Compilers for Embedded Processors

Peter Marwedel
Universität Dortmund, Informatik 12
44221 Dortmund, Germany
e-mail-alias: marwedel@acm.org

Abstract— This tutorial responds to the increasing use of embedded processors for implementing systems-on-a-chip. We will provide an introduction to embedded processors and we will show that current compilers do not provide the required efficiency. We will give an overview over new compiler optimization techniques, which aim at making assembly language programming for embedded software obsolete. Finally, we will present techniques for retargeting compilers to new architectures. One of the approaches closes the gap between electronic CAD and compiler generation.

I. Embedded Processors, Core Processors and Embedded Systems

There has recently been a huge amount of interest in embedded processors in general and in embedded core processors in particular. What is the main reason behaind this huge interest?

The main reason is *flexibility*. It is possible to change the overall behavior of a processor-based design by changing the program that is executed on the processor. This way, it is possible to accommodate late changes of the specifications. These days, sepcifications change even after the designers have already started to generate a design. If this happens, ASIC designers may have to start all over again. Designers of processor-based systems are in a better position: they modify the program, re-compile and re-load it.

Flexibility also simplifies upgrading a design. Upgrading does in fact include a number of things: it includes, for example, both the generation of a more enhanced product and downloading new *firmware code* into some product already delivered to the customer.

In the special case of processors cores and other off-the-shelf processors, there is one additional advantage and that is *reuse*. Future chips, which are expected to contain more than 10^8 transistors can only be designed in acceptable time frames if complex components are reused.

Reuse of complex components has a number of advantages:

- It cuts down design time to the required level.
- Reuse of cores also improves the efficiency of the design, since cores are usually highly optimized.
- Testing is simplified because test engineers know the components they have to test from previous designs.

Normally, it is very difficult to provide flexibility and reuse at the same time. These two features are in fact in many cases mutualy exclusive. Processors cores, however, exhibit both features at the same time. It is this combination of features, which makes them so popular[1].

Where will those processors and processor cores be used? The various types of processors are intended to be used in so-called *systems-on-a-chip*. In such systems, there is a variety of different components: processors, RAM and ROM memory, various converters and possibly some full custom glue logic. A very important consequence of chip-level integration is that the size of RAM and ROM is extremely important. This, in turn, means that the code generated from application programs has to be very compact and has to use RAM locations for variables very efficiently. This is different from board-level integration, where the size of ROMs and RAMs does not matter that much.

What are actually the main applications areas for systems-on-a-chip? According to market analysts [22], the fastest growing market in general and for systems-on-a-chip in particular is the *embedded systems market*. In this context, *embedded systems* can be defined as *systems reading, processing and controlling physical quantities*. For embedded systems, market analysts are predicting so-called double-digit growth rates, or growth rates beyond ten percent. This indicates that the embedded systems market should receive an adequate amount of attention. Information processing is not restricted to what people come into contact with at the universities and at their desk. Embedded systems are frequently not realized as being present, because they do not come with a screen and a keyboard.

Most of the functionality of embedded systems can be implemented with software and embedded processors. Due to the flexibility of software solutions and the trend

[1] FPGAs are the only others component providing the same combination of features. But FPGAs are by far not as efficient as processors.

towards faster and faster processors, special hardware solutions will become less and less important. The trend towards software solutions does already exist for some time and has already led to the wide-spread use of processors in embedded systems. Two examples show that this use has already exceeded the level that is expected by most people:

1. According to Camposano [4], the New York Times has estimated (in 1995) that the average American comes into contact with about 60 microprocessors every day. Even today (1997) most people believe this number to be lower.
2. Due to personal information, some top-level cars include at least 100 microprocessors.

These numbers indicate that quite some fuctionality of embedded systems has already been implemented by software and processors.

The different processors that are used for embedded systems can be classified by using three main characteristics of processors.

The first characteristic (see fig. 1) is the form in which processors are available. Processors are called core processors, if they are available as an entry in a CAD library. There are, of course, other forms in which processors are available (for example, in packaged form).

