MAT2DSP – A TOOL FOR EVALUATING IMPLEMENTATION COMPLEXITY OF SIGNAL PROCESSING ALGORITHMS

Benjamin Friedlander
Dept. Elec. & Comp. Eng.
University of California
Davis, CA 95616

ABSTRACT

MAT2DSP is a MATLAB toolbox, currently under development, whose function is to estimate the implementation requirements of algorithms specified in the form of a MATLAB program. This toolbox is aimed at providing researchers developing advanced signal and image processing algorithms, a quick and convenient way of estimating what would be needed to implement their algorithm on a specified processor. MAT2DSP analyzes the user program and generates reports on its computational requirements.

1. INTRODUCTION

The development of advanced signal processing systems starts with the design of the processing algorithms. Only after the algorithms have been specified (in the form of a mathematical description or a flow-diagram) does the hardware/software design process begin. A number of mathematical analysis packages are available today for aiding algorithm development. These tools, which are widely used by the research community, include commercial mathematical software packages such as MATLAB, Mathematica, XMath, and others.

These mathematical analysis packages make it relatively easy to evaluate the performance of the algorithm (in terms of detection probabilities, classification probabilities, image quality, *etc.*). However, none of the existing algorithm design tools provide adequate information about the cost of implementing a given algorithmic solution. Yet the algorithm chosen to solve a particular problem can have a strong impact on the complexity and cost of the final implementation. Since there are often multiple algorithmic solutions for any given problem, having different cost/performance trade-offs, it is important to have the ability to perform these trade-offs.

This work was supported by the US Air Force under contract no. F33615-93-C-1312, sponsored by the Advanced Research Projects Agency.

In this paper we describe a software tool, currently under development, which is designed to provide the algorithm developer with an analysis of the implementation costs of a given algorithm. This tool is being implemented as a MATLAB toolbox, which we named MAT2DSP. However, the same methodology can be used to develop versions of this program which will work with other mathematical software packages.

Giving the researcher responsible for developing advanced algorithms such a tool will speed up significantly the algorithm development process by quickly eliminating approaches which clearly fail to meet the system constraints, and focusing time and energy on the practical approaches. It should be emphasized that the tool does not have to be precise – approximate estimates of cost would be quite adequate at this early stage of the design process. Our main objective is to create a tool which is convenient to use, and which can be easily modified to suit the needs of the user.

Clearly, there is a trade-off between the complexity of the MAT2DSP program and the accuracy of the estimates it can provide. In order to provide very accurate results the program will have the capture the full details of the implementation. This is tantamount to performing a complete hardware/software simulation of the intended implementation. This is very definitely *not* our intention. Our goal is to provide the best performance estimates possible using a relatively crude, high-level representation of the intended implementation.

2. THE APPROACH

The computational part of MATLAB, and other high-level signal processing programs, is built from a set of primitive functions such as: vector-vector and vector-matrix multiplies, FFT, convolution, filtering, solution of a set of linear equations, eigenvalue decomposition of a matrix, and so on. These functions operate on data vectors or arrays. Advanced signal processing functions of arbitrary complexity can be composed from these primitives.

0-7803-2431-5/95 $4.00

We start with a MATLAB program, which we refer to as the 'target program', which implements a given signal or image processing algorithm. The target program consists of one or more M-files and functions. The MAT2DSP program (which is also written in the MATLAB language) operates on this program and produces one of several user selected reports which contains information about the computational requirements of the algorithm and an estimate of its runtime on a user specified processor or mix of processors. In particular, the report contains a detailed breakdown of the computational primitives used by the target program.

The generation of this report involves three separate steps:

1. In the first step the target program is translated into a modified version of the program. This modified program is another MATLAB program which, when executed, will perform all the computations of the original program, and in addition, will generate a detailed record of what took place. We refer to this record as the 'primitive list'.

2. In the second step, the modified program will be executed to generate the primitive list. This list contains detailed information about the number and types of computations performed by the target program, and provides the "raw material" from which various reports can be generated. This information characterizes the signal or image processing algorithm implemented by the target program, and is independent of the hardware on which the algorithm will be eventually implemented.

3. In the third step we run the report generator program. This program uses the primitive list and a database, to estimate the runtime of the target program on a user specified processing hardware. The database contains information about the runtimes of the computational primitives on different hardware software platforms. Different types of reports of varying levels of complexity can be generated based on the data contained in the primitive list and the database.

