MODULO-ADDRESSING UTILIZATION IN AUTOMATIC SOFTWARE SYNTHESIS FOR DIGITAL SIGNAL PROCESSORS

Markus Willems, Holger Keding, Vojin Živojnović and Heinrich Meyr

Institute for Integrated Systems in Signal Processing
Aachen University of Technology
Templergraben 55, 52056-Aachen, Germany
{willems,keding,zivojnov,meyr}@ert.rwth-aachen.de

ABSTRACT

Digital Signal Processors (DSPs) have become key components for the implementation of digital signal processing systems. With DSPs moving into new application domains and the increasing complexity of modern DSP architectures, efficient programming support receives major interest. Therefore, an optimizing compiler becomes a must for future DSP-architectures. Todays DSP compilers result in significant overheads both in memory consumption and program execution time compared to hand-written assembly code. This is mainly due to an inefficient compiler support of the DSP specific architectural features, such as the modulo-addressing capability which is an enabeling feature for a large class of DSP algorithms. Within this paper we analyze why existing compilers fail short in supporting the modulo-addressing mode and present a compiler concept that allows the efficient utilization of this feature. We describe how an advanced compiler optimization strategy allows a near optimum support of the modulo-addressing mode, and point out why this concept is favorable to DSP-specific language extensions.

I INTRODUCTION

When it comes to the implementation of digital signal processing systems, the designer faces several, apparently contradictory, requirements:
systems of ever increasing complexity have to be realized within a reduced time resulting in minimum costs and short time-to-market. In order to match all these requirements, advanced tool support is crucial.

If the implementation calls for a programmable architecture, digital signal processors (DSP) appear to be favorable. DSP-features include:

- single cycle multiply-accumulate
- distributed data memory (modified Harvard architecture)
- small (and in general heterogeneous) register sets
- advanced addressing modes, such as bit-reversed and modulo

System design starts with a high level language (HLL) description of the system. On the other hand, DSP-implementations are realized using hand-written assembly code which allows to result in runtime and memory efficient implementations. This manual assembly coding process is time-consuming and code maintainance is a difficult and error prone task. Keeping the design constraints in mind, hand coded assembly will not be a future option.

A significant speedup in the design process could be achieved if optimizing HLL-compiler were available. Up to now, this approach is restricted by the suboptimum code generation capabilities of existing compilers. This has been indicated especially by the DSPstone compiler benchmarking project [1,2]. So the advantages in design time would be more than compensated by the higher price per shipped device since more memory and a faster chip might be necessary.

Because of the urgent need for a design speedup, it becomes a must to significantly improve the design flow *HLL* $\rightarrow$ *compiler* $\rightarrow$ *target code*. An analysis of existing compilers shows that these fail short in addressing the DSP-specific architectural features.

In order to identify possible improvements in the compilation process, it is necessary to take a closer look at the compiler principle (fig.1).

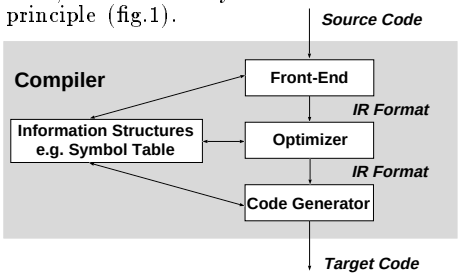

Figure 1. The compiler principle

Four reasons might be encountered for code inefficiencies:

- the limited expressive power of the specification language does not allow to utilize the architecture specific features - this would call for the introduction of DSP-specific language extensions [3,4].
- the source code formulation prohibits the efficient utilization of the DSP-specific features - this would call for programmer's guidelines.
- the compiler itself is sub-optimum and does not utilize the information that is inherent to the code
  - because the intermediate representation (IR) does not allow to represent the information - this would call for an extension of the IR
  - because the code generator does not make use of the information - this would call for more intelligent optimization strategies

Recently, advanced concepts to improve DSP-compilers with regard to memory bank and register allocation have been presented [5,6,7]. For these specific problems it has been identified that the inefficiencies are rooted in the code generator, so that there would be no need for a programmer directed allocation of data via language extensions to fully exploit these DSP-specific capabilities.

Within this paper we analyze the compiler optimization problem with a focus on the DSP-specific modulo-addressing mode which is of major importance for digital filter realizations. After a short introduction to the modulo-addressing principles in sec. 2, sec. 3 comes with a thorough analysis of the implications of different C-coding styles. The internal representation and the code generation principles are subject of sec. 4. In sec. 5 we present a code-generator concept that allows to efficiently transfer the modulo-information of the IR into optimized code. This is based on the introduction of

the *Circular Buffer Table* concept. Finally, benchmarking results for different C-code versions and different optimization strategies come with sec. 6.

