Using Compilers for Heterogeneous System Design

Rainer Leupers, Peter Marwedel
University of Dortmund, Dept. of Computer Science 12, D-44221 Dortmund, Germany
email: *leupers|marwedel@ls12.informatik.uni-dortmund.de*

Abstract

Heterogeneous systems combine both data and control processing functions. A programmable DSP core forms the central component. The design of such systems establishes a new application of compilers in electronic CAD: In order to meet given real-time constraints and optimize chip area consumption, the DSP core needs to be customized for each application. In turn, this requires compiler support for evaluating different architectural alternatives. This paper discusses the importance of retargetable compilers in heterogeneous system design. [1]

1 Introduction

Regarding digital signal processing (DSP) system components two classes of functions can be distinguished:

- **Real-time data processing functions:** The inputs are data streams that have to be processed with a certain rate (sample frequency or throughput). A valid implementation has to meet given throughput constraints, whereas other system parameters (e.g. chip area) are subject to minimization.
- **Control processing functions:** The inputs are variables which have to be processed at irregular points of time. Normally there are only few timing constraints. Essentially those functions guarantee correct communication between the system and its environment.

A system comprising both kinds of functions is called a heterogeneous system. By means of increasing integration scales it has become possible to implement complete heterogeneous systems on a single chip. Single-chip implementations result in smaller physical volume and lower production costs. According to the two function classes mentioned above, these systems show a heterogeneous architectural style (fig.1): a programmable DSP core (i.e. a complete processor macrocell), additional specific data path elements for realising data processing functions, and a small amount of on-chip memory for storing programs and internal constants and variables. Obviously, programmable cores provide a much higher flexibility than pure hardware solutions.

[1] **This paper was published at the Int. Conf. on Parallel Architectures and Compilation Techniques (PACT) 1995, Limassol/Cyprus, ©1995 IFIP**

Decisions regarding the appropriate accelerators can be facilitated by compiler support: Different architectures and instruction sets can be evaluated by repeatedly re-compiling the DSP algorithm that describes the system behavior onto an instruction set, and trading cycle count against chip area. A compiler supporting this tradeoff must be capable of handling flexible target DSPs and available instruction-level parallelism.

The purpose of this paper is to describe a methodology for efficient evaluation of architectural alternatives for a DSP core within a heterogeneous system, based on a retargetable compiler.

The idea of incorporating compilers into the system design process is also proposed in [1]. That approach, however, is currently limited to evaluation of *operation chaining* in DSP algorithms. In CAPSYS [2], compilation techniques are used for generating a programmable hardware structure for a given application. Since CAPSYS assumes a certain generic VLIW architecture, it cannot be applied for code generation for arbitrary predefined DSP cores. The approach presented in this paper extends the applicability of compilers in system design in the sense that arbitrary processor structures can be handled.

In the remainder of this contribution we discuss the role of compilers in heterogeneous system design, and we describe techniques for retargetable DSP code generation. The practical applicability of our approach is discussed using a real-life example.

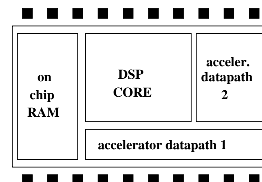

Figure 1: Heterogeneous architectural style

2 Hardware/software codesign for heterogeneous systems

The central issue of HW/SW codesign is to partition a system behavioral description into hardware and software com-

ponents in such a way that the system implementation fulfills given timing restrictions and minimizes the necessary amount of application-specific hardware modules, and thereby design and production costs. After the partitioning step hardware components are synthesized and software components are mapped onto the microprocessor by a compiler. The basic process is shown in fig.2. Since decisions during the partitioning step are often based on rough estimations, in general some iterations are required until a valid implementation is found.

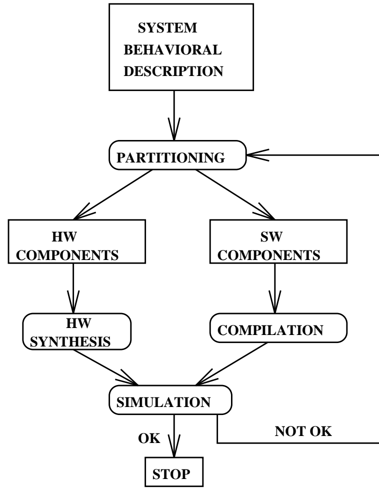

Figure 2: HW/SW codesign flow

In case of single-chip implementations as depicted in fig.1, the DSP core together with the accelerator datapaths can be regarded as one application-specific processor whose instruction set varies during the design iterations due to changing datapaths. Making use of varying instruction sets requires either manual compiler adaption or manual machine code adaption.