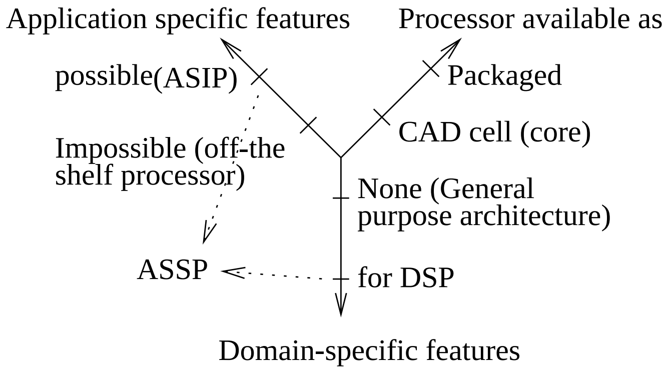


Fig. 1. Cube of processor types

The second characteristic is the availability of domain-specific features. Of course, some processors do not have any of these domain-dependent features. They are called *general purpose processors*. Some processors come with domain-specific features. The goal of adding such features is to find a good match between application domain and processor architecture, leading to a high overall efficiency. For example, processors for digital signal processing include saturating arithmetic, multiply/accumulate instructions and special addressing modes for implementing digital filters and Fourier transforms. Others processors exhibit special features for microcontroller applications, such as sophisticated bit manipulation instructions.

Even more efficient designs are feasible with application-specific features. If such features are present, the corresponding processor is called an ASIP (*application-specific instruction set processor*). There is some research on how optimized ASIPs can be designed for given applications [1, 9]. It has also be proven, that ASIPs can in fact be more efficient than domain-specific or general purpose processors [8]. *Off-the-shelf processors* are processors which do not include any application-specific features. *Application-specific signal processors* (ASSPs) are application-specific processors including special features for digital signal processing. In a 3D representation of processors by a cube, ASSPs correspond to one of the edges.

In addition to the three coordinates, there are of course other criteria for classifying processors.

II. Efficiency of Compilers for Embedded Processors

A. Existing Compilers

Now, the next question is: is it really a problem if future systems will be implemented in software? There are many mature software design environments which now could also be used for the design of embedded systems. The customer base for these environments is larger than that for ECAD tools and hence more money is available for their development. This could be used to make them more reliable and robust than currently used hardware design systems, for which the number of copies sold is usually quite small. Does this mean than designers of embedded systems will find all the tools they need on the market?

This would be nice, but this will most likely not be the case. Development tools for processor-based systems currently have some severe limitations. This applies to various kinds of development tools, but in this contribution we will focus on compilers.

Problems with using current compiler technology for embedded processors have been mentioned by quite a number of industrial designers. Detailed numerical data has been published by a group of reseachers working at the University of Aachen. Researchers at Aachen have compared the size and the speed of assembly language library routines with the size and the speed of compiled code. According the results of this DSPStone benchmark project [30], overhead of compiled code (in terms of code size and clock cycles) typically ranges between 2 and 8.

As an example, we consider the results that have been found for the ADPCM algorithm, which is a very common algorithm for speech encoding. For this algorithm and three different processors, the data memory overhead ranges between about 170 % and almost 400 %. Thus, if compilers are used, the data memory has to be up to almost 5 times larger than for assembly code.

For the same benchmark, the situation is even worse if the speed overhead is taken into account. This overhead ranges between about 500 and almost 700 %. This means that up to 7/8th of all processor cycles are waisted if compilers are used. This translates into a huge loss of performance and electrical power and is clearly not acceptable for embedded systems. *Optimizations for low-power design should not be constrained to the hardware level. From an overall point of view, a highly optimizing compiler is one of the most important contributions to low power design.*

Due to the current lack of highly optimizing compilers, a major amount of applications is implemented in assembly languages. The exact percentage varies from company to company and from application to application. Paulin has computed this percentage for a closed set of applications [20]. He found that a major percentage of DSP applications and of controller applications is written in assembly languages.

