potential mappings and schedulings. However, finding the optimal solution is difficult. 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 automate the design and implementation of parallel DSP applications [1, 2]. 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 that can be directly compiled and linked. In order to generate an optimal mapping and scheduling, PEPSY estimates accurately the performance of the parallel application. The estimated computation and communication times as well as the memory usage are used to verify the design goals of the parallel application prior to its implementation.

In this paper, we focus on PEPSY's code synthesis and the experimental evaluation of PEPSY's performance evaluation. We start with a brief overview of PEPSY. We then describe PEPSY's code synthesis in more detail and demonstrate the performance of our prototyping environment by the (automated) parallelization of a complex audio application. A brief discussion and a summary of related work conclude this paper.

2. PROTOTYPING MULTI-DSP SYSTEMS

Figure 1 depicts the overall architecture of PEPSY. PEPSY automates the parallelization of data-flow oriented applications onto heterogeneous multi-processor systems.[1] Data-flow oriented applications are common in the field of signal processing.

Two models serve as the primary input for the prototyping environment. The application model describes the overall DSP application in form of an extended data flow graph [3], i.e., the nodes and arcs of this graph are augmented by additional information such as task execution times, required memory and amount of transferred data. The hardware model describes the multi-processor system onto which the DSP application is mapped. Each processor is characterized by parameters such as its execution speed and size of local memory. Physical point-to-point connections are described by the features of the communication interfaces. Mapping constraints between application and hardware model may be specified and serve as an optional input to the optimizer.

The optimizer approximates an optimal mapping and scheduling for all tasks given the application model, hardware model and mapping constraints. PEPSY's optimizer is based on simulated annealing which allows a formal specification of different optimization objectives. In order to calculate this approximation, the optimizer determines the

[1] In the current implementation, we only consider distributed memory systems with point-to-point connections.

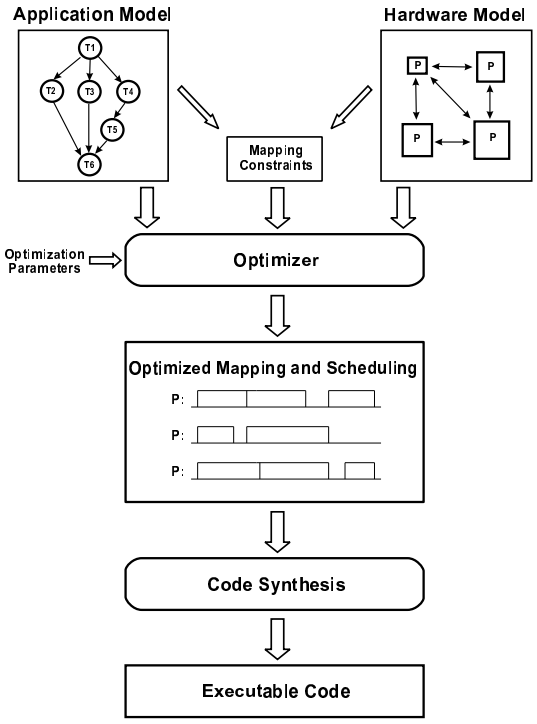

Figure 1: Overall architecture of PEPSY, our prototyping environment for multi-DSP systems.

memory usage as well as the execution and communication times of all tasks mapped on a single processor. To decouple sender and receiver, data transfer between tasks is realized by dedicated communication buffers. A task writes its data into a communication buffer of sufficient size; the task(s) receiving this data read(s) from that buffer. Inter-processor communication is realized by introducing a dedicated sender task on one processor and a receiver task on the other processor.

The optimized mapping and scheduling generated by the optimizer consists of a task list for each processor. This task list includes the application tasks as well as the sender and receiver tasks introduced for inter-processor communication. For each task, start and end times are estimated by the optimizer using our communication model for buffered data transfer [2]. This communication model is the basis of PEPSY's performance estimation.

The final step in our prototyping environment is automatic code generation and synthesis. The goal of this step is to generate the complete multi-processor source code that can be directly compiled and linked.