II MODULO ADDRESSING: BASICS AND APPLICATIONS

Modulo addressing allows to access data from a predefined memory partition defined by a base address `base` and the buffer size `N`, in a specific order. A mathematical description of the actual address, where `count` is an arbitrary counter variable, is as follows

$$\begin{aligned} addr &= base + (count \bmod N) \\ &= base + (count - \lfloor count/N \rfloor * N) \end{aligned}$$

The `mod` operator guarantees that once `addr` crosses the upper or the lower bound, it is wrapped around. Figure 2 illustrates the access for different values of `count`.

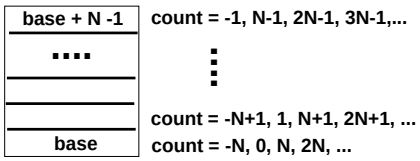

Figure 2. Memory access for different values of `count`

This addressing mode is very attractive for the realization of digital filtering operations working on streams of data. A typical mathematical formulation of such a filtering operation is the "gliding window" structure of a convolution where the result `y` at time `n` depends on N subsequent input samples `x`.

$$y(n) = \sum_{k=0}^{N-1} c_k x(n-k) \qquad (1)$$

Only the N last input samples are necessary. For calculating output sample `y(n+1)`, the oldest sample `x(n-N-1)` is not required anymore but a new sample `x(n+1)` has to be included. We achieve an efficient realization using modulo addressing with a memory partition of length `N`.

III C-CODE ANALYSIS

With respect to supporting the modulo-operation, one must analyze the C-code with regard to the explicit allocation of data to memory.

Fig.3 shows a straightforward C-formulation of the algorithm described by eqn.(1). This code calls for a specific

```
(1)    accu = 0;
(2)    for (i=0 ; i < (N-1) ; i++){
(3)        accu += c[i] * x[z--];
(4)        x[z+1] = x[z];
       }
(5)    accu += c[N-1] * x[z--];
(6)    x[z+1] = next_X;
```

Figure 3. C-code: FIR filter simulating a gliding window

data allocation in memory: after processing a data sample `x` (line(3)) it is explicitly shifted in memory (line(4)). Therefore the newest data sample is bound to the base address. The compiler cannot decide whether the algorithm has been programmed that way by intention, e.g. to guarantee the correct data allocation in case of an interrupt or a DMA, or if the programmer simply stayed close to the algorithmic formulation and did not care about the actual data allocation. As a consequence, the memory shift has to be implemented in the target code, resulting in a data allocation as presented in figure 5a.

A functional equivalent formulation of the filtering operation without explicit memory shift performs the data access

```
(1)    accu = 0;
(2)    for (i=0 ; i <= (N-1) ; i++)
(3)        accu += c[i] * x[(z-- % N];
(4)    x[z-- % N] = next_X;
```

Figure 4. C-code: FIR filter using circular addressing

based on a circular buffer realization. Fig. 4 shows the C-code. `x` defines the base address of the buffer, `N` its length. Therefore the circular buffer is completely specified. With this formulation, the newest sample resides anywhere in the circular buffer, the C formulation does not impose any restriction to the location within the specified memory section. The data allocation information inherent in the C code is that input data has to be allocated in a circular buffer way. As in the previous formulation, the compiler has no freedom in its decision how to arrange the memory allocation. Fig. 5b illustrates the resulting data allocation.

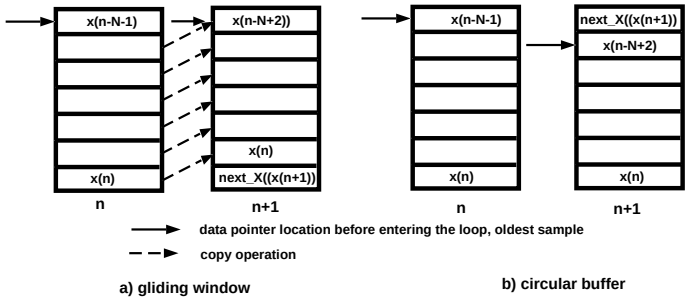

Figure 5. Data allocation for the (a) gliding window formulation (b) circular buffer formulation

Since the compiler is forced to allocate the data in a circular buffer way, the modulo-formulation of the filtering operation obviously is not suited if there is no circular buffer support by the hardware. This would mean to emulate a circular buffer. Hence, for a general purpose processor without modulo-addressing capability the direct formulation is favorable. For a DSP coming with the modulo-addressing capability the circular-buffer formulation should clearly be beneficial. An optimizing compiler should be able to extract the source code information about the specific way of data allocation.