Implementing complex systems using assembly languages has all the well-known disadvantages, for example a long time to the market, a low product quality and the inability of retargeting the application to new processors.

Before we try to change this situation, we should analyze the reasons behind this poor performance of current compilers. These reasons can be understood, if we look at the how currently available architectures for embedded processors were designed.

For most of these processors, the design goal was to have very efficient processor hardware. This hardware was designed to fit typical applications in the application domain as good as possible. Ease of compilation was never really a design goal.

As an example let us consider the TMS320C25, a very popular digital signal processor. The microarchitecture of this processor contains a variety of registers. There are registers at the input and the output of the multiplier and a special accumulator at the output of the adder. Furthermore, there are special address registers, a special address register pointer register and there is a small data RAM. All these registers have a different functionality. Due to this, the register set is said to be *heterogenous*. This constrasts with the homogenous register files found in most standard RISC architectures.

Having a heterogenous register set does indeed make sense for DSP architectures. The registers at the multiplier and the adder can very efficiently be used to implement digital filters. The accumulator is appropriate for storing partial sums during the filtering operation. Replacing these specialized registers by homogenous registers files would possibly extend the critical path for multiplications, would not speed up digital filtering, would increase the chip area and it would restrict the maximum number of accesses to registers in any clock cycle. The only 'advantage' would be to make the design of a compiler easier.

Since embedded systems have to be efficient, processors with special features for embedded systems, in particular for DSP and microcontroller applications, will probably be used for many years. This should generate interest in compiler techniques supporting those features.

B. New Optimization Techniques

B1 Overview

As will be shown in the next sections, it is actually possible to significantly improve the performance of compilers for embedded processors. Some researchers have recently designed new optimization algorithms exploiting special features of embedded processors.

Wess[28], Araujo and co-workers [2] have proposed register assignment techniques for heterogenous register sets. These techniques extend register assignment techniques for homogenous register sets that are used for standard RISC- and CISC-processors.

Different operation *modes* are another typical feature of embedded processors. Many DSP processors provide a choice between saturating and wrap-around arithmetic modes and also use modes for selecting significant bits of fixed point numbers after multiplication. The instruction sets of these processors include instructions for switching between these modes (in microprogramming, the same concept was called *residual control*). Compilers should try to generate a minimized number of mode switching instructions. Liao has published an algorithm performing this type of optimization [14].

The recently announced 'C60 processor of Texas Instruments offers a significant amount of instruction-level parallelism. Up to 8 instructions of 32 bits each can be executed in parallel. The parallelism of this and similar machines can only be exploited if techniques for globally rearranging code exist. These techniques are called global scheduling techniques. *Mutation scheduling* (by Nicolau) [19] is possibly the most sophisticated global scheduling technique. It incorporates global code movement, consideration of available hardware resources and the application of algebraic rules.

Nicolau and his group have also worked on the integration of loop unrolling and register allocation [10], a technique required for finding a compromise between code density and execution speed for loops.

Another feature, which is commonly found in many processors is the presence of an address generation unit. Such units allow address computations to be performed concurrently with other arithmetic operations. Bartley [3], Liao [15], Leupers [11], and Sudarsanam [25] have recently proposed algorithms exploiting this extra hardware.

Two optimizations will be explained in a little more detail in the next section.

B2 Exploitation of instruction level parallelism

A number of architectures (such as the popular TMS 320C25) allow several register transfers to be encoded into a single instruction [27]. This can actually be exploited in practical examples. For a simple set of two C-statements, we can generate a sequence of register transfers that are legal for the 'C25. Already in this small example, a set of three transfers can be identified, which can be encoded in the same instruction and executed in one cycle. The result is a reduction from 9 to 7 cycles.