3. AUTOMATIC CODE GENERATION AND SYNTHESIS

Figure 2 shows the structure of PEPSY's code synthesis. In order to generate the complete multi-processor code, we need the source code for the application tasks, the communication and synchronization routines, the memory allocation and an *executive*. The executive processes all mapped tasks in the order given by the schedule for each processor. It includes the allocation of all required communication buffers and calls to functions implementing the application and communication tasks. Code for the application tasks is specified by the user; code for the communication tasks is hardware-specific and has to be provided for an individual multi-processor system. PEPSY automatically generates the complete source code for the executive.[2] Finally, executable code is generated for each processor by using target-specific (and commercial) tools for compiling and linking.

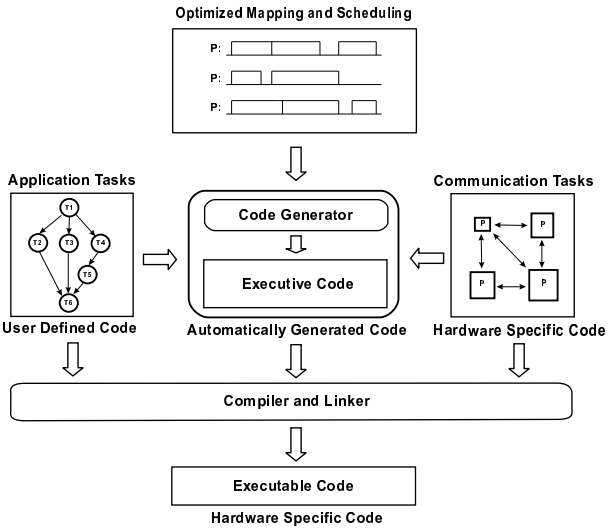

Figure 2: Code Synthesis in PEPSY.

3.1. Application Tasks

The user provides the code for all application tasks in form of a source code library. The application code has to follow our buffered communication model, i.e., the input and output to the task is provided by (pointers to) communication buffers. In the source code library, each application task T_i is realized as an individual function with a well-defined interface (function prototype). Each function has a unique name which is also 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.

3.2. 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. There are only two instances of communication routines implemented on each processor: a sender function `send` and a receiver function `receive`. These functions are hardware-specific and

[2]In the current implementation, PEPSY generates ANSI-C source code.

their code has to 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.

To implement a specific communication task, the sender or receiver function must be called with corresponding parameters. Thus, the code for a communication task sending data from buffer $outb_k$ to processor i or receiving data from processor i to buffer inb_k looks like:

$$\texttt{send}(outb_k, i)$$
$$\texttt{receive}(inb_k, i)$$

3.3. Memory Allocation

In our buffered communication model, a communication buffer b_k is required for each arc in our extended data flow graph. The size s_k of the communication buffer is determined by the amount of transferred data which is also specified in the extended data flow graph. For the code synthesis, buffers with sufficient size have to be provided. These buffers can be provided either by *static* or *dynamic* allocation. Statically allocated buffers result in a faster execution of the executive. 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. Code for dynamic memory allocation (`allocate` and `release`) is target-specific and has to be provided for each target system.

3.4. 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 of 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.

Figure 3 depicts two fragments of a code automatically generated for the executive. In the left column, the communication buffers are statically allocated. A unique buffer is allocated for each arc in the data flow graph. The entry point (function name) for each task is taken from the task list generated by the optimizer. In the right column, the code for the same fragment is generated using dynamic buffer allocation. The communication buffer is allocated before the first task writes data to it and is released after the last task has read all data from it.

4. EXPERIMENTAL EVALUATION

We demonstrate the performance of our prototyping environment by the parallelization of a complex audio application, i.e., a simulator of the human peripheral auditory system is automatically distributed onto a multi-DSP system. Based on a functional model of the human ear, this simulator generates the excitation pattern for the auditory

```
D_TYPE b1[BSIZE1];
D_TYPE b2[BSIZE2];
D_TYPE b3[BSIZE3];
void executive()
{
  receive(b1,p1);
  task1(b1,b2);
  send(b2,p2);
  task3(b2,b3);
  send(b3,p3);
}
```

```
D_TYPE b1;
D_TYPE b2;
D_TYPE b3;
void executive()
{
  allocate(b1,BSIZE1);
  receive(b1,p1);
  allocate(b2,BSIZE2);
  task1(b1,b2);
  release(b1);
  send(b2,p2);
  allocate(b3,BSIZE3);
  task3(b2,b3);
  release(b2);
  send(b3,p3);
  release(b3);
}
```

Figure 3: Automatically generated code for the executive with statically allocated (left column) and dynamically allocated (right column) communication buffers.

