Complex Instruction and Software Library Mapping for Embedded Software Using Symbolic Algebra

Armita Peymandoust
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Tajana Simunic
HP Labs & Stanford University
1501 Page Mill Rd., MS 3U-4
Palo Alto, CA 94304

Giovanni De Micheli
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

{armita, tajana, nanni}@stanford.edu

ABSTRACT

With growing demand for embedded multimedia applications, time to market of embedded software has become a crucial issue. As a result, embedded software designers often use libraries that have been pre-optimized for a given processor to achieve higher code quality. Unfortunately, current software design methodology often leaves high-level arithmetic optimizations and the use of complex library elements up to the designers' ingenuity. In this paper, we present a tool flow and a methodology, SymSoft, that automates the use of complex processor instructions and pre-optimized software library routines using symbolic algebraic techniques. We use SymSoft to optimize a set of examples for the SmartBadgeIV [2] portable embedded system running embedded Linux operating system. The results of these optimizations show that by using SymSoft we can map the critical basic blocks of the examples to the StrongARM SA-1110™ instruction set much more efficiently than the commercial StrongARM compiler. SymSoft is also used to map critical code sections to commercially available software libraries with complex mathematical elements such as `exp` or the `IDCT` routine. Our measurements show that even higher performance improvements and energy savings are achieved by using these library elements. For example, the final optimized MP3 audio decoder runs four times faster than real-time playback while consuming four times less energy. Since the decoder executes faster than real-time playback, additional energy savings are now possible by using processor frequency and voltage scaling.

KEYWORDS

Embedded systems, Performance optimization, Power minimization, Symbolic algebra, Software optimization, Automated software library mapping.

1. INTRODUCTION

The principal requirement in system-level design of embedded multimedia appliances is to reduce cost and time to market. In embedded system design environment, the degrees of freedom in software design are often much higher than the freedom available in hardware design. As a result, the primary requirement for embedded system-level design methodology is to effectively facilitate code performance and energy consumption optimization. Automating as many steps in the design of software from algorithmic-level specification is necessary to meet time to market requirements. Unfortunately, current available compilers and software optimization tools cannot meet all designers' needs. Typically, software engineers start with algorithmic level C code, often developed by standards groups, and manually optimize it to execute on the given hardware platform such that power and performance constraints are satisfied. Needless to say, this conversion is a time-consuming and often error-prone task, which introduces undesired delay in the overall development process.

Pre-optimized software libraries and complex processor instructions are often available for embedded system design. But, most compilers are unable to use these complex assembly instructions and pre-optimized library elements efficiently while compiling C code for embedded processors. Therefore, software engineers need to design key routines in assembly [1] or manually map a code section to a pre-optimized library element. Example of complex instructions available range from the simple multiply-accumulate (MAC) to a library of more complex instructions, such as those developed by Tensilica tools [6]. There are several pre-optimized software libraries commercially available. Intel recently released a library targeted at multimedia developers for StrongARM SA-1110™ embedded processor [14], and TI has a similar library for TI'54x DSP [15]. Embedded operating systems typically provide a choice from a number of mathematical and other libraries [16][17]. When a set of pre-optimized libraries is available,

the designer has to choose the elements that perform best for a given section of code. For example, consider a section of code that calls the `log` function. The library used in mapping consists of four different `log` implementations: double, float, fixed point using simple bit manipulation algorithm [18], and fixed point using polynomial expansion. Each implementation has a different accuracy, performance, and energy trade-off. A designer would need to estimate which of the four implementations would work best, test the hypothesis, and iterate until the best result is found. Designers are faced with an even more complex problem when attempting to map a software implementation of IDCT already present in MP3 standards code into an embedded software library. There are many ways to implement IDCT on a given processor, and it may be difficult for a designer to determine which library element is most appropriate.

Our objective is to improve the quality of compiled code for embedded systems and facilitate the software design process. In this paper, we propose a new methodology based on symbolic manipulation of polynomials and energy profiling which reduces manual intervention. Our methodology automates the process of identifying the code sections that benefit from complex library mapping, and then performs the mapping using symbolic techniques. We apply a set of techniques previously used in algorithmic-level hardware synthesis [28][29] and combine them with energy profiling, floating-point to fixed-point data conversion, and polynomial approximation to achieve a new embedded software optimization methodology. The combination of these tools and standard compiler optimization techniques allow novel automatic code transformations.