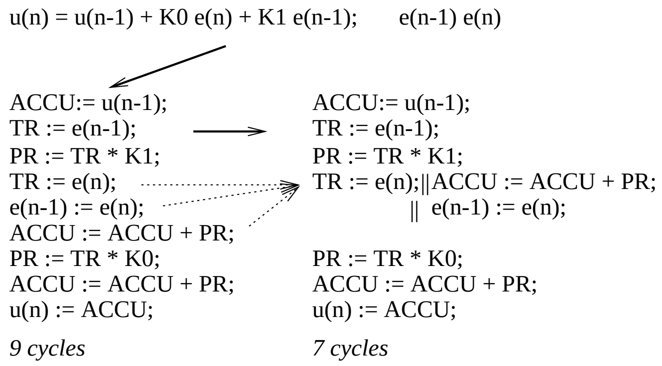


Fig. 2. Compaction

Techniques for generating such compacted code have already been studied in the context of microprogramming. In the meantime, however, the speed of available computers, the quality of integer programming packages and the knowledge about modeling techniques have been improved to an extend making optimal algorithms applicable for not too small basic blocks. The technique developed by Leupers [13] can be applied to basic blocks of up to about 50 transfers. This is sufficient for most practical examples. Partitioning of larger basic blocks into blocks of this size not required except for few exceptional cases.

The results for compaction are excellent. For various entries in the DSPStone benchmark suite, code size reductions between about 5 and more than 30% have been observed. There is only a single case (lattice2), in which the reduction does not exceed 10 %. The execution times for the compaction algorithm are very moderate: they range from 2 to 35 seconds.

B3 Exploitation of multiple memory banks

In order to study the another optimization technique, we consider the microarchitecture of the Motorola 56K processor containing two memory banks. Each bank comes with its own address generation unit (AGU). A move between the X-memory bank and the the X-registers and another move between the Y-memory and the Y-registers can be performed in parallel. In addition, there are cases in which yet another operation can be performed in parallel.

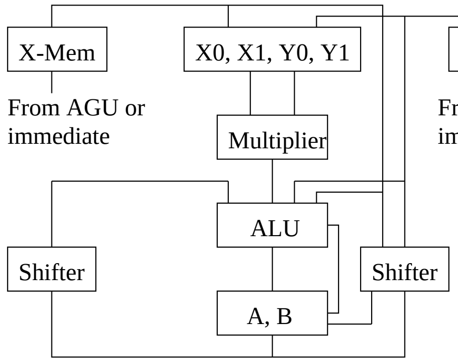


Fig. 3. Mototorola 56k processor architecture

Techniques for allocating variables to memory such that the total number of parallel operations is maximized have been proposed by Sudarsanam and Malik [26]. They are based on conflict graphs. In such graphs, nodes denote variables and symbolic registers. Edges between any pair of nodes model potential parallel accesses to the corresponding values and hence indicate that these values should be held in different memory banks or register sets.

An algorithm based on simulated annealing tries to respect as many of these edges as possible. The results are quite impressive: code size reductions for the DSPStone benchmark range between 20 and almost 60 %.

These optimization techniques show that substantial improvements of available compilers are feasible. There is some hope that future compilers for embedded processors will be much better than those that are currently available and that eventually this will lead to a replacement of assembly language programming for embedded systems.

This concludes the description of new optimization techniques. Next, we turn our attention towards retargetability.

III. Retargetable Compilers

A. What is retargetability?

Let us first try to define the term. A compiler is said to be retargetable, if it can be applied to a range of target processors.

Actually, we can distinguish between different levels of effort for switching to a new target:

- A high effort is required for so-called *portable* compilers. Porting a compiler to a new target possibly includes rewriting some parts of the compiler.

- More precisely than above, we say that a compiler is retargetable if it includes almost no processor specific code. The characteristics of the target processor msut be captured in a separate *target description*.

B. Why retargetability?

Retargetability is difficult to implement. Why then do people investigate techniques for generating retargetable compilers?

