WRITING AND COMPILING DSP ALGORITHMS IN AN ASYNCHRONOUS APPLICATIVE LANGUAGE

Markus Freericks *Alois Knoll*

Technische Universität Berlin Sekr. FR 2-2
Franklinstr. 28-29 1000 Berlin 10
mfx@cs.tu-berlin.de

ABSTRACT

The functional programming language ALDiSP, which is specially tailored to the needs of DSP system specification, is briefly presented. ALDiSP is based on asynchronous concepts that make it easy to specify interrupt-driven control flow. It is shown how ALDiSP specifications can be translated into efficient code using techniques of *abstract interpretation* and *partial evaluation*. By applying abstract interpretation to the set of possible states that a program can encounter, a static schedule can be found if one exists. Practical problems of this approach are discussed and solutions outlined.

1. INTRODUCTION

For a variety of reasons the development of a language for the specification and implementation of DSP algorithms is a demanding challenge. The requirements are even higher when real-time systems have to be specified.

The traditional doctrine suggests that execution speed can only be achieved in "low-level" languages, i.e., ultimately, by using the machine language. In fact, most real-time DSP applications today are still programmed in assembler, usually supported by DSP macro libraries. There have been adaptations of traditional imperative languages, most notably "C", but these have met with limited success. Special-purpose languages have also been developed. One of the more prominent ones, which serves as our comparison, is SILAGE[9].

In 1989-90, two languages, ImDiSP[1], an imperative one, and ALDiSP[5, 6], which is based on functional concepts, were developed at TU Berlin in the context of the CADiSP project[7]. An ImDiSP-compiler for the Motorola 96000 has been written.

The ALDiSP compiler **ac** is under development, after a first interpretive implementation of the language made it possible to evaluate the novel concepts.

This paper gives a short introduction to the key real-time mechanism of ALDiSP, the *suspension*, and describes the techniques on which that compiler is based.

2. THE LANGUAGE

The language ALDiSP was already presented in [4] and was first fully described in [3]. Therefore, only a short overview is to be given here.

ALDiSP is a call-by-value functional language with the following features:

- a *delay* form provides lazy evaluation on demand
- a *module* facility

This work was in part supported by an Ernst-von-Siemens grant.

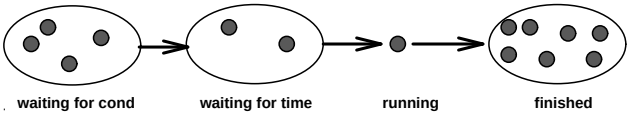

Figure 1: suspension state model

- overloaded function definitions
- user-definable types; arbitrary predicates can be used as types
- exception mechanism: both aborting and returning exceptions, dynamically bound
- rounding and overflow expressible via exceptions
- automatic mapping of all functions on arrays and signals
- higher-order functions
- no restrictions due to type-checking
- i/o functions
- time and i/o expressed via suspensions

A formal semantics and a set of DSP application examples can be found in [6].

3. THE SUSPENSION MECHANISM

All real-time functionality an ALDiSP is based on only one new construct, the *suspension*. This construct also makes the implementation of data-driven evalatuation possible, a feature not found in other functional languages.

An expression as

`suspend` *expr* `until` *cond* `within` *limits* `end`

evaluates to a placeholder object ("a suspension"). When, after some time has elapsed, the condition becomes true (*cond*s are usually tests depending upon global state, or the `true`), the expression will be evaluated within the time range defined by the limits (relative to the point of time when *cond* becomes true). After the evaluation has taken place, the result replaces the placeholder (just like a promise in MultiLisp[10] is replaced).

The evaluation model of the program as a whole is depicted in fig.1: There are three pools containing suspensions, with one currently executing suspension. Newly created suspensions are placed in the 'waiting' pool. When their *cond* becomes true, they advance to the 'ready' pool; a waiting counter is set to an arbitrary number within the *limit*s.

When the expression of the current suspension is finished (a process in which many new suspensions may have been created and put into the 'waiting' pool), the result is placed in the 'result' pool, and some arbitrary suspension whose counter is 0 is selected from the 'ready' pool to be evaluated next. If there are no eligible suspensions, program

time advances[1], and the counters of the waiting suspensions are decremented. The whole program stops when the first two pools are empty and the evaluation of the current suspension's expression terminates, though most real-time programs do not terminate.