Example 1. As a ***motivating example***, let us look at the following code segment:

```
for i=1..3
    y = y + cos(i*x);
```

Using standard loop unrolling, the given code is transformed into the following:

```
y = cos(x) + cos(2*x) + cos(3*x);
```

Now assume that for a given application cos(x) can be approximated into a Taylor series with three terms without noticeable degradation on the output. Many multimedia applications tolerate computational inaccuracy well, as long as the resulting effects (e.g. audio, video degradation) are limited. Therefore, y can be approximated as a polynomial:

$$y = 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 + 1 - \frac{1}{2}2^2x^2 + \frac{1}{24}2^4x^4 + 1 - \frac{1}{2}3^2x^2 + \frac{1}{24}3^4x^4$$

This polynomial can be further simplified using the expand routine in symbolic algebra:

$$y = 3 - 7x^2 + \frac{49}{12}x^4$$

Assuming that the embedded processor used to execute this code has a multiply accumulate (MAC) instruction, another symbolic routine called the Horner transform can be used on y:

$$y = 3 + (-7 + \frac{49}{12}x^2)x^2$$

The new equation can be mapped to one multiply instruction and two multiply-accumulates. Obviously, this mapping is much more efficient than three calls to the cosine library function. Unfortunately, to our knowledge, there is no available software optimization tool that performs this simple optimization automatically. Thus, it would be up to designers to manually implement such optimizations. ■

This paper presents a tool-flow, called SymSoft, that performs algebraic manipulations such as the one shown in Example 1 simultaneous with automatic complex instruction and library mapping. First, a characterization function is derived for the pre-optimized library elements and complex assembly instructions. Then, the performance and energy critical code sections are identified using the energy profiler. If necessary, a tool such as Fridge [4] can be used to help transform floating-point data types into fixed-point. Next, complex nonlinear arithmetic functions in critical blocks are approximated as polynomials such that the final output is within the acceptable tolerance limits. Finally, symbolic algebra is used to map the polynomial representations of the critical basic blocks to the instruction set and library elements available automatically such that performance and power consumption are optimized.

The paper is organized as follows: Section 2 discusses previous work in software optimization for energy and performance. Section 3 describes the software and hardware platform and the measurement setup we are using in our experiments. Section 4 presents the SymSoft flow, and gives an overview of each of its steps and components. The results of SymSoft optimizations on several software examples for the SmartBadgeIV system are presented in Section 5. SymSoft lowers the execution time and energy consumption of these examples by using a pre-optimized software library available for StrongARM and the StrongARM instruction set. Finally, Section 6 summarizes the contributions of this work.

2. RELATED WORK

Designers have used software performance and size optimization methodologies and tools of for many years. Generally, compilers are used to translate a high-level specification into optimized machine code for a target processor. Several researchers have worked on optimizing compilers in last few years [7]. Prototype research compilers have shown impressive results [8]. Most optimizing compilers target high-

performance and/or general-purpose computers. Relatively little effort has been dedicated to create powerful optimizing compilers for embedded processors. Even though several researchers are studying automatic code retargeting techniques for embedded processors [9][10], currently, most embedded processors (or DSPs) are programmed directly by expert programmers and code optimization is mostly based on human intuition and skill. In addition, due to recent growth in market demand for portable devices, optimization of software for power consumption is gaining importance. As a result, one of the primary requirements for system-level design methodology of embedded devices is to effectively support code performance and energy consumption optimization.

Several optimization techniques for lowering energy consumption have been presented in the past. Numerous methodologies for optimizing memory accesses have been introduced that combine automated and manual software optimizations [11]. Tiwari et al [12][13] used instruction-level energy models to develop compiler-driven energy optimizations at assembly level such as instruction reordering, reduction of memory operands, operand swapping in the Booth multiplier, efficient usage of memory banks, and a series of processor specific optimizations. Several other optimizations such as energy efficient register labeling during the compile phase [19], procedure inlining and loop unrolling [20] as well as instruction scheduling [21] have also been suggested. In addition, various compiler optimizations have been applied concurrently and the resulting energy consumption was evaluated via simulation [22]. All of these techniques focus on automated instruction-level optimizations driven by the compiler. Unfortunately, current available compilers have limited capabilities. Specifically, they are incapable of handling arithmetic optimizations such as shown Example 1.

In the previous work [34], MP3 audio decoder software available from the standards body [3] was manually optimized for the SmartBadge embedded system [2]. This work required the designer to first

implement a fixed-point library and then to replace all floating-point operations with fixed point. Then, the designer needed to fully understand the details of the SmartBadge's design, so that the critical arithmetic operations can be manually optimized with inline assembly code. The manual optimization process lasted several days. This experience is similar to the typical industrial settings, where the software needs to be ported and optimized to the newer versions of hardware.