We would like to mention four different reasons:

1. Retargetable compilers are required to support using ASIPs. Many different instruction sets can be defined by choosing values for the generic parameters of ASIPs. For each set of values, there will possibly be only a small number of designs. For this small number, it will not be economically feasible to design a specialized compiler. So, there should be a retargetable compiler which can generate code for all legal value sets of generic parameters.

   As far as ASIPs are concerned, no retargetability for very different processors is required. Retargetability within the range of parameters is sufficient.

   Why do we need ASIPs? The main reason for ASIPs is that embedded systems require maximum efficiency and maximum efficiency can only be obtained through customization. It has been observed that customization of processors results in more computations per Watt than can be achieved for standard processors. Fully application-specific hardware (ASICs) would achieve a even higher number of computations per Watt, but ASICs lack flexibility. Hence, ASIPs are a good compromise between power consumption and flexibility. Due to their low power consumption, it has been observed that first generation products are sometimes implemented with standard processors and these are later replaced by ASIPs in second generation products.

2. For embedded systems, there is a large range of applications. This range includes, for example, health care applications in which the systems are inserted into the human body, applications in telecommunications and applications in transportation systems.

   Due to the large variety of applications, a large variety of processors is also required. Due to the mutual dependencies between instruction set and microarchitecture, we believe that there will be different instruction sets, each of which will provide a good match for some applications.

   This means that we also need compilers for different instruction sets which are used for small or medium number of applications. This will be economically feasible only if compilers can be easily retargeted to different instruction sets.

3. With retargetable compilers, it is possible to analyse tradeoffs between adding more processor features and the resulting size, speed and possibly also the power consumption of the processor. At a high level, these features may be instructions that can be added or deleted. At a lower level, one could also experiment with hardware features, provided that the compiler can be generated from a hardware description.

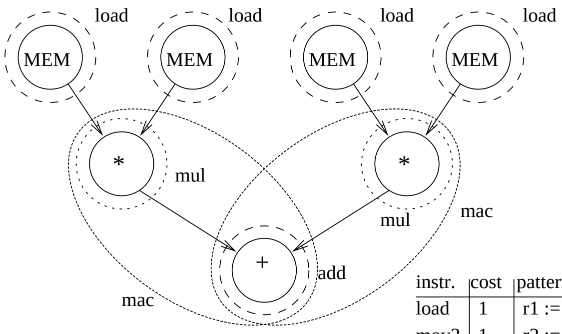
4. Working on retargetable compilers also provides some insight into mutual dependencies between compilers and architectures. This insight can also help in designing a target-specific compiler. Hence, there may be a benefit from research on retargetable compilers even if only target-specific compilers will be used.

C. Code selection

C.1 Objective

Let us now discuss one of the key operations of any compiler and let us see how it can be implemented in a retargetable compiler. Possibly the most important operation in any compiler is *code selection*. In code selection, machine instructions are selected.

Let us consider an example in which the program to be compiled requires the computation of an expression including two multiplies and one add (see fig. 4). This expression includes references to four variables. In fig. 4, they are denoted by nodes labeled 'MEM'.



instr.	cost	pattern
load	1	r1 := MEM
mov2	1	r2 := r1
mov3	1	r3 := r1
add	2	r1:= r1 + r2
mul	2	r1 := r1 * r2
mac	3	r1 :=(r1*r2)+r3

Fig. 4. Expression (a * b) + (c * d) and instruction patterns

Each of the nodes in fig. 4 has to be implemented by some instruction of the available instruction set. According to fig. 4, we have one `add` instruction which adds the contents of registers 1 and register 2 and stores the result into register 1. There is also a `multiply` instruction using the same input and output registers. In addition, there is a `mac` instruction and a `load` instruction. Finally, there are `move` instructions for moving data from register 1 to registers 2 and 3. In fig. 4, the dotted ovals and circles represent instructions. These do not yet include `move` instructions that are required for moving data from the output register of one instruction to the input register of the next instruction.

