MATCHING DSP ALGORITHM TRANSFORMATIONS FOR POWER, PERFORMANCE AND MEMORY TRADE-OFFS

Vytautas Stuikys\textsuperscript{1}, Robertas Damasevicius\textsuperscript{1}, Jevgenijus Toldinas\textsuperscript{2}, Giedrius Ziberkas\textsuperscript{1}

\textsuperscript{1}Kaunas University of Technology, Software Engineering Department, Studentų 50-405, Kaunas, Lithuania, vytautas.stuikys@ktu.lt, damarobe@soften.ktu.lt, giedrius.ziberkas@ktu.lt
\textsuperscript{2}Kaunas University of Technology, Computer Department, Studentų 50-209, Kaunas, Lithuania, eugenijus.toldinas@ktu.lt

Abstract. The traditional development of complex digital signal processing (DSP) applications is now reaching its limits due to the intense pressure on design cycle and strict performance constraints. The new approach, called Algorithm-Architecture Matching, aims to leverage the design flow by a simultaneous study of both algorithmic and architectural issues, taking into account multiple design constraints, as well as algorithm and architecture optimizations. The aim of this paper is to consider the matching between a family (variants) of the given representative DSP algorithms and a given set of prescribed constraints (performance, energy consumption, accuracy, memory). We propose to use Feature Diagrams for the representation of the algorithm variability features per se and as well as for the matching representation, which is based on the algorithm feature and constraint trade-offs. Experimental results for the cosine look-up table, Fast Fourier Transform (FFT) and sparse matrix multiplication algorithms are presented.

Keywords: DSP algorithms, power consumption, energy awareness, embedded applications, pocket PC.

1 Introduction

The traditional development of complex digital signal processing (DSP) applications, which is based on the consecutive design flow (a theoretical study of the algorithms, a study of the target architecture, and finally the implementation), is now reaching its limits due to the intense pressure on design cycle and strict performance constraints. The Algorithm-Architecture Matching approach aims to leverage the design flow by a simultaneous study of both algorithmic and architectural issues, taking into account multiple design constraints, as well as algorithm and architecture optimizations. The matching problem, if we want to consider it systematically, requires a thorough study of the domain in the whole. The first step towards dealing with the problem systematically is the identification of the best strategy that has been already approved in the related areas such as software engineering.

Current approaches for architectural design of systems or components in software engineering (including embedded software) predominantly use the product line (PL) concept as a strategic goal. A software PL is a set of software systems that share a common, managed set of features satisfying the specific needs and are developed from a common set of core assets in a prescribed way [1]. The concept of PLs, if applied systematically, allows for the dramatic increase of software design quality, productivity, provides a capability for mass customization and leads to the ‘industrial’ software design [2].

We suggest to apply and adapt the PL concept focusing on the analysis of DSP algorithms domain, the analysis of constraints and identifying the adequate high-level models in order to be able to build a matching between these domains at the implementation phase. The key for the PL concept implementation is the use of domain analysis and domain modelling methods with a subsequent representation of the results explicitly. The basic concept we exploit in the analysis is variability of the domain (or more precisely, the commonality-variability relationships [3]).

In general, variability has many dimensions (e.g., variability of DSP algorithms even for the same task, variability of ever-growing technology capabilities, variability of constraints which are inspired by market demands, technology itself, user requirements, new appliances such as ambient intelligence, mobile computing, etc.). As a result of the previous observation, one can predict the complex relationships between various kinds of variability dimensions in the DSP domain. For example, variability of algorithms can be considered not only as an optimization problem (e.g., inventing more efficient algorithms) but also as a specialization problem (e.g., the one which is based on the extraction of useful data properties, algorithmic properties or application properties with respect to the given constraints). Thus the boundaries of matching between DSP algorithms, constraints and architecture are becoming extremely wide, especially having in mind such an important constraint now as energy awareness at the application level (e.g., in the context of mobile computing).

The aim of this paper is to consider the matching between a family (variants) of the given DSP algorithms and a given set of prescribed constraints (e.g., performance, energy consumption awareness, accuracy, memory and various trade-offs amongst the constraints). We suppose that having this matching a
designer could be able to justify the selection of the appropriate architecture at the implementation phase. On the other hand, a designer has two possibilities to implement algorithms: to use the hardware architectural solution or to implement the given algorithm as a part of application software.