Our proposed methodology and tool flow uses profiling to identify the code sections that would benefit most from algebraic optimizations, and then automatically performs the optimizations using symbolic techniques. Such symbolic techniques have been previously used in algorithmic level synthesis of data intensive circuits [28][29]. SymSoft uses the same principles previously used for high-level component mapping of hardware and applies them to the software optimization problem. The outcome of our mapping algorithm is software that runs faster and consumes less energy on the SmartBadgeIV [2] embedded system while compared to the output of the commercial StrongARM compiler.

3. EXPERIMENTAL SETUP

We used SymSoft to optimize a set of examples on the SmartBadgeIV [2]. SmartBadgeIV, as shown in Figure 1, is an embedded system powered by batteries through a DC-DC converter. It consists of StrongARM SA-1110™ processor with StrongARM SA-1111™ companion chip, audio CODEC with microphone and speakers, Lucent's WLAN card, sensors and three types of memory: SRAM, SDRAM and FLASH. SmartBadgeIV currently runs eCos [16] and an embedded version of Linux operating system [17]. In this work we use Linux OS since the software library available to us is implemented for Linux. SmartBadgeIV 's Linux has the main parts of the OS, including a small file system, residing in SRAM. The larger file system is remotely mounted from the server via the WLAN card. In our

experiments, the program files and their input data reside in the directory structure on the server and are accessed via the wireless link on the SmartBadgeIV.

All of the measurements were performed using National Instruments Data Acquisition (DAQ) measurement system capable of 1.25 Msamples/second. We found a sampling speed of 1 kHz to be sufficient. In our setup, we used one PC to measure system, processor, and WLAN currents via the DAQ interface, and the other PC to act as a remote file server for the SmartBadge IV. The execution time of the code was measured by accessing StrongARM SA-1110™ on-board timer.

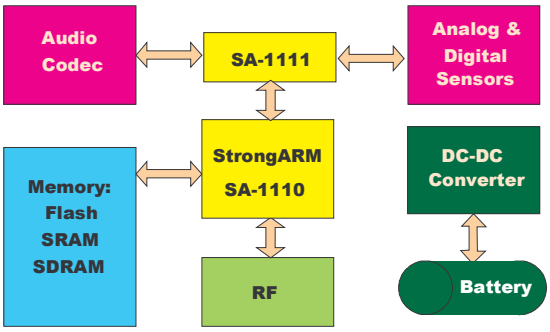

Figure 1. SmartBadgeIV Architecture

4. SYMSOFT METHODOLOGY AND TOOL FLOW

Ideally, the software designer would write an algorithmic-level description of the software and have a compiler-like tool optimize it for the given hardware platform. However, optimum implementation of calculation intensive routines for the particular hardware design is not possible with traditional compiler optimizations alone. Commonly, the designer does most of such optimizations by hand. Automating even a portion of this process can save much design time.

Here we present a methodology and a tool flow, SymSoft, which facilitates embedded system software optimization with automating library and complex instruction mapping for a given embedded

processor. Figure 2 shows the SymSoft flow. The mapping methodology consists of three main steps: library characterization, target code identification, and mapping.

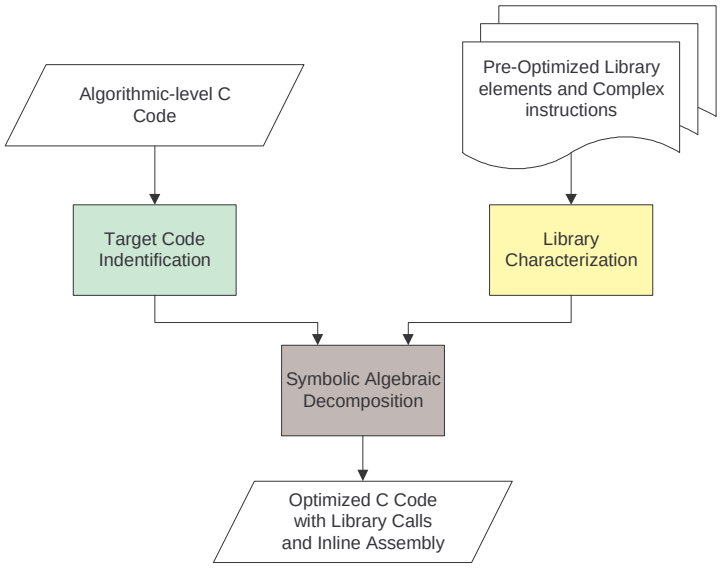

Figure 2. SymSoft Tool Flow

The first step is to characterize the library elements. The characterization not only includes performance and energy consumption of the complex element for a given hardware architecture, but also the expected input and output format, accuracy and a polynomial representation.