Implementing all operations of the expression is equivalent to finding a cover of that expression by instructions. Even our small example allows different covers to be generated and the question is: how can we generate the cover for which the cost (representing the total number of instructions or cycles) is minimal?

A number of techniques for generating optimum covers for data-flow trees have been published in the late eighties[2].

C2 Code Selection by Tree Parsing

As an example, we will consider cover generation using `iburg`[5]. `iburg` is publicly available from Princeton University. `iburg` models cover generation as a language parsing problem. For this purpose, the instruction set has to be described as a grammar. In this grammar, permanent memory and operators are terminals and transient registers are non-terminals. The start symbol is actually not very important and can be defined in a number of ways. Instructions correspond to rewrite rules of the grammar.

`iburg` computes an optimum cover by first annotating data-flow trees with triples representing target registers, instructions used for storing results into those target registers and the cost for storing results into target registers.

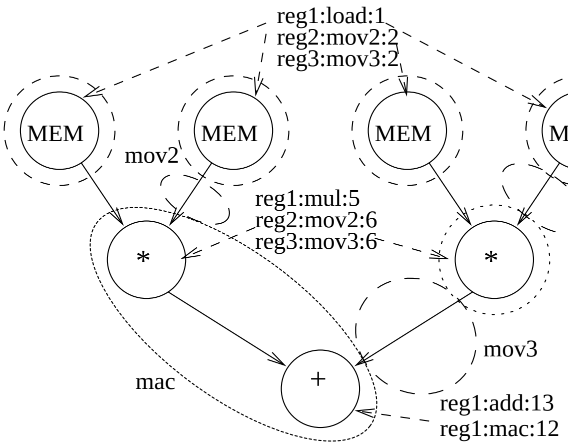


Fig. 5. Annotation and final cover generated by iburg

For example, MEM nodes are annotated with a triple describing the `load` operation into target register 1. Furthermore, there is another triple using register 2 as the target register and using `mov2` as the last instruction. Its cost is 2, since both the `load` and the `mov2` instruction are required. A similar triple exists for target register 3.

Annotating the tree is also done for the multiply operation. The result of this operation can be stored in register 1, 2 or 3, resulting in costs of 5, 6 and 6 respectively.

[2]The problem of covering data-flow graphs resulting from expressions containing common subexpressions is more complex. In this text, we will restrict ourselves to data-flow trees.

Finally, there are two triples for the root node. The triple using the `mac` instruction has lowest cost and is seleted by `iburg` for the tree cover.

Interestingly enough, this cover includes ovals representing `mov2` and `mov3` instructions. These are automatically selected by `iburg`. This is an improvement over earlier approaches which used separate *data routing* phases to cope with heterogenous register sets.

D. Retargetable Code Generator RECORD

`iburg` alone does not make a full retargetable compiler, but it is a good building block. `iburg` is used in the retargetable compiler RECORD designed by Leupers [12].

RECORD reads the program to be compiled and the instruction set and then uses the `iburg` pattern matcher generator to generate instruction covers for this program.

In contrast to `iburg`, RECORD exploits various algebraic rules in order to find the best match between the instruction set and the program. For this purpose, algebraic rules are applied to the original expression trees. Transformed trees are also used as input to the pattern matcher and tree leading to the cover with minimum cost is selected.

RECORD also includes comprehensive *post-processing*. It includes code compaction and address generation unit support as described in section B of this text. Also, there are special optimizations for digital signal processing, such as optimization for delayed signals that are needed for digital filters.

The code generated by RECORD is surprisingly good. It generates code that is more compact than the code produced by the target-specific TI compiler for the 'C25, even though RECORD does not yet include any of the standard optimizations found in compiler text books and even though it is a retargetable compiler. Using a set of DSP-Stone examples, RECORD outperforms the TI compiler in six cases, there are two cases in which both compilers produce the same amount of code and two cases in which the TI compiler generates more compact code.

E. Target Models

Retargetable compilers use a variety of target models.