In this paper we propose to use Feature Diagrams [4] for the representation of variability features of the analysed domains per se and as well as for the matching representation, which is based on algorithmic properties and constraint trade-offs identified either analytically or empirically.

The paper is organized as follows. Section 2 discusses the related works. Section 3 presents a general framework for the analysis of the algorithm-architecture matching task. Section 4 describes representation of narrow DSP sub-domains using Feature Diagrams. Section 5 considers algorithm and data specialization problem for the DSP domain. Section 6 presents the experimental results with representative algorithms. Finally, Section 7 presents the conclusions.

2 Related Work

The related works can be categorized into two research streams: 1) analysis of the architectures and design techniques for power, execution performance and/or calculation accuracy trade-offs, and 2) transformation-based optimization of embedded software for high performance or low power.

1) Power and performance trade-offs. A relationship between execution time and power consumption of different instructions for the i960 RISC (Reduced Instruction Set Computer) processor is analyzed by [5]. Since there has been no significant statistical variation between two criteria established, minimizing execution time also means minimizing energy consumption. Therefore, for DSP applications, instruction reordering, instruction packing, operand reordering, register allocation, and memory assignment can result in power savings. Aspects of DSP code generation for performance, code size, power and retargetability requirements are discussed by [6]. Graph-based solutions to embedded DSP software synthesis trade-offs are analyzed by [7]. Different processor configuration parameters for performance/energy optimization of DSP transforms are analyzed by [8]. An energy-aware source compilation framework for DSP applications based on multiple energy saving criteria (cache hits, processing units, anticipated scheduling factor, central processing unit (CPU) bus activity, binary code size) is described by [9].

2) Transformation-based software optimization. In [10] energy consumption is optimized using two well-known transformation methods – loop unrolling, where it aims at reducing the number of processor cycles by eliminating loop overheads, and loop blocking (tiling), where it breaks large arrays into several pieces and reuses each one without self interference. Compiler optimizations such as linear loop transformations, tiling, unrolling, fusion, fission, and scalar expansion are also considered by [11]. However, only loop unrolling has been shown to decrease the consumed energy. Software pipelining and recursion elimination for software-level energy optimization are also considered by [12]. Various source code transformations for software power optimization are discussed in [13, 14]. Different search strategies in optimization space for finding good (in terms of performance) source-level transformation sequences for typical embedded programs written in C are considered by [15].

3 General Framework for the Analysis of the Matching Task

A general framework of the matching task is depicted in Figure 1. At the core of the framework is the Y-diagram that links together three basic constituents of the task: algorithms, constrains (criteria) and architectures. The framework also outlines a context of the task (e.g., possible methods of the solution domain such as transformation, optimization, etc.), possible implementation technologies (e.g., in software or hardware), and applications) and identifies the sub-domain spaces for each constituent. The intersection between the constituents identifies the space for the matching task presented by a circle in Figure 1.

![Figure 1. DSP sub-domain spaces](image-url)
The space of the DSP algorithms is to be considered for each class of algorithms dedicated for a specific DSP task. If we look at the DSP algorithm taxonomies and the role a specific algorithm plays in the DSP domain, one can identify representative algorithms of the domain. We call representative algorithms those which are 1) most sensible to achieving the relevant criteria/constraints and 2) applications use them intensively. Examples are cosine calculation, sparse matrix multiplication, Fast Fourier Transform (FFT), etc.

The space of basic constraints is as follows: performance (time), energy, accuracy, memory and various trade-offs of the previous mentioned items. These constraints (criteria) usually are common to the entire DSP algorithm domain. However, some specific criteria cannot be relevant to a specific algorithm (e.g., accuracy for matrix multiplication algorithm). Each representative algorithm is to be matched to this set of constraints that are relevant to a given application. On the other hand, a representative algorithm has not a unique realization but rather a family of possible solutions. This family is composed having in mind such aspects as functional variants of algorithms (e.g., the same functionality can be implemented in a different way), properties of data in a domain. The other source of variability is the variants of selected constraints or their tradeoffs. As the scope of variability can be actually large, we need to manage this variability in some well-established way. In the next section we present a systematic approach to dealing with the problem using Feature Diagrams.

4 DSP Sub-Domain Representation Using Feature Diagrams

Here by the DSP sub-domain we mean the representative DSP algorithms and various constraints/criteria trade-offs which are induced by a given application. Figure 2 outlines a model of the sub-domain, which is represented using a feature diagram.