Since both manual compiler and machine code adaption are rather error-prone and time-consuming tasks, retargetable compilers should be applied for mapping software components onto varying processor structures and instruction sets. Retargetable compilers read both an algorithm (given in an HLL) and a target processor description and emit binary code implementing the required behavior on the given programmable hardware structure if possible. Existing retargetable compilers [3, 4, 6] map algorithms onto microcode for the given hardware structure. This requires detailed knowledge about the processor controller structure. In case of off-the-shelf DSP cores, however, this information in general is not publicly available, and code cannot be generated. Commercial compilers for standard DSPs still do not provide sufficient code quality, due to the lack of sophisticated code generation techniques. Furthermore, they only work for fixed architectures. In case of non-standard DSPs, it is often the case that no compiler is available at all.

In order to fill this gap, we present a technique for assembly code generation independent of the target processor. This technique is based on the retargetable microcode compiler MSSC.

3 Retargetable microcode generation

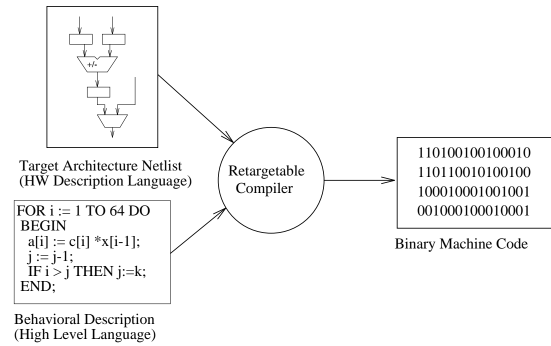

Figure 3: Functionality of MSSC

The microcode compiler MSSC has already been described in detail in [6, 7], therefore we only give a brief overview here for better understanding the process of retargetable assembly code generation. The functionality of MSSC is depicted in fig.3. It processes an algorithm given as a PASCAL or RT-level program and a model of a target processor in the MIMOLA hardware description language. The processor model closely reflects the actual hardware structure. One register has to be labeled as program counter and one storage module as instruction memory. The program is translated into binary microcode that executes the program on the hardware structure. Note that the program is extracted from the pure target machine structural description. As a consequence of that, MSSC needs not to be recompiled for different target machines and thereby guarantees short turnaround times when being retargeted to an alternative processor architecture.

4 Retargetable assembly code generation

Due to the fact that many DSPs are not microcoded, but show a two-level interpretation scheme of instructions, microcode compilers cannot be directly applied.

The solution to this problem is a technique that permits assembly level code generation by a two-phase use of MSSC. Firstly, MSSC is used to obtain the required controller description which implicitly includes the set of valid instructions. Using this controller description, MSSC maps DSP algorithms to assembly-level instructions. Fig.4 gives an overview of the whole procedure.

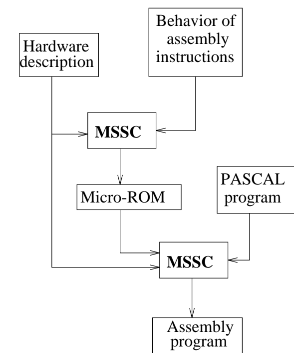

Figure 4: Overview of code generation technique

The first step is to model the target processor's RT-

structure in MIMOLA. The model consists of a netlist of storage and combinational modules. The behavior of available machine instructions is modelled on the RT-level. RT-level structure and the instruction behavior form the input for MSSC in phase 1. The result of the first MSSC run is a binary microprogram in which each microinstruction corresponds to exactly one assembly instruction. This microprogram is never executed, only its binary values are important for the following steps. The microprogram is transformed into a MIMOLA decoder module, that serves as an instruction decoder in phase 2. This decoder is integrated into the hardware structure description used in phase 1. The software input now can be any PASCAL program. MSSC is applied to the extended hardware structure and the program and translates it into a machine program executable on the target processor. The decoder forces MSSC to generate assembly level code instead of microinstructions. Phase 1 has to be performed only once for each target structure, after that any program can be compiled into assembly instructions. This technique preserves retargetability.

We will now explain the several steps of both phases in more detail, using the DSP TMS320C25 as an example target processor.

1.1 RT structure modelling: The target processor's RT-level structure is modelled in MIMOLA.

When developing the model, only a rough structural description has to be available to the designer. In case of the TMS, this information can be found in the data book. Regarding the controller, no information at all is required in phase 1. Instead we use a very simple controller consisting of a microprogram counter (MPC), a microinstruction storage (MIS) and an incrementer.