If this description seems to evince a very inefficient evaluation mechanism, it should be remembered that a suspension is roughly equivalent to an interrupt handler. Furthermore, this description gives the *semantics* of suspensions, not their *implementation*. For example, the "arbitrarily chosen" counter is in reality determined by the reaction time that can be statically guaranteed for a given event.

A suspension is an expressions that is "suspended" until some condition holds. This condition relates either to other suspensions (waiting for them to evaluate) or to i/o state. Accessing the value of an as yet unevaluated suspensions suspends the accessing function itself. A predicate *isAvailable* tests fo the availability of a suspension's value.

A good example for the use of suspensions is the valve controller:

```
func guard_closed_valve()=
   suspend open_valve();
           guard_open_valve();
   until current_pressure()>100.0
   within 0ms,2ms

func guard_open_valve() =
   suspend close_valve();
           guard_closed_valve()
     until current_pressure()<95.0
    within 0ms,5ms
```

These functions implement a two-state controller in a direct way. A `suspend` expression consists of three parts: the expression that is to be suspended; the condition upon which the evaluability depends, and two "duration" values. The latter determines the time frame, relative to the point where the condition becomes true, in which the expression must be evaluated. Providing a condition that is constantly "true" allows the user to request fixed timing delays.

The execution of an ALDiSP program is a two-level process, managed by a scheduler and an evaluator. The scheduler controls the i/o and activates suspensions; each suspension's value is then computed by the evaluator. If, during the evaluation, a side-effect is attempted, the evaluation blocks and returns a suspension. The scheduler then effects the side-effect and, later, re-awakens the suspended evaluation. Thus, evaluation is purely functional.

4. STREAMS

Recent trends in incorporating i/o into functional languages use the concept of *streams*. A stream is a (usually infinite) sequence of values produced by a function. Only a finite prefix of a stream must be represented extensionally, i.e. as a data structure; the infinite remainder of the stream is can be represented as a function.

Streams can be used to model an output-driven system: whenever an output value is required by the printing function, a new stream element is computed. Input can be provided as a stream, too; e.g., a file can be modelled as a (finite) stream of characters.

Standard streams do not lend themselves to modelling input-driven behaviour. A suspension-based analogon to streams, dubbed *pipes*, provides for input-driven i/o.

[1] "Time" here is, of course, "virtual" time. It depends upon the speed of the compiled program whether this virtual time can be as fast as the real time.

The following is a complete ALDiSP program that reads an input at a fixed rate, applies a simple second-order FIR filter to it, and writes it out again:

```
func ReadFromRegister(Rate,Reg) =
  read(Reg) ::
     suspend readFromRegister(Rate,Reg)
       until true within Rate,Rate

func WriteToRegister(Reg,OutputPipe) =
  {write(Reg,head(OutputPipe));
   suspend WriteToRegister(Reg,tail(OutputPipe))
     until isAvailable(tail(Reg))
    within 0ms, 0.1ms}

func FIR(a0,a1,a2)(inp) =
  let s0 = 0::inp \\ :: implements the t^-1 operator
      s1 = 0::s0 \\ the '0's are the initial values
      s2 = 0::s1
  in
      inp + a0*s0 + a1*s1 + a2*s2
  end

filter1 = FIR(0.98132343,-0.3225834,0.12465574)

net
 SamplingRate = 1 sec / 44000
 Input = ReadFromRegister(SamplingRate,StdIn)
in
 WriteToRegister(StdOut,filter1(FilteredInput))
```

The `ReadFromRegister` function samples a given input register (e.g., an ADC) at a frequency of 44kHz. The resulting pipe is then filtered and written to an output register (e.g., a DAC).

5. ARRAY MANIPULATION

Mathematical and DSP code is usually very much concerned with the efficient handling of arrays (vectors and matrices). There are two approaches to processing arrays:

- In imperative languages like FORTRAN, arrays are manipulated element-wise: The array is seen as a data structure that holds values. Programs in such languages are usually heavily loaded with nested loops.
- In functional and special-purpose array languages like APL, arrays are treated as entities, i.e. as values. Element extraction is then just one of the operations defined on arrays. This approach employs higher-order functions such as the "reduce" operator of APL, that applies a binary operator pairwise to all elements of an array, thus reducing it to a scalar. For example, "reduce +" gives the sum of all elements of an array.