As pointed out, different architectural classes require specific functional descriptions even in a HLL to enable an efficient utilization of the hardware capabilities. So why not using language extensions which appear to be specific programming styles, too? The key is portability, which covers

- functional portability: the description can be transferred to a different target and the compiler can generate functionally correct code
- performance portability: when transferring the description to a different the compiler can generate efficient code.

Up to now language extensions are processor specific [4], so no portability can be found at all. Even if a standardization of DSP-specific language extensions would be reached [3], functional portability would be limited to the architectural class of DSPs only. It is impossible to test the functionality on a host machine without a compiler support of the specific extensions.

In contrast to this, the modulo-specific C-coding style enables performance portability for all modulo-addressing architectures with no need for an extension of the standard. Most important, functional portability is guaranteed for all machines that come with an ANSI-C compiler.

As a consequence, DSP-specific language extensions should be limited to those architectural features that can not be utilized from the information that is inherent to the C-code.

IV THE IR AND TREE-REWRITING RULES

The intermediate representation (IR) serves as an interface between front-end and code generator. In majority, it is based on tree structures [9], where operators constitute the nodes, and operands constitute the leaves of the tree. Fig. 6 shows an IR tree for the assignment statement `r := a * b[z]`. The leaves in the tree are type attributes with subscripts; the subscript indicates the value of the attribute. The `ind` operator makes its argument a memory address.

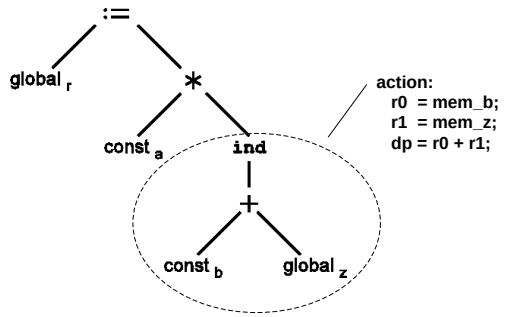

Figure 6. Intermediate Code Tree for `r := a * b[z]`

One reliable way of target-code generation is to represent the instructions of the target machine by a set of tree-rewriting rules, called a *tree-translation-scheme* [10]. A tree-rewriting rule is a statement like

$$replacement \longleftarrow template\ \{\ cost\ \} = \{\ action\ \}$$

where a *replacement* is a single node that replaces the IR tree, a *template* is a tree that has to be matched with the IR tree, an *action* is an emitted target code fragment and *cost* indicates the costs associated with this template. Fig.6 shows the *action* for the indirect addressing, the template comes with *cost = 3*. An optimizing code generation matches all templates of the tree-rewriting rules of the target machine against the subtrees of the IR tree during a depth- first traversal of the tree, searching for a minimum cost matching.

For an efficient tree matching it is necessary to have a template tree that can be easily identified within the complete tree. This allows to keep the complexity of the tree matching procedure manageable. Since the modulo-addressing information is useful for the code generation as it is, a `%`-node must be available in the IR. The representation of `base[count % N]` using the `%`-node can be found in fig.7b. The `%`-node is already available for some IR. So far, even if this node exists, handling the information is far from being efficient. This is due to the fact that the compiler does not distinguish whether the modulo-operation is used as an arithmetic expression or as an address calculation. Whenever a `%`-operator is identified, it is transferred into a functional equivalent representation, e.g.:

$$count\ \ \%\ \ N$$

$$\text{if } N = 2^k \quad : \quad count\ \ \&\ \ 0\underbrace{11..11}_{k} \tag{2}$$

$$\text{else} \quad : \quad count - int(\frac{count}{N}) * N \tag{3}$$

(Notice that eqn.(3) is a compiler dependent representation of the modulo-operation, since ANSI-C refers it to the compiler how to handle negative values of `count`.) The representation of a modulo-addressing mode following eqn.(3) is presented by fig.7a. Keeping in mind the lack of the modulo-addressing mode for general purpose processors, a unique handling of modulo-arithmetic and modulo-addressing does not come as a surprise. Most DSP-compilers simply have been ported from compilers that originally have been built for these general purpose architectures.