This simple controller shows a VLIW architecture, the MIS has a width of 150 bits. Such a VLIW controller would allow many more valid instructions than actually available in the TMS instruction set. In order to restrict MSSC to set set of valid assembly instructions, we first model the behavior of those instructions.

1.2 Assembly instruction modelling: The behavior of available assembly instructions is modelled in MIMOLA on the RT level. We consider the TMS "LACK" instruction (load accumulator immediate short): Its behavior is modelled by:

```
LACK:
 PARBEGIN
  ACC := ZeroExtend24(ROM[PC].(7:0));
  PC := "INCR" PC;
 PAREND;
```

For each assembly instruction a label of the same name is declared. The `PARBEGIN-PAREND` construct denotes parallel execution of the included statements. In case of `LACK` the 32-bit accumulator `ACC` is assigned an 8 bit immediate value stored in the program ROM, addressed by the program counter `PC`, extended by 24 zero bits. The program counter is incremented in parallel. In this manner all TMS instruction behaviors are described sequentially, resulting in an RT-level "program" to which MSSC is applied in phase 1.

1.3 Decoder generation: MSSC is applied to the structural and behavioral models developed in steps 1.1 and 1.2. It translates the assembly instruction behaviors into a set of binary microinstructions. The resulting microcode is stored into the MIS module containing one line for each modelled assembly instruction then (fig.5). The initialized MIS can be regarded as an instruction decoder. The decoder input is an address, and the output is the corresponding microinstruction. Integrating this decoder constructed by MSSC into the TMS processor model and applying MSSC again to the extended model is the basic idea of phase 2 in our approach, which is described in the following.

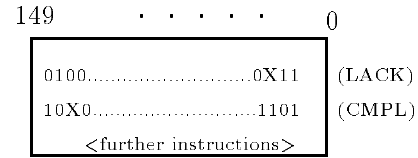

Figure 5: Contents of MIS

2.1 Decoder integration: The decoder generated in phase 1 becomes part of the controller structure. Microprogram counter and incrementer of phase 1 are dropped. The new controller structure is shown in fig.6.

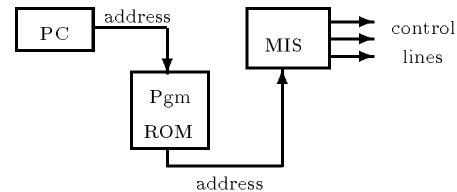

Figure 6: Controller in Phase 2

The program counter `PC` addresses one line in the TMS program ROM. The program ROM in turn steers the instruction decoder, which directly controls all modules. As mentioned in section 3, MSSC is capable of passing control codes to modules via decoders. Due to the controller structure, the only way to control modules is now via the instruction decoder, which in turn only contains microinstructions implementing valid assembly instructions. Thus, having the program ROM marked as location for instructions, MSSC is forced to generate control codes for the instruction decoder during code generation. These control codes can be interpreted as encoded assembly instructions, since each possible decoder output corresponds to exactly one assembly instruction. Therefore, MSSC can now be used for translating HLL programs into TMS machine code.

2.2 HLL program translation: MSSC is applied again, this time to the extended hardware structure including the instruction decoder, and an arbitrary PASCAL program. In order to explain the basic idea, we consider code generation for the single statement

```
x := 42;
```

Since this statement cannot be allocated in the TMS hardware within a single cycle, MSSC splits it into the sequence

```
(1) ACC := 42;
(2) DataRAM[1] := ACC;
```

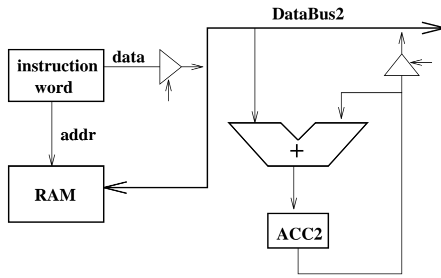

Figure 7: New adder-accumulator unit

using the TMS accumulator as a temporary cell and assuming variable **x** has been mapped to data RAM cell 1. Each statement can now be allocated within a single cycle, since the TMS provides suitable instructions: LACK for loading the accumulator with an immediate constant and SACL for storing the accumulator into a data RAM cell. MSSC finds corresponding microcode within the instruction decoder and generates intermediate code for addressing the decoder. The intermediate code can easily be translated into real TMS machine code by table lookup. In our example one obtains:

Mnemonic	Machine Code
LACK 42	1100101000101010
SACL 1	0110000000000001

5 Exploring architectural alternatives

The main purpose of our retargetable assembly code generation technique is to support the designer of a heterogeneous system in studying performance effects of different accelerator datapaths without compiler redesign.