ALDiSP facilitates the second approach[2], because it is both easier to program and easier to compile efficient code for it.

The compilation of code for arrays in ALDiSP employs the duality of arrays and functions. As every array can be seen as a function of indices to values, every function of a finite data domain can be tabulated as an array. For the ALDiSP compiler this means that there are two possible optimizations to perform: whenever the compiler encounters a function that is called only with a small range of possible arguments (and that does no I/O), the compiler can tabulate the possible results. A simple heuristic would be to measure the code size needed to implement the function, and tabulate the function whenever the resultant array is not much bigger than the code (e.g., less than a factor of 2). An example of utilizing this optimization is the FFT

[2] Eeven FORTRAN-90 does, to a small extent!

function, where a table of twiddle factors (complex units of root) has to be held. In an imperative implementation, a vector of such factors would have to be initialized before using the FFT function the first time; in an ALDiSP implementation, one would just write down the factor ($e^{in/m}$) and let the compiler create the table and the lookup computations.

The second optimization is of more importance in practice. Whenever an operation computes a new array value, this array can be represented *extensionally* (tabulated as an array data structure) or *intensionally* (as a function). For example, given two vectors A and B, the call $A + B$ can either result in a new data structure, or can be laid down as the function $\lambda i.A[i] + B[i]$. This second approach has a number of advantages: if the resultant matrix is not used totally, e.g. if only a small number of elements is ever accessed, the function (+) will only be called for those elements. Also, no memory is needed to store the extra elements. In fact, the only time that an extensional representation of an array is needed is when there is more than one access to many elements later in the program.

The ALDiSP compiler uses the intensional representation as a default and decides what functions/arrays to dump when the partial evaluator builds the final program.

6. COMPILATION TECHNIQUES

The two "traditional" compilation techniques for functional programming languages are combinator-based compilation and stack machines. The first is based on the translation of the program and its input into a graph which is then reduced in a rather arbitrary order. Stack machines have a more conventional machine model in mind; they are mainly used in the implementation of strict and impure functional languages. The LISP machines were hardware-implementations of this approach.

Both techniques are not well matched to ALDiSP, mostly because they rely on a tagged memory heap providing dynamic storage and garbage collection facilities.

Because of this, we utilize two new methods: *abstract interpretation* (AI) and *partial evaluation* (PE). An AI of a program is a "simulated execution" on *abstract input values*. For example, an operation that reads an input register will return, if interpreted abstractly, a symbolic value that denotes "some 16-bit integer". Furthermore, when a truth value evaluates to "unknown" in the context of an `if` expression, both branches of the `if` have to be simulated. As a consequence, the AI process is in danger of not terminating: if the program contains an infinite loop, the abstract interpretation may well be infinite, too. To prevent such situations, the AI maintains a global table of function calls "open" at any time. Since, in ALDiSP all looping has to be implemented via recursion (or implicitly, by automapping), this guarantees that every loop has to register as a call to a function for which an "open" call exists, i.e., one for which no "return" is registered. In the case of a loop, the recursive function call is aborted and a dummy value ("bottom", the least defined object) is returned.

The call table (also known as *call cache*) has a second purpose: after the AI has been performed, it is known for each function how many times it has been called and with what types of arguments. Using this knowledge, a PE rewrites the program by *specializing* functions, possibly inlining them.

Abstract interpretation and partial evaluation are a generalization of traditional optimization techniques based on data-flow analysis, such as constant-propagation, inlining, and dead code removal: AI is data-flow analysis via simulated execution; and PE is a combined function inlining and constant propagation using the information gained in AI. In contrast to standard optimization techniques, AI and PE go beyond the boundaries of function definitions and rearrange the whole program. Standard techniques are usually restricted to single function definitions or even basic blocks only.