In general, a feature diagram is the tree-like notation or directed acyclic graph (DAG) that consists of a set of nodes, a set of directed edges, a set of edge decorations, relationships and constraints among features. A feature is understood as an externally visible characteristic of an item (i.e., concept, entity, algorithm, system or domain per se). The root represents the top level feature. The intermediate nodes represent compound features and leaves represent atomic features that are non-decomposable to smaller ones in a given context. The edges are used to progressively decompose a compound feature into more detailed features. Edges of the graph also denote relationships or dependencies between features. One can learn more about the notation from [16, 17].

In Figure 2, mandatory features are denoted by black circles above a feature box, while optional and alternative features are represented by white circles. Mandatory features express common aspects of the concept, whereas optional and alternative features express variability. All basic features may appear either as a solitary feature or in groups. If all mandatory features in the group are derived from the same parent in the parent-child relationship we can speak about the and–relationship among those features. An optional feature is the one which may be included or not if its parent is included in the feature model. Alternative features, when they appear in groups as derivates from the same parent, may have the following relationships: or, xor, case, etc. The xor–relationship among features derived from different parent also can be treated as a constraint. Usually this relationship appears as a constraint when features are derived from different parents.

![Figure 2. DSP sub-domain expressed through essential features and their ‘parent-child’ relationships (P-performace, A-accuracy, M-memory, E-energy)](image-url)
The presented model focuses on the specialization of the DSP algorithms. We discuss the specialization problem in more details in Section 5. We admit that for each representative algorithm there are a set of specialized algorithms that are identified as variants from 1 to n (see Figure 2). What is needed to solve the matching problem is to build the relationships between variants of constraints and variants of specialized algorithms. The relationships are summarized in Table 1.

Table 1. Possible relationships (+) and limitations (−) among constraints and specialized algorithms

<table>
<thead>
<tr>
<th>Constraints/ criteria space trade-offs</th>
<th>Cosine computation with Taylor series using Horner scheme</th>
<th>Cosine computation using look-up tables</th>
<th>Multiplication of sparse matrices</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy-Performance</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Energy-Accuracy</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>Energy-Memory</td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Energy-Performance-Accuracy</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>Energy-Performance-Memory</td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Performance-Accuracy</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>−</td>
</tr>
</tbody>
</table>

5 Algorithm and Data Specialization Problem for DSP Domain

Algorithms and data structures of an application are two interrelated attributes that have a direct impact on characteristics of the implementation. For a specific task, both algorithm and its data structure (if it is relevant to the given task) can be represented in two different modes: either as a generic or as a specialized representation. The first gives a general solution of the problem under consideration but, in many cases, this solution may be either not optimal, or not satisfying the given constraints. The second representation is a reconstruction of a generic algorithm (or perhaps together with its data structure) aiming at achieving some balance between optimality and given constraints. We call it the specialization of an algorithm. The specialized version of an algorithm is derived from the generic one through some reconstruction process.

In general, the problem of algorithm specialization is a generalization of the well-known program specialization problem [18]: a specifically annotated program (metaprogram) together with some known portion of its input (parameter value), is transformed into a specialized version obtained by pre-computing the parts of the program that only depend on this input. What is common for those problems is that both are aiming (1) to improve some characteristics, usually the same from the implementation viewpoint; (2) both are based on analysis and elicitation of some properties from the generic solution in order to realize the specialization. What is different is that for the first task analysis is performed at a higher abstraction level with the elicitation of algorithmic-based features (such as operations and data structures), while the second task considers program-related features (e.g., loops, variables, etc.).

We define program specialization more formally as a technique for separating a computation process into two parts, so that given \( f(x, y) \), it is rewritten as \( f_s(y) \), where \( x \) constitutes the static (early, constant) part of a computation and \( y \) – the dynamic (late, variable) part of it.

\[ T_s : f(x, y) \rightarrow f_s(y), x = \text{const} \]  

A separate case of specialization is data specialization [19]. This method aims at encoding results of early computations in data structures. The execution of a program is divided into two stages. First, computations on known input values are performed and their result is saved in a data structure (specialized cache or look-up table (LUT)). Then the algorithm is modified to use a specialized data structure instead of a performance-costly function. Of course, caching an expression is beneficial only if its execution cost exceeds the cost of a cache reference, i.e. is recommended only for such performance-costly functions as sine, cosine, logarithm, etc.