The next step identifies the target code for optimization. In this step, an initial check is performed to see whether data representation used in the algorithmic-level C code matches the target hardware. Most embedded processors support only fixed point computation, but many multimedia algorithms utilize floating-point operations. The profiler, described in Section 4.2.2, detects if data representation is an issue within several seconds. Then, if needed, floating-point operations are replaced with fixed-point operations with the help of a floating-point to fixed-point converting tool [4][5]. The profiler also reports the performance and energy critical functions of the code. The polynomial representations of the arithmetic sections of the critical routines are calculated with help of traditional compiler techniques

such as loop unrolling. When necessary, polynomial approximation techniques are used. Accuracy is checked at the end of the target code identification step to make sure that the code still meets the specifications, as some rounding occurs both during the data representation conversion and during the polynomial formulation.

Finally, the target code represented by polynomials is automatically mapped into the library elements and complex processor instructions. Our key contribution in SymSoft is a new method to map critical code segments into pre-optimized software library elements and complex assembly instructions using symbolic polynomial manipulation. The mapping process selects the solution that offers best performance with sufficient accuracy. Since our methodology is compliant with other software optimization techniques, additional benefits are gained by combining it with traditional complier optimization algorithms, such as constant and variable propagation, dead code elimination, and loop unrolling. The next sections describe each part of the SymSoft flow in detail.

4.1 Library Characterization

The target library consists of pre-optimized software libraries and complex arithmetic instructions available for the target processor. Complex arithmetic instructions vary from the simple multiply-accumulate (MAC) to more complex instructions, such as those developed by Tensilica tools [6]. Pre-optimized software libraries include traditional embedded system libraries, such as the IEEE floating-point mathematical library for Linux operating system [17], commercial libraries available for the particular processor, such as Intel's integrated performance primitives (IPP) [14], and a set of in-house pre-optimized routines. Table 1 shows a sample of elements of the IPP library. Library characterization is done on element-by-element basis. Each element is labeled with the type of inputs and outputs, performance, accuracy, energy consumption, and finally its polynomial representation.

Table 1. Sample of IPP Library Elements

Library Elements	Description
Exp	Exponentiation
Ln	Natural logarithm
DotProd	Vector dot (inner) product
Mean	Vector arithmetic mean
FIR	Finite impulse response filter
IIR	Infinite impulse response filter
Conv	Convolution
WinHamming	Hamming window
FFT	Fast Fourier transforms
HuffmanDecode	Decodes Huffman symbols
SubBandSynthesis	Stage two of hybrid synthesis filter bank
IMDCT	Inverse modified discrete cosine transform

The format of library element inputs and outputs is determined from the library include files or documentation available with the library element. Techniques discussed in Section 4.2.3 can be used to extract the polynomial representations from the source code if the code is available. Otherwise, either the distributor needs to provide the equivalent polynomial representation or it might be obtained from the documentation. Important part of library characterization is the determination of accuracy, performance and energy consumption. This information is used to guide the selection process when more than one library element has same functionality. Most embedded systems have OS timers that can be used for fine-granularity performance measurements on hardware. However, often there is not an easy way to measure processor and memory power consumption. Alternatively, a cycle-accurate energy consumption simulator [24] easily provides energy and performance estimates of library elements. Note that the library characterization step is yet to be automated.

Examples of two characterized complex library elements, SubBand Synthesis and IMDCT, are shown in Table 2. The library has three different versions of each library element: the open-source floating point version from the MP3 standards library [3], fixed-point in-house pre-optimized routine, and a

version from Intel's integrated performance primitive (IPP) library for StrongARM SA-1110™ processor [14]. For each library element, we have measured its performance on the SmartBadgeIV hardware. All entries in Table 2 are represented using polynomials. Since polynomials for complex library elements can be quite large, we show only a critical portion of IMDCT polynomial in Equation 1. Equation 1 shows how *n/2* windowed samples, y_k, are transformed into *n* x_i samples. Note that this is just a first order polynomial, since $\cos(\frac{\pi}{2n}(2i+1+\frac{n}{2})(2k+1))$ can be calculated in advance for all *i*, *k* and *n*.

$$x_i = \sum_{k=0}^{\frac{n}{2}-1} y_k \cos(\frac{\pi}{2n}(2i+1+\frac{n}{2})(2k+1)) \qquad \textbf{(1)}$$

Table 2. Characterized Complex Library Elements

Library Element	Execution time	Input Type
float SubBandSyn	0.95	64 bit float
fixed SubBandSyn	0.01	32 bit fixed
IPP SubBandSyn	0.002	32 bit fixed
float IMDCT	0.39	64 bit float
fixed IMDCT	0.014	32 bit fixed
IPP IMDCT	0.0002	32 bit fixed

4.2 Target Code Identification

The input to the target code identification step is the algorithmic-level C code of the embedded software. The output of this step is a set of polynomial representations of the critical code segments that would benefit most from mapping to complex instruction and pre-optimized library elements. Target code identification consists of three stages as shown in Figure 3. First, the profiler checks to see whether floating point operations are on the critical path. If needed the floating-point operations are transformed

into fixed-point operations by data representation conversion. Next, the energy and performance critical procedures are identified. This step can be done either with simulation using the energy profiler [24], or by profiling directly on the hardware. Finally, when the power and performance critical procedures are identified, they are formulated as polynomials suitable for mapping into library elements. In the next sections, we will take a closer look at each stage of the target code identification step.