As an example for a datapath change we consider an additional adder-accumulator unit. Although this datapath modification results in a changed instruction set, the HLL system behavioral description can remain unchanged and can be mapped onto the new structure by our retargetable compiler. In order to recompile programs onto the new structure the designer simply repeats phase 1, i.e. adding the new datapaths elements to the structural description, specifying the behavior of the resulting new assembly instructions, and using MSSC for generating the corresponding new instruction decoder.

We consider the effects of placing the second adder-accumulator unit into the datapath in order to enable performance enhancements by instruction-level parallelism. Fig.7 shows the architectural extensions.

The new unit works in parallel to the original TMS ALU-accumulator unit, thereby allowing parallel additions and data RAM manipulations. By supplying separate control signals to the new modules and assuming the adder to have a transparent mode, five new instructions arise from this measurement:

LDA <memadr>: (*load ACC2*)
ACC2 := DataRAM[<memadr>]
LDI <cnst>: (*load ACC2 immediate*)
ACC2 := <cnst>
ADA <memadr>: (*add to ACC2*)
ACC2 := ACC2 + DataRAM[<memadr>]
ADI <cnst>: (*add to ACC2 immediate*)
ACC2 := ACC2 + <cnst>
STA <memadr>: (*store ACC2*)
DataRAM[<memadr>] := ACC2

As a case study, we mapped two well-known DSP benchmarks onto the original and the extended TMS structure: the *differential equation solver* and the *elliptical wave filter*. Due to the limited space we only mention the results here:

prog	PASCAL statements	original TMS320C25	extended TMS320C25	CPU sec
ellip	38	184	105	54
diffeq	35	100	73	64

Columns 3 and 4 show the number of machine instructions that have been generated for the original TMS instruction set and the extended instruction set allowing parallel additions, as well as the compilation time on a SPARC-10.

The above data indicate, that using the new parallel adder-accumulator unit accelerates execution of *ellip* by 43 % and execution of *diffeq* by 27 % at the expense of larger chip area. Using our exploration technique, a system designer can quickly obtain this information by adapting the hardware model and recompiling the programs. Compilation speed is slow compared to a target-specific compiler, but this is more than compensated by retargetability in this context.

6 Conclusions

A hardware/software codesign strategy for heterogeneous information processing systems built around a DSP core was presented, which establishes a new role of compilers in electronic CAD tools for system design: The compiler does not just generate the required code, but is involved in the iterative system design process. Furthermore, contemporary DSP architectures require more sophisticated code generation techniques than currently available DSP compilers offer. Therefore, we propose the usage of a retargetable compiler which maps algorithms to flexible DSPs and which exploits available instruction-level parallelism. Compiler retargetability ensures relatively short turnaround times. [2]

References

[1] F. Onion, A. Nicolau, N. Dutt: Incorporating Compiler Feedback into the Design of ASIPs, European Design & Test Conference (ED & TC), Paris, March 1995

[2] G. Menez, M. Auguin, F Boeri, C. Carriere: Contribution of Compilation Techniques to the Synthesis of Dedicated VLIW Architectures, PACT-93, pp. 217-228

[3] S.R. Vegdahl: Local Code Generation and Compaction in Optimising Microcode Compilers, PhD Thesis and Report CMUCS-82-153, Carnegie-Mellon-University, Pittsburgh, 1982

[4] T. Baba, H. Hagiwara: The MPG System: A Machine-Independent Efficient Microprogram Generator, IEEE Trans. Comp., Vol C-30, 6(1981), pp. 373-395

[5] P. Marwedel, W. Schenk: Cooperation of Synthesis, Retargetable Code Generation and Test Generation in the MIMOLA Software System, European Design & Test Conference (ED & TC), 1993, pp. 63-69

[2] This work has been partially supported by the European Union, ESPRIT BRA project 9138 (CHIPS)

[6] L. Nowak: Graph Based Retargetable Microcode Compilation in the MIMOLA Design System, 20th Annual Microprogramming Workshop (MICRO-20), 1987, pp. 126-132

[7] R. Leupers, W. Schenk, P. Marwedel: Microcode Generation for flexible parallel target architectures, PACT-94, pp. 247-256