If a language has a formal denotational semantics (which can usually be implemented as a functional program rather easily), it is possible to prove that some abstract interpretation of this language is "correct". There are a variety of possible abstract interpretation schemes for a give language, depending on

- what abstract values are needed: a simple abstract value would be "some number", a highly sophisticated one would be "a vector of 32 floating-point values, none of which is negative or zero, which should be held in cache". During an abstract interpretation, all kind of extra information can be lugged around in the abstract values.
- how recursion is treated: the abstract interpretation of an `if` construct needs to follow both arms of the conditional. In the context of a recursive function, this leads to nontermination. By caching the calls to functions, these can be avoided, but it is known that the cost of finding a fixed point for a recursive function can be exponential in the height of the abstract data domain – that is, quite expensive.
- How "context information" is used: when the arms of an `if` expression are evaluated, a context is assumed in which the condition of the `if` is either true or false. This information can be carried along and used to steer the progress and, later, the specialization process.

These, among others, are active areas of research[11, 12].

6.1. STATIC CONTROL FLOW

Real-time algorithms, especially those in the realm of DSP, tend to have a very regular (or static) control flow. This is the reason to expect that a program that specifies such an algorithm will behave very well under abstract interpretation, even if the language very "dynamic" one.

Typical ALDiSP programs are indeed reduced to the expected core functionality by the partial evaluation process. In analogy to the well-known *syntactic sugar*, a term that denotes all "syntactic niceties" of a language that can be stripped away in a parser/preprocessor without any deeper understanding of the program, we would like to introduce the term *semantic sugar* to denote all mechanisms that can be removed by a partial evaluator. These features include higher-order functions, overloading, most type checking, automatic mapping, exceptions, etc.

It is still possible to write programs that the partial evaluator cannot handle – but these are mainly contrived examples. Most "natural" DSP applications are well-behaved.

6.2. STATE ANALYSIS

When partial evaluation is completed, the functions that constitute the program are simplified as far as possible. Still, the other part of the run-time model has to be handled: the scheduler.

Here, too, the abstract interpretation approach is applied. When running the program, there are two main *states*: The system is *resting* when all pending suspensions are either waiting for input to occur (or time to elapse), or are dependent upon suspensions that are waiting for input. As time passes, input takes place and suspensions are activated. The evaluation of these input-dependent suspensions enables other suspensions to run, and so on. Eventually, the system comes to rest again.

The above outlined process can be "abstracted" by providing abstract input whenever the system is resting. If the control behaviour of the program is independent of the actual input values, the only thing to model is the absence

or presence of input. Otherwise, all possible input values must be simulated. At the moment, our compiler does not do the latter; the overhead would be too high.

Since a program may be waiting for a number k of input sources simultaneously, all possible combinations of input events must be simulated. For each such combination, a new abstract state is reached after the system comes to rest.

Two abstract states are said to be "similar" if all suspensions they contain are waiting for the same conditions, and only their particular variable bindings (modelling the state) differ. Furthermore, the differing variable bindings must be of scalar type; i.e., not contain differently-shaped data structures. Those would prohibit a static memory layout.

If the abstract interpretation of the scheduler finds that the set of possible states at time k (denoted S_k^2) is "similar" to the state k times steps later (S_{k+c}^2), it has found a static schedule. In a purely synchronous program, S_k^2 will be similar to S_{k+1}^2, i.e., the set of possible states is the same at all points in time.

Once a stable schedule is found, compilation can follow. The scheduler can be executed "in the compiler's mind", and no run-time scheduling is needed any more.

7. PROBLEMS

There is one practical problem with state analysis: In a program with two sampling frequencies, the constant c will be the lowest common denominator of the frequences – quite a large number, and a real problem for the compiler. In addition, even if the number of input sources is small (say, 4 or 5), there are 2^k possible combinations. Heuristics are needed to speed up this simulation process.

On the other hand, it is quite possible for the partial evaluator to simply take each suspension for its own and compile code for it; but then it might be possible that incorrect programs could be created. For example, our abstract machine assumes that there is one "interrupt handler" for each possible input event. If there are two suspensions that install themselves into the same "interrupt slot", one will overwrite the other. (If such a situation were detected in the state simulation, those suspensions would be merged into one.)