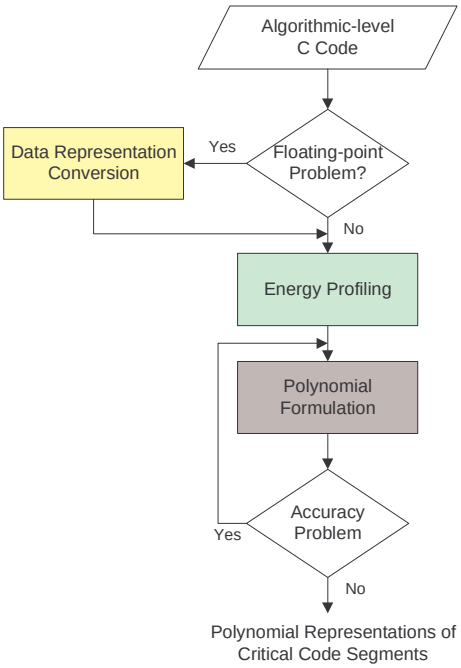

Figure 3. Target Code Identification

4.2.1 Data Representation Conversion

Signal processing algorithms are generally developed using ANSI-C with IEEE floating-point data types. However, these algorithms are often implemented in embedded systems using fixed-point data types in order to meet the power, cost, and performance requirements. In this stage, it is checked whether floating-point operations are capturing most of the execution time and power consumption of the algorithmic-level C code. In that case, floating-point operations are considered critical and they must be converted to fixed-point operations. Converting a floating-point algorithm to a fixed-point algorithm

is a time consuming and error prone task. Facilitating and semi-automating this conversion has been the target of many research projects [4][5]. Such tools use interpolative analysis or analytic techniques to convert floating-point operations into appropriate fixed-point operations while reducing the manual work and the number of simulations required. In our tool flow, we opt to use a tool like Fridge (a.k.a. CoCentric fixed-point designer) to automate this stage of optimization.

4.2.2 Energy Profiling

Code optimization requires extensive program execution analysis to identify performance and energy-critical bottlenecks and to provide feedback on the impact of code transformations. Profiling is typically used to relate performance to the source code for CPU and L1 cache [23]. Energy profiler enables easy identification of energy-critical procedures. It also facilitates analysis of code transformations' impact on the processor energy consumption, the memory hierarchy, and the system busses.

The profiler exploits a cycle-accurate energy consumption simulator [24] to relate the embedded system energy consumption and performance to the source code. Thus, it can be used for analysis (i.e., to find energy-critical sections of the code), and for validation (i.e., to assess the impact of each code optimization). The profiler architecture is shown in Figure 4.

Source code is compiled using a compiler for a target processor. The output of the compiler is the executable represented as assembly code and a map of locations of each procedure in the executable. The profiler of the cycle-accurate simulator periodically samples the simulation results (by user defined sampling interval) and maps the energy and performance to the function executed using information gathered at the compile time. Sampling is used to improve profiling speed while maintaining accuracy. Once the simulation is complete, the energy consumption and execution time of each function are displayed.