RECORD is based on an HDL model of the target processor. The HDL model can be a model either of the instruction set or of the register transfer architecture or any mixture thereof. From this HDL model, RECORD computes the information that is required for the *iburg* pattern matcher. For this purpose, *instruction extractions* [12] is used.

Due to this approach, no separate target modeling language is needed and the gap between ECAD and compiler worlds has been bridged.

There are, however, other approaches for modeling targets. One of these is used in the CHESS compiler designed at IMEC in Leuven, Belgium [21]. CHESS is based on a target model written in a language called nML. These main idea of nML is to capture the information in the processor reference manual as precisely as possible in textual form. Hence, various alternatives for opcode formats are reflected in so-called OR-rules in nML. Applicability of nML for simulation and code generation is decribed in a paper at DAC 97 [7].

Another approach is used by Paulin and his group at SGS Thomson [16]. They represent the information for the compiler by a set of rules. Symbolic instructions are emitted, whenever a matcher finds a match between a rule and the internal representation of the program to be compiled. Hence, rule sets have to be specified everytime a new processor has to be supported by a compiler. Symbolic instructions are later replaced by real instructions of the target processor during post-optimization.

F. More Approaches

In addition to the work mentioned so far, a couple of other approaches exist:

- ISDL is a language that has been proposed for the description of target processors [6].
- The GNU C compiler GCC [23] is frequently mentioned as a retargetable compiler. This compiler does indeed generate code for various RISC processors with homogenous register sets. However, it has not been designed for heterogenous register sets and attempts to do so usually fail.
- Flexware [17] is an earlier approach of Paulin and co-workers. It has been replaced by the rule-based approach mentioned above.
- At the University of Guelph, Wilson has proposed models for integer programming-based compilation [29].
- At the University of Vienna, Wess is using Trellis diagrams as a different way of modeling targets [28].
- For the special case of transport-triggered architectures, Corporaal has designed compiler algorithms (see [18]).
- Mavaddat is working on theoretical models called Lindenmayer- or L-systems (see [18]).
- At Philips, there is work for both audio and video applications [24].
- For VLIW machines, available parallelism is visible at the instruction set level. Hence, scalability is a problem, because more parallel machines come with a different instruction set. Binary code compatability even within a family of processor is not easy to obtain. Rau and co-workers at hp-labs are working on smart loaders. These loaders retarget symbolic code to the actual architecture.
- Finally, the Stanford University Intermediate Form (SUIF) has recently shifted from providing a framework for parallelizing compilers towards a generally usable compiler framework.

IV. Conclusion

The following conclusions cane be drawn from this tutorial:

There is currently a clear trend towards cores in general and towards embedded core processors in particular.

Current compilers for embedded processors do not produce high-quality code. However, recent achievements indicate, that substantial improvements are feasible and future compilers for DSPs and microcontroller could potentially replace assembly language programming.

CAD companies and designers have to become aware of the issues in generating software for embedded processors. There are many such issues, including problems with currently available compilers, both also many more problems with embedded debugging, software/hardware partitioning and software/hardware interface synthesis. Assuming that products solving these problems can be designed by subcontractors will potentially lead to problems such as the poor code quality described in section B.

In the last section of this text, we have explained the motivation behind research on retargetable compilers and how they can be designed. It has been shown that the gap between ECAD and compiler worlds can be bridged and that retargetable compilers can compete favourably with available target-specific compilers.

The current tutorial was made possible by the progress made on code generation for embedded processors in recent years. I would like to thanks those people, since it is their work that made this tutorial possible.

References

[1] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. An ASIP instruction set optimization algorithm with functional module sharing constraint. *Int. Conf. on Computer-Aided Design (ICCAD)*, pages 526–532, 1993.

[2] G. Araujo and S. Malik. Optimal code generation for embedded memory no-homogenous register architectures. *8th Int. Symp. on System Synthesis (ISSS)*, pages 36–41, 1995.