As stated above, when the modulo-operation is used to specify an address, there is no use in transferring the modulo information into a (functionally equivalent) arithmetic representation. Therefore, if an architecture supports the modulo-addressing mode, the compiler has to distinguish whether an arithmetic operation or an address calculation is performed using the modulo-operator. Different from todays concepts, the IR remains unchanged during optimization steps if the `%` node can be identified to be part of an address calculation. Identification is possible by an upward traversal of the tree: if an `ind`-node is found, the structure remains unchanged, otherwise the node is replaced in the traditional way.

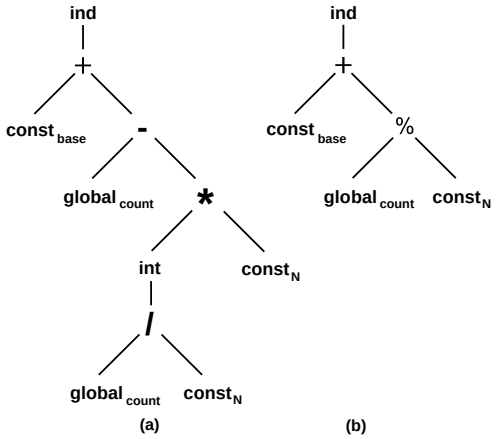

Figure 7. (a) arithmetic expansion, (b) modulo node `%`

V OPTIMIZING CODE GENERATOR

Using the information inherent to the IR-tree, it should be possible for the code generator to identify the modulo-addressing in principle. If we assume that code generation is based on tree matching, it must be analyzed which templates are available for matching the modulo-addressing representation within the IR. As an example, we use the NEC μPD7701x [11] instruction set which serves as a reference target for the results in sec.6. Fig.8 identifies the sequences of actions that are related to the C-formulation (`r=base[count%N]`) and the assembly code statement (`r1 = *dp1 %%`), respectively.

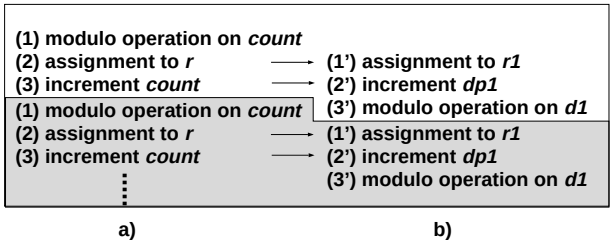

Figure 8. Modulo addressing (a) C-code,(b) assembly

Obviously, directly matching one modulo instruction with one assembly instruction (white background) results in a considerable overhead, since action (1) has to be emulated in assembly first, and action (3') has to be reset. However, when there are successive accesses to the circular buffer (gray background), a much better matching and a reduced relative overhead result. By taking a closer look at the C-code for typical DSP-applications that utilize the modulo-addressing capabilitiy, e.g. the FIR-filter (fig.4), it becomes clear that these call for exactly this periodic access. To identify the

periodic matching the code generator has to identify *adjacent accesses* to a circular buffer i.e. different accesses to the same buffer, where there is a path in the program flow graph between the two buffer accesses with no other access to the same buffer. This is impossible by just using the simple tree rewriting rules but requires an additional information structure, introduced by us as a *Circular Buffer Table (CBT)*.

The CBT-concept is derived from the well known symbol table concept [9]. Fig. 9 illustrates the concept by an example.

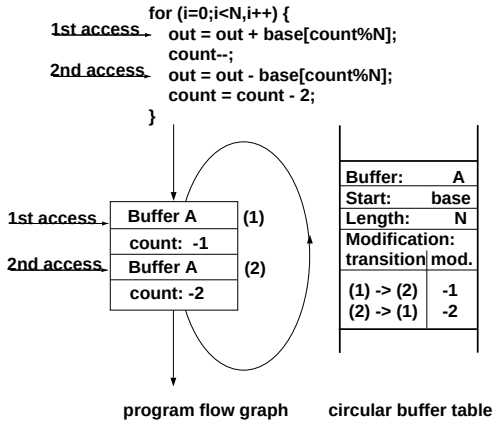

Figure 9. The *Circular Buffer Table* (CBT) concept

The example shows the stored information for a single CBT entry. This information consists of 4 sections:

- *Buffer* assigns a symbolic name to the circular buffer,
- *Start* indicates the lower bound of the circular buffer,
- *Length* informs about the buffer length,
- *Modification* informs about the step size between two adjacent accesses