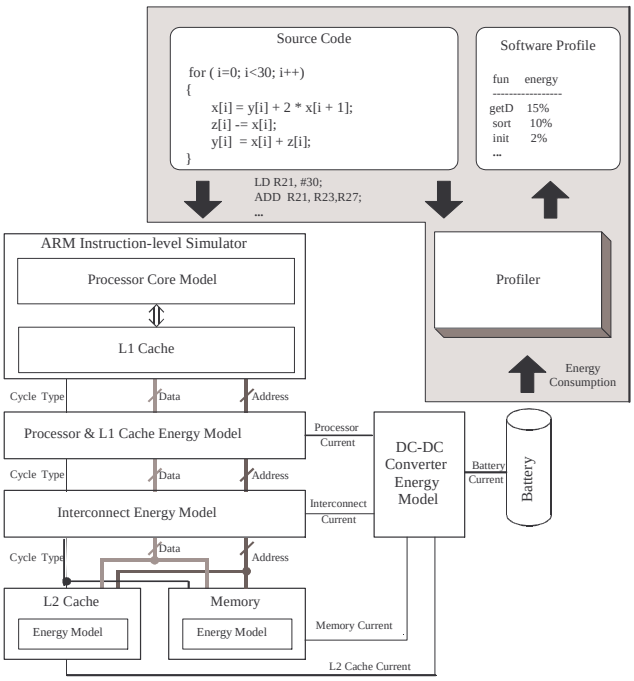

Figure 4. Profiler Architecture

With the profiler, SymSoft can obtain energy consumption breakdown by procedures in the source code and thus can quickly identify the sections of the source code whose optimization can provide the largest execution time and energy savings. In addition, with the cycle-accurate simulator that is at the heart of the profiler, SymSoft can get detailed information about performance and energy consumption of smaller subsections of code. The identified critical code segments are then passed as inputs to polynomial approximation and symbolic mapping tools that can optimally map the code section into complex library elements and assembly instructions in few minutes.

4.2.3 Polynomial Formulation

Our goal is to automatically map the critical code segments selected by the profiler into pre-optimized library elements or complex assembly instructions such that optimum execution time and power consumption are achieved. The symbolic mapping algorithm, described in Section 4.3, takes as input the

polynomial representations of the critical code segments and the polynomial equivalence of complex arithmetic assembly instructions and pre-optimized library elements. The polynomial formulation step prepares the first set of inputs required by the symbolic mapping algorithm by calculating the polynomial representations of the critical code segments. The second set of inputs is calculated in the library characterization step as described in Section 4.1.

The polynomial representation of a basic block can be directly extracted from the C code if the basic block calculates a polynomial function. If the basic block performs a series of bit manipulations or Boolean functions, interpolation-based algorithms [31][32] can be used to formulate the equivalent polynomial representation. When the basic block implements a transcendental function, we use an approximation, such as the Taylor or Chebyshev series expansion, as its polynomial. The chosen polynomial approximation has to be verified by simulation to ensure that the software constraints, such as audio quality, are satisfied. A good approximation can result in large performance and power improvements for multimedia applications, since these applications can tolerate a slight degradation in the output. For example, to verify the accuracy of the MP3 decoder we have used the compliance test provided by the MPEG standard where the range of RMS error between the samples defines the compliance level [25]. If the approximation is not sufficient to satisfy the accuracy constraints, the quality of approximation is changed and verified again through simulation.

The objective of this step is to formulate polynomials that cover as much of the source code as possible. Consecutively, the likelihood of finding a more complex library element that matches at least a portion of the formulated polynomial increases. This objective can be accomplished by using code transformation techniques such as loop unrolling, constant and variable propagation to form larger basic blocks.

4.3 Symbolic Mapping Algorithm

The symbolic mapping algorithm requires two sets of inputs: a set of polynomial representing the critical code segments and another set of polynomials representing the pre-optimized library elements and complex instructions. The former has been generated in the target code identification step and the latter is the output of the library characterization step. The goal of the symbolic mapping algorithm is to decompose the polynomial representations of the critical code segments (CCS) into the polynomial representations of the target library such that execution time and power consumption are minimized. The power consumption and execution time of each library element are provided to the mapping algorithm as constants by the library characterization step as described in Section 4.1. As opposed to tree covering based algorithms, in our algorithm, mapping is performed simultaneously with algebraic manipulations.

Symbolic computer algebra is a set of algorithms capable of algebraic manipulation of expressions containing undetermined values (symbols), such as variable `x` in `(x+1)*(x-1)`. Several commercial symbolic computer algebra softwares are available on the market; Maple [26] and Mathematica [27] are most widely used. The algebraic object to be symbolically manipulated is a set of multivariate polynomials that represent a critical basic block identified in the profiling step. Most interesting symbolic polynomial manipulations are based on Gröbner bases [30]. Gröbner bases also solve variable elimination in a set of polynomials and ideal membership problems, which is the core of the simplification modulo set of polynomials [30]. We use the following set of symbolic techniques: factorization, expansion, Horner transform, multivariate polynomial substitution, and variable elimination. We have described the complex underlying theory in the contest of hardware design

elsewhere [29][28]. In this section, we show the power of symbolic algebra by means of few of the routines applied to simple examples.