[3] D.H. Bartley. Optimizing stack frame accesses with restricted addressing modes. *Software - Practice and Experience*, 22:101–110, 1992.

[4] R. Camposano and W. Wolf. Message from the editors-in-chief. *Design Automation for Embedded Systems*, 1996.

[5] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems, volume 1*, pages 213–226, 1992.

[6] Hadjiyiannis97. ISDL: An instruction set description language for retargetability. *Design Automation Conference*, 1997.

[7] Hartoog. Generation of software tools from processor descriptions for hardware/software codesign. *Design Automation Conference*, 1997.

[8] I.-J. Huang. Synthesis and exploration of instruction set design for application specific symbolic computing. *2nd Workshop on Code Generation for Embedded Processors (unpublished)*, 1996.

[9] I.-J. Huang and A. Despain. Generating instruction sets and microarchitectures from applications. *Int. Conf. on CAD (ICCAD)*, pages 391–396, 1994.

[10] D.J. Kolson and A. Nicolau. Optimal register assignment to loops for embedded code generation. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 1996.

[11] R. Leupers. Algorithms for address assigment in DSP code generation. *ICCAD*, 1996.

[12] R. Leupers. *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, 1997.

[13] R. Leupers and P. Marwedel. Time-constrained code compaction for DSPs. *Int. Symp. on System Synthesis (ISSS)*, 1995.

[14] S. Liao, S. Devadas, K. Keutzer, and S. Tijang. Code optimization techniques for embedded DSP microprocessors. *32nd Design Automation Conference*, pages 599–604, 1995.

[15] S. Liao, S. Devadas, K. Keutzer, S. Tijang, and A. Wang. Storage assignment to decrease code size. *Programming Language Design and Implementation (PLDI)*, 1995.

[16] C. Liem. *Retargetable Compilers for Embedded Core Processors.* Kluwer Academic Publishers, 1997.

[17] C. Liem and P. Paulin. FlexWare – A flexible firmware development environment. *Proc. European Design & Test Conference (EDAC-ETC-EUROASIC)*, pages 31–37, 1994.

[18] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors.* Kluwer Academic Publishers, 1995.

[19] S. Novack, A. Nicolau, and N. Dutt. A unified code generation approach using mutation scheduling. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer Academic Publishers*, 1995.

[20] P. Paulin, C. Liem, T. May, and S. Sutarwala. DSP design tool requirements for embedded systems: A telecommunications industrial perspective. *Journal of VLSI Signal Processing*, pages 23–47, 1995.

[21] J. V. Praet, D. Lanneer, G. Goossens, W. Geurts, and H. De Man. A graph based processor model for retargetable code generation. *European Design & Test Conference*, 1996.

[22] M. Ryan. Market focus – insight into markets that are making the news in EE Times. *http://techweb.-cmp.com/techweb/eet/embedded/embedded.html (Sept. 11)*, 1995.

[23] R. M. Stallman. Using and porting GNU CC. *Free Software Foundation*, 1993.

[24] M. Strik, J. van Meerbergen, A. Timmer, and J. Jess. Efficient code generation for in-house DSP cores. *European Design & Test Conference*, pages 244–249, 1995.

[25] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. *Design Automation Conference*, 1997.

[26] A. Sudarsanam and S. Malik. Memory bank and register allocation in software synthesis for ASIPs. *Intern. Conf. on Computer-Aided Design (ICCAD)*, pages 388–392, 1995.

[27] E. Timmer. Conflict modelling and instruction scheduling in code generation for in-house DSP cores. *32th Design Automation Conference*, 1995.

[28] B. Wess. Code generation based on Trellis diagrams. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer Academic Publishers*, 1995.

[29] T. Wilson, G. Grewal, S. Henshall, and D. Banerji. An ILP-based approach to code generation. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer*, 1995.

[30] V. Zivojnovic and et al. DSPstone: A DSP-oriented benchmarking methodology. *Proc. of the Intern. Conf. on Signal Processing and Technology*, 1994.