As specialization is based on properties of algorithms and data structures (e.g., level of sparseness, distribution of zeroes for the sparse matrix multiplication, etc.), different values of properties may imply different specialized algorithms [20]. Having in mind also other features (e.g., different generic algorithms, even for the same task, such as FFT; different extrapolations schemes for cosine calculation from look-up tables, etc.), which increase the number of possible specializations, variability of the algorithm space can be indeed large in terms of possible relationships (see Figure 2, the internal algorithmic features are missing). Some kind of relationships (e.g., performance and type of a specialized algorithm) can be derived through the use of analytic methods. However, a vast majority of evaluations for energy consumption awareness and algorithms
characteristics require empirical investigation. With the advent of mobile computing, this kind of evaluation is becoming as crucial as never before though other characteristics and constraints are important as ever.

6 Experimental Results with Representative Algorithms

6.1 Cosine Calculation using Look-Up Table (LUT)

The cosine calculation has been chosen as a representative algorithm of the calculation-intensive application. We analyse three variants of the representative algorithm in C#: standard C# library `Math.Cos()` function, cosine LUT and cosine LUT with linear interpolation. LUT-based cosine implementations consist of the automatically generated LUT of a predefined size and a wrapper function.

The trade-off here is that accuracy of the result depends upon the size of the LUT (Figure 3). Here accuracy is expressed in terms of max. error of the cosine LUT with respect to the standard `Math.Cos()` function on a mobile device. In a simple LUT without interpolation of function values, the value of a function argument is rounded to the nearest value for which a function value in a LUT exists. Thus the accuracy of this approach is not fine. A more complex approach includes a LUT with linear interpolation of the function values for these arguments of a function, which are not available in a LUT.

The complexity of the LUT based method without interpolation is constant. It requires only 1 multiplication for the calculation of a LUT index and does not depend upon the size of a LUT. The complexity of the LUT with linear interpolation is also constant. It requires 2 multiplications and 4 additions, and does not depend upon the size of a LUT.

The experiments were performed on a Compaq iPAQ H3900 (Pocket PC platform, Intel PXA250 400 MHz CPU, 32 MB RAM, Windows CE 3.0 OS). The calls to a standard `Math.Cos()` function, cosine LUT, and cosine LUT with linear interpolation were put into a loop and repeated 1E8 times. The results of the execution time and battery voltage drop measurements are expressed in percents with respect to the corresponding measurement results of the standard `Math.Cos()` function (i.e., we assume that execution time and voltage drop of `Math.Cos()` is equal to 100%). For the LUT-based cosine implementations with and without interpolation the execution time and voltage drop values are flat (see Figure 4).
The LUT without interpolation has the lowest energy consumption (about 5-6% of \(\text{Math.Cos}()\) energy consumption) and the best performance (about 4% of \(\text{Math.Cos}()\) execution time). The LUT with linear interpolation has worse energy consumption (about 12-25% of \(\text{Math.Cos}()\) energy consumption) and performance (about 9% of \(\text{Math.Cos}()\) execution time), but higher accuracy. However, there is a significant data memory usage overhead for the LUTs, which is a relevant concern for mobile devices. Therefore, here we have a Performance-Accuracy-Energy-Memory trade-off between standard \(\cos\) function and two LUT-based implementations of cosine. More details about this experiment can be found in [20].

### 6.2 Sparse Matrix Multiplication

Sparse matrices often appear in various scientific and engineering applications such as finite element computation. When sparse matrices are stored and processed on a computer, it is beneficial to use specialized algorithms and data structures that take advantage of the matrix sparsity. Standard matrix structures and algorithms are slow and consume large amounts of memory when applied to large sparse matrices. Sparse data is by nature easily compressed and this compression almost always results in less memory usage, higher processing speed and lower power consumption. An important problem for the engineers is to find a sparsity limit, from which the application of the matrix compression pays off in terms of memory usage, power consumption and multiplication time. Here we compress sparse matrices using “Storage-by-columns” algorithm [21], where a matrix is substituted with 3 vectors, which store the nonzero elements of the matrix and their position. Such specialized matrix data structure is used by a matrix multiplication algorithm. Both algorithms for uncompressed and compressed matrix multiplication were implemented in C#.

The experiments were performed on a Compaq iPAQ H3900 (Pocket PC platform, Intel PXA250 400 MHz CPU, 32 MB RAM, Windows CE 3.0 OS) with 100x100 randomly generated matrices for both uncompressed and compressed sparse matrices. Multiplication operation for both cases was performed 1500 times. The results of the experiments are presented in Figure 5.