Example 2: ***Factor*** and ***expand*** are inverse operations. Consider using Maple to factor and expand the following polynomial:

```
› S := x^2*(x^14+x^15+1);
› P := expand(S);
        P := x^16+x^17+x^2
› factor(P);
        x^2*(x^14+x^15+1)
```
■

Example 3: ***Horner*** form of a polynomial is a nested normal form with minimal number of multiplications and additions. Any polynomial can be rewritten in Horner, or nested, form. An example of Horner form polynomial for multiple variables is shown below:

```
› S:= y^2*x+y*x^2+4*x*y+x^2+2*x;
› convert(S, 'horner', [x,y]);
        (2+(4+y)*y+(y+1)*x)*x
```
■

Example 4: ***Simplify*** implements substitution and variable elimination for multivariate polynomials:

```
› S:= x + x^3*y^2 –2*x*y^3;
› simplify(S, {p = x^2–2*y}, [x,y,p]);
        x+y^2*x*p
```
■

The core of the library-mapping algorithm is the simplification modulo set of polynomials (*simplify*) routine. The polynomial representations of critical code blocks are simplified modulo a subset of polynomials representing the library elements called the side relation set. Choosing the side relation set is a non-trivial and important task, especially since different side relation sets results in different solutions. In previous work [28], an algorithm was introduced to select the side relation set such that the

hardware implementation of a (portion of) data path with a given component library has minimal critical path delay. In this paper, we use the algorithm to optimize execution time of the critical code segments of software by mapping to pre-optimized library elements and complex assembly instructions. Since evaluating all subsets of the library is exponentially expensive, the library-mapping algorithm uses the branch-and-bound method with execution time and energy consumption as bounding functions to prune the search space. All previously described symbolic manipulations except *simplify* are used as guidelines in formulating different side relation sets to speed up the mapping algorithm.

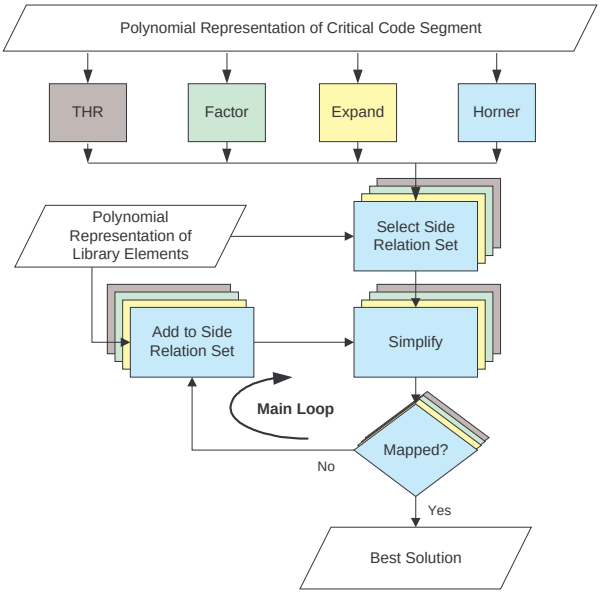

Figure 5. Overview of the Library Mapping Algorithm

Figure 5 gives an overview of the mapping algorithm. Inputs to the algorithm are the polynomial representations of the critical code segments (CCS) and the polynomial representations of the target library elements. Initially, tree-height reduction, factorization, expansion, and Horner-based transform are applied to the polynomial representation of the CCS resulting in several different polynomials representing the same code segment. Each of the different polynomial representations is used to select a side relation from the target library. These guidelines are used to increase the speed of finding the

desirable mapping. The polynomial representation of the CCS is simplified modulo the selected side relation sets in parallel. If the result of *simplify* matches a library element then the CCS is mapped. Otherwise, we need to continue to add to the side relation set until the CCS is fully mapped to our library. The iterative part of the algorithm, denoted in Figure 5 as *main loop*, is implemented using branch-and-bound algorithms.

Table 3 shows the pseudo-code of the library-mapping algorithm. Inputs to this algorithm are the polynomial representation of the critical code section (*CCS*) and the polynomial representations of the library elements (*L*). The bounding function is defined as the best execution time for *CCS* seen so far. The lower bound computed at each decision branch is the execution time of the library elements in the side relation set in view of data dependencies. If this lower bound is greater than the best execution time seen so far, the corresponding decision branch is pruned. Decision tree (*decision_tree*) implements the branch-and-bound algorithm. The algorithm starts by initializing the root of *decision_tree* to the polynomial representation of *CCS* and calculating an initial bound. The bounding variable is initialized to the execution time of calculating the *CCS* polynomial solely with add and multiply instructions, the lexicographical mapping (*LexMap*). Nodes are added to this tree in breadth-first manner. These nodes store the polynomial result of *simplify* of their parent node and the chosen side relation set. When a simplification result corresponds to a polynomial representation of a library element, a possible solution is found and the corresponding tree node is marked accordingly. If the execution time of the solution is less than previously encountered solutions, we set the bounding variable to the current value. In case the simplification result stored in a tree node does not correspond to any library elements, we apply the same steps to the new tree node until either a solution is found or the corresponding branch is pruned. Since *CCS* is a polynomial and add and multiply instructions are always available in our library, we are

guaranteed to have a solution. However, our mapping algorithm searches for a solution that best exploits the given software library.