In the following sections we describe the three components of the MAT2DSP program in more detail. It should be emphasized, however, that this paper presents one possible implementation of the program. We are currently exploring alternative implementations, and it is quite possible that the final program will be quite different from the preliminary version which is described here.

2. THE AUTOMATIC TRANSLATOR

The automatic translator is a special compiler for the MATLAB language, which generates a modified MATLAB program. The modified program performs the tasks of the original program, and also records the number of primitive calls (calls to the set of internal functions) and the parameters used in each call. The set of internal functions includes all MATLAB internal functions. The user may add additional functions or script files to the set of internal functions. Any external function which is called from the input program is translated in the same fashion. For example, suppose that our input program is `prog`, and it calls the external functions `func1` and `func2`, and `func1` calls the external function `func3`. The translator will generate the following modified MATLAB programs: `tprog, tfunc1, tfunc2, tfunc3`. These are the modified versions of `prog, func1, func2, func3`, respectively.

Internal MATLAB where expressions such as `x+y` are replaced by `madd(x,y)` and expressions such as `fft(x)` are replaced by `mfft(x)`. The functions `madd` and `mfft`, in addition to performing the addition and `fft` operations, record the parameters used in each call. As other compilers, the translator reports to the user the presence of errors in the source program.

There are two parts to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation.

The analysis usually consists of three phases:

1. *Lexical analysis*, in which the stream of characters making up the source program is read from left-to-right and grouped into *tokens* that are sequences of characters having a collective meaning.

2. *Syntax Analysis*, in which tokens are grouped hierarchically into nested collections with collective meaning. In compilers syntax analysis is called parsing.

3. *Semantic analysis*, in which certain checks are performed to ensure that the components of a program fit together meaningfully. The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code generation phase.

The translator has a simpler structure than that of a full-blown compiler. It has only two blocks: the lexical analyzer and the syntax directed translator. The lexical analyzer converts the stream of input characters into a stream of tokens that becomes the input

for the following phase - the syntax directed translator. The syntax-directed translator is a combination of syntax analyzer (parser) and code generator. As in most compilers, the interaction between the lexical analyzer and the parser is implemented by making the lexical analyzer a subroutine of the parser. Upon receiving the "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token. The current token being scanned is usually referred to as the *lookahead* symbol.

We have chosen to use the recursive descent parsing method described in [1]. Recursive descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. In general, the selection of a production for a non-terminal may involve trial and error. In other words, we may have to try a production and backtrack to try another production if the first is found unsuitable. Most production in the above syntax directed definition of MATLAB do not require backtracking.

The translator has two main routines: The parsing routine (`parse`) and the lexical analyzer routine (`lexan`). The translator is activated by the MATLAB line `parse(progname)` where progname is a string representing the name of the MATLAB program to be translated. The parsing routine calls `lexan` whenever it needs a new token.

It is worth noting that in MATLAB the distinction between function names and identifier names is not trivial. Consider, for example the statement `y = cos(2*pi*k*n/N)` in which `cos` is interpreted as the MATLAB internal function `cos`. On the other hand, the `cos` in `cos=sqrt(1-sin^2(alpha))` is interpreted as an identifier. The lexical analyzer must therefore be able to determine for each alphanumeric string if it is a function or identifier. This is implemented by generating and maintaining a list of all existing identifiers called varlist. At the beginning of the parsing procedure this list is empty. Whenever an identifier is observed in a global declaration statement or in the left side of an assignment statement, it is added to varlist. When the lexical analyzer identifies an alphanumeric string it checks if it appears in varlist. If it does, the current token is ID (identifier). Otherwise, it is either IDPRIM (MATLAB primitive) or IDFUN (an external MATLAB function).

3. THE PRIMITIVE LIST GENERATOR

The modified program created by the automatic translator will next be run to generate the "raw" list of primitives. Each entry in the list consists of:

(i) the name of the primitive

(ii) the input type (real, complex, string)

(iii) the input size (dimensions)

(iv) a count of the number of times that particular entry was recorded

The "raw" list can be sorted and condensed to generate a summary list in which each type of primitive appears only once, together with statistics of the input size.

The computational primitives which appear in the list, include the MATLAB primitives, and may also include any user defined functions. If a certain DSP function will be implemented by a special purpose hardware, we will declare it to be a primitive, so that it will appear on the primitive list, rather than be decomposed into the MATLAB primitives which implement it.