A second problem is that of *code explosion*: a naive partial evaluator will unroll all loops and over-specialize many functions. Here, too, heuristic guides are needed to control the expansion process. One simple heuristic is the "20-rule": an independent counter attached to each function definition assures that a given function is expanded not more than 20 times. For loops showing a "logarithmic" behaviour (e.g. binary search in an array), this results in total unfolding for data structures smaller than a million entries (2^{20}); otherwise, a factor of 20 is small enough not to prove disastrous if a loop is unrolled that should not.

8. RELATED WORK

Partial evaluation initiated from attempts to automatically generate compilers from interpreters. A lot of current work in abstract interpretation and partial evaluation tries to extend the range of languages and features these methods can be applied to. A seminal collection of papers can be found in [11]; the latest in a series of workshops is [12].

Berlin[13] is one of the first articles describing the use of partial evaluation for "number crunching" applications.

A recent article by Nirkhe and Pugh describes applications of partial evaluation techniques to an imperative DSP language. Their approach is based on a language with user-annotated binding time information and fixed maximum loop counts.

9. CONCLUSION

We have shown that it is possible to compile a language equipped with a number of semantic complexities targeted at a demanding realm such as DSP programming, in a correct way into efficient code. While the compilation process is a costly undertaking, we hope that it is more than made up by the gained correctness and ease of programming.

Our compiler is now able to run the first phases of abstract interpretation and partial evaluation; the state analysis technique outlined in the last section is being implemented.

As an added bonus, the abstract interpreter, being a superset of an "actual" interpreter, can be used to simulate programs (albeit slowly – the abstract interpreter is burdened by a lot of bookkeeping chores.)

For now, no specific target architecture is envisioned. The output of the compiler is a pseudo-code similar to "C" or FORTRAN.

10. REFERENCES

[1] V. Kruckemeyer, A. Knoll: *Eine Imperative Sprache zur Programmierung digitaler Signalprozessoren*, Forschungsberichte des Fachbereichs Informatik Nr. 1990/10, TU Berlin

[2] U. Blenk, A. Fauth, A. Knoll: *An Imperative Language for Digital Signal Processing*, in: Int. Conf. on Signal Processing '93 (ICSP'93), Beijing

[3] M. Freericks, A. Knoll: *ALDiSP - eine applikative Programmiersprache für Anwendungen in der digitalen Signalverarbeitung*, Forschungsberichte des Fachbereichs Informatik Nr. 1990/9, TU Berlin

[4] A. Knoll, M. Freericks: *Aldisp - An Applicative Real-Time Language for DSP Programming*, CIT/IEEE International Conference on Signal Processing '90 (ICSP'90), Beijing

[5] A. Knoll, M. Freericks: *An applicative real-time language for DSP-programming supporting asynchronous data-flow concepts*, in: Microprocessing and Microprogramming, Vol.32, No. 1-5 (August 1991 - Proceedings Euromicro '91), pp. 541-548

[6] M. Freericks, A. Knoll, L. Dooley: *The Real-Time Programming Language ALDiSP-0: Informal Introduction and Formal Semantics*, Forschungsberichte des Fachbereichs Informatik Nr.92-26

[7] A. Knoll, R. Nieberle: *CADiSP - a Graphical Compiler for the Programming of DSP in a Completely Symbolic Way*, Proc. IEEE-ICASSP'90, pp. 1077-1080

[8] V. Nirkhe, W. Pugh: *Partial Evaluation of High-level Imperative Languages, with Applications in Hard Real-time Systems*, ACM SIGPLAN-SIGACT POPL'92, pp. 269-280

[9] *Silage User's and Reference Manual*, prepared by Mentor Graphics/EDC, June 1991

[10] R. Halstead: *Multilisp: A Language for Concurrent Symbolic Computation*, in: ACM TOPLAS, Oct. 1985, pp. 501-538

[11] D. Bjœrner, A. P. Ershov, N. D. Jones: *Partial Evaluation and Mixed Computation*, Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, North-Holland, 1988

[12] *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Yale Univ., June 17-19, 1991, published as: SIGPLAN Notices, Vol.26, No.9, Sept. 1991

[13] A. Berlin, D. Weise: *Compiling Scientific Code Using Partial Evaluation* IEEE Computer, Dec. 1990, pp. 25-37