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

Table 1: 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.

nerve given an audio signal as input [4]. The model of the human ear is compromised of various (non-linear) filters and transformation functions.

A single-processor implementation of the simulator written in C and assembler serves as the starting point for our evaluation. In order to parallelize the simulator using our prototyping environment, we have to provide a hardware model and an application model. 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. To model the simulator as an extended data flow graph, we partition the simulator into 93 tasks. The execution times of all 93 tasks have been measured using the simulator running on a single TMS320C40 processor. The simulation of a block of 1024 data samples requires 251.7 ms on a single processor and serves as reference for our evaluation.

Table 1 summarizes the result of the optimization step for the mapping and scheduling of the simulator onto 4 processors labeled A to D. The columns labeled AT and CT show the distribution of the application and communication tasks among all processors. The optimizer introduces a total of 156 communication tasks to implement the data transfer among all 4 processors. The fourth column in Table 1 shows the number of data words transferred from and

	Optimizer		*Implementation*	
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 2: 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.

to each processor and thus 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 2 compares the computation and communication times estimated by the optimizer 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.

5. DISCUSSION

Optimized design and implementation of parallel DSP applications require an accurate estimation of the computation and communication times of all processors. As demonstrated in the complex audio application, PEPSY's estimation is very close to the measured times. This is due to the following reasons. First, the optimizer uses *measured* task execution times. The tasks in this 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 [2].

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. Thus with PEPSY, a parallelization onto a different number of processors and/or a different target system can be realized within minutes.

There are related prototyping systems based on data flow graphs known in the literature. Fresse et al. [5] have developed a prototyping environment for a multi-TMS320C40 system targeted for image processing applications. The GRAPE-II [7] environment uses synchronous and cyclo-static data flow graphs as application model. Data transfer is also realized using communication buffers [6].

Nevertheless, PEPSY's design flow using a formal optimizer with a communication model results in an accurate performance estimation for the multi-processor implementation. The design goals of the parallel application can be verified prior to its implementation, and, therefore, performance measurements on an intermediate multi-processor implementation can be avoided.

There are many sources for improvement and further work in PEPSY. First, the code generation and synthesis will be improved to better re-use the communication buffers and to automatically introduce code for inter-processor communication using DMA transfer. Second, PEPSY will be extensively evaluated on various DSP applications. Finally, a mid-term goal of this research is to extend PEPSY to a codesign framework for heterogeneous (multi-DSP and FPGA) systems.

6. REFERENCES

[1] C. Mathis, M. Schmid, and R. Schneider, "A Flexible Tool for Mapping and Scheduling Real-Time Applications onto Parallel Systems," in *Proc. Third Int. Conf. on Parallel Processing & Applied Mathematics*, Kazimierz Dolny, Poland, Sept. 1999, pp. 437–444.

[2] C. Mathis, B. Rinner, M. Schmid, R. Schneider, and R. Weiss, "A New Approach to Model Communication for Mapping and Scheduling DSP-Applications," in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing* , Istanbul, Turkey, June 2000, pp. 3354–3357.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *J. of VLSI Signal Processing Systems*, vol. 21, no. 2, 1999.

[4] C. Mathis, R. Weiss, and R. Buechel, "Design and Experimental Evaluation of a Multi-DSP based Simulation of the Human Peripheral Auditory System," in *Proc. Int. Conf. on Signal Processing Applications & Technologies (ICSPAT).* Sept. 1998, pp. 1098–1102.

[5] V. Fresse, M. Assouil, and O. Deforges, "Rapid Prototyping of Image Processing Applications onto a Multiprocessor Architecture," in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, Mai 2000.

[6] L. De Coster, R. Lauwereins, and J.A. Peperstraete, "Data Routing in Dataflow Graphs," in *Proc. of the 8th Int. Workshop on Rapid System Prototyping* , 1997, pp. 100–106.

[7] R. Lauwereins, M. Engels, M. Ade, and J.A. Peperstraete, "Grape-II: a System-Level Prototyping Environment for DSP Applications," *Computer*, vol. 28, no. 2, pp. 35–43, Feb. 1995.

Papers of the authors are available from the institute's home page at `http://www.iti.tu-graz.ac.at`.