Whenever a modulo addressing is identified in the tree structure, it is checked whether it is an access to a circular buffer that is already included in the CBT. Otherwise a new entry to the CBT is generated. Compared to the concept of language extensions the CBT approach comes with several advantages:

- monitoring of each circular buffer for the complete program flow. Initialization of the circular buffer is necessary only once.
- no limitation on the number of circular buffers. If there are more circular buffers in the program than can be handled by the hardware, a context switch is possible, similar to register spills.
- no explicit initialization of the registers of the circular buffer by the programmer. He can concentrate on the functional part of the algorithm

VI RESULTS

The following benchmarking results have been achieved for the NEC μPD7701x architecture [11] for a simple 4-tap FIR-filter. The '-ref' indicates the hand-optimized assembly reference code, '-gw' the gliding window C-code formulation of fig.3, '-cb' the circular buffer C-code formulation of fig. 4. Since the NEC-compiler supports DSP-specific language extensions, we included the benchmarking results as well,: '-cb-ext' indicates only the extensions for circular buffer support have been applied, '-all-ext' stands for all possible extensions.

Results indexed with 'COTS (commercial-of-the-shelf)' were obtained using the compiler delivered by the vendor, 'optimizing (CBT)' gives the results when modulo-addressing is supported in the way described above. 'all optimizations' indicates that in addition to the CBT-approach optimum data memory allocation [5,6] as well as data type matching [12] has been done by the compiler.

code	compiler	clock cycles	program words	data words
fir-ref.asm	no	20	6	34
fir-gw.c	COTS	436	36	35
fir-cb.c	COTS	582	57	35
fir-cb-ext.c	COTS	84	9	33
fir-cb.c	optimizing (CBT)	84	9	33
fir-all-ext.c	COTS	21	7	34
fir-cb.c	all optimizations	21	7	34

The results indicate that advanced compiler concepts allow to significantly improve the code quality. It also states that language extensions can be seen as a short term solution for improving existing compilers. For the specific language extensions used here (memory bank allocation, fractional data type, modulo-addressing) the results show that the information can be extracted by an optimizing compiler.

VII SUMMARY

The design requirements for digital signal processing systems call for a significant improvement of the compiler generated code quality. Analysis indicated the inefficient utilization of DSP-specific architectural features. Within this paper it has been pointed out that the propagation of inherent source code information might require an adaption of the internal representation of the compiler to the target code capabilities. As well, a more complex and sophisticated code generator is necessary, accounting for most of the effort that has to be spent for compiler improvements. All compiler optimizations are worthless if the source code does not include the necessary information. As a consequence, the DSP-programmer must have a basic understanding of the DSP-specific hardware features but is not asked to be familiar with a specific instruction set.

REFERENCES

[1] V. Živojnović, J. Martínez, C. Schläger, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," in *Proc. of ICSPAT'94 - Dallas*, Oct. 1994.

[2] M. Willems and V. Živojnović, "DSP-Compiler: Product Quality for Control-Dominated Applications?," in *Proc. of ICSPAT '96*, (Boston), Oct. 1996.

[3] D. Ombres, "C and C++ Extensions Simplify Fixed-Point DSP Programming," *EDN*, pp. 135–138, Oct. 1996.

[4] M. Willems and V. Živojnović, "DSP Specific Language Extensions to ANSI C - An Overview," submitted for publication.

[5] M. Saghir, P. Chow, and C. Lee, "Automatic Data Partitioning for HLL DSP Compilers," in *Proceedings of ICSPAT '95*, pp. 866–871, Oct. 1995.

[6] A. Sudarsanam and S. Malik, "Memory Bank and Register Allocation in Software Synthesis for ASIPs," in *Proceedings of ICCAD '95*, Oct. 1995.

[7] G. Araujo, S. Malik, and M. Lee, "Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures," in *Proceedings of DAC'96, Las Vegas*, Oct. 1996.

[8] M. Willems, M. Jersak, and V. Živojnović, "DSP-bezogene Spracherweiterungen - Möglichkeiten und Grenzen," in *Proc. of DSP Deutschland 95, Munich*, pp. 100 – 110, Sep 1995. in German.

[9] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[10] A. Aho, M. Ganapathi, and S. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," pp. 491–516, 1989.

[11] NEC Corporation, *NEC μPD7701x Family User's Guide*, 1995.

[12] H. Keding, "DSP specific Compiler Optimizations," Diplomarbeit D 333, Institute for Integrated Systems in Signal Processing, Aachen University of Technology, 1996.