The summary list described above does not preserve information about important aspects of the program such as sequencing of operations, and the presence of loops. We are currently developing a more sophisticated version of the primitive list generator which creates a hierarchical block representation of the program. This block representation preserves sequence information and identifies the presence of loops and branch points. The hierarchical nature of the representation makes it possible to generate reports containing a desired level of detail.

4. THE DATABASE

The database contains information about the implementation requirements of different primitives on different hardware/software platforms. The database is designed so that it can grow over time by including information about new types of hardware, as well as more detailed information about the hardware already included in the database. In its present form the database consists of two tables: one containing primitive related data, and the other containing processor related data.

The primitive table contains a listing of all the computational primitives and their attributes, and the name of the target processor. By default, all the primitives are assumed to run on a single target processor. However, the user can define an arbitrary mix of processor. This is useful in cases where special purpose processors are used to speed up the implementation. For example, the main program may run on a general purpose DSP chip, while the FFTs may be performed on an FFT chip.

The processor table contains a listing of target processors and their characteristics (fixed point, floating point, numerical precision, *etc.*). For each processor and each computational primitive, the table contains benchmark data on run-time vs. input size. Later version of the program will also include information about memory requirements.

At the present time benchmark data was collected only for MATLAB programs running on the same workstation as the MAT2DSP program. This data is being used to test how well the MAT2DSP program is able to predict runtimes of MATLAB programs. Next we plan to collect data for 'C' programs running on selected workstations. The ultimate goal is to incorporate in the database information about execution times on commercial DSP chips (such as TI, MOTOROLA, AT&T).

A number of alternative methods for constructing the database are being considered. At the two extremes we have a database constructed entirely from measured data, and one constructed entirely from computed data. The first means that all the data in the database was measured by running appropriate benchmarks on the target processor. The second means that all the data was computed from a few basic numbers (such as the number of machine cycles needed to perform addition, multiplication and memory access) using theoretical formulas for the execution times of the various primitives. Data based on measurements will, presumably, lead to more accurate performance predictions. However, the collection of such data is relatively costly and time consuming. Computed data is, presumably, less accurate, but much easier to collect, modify, and maintain.

5. THE REPORT GENERATOR

The report generator combines the information in the primitive list and the database to provide a report on the implementation requirements of the algorithm. A variety of reports, of various levels of complexity, can be generated. Initially we generate estimates of runtimes by type of primitive, and an estimate of the total runtime. In particular, the report provides a list of all the primitives used by the program, and the fraction of the total runtime used by that primitive. This list make it possible to identify quickly computational bottlenecks, and where to put most of the implementation effort.

Later versions of the report generator will take into account the more detailed program flow and the ability to pipeline or parallelize certain primitives or combinations of primitives. It will also take into consideration more detailed information about the processor architecture and examine potential I/O or CPU to Memory transfer bottlenecks. It is possible to incorporate a considerable amount of intelligence into the report generator, using the kind of reasoning employed by an expert hardware/software designer.

It should be emphasized that the proposed software tool does not have to provide extremely precise estimates. What is needed is to give the algorithm developer a rough idea of what is needed to implement a given algorithm, so that he can do an approximate cost/performance trade-off, to see which of several possible solutions will fit the system constraints. Once a given algorithm is selected and specified, its precise implementation requirements will be determined during the hardware/software design process.

We believe that a report generator of a relatively modest level of complexity will be adequate to achieve the stated goals of the MAT2DSP toolbox.

6. CONCLUSIONS

The MAT2DSP software tool described above is at a preliminary stage of development. Our intention is to develop a toolbox which can be expanded and modified over time, either by the developer or by the end user. Initially the toolbox will provide runtime estimates based on straightforward addition of the runtime estimates of the primitives from which the program is composed. Later versions will take into account more detailed information related to dataflow, program overhead, and data transfer times. The database will also evolve over time to incorporate more hardware/software platforms and more detailed timing information.

Our plan is to make it possible for researchers who develop advanced signal processing algorithms to get quick estimates of what would be required to implement their algorithms. They can do this without having to know much about the hardware and about the process of hardware/software design, since the necessary information is embedded in the MAT2DSP program and its database. This will allow the algorithm developer to be aware of the cost/performance tradeoffs of different solutions and come up with the best practical solution which fits the system constraints.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1988.