From the results, we can see that compression is efficient when sparsity of matrices is above 67%, multiplication time is lower then sparsity is above 63% and power consumption is lower then sparsity is above 63%.

![Figure 5. Sparse matrix multiplication results: a) data memory usage, b) multiplication time and c) energy consumption expressed via voltage drop](image)

The experiments were performed on a Compaq iPAQ H3900 (Pocket PC platform, Intel PXA250 400 MHz CPU, 32 MB RAM, Windows CE 3.0 OS) with 100x100 randomly generated matrices for both uncompressed and compressed sparse matrices. Multiplication operation for both cases was performed 1500 times. The results of the experiments are presented in Figure 5.

From the results, we can see that compression is efficient when sparsity of matrices is above 67%, multiplication time is lower then sparsity is above 63% and power consumption is lower then sparsity is above 63%.
6.3 FFT specialization

FFT (Fast Fourier Transform) is widely used in various DSP applications and image processing. Here we specialise a FFT algorithm [22] with respect to the number of points (data block size). FFT loops were unrolled and calls to the performance-costly sine and cosine functions were replaced with the early pre-computed function values. The annotations for loop unrolling and value insertions were coded in a metaprogram using a metalanguage, and the generation of the specialized FFT algorithm (in C#) is performed automatically. The experiments were performed on a ASUS P750 (Pocket PC platform, Intel PXA270 520 MHz CPU, 64 MB RAM, Windows Mobile 6 Professional OS).

The results for an original FFT algorithm and a FFT algorithm specialized with respect to the data block size using program specialization technique are presented in Figure 6. Each FFT operation was repeated 10E6 times. We can see that FFT specialization can improve performance and power characteristics by 60% (see Figure 6, b and c). However, the specialised FFT uses much more program memory than the original version of the FFT algorithm. While the size of the executable file of the original FFT algorithm remains constant, the size of the executable file of the specialized FFT algorithm grows exponentially (see Figure 6, a; note that plot axes are logarithmic).

![Figure 6. FFT specialization results: a) program memory usage, b) time and c) energy consumption expressed via voltage drop](image)

6.4 Evaluation of Results

The matching between different variants of the DSP algorithms and a given set of prescribed constraints (e.g., performance, energy consumption, accuracy, program and data memory and various trade-offs amongst the constraints) can help an embedded software developer to choose the most suitable implementation for his design task. E.g., for sparse matrix multiplication task, the experimentally established trade-off between using uncompressed and compressed matrices is about 63-67% (see Figure 5).

A specialized program is almost always more efficient (in terms of execution time or energy consumption) than the general program, but the general program tends to be easier to write, debug and maintain; and it does not need to be rewritten for each new case. On the other hand, when the size of the specialization problem is large, code explosion can occur (see Figure 6, a). For example, this situation can occur when a loop needs to be unrolled and the number of iterations is high. It may degrade the execution time of the specialized program, because of the instruction cache misses.

In case of data specialization, where calculation-intensive computations are substituted with memory access operations, the performance and energy consumption values usually make only a small fraction of the
values shown by the calculation-intensive function (see Figure 4). However, the primary concerns here are low accuracy, which can be improved 1) by using some additional computation such as linear or quadratic interpolation of function values, or 2) by increasing the size of a data look-up table. The latter approach is however limited by small random access memory (RAM) and cache sizes on mobile devices.

7 Conclusions

We have presented the Algorithm-Architecture Matching framework of the methodology for managing embedded DSP software development trade-offs (speed, energy, accuracy, memory), which uses feature diagrams for describing trade-off models at a high level of abstraction.

The proposed framework considers the analysis of the problem and solution domains at the program construction time. The problem domain identifies the influential factors, the awareness factors, the type of the problem, single criteria, and trade-offs of the criteria and relationships. Analysis of the problem domain consists of identification of the influential and awareness factors, identification of the performance, accuracy, memory, and energy criteria and related trade-offs. Analysis of the solution domain consists of identification of the calculation-intensive algorithms for embedded applications, selection of representative algorithms for embedded applications, and specialization of the representative algorithms for different criteria. For specialization, we use two methods: algorithm specialization for known input values and data structure specialization. Our methodology allows the developer to represent embedded program design trade-offs in energy-speed and energy-accuracy dimensions, and to select a program implementation that best matches system requirements and/or constraints.

Future work will focus on the extension of the proposed methodology for other types of algorithms such as communication-intensive algorithms.

References