Table 3. Pseudo Code of the Library Mapping Algorithm

```
function Decompose (exp_tree, boundVal, L) {
  // initialize the decision tree
  decision_tree ← tree (exp_tree)
  Depth ← 0
  Bound ← boundVal
  for all n ∈ decision_tree with depth == Depth  do{
     if Depth == 0
          choose sr ∈ L to preserve the exp_tree structure
      else for all sr ∈ L {
          result = simplify (n, sr);
          AddChild (n, result)  // make result a child of node n
          if result ∈ L   // solution is found
             Bound = Min(cost of node result, Bound);
      }
       if no more n ∈ decision_tree with depth == Depth
          Depth ← Depth + 1
  }
  return the best solution
end Decompose

procedure main (CCS,L)
   exp_tree [1 .. NoManipulations] = AllManipulations (CCS);
   for i = 1 to NoManipulations {
      boundVal[i]=LexMap(exp_tree[i]);
      solution[i] = Decompose(exp_tree[i],boundVal[i]) }
   return the best solution in solutions[i]
end main
```

The branch-and-bound algorithm in Table 3 is applicable to most practical problems and its runtime is in the matter of minutes. But, as for all branch-and-bound algorithms, the worst-case complexity remains exponential. The speed of this algorithm depends on the initial polynomial and the initial side relation set. Here, we use a set of library independent symbolic manipulations on the original *CCS* polynomial to help with the selection of initial side relation element. These manipulations improve the execution time without hampering the quality of the solution. First, we apply tree-height reduction,

factorization, expansion, and Horner-based transform to *CCS* in the *AllManipulations* function. As a result, we have several different polynomials (*exp_tree*) representing the same code section. Each of these representations can result in the desirable implementation based on the available library elements.

To select the initial member of side relation sets, we start with the primary inputs and cover the expression tree with the library elements. We choose all library elements that cover the primary inputs and a portion of the expression tree as initial elements of the different side relation sets used to simplify the root of the *decision_tree*. If the result of simplify is not a library element, we add more elements to the side relation set without further guidance from the expression tree and decompose the result. Note that in selecting the side relations from the library, all different permutations of the variables with the same data-type are considered. This algorithm is implemented in C with calls to Maple V for the symbolic manipulations.

Example 5: In order to demonstrate the power of our ***library mapping*** algorithm, consider a basic block implementing Equation 2:

$$d = \cos\left(\frac{\pi}{72}(2p + 1 + \frac{N}{2})(2m + 1)\right) \qquad \textbf{(2)}$$

Equation 2 is approximated using Pade approximation to the polynomials shown in Equation 3 in the previous step of the SymSoft flow as described in Section 4.2.3.

$$x = \frac{\pi}{72}(2p+1+\frac{N}{2})(2m+1)$$
$$d \cong \frac{1-\frac{3665}{7788}x^2+\frac{711}{25960}x^4-\frac{2923}{7850304}x^6}{1+\frac{229}{7788}x^2+\frac{1}{2360}x^4+\frac{127}{39251520}x^6} \qquad \textbf{(3)}$$

The simplification modulo set of polynomials routine can be used to map the numerator and denominator of Equation 3 to the available instruction set. Let dn be the numerator of Equation 3

with a, b, and c the constants of the polynomial. In addition, we define siderels as a subset of the available instructions with renamed variables. We have:

```
> dn:=1+a*x^2+b*x^4+c*x^6:
  siderels:={w=x^2, y=b+c*w, z=a+y*w}
> simplify(dn, siderels,[x,w,y,z]);
   1+z*w
```

Note that the first element of the side relation set (w=x^2) corresponds to the square or multiply instruction and the other two elements of the set (y=b+c*w, z=a+y*w) and the result of simplify (1+z*w) correspond to the MAC instruction. The side relation set can be any subset of the available instruction set with proper renaming of the variables. Different side relation sets result in finding other possible solutions for the specification. The above implies:

```
dn=1+a*x^2+b*x^4+c*x^6=1+z*w
  =1+(a+y*x^2)*x^2=1+(a+(b+c*x^2)*x^2)*x^2
```

Therefore, the numerator of Equation 3 can be mapped to one square and three MACs instructions. Assuming R1, R2, R3, R4, and R5 hold 1, a, b, c, and x, respectively, the resulting assembly code is:

```
MULT R6, R5, R5
MAC  R7, R3, R4, R6
MAC  R8, R2, R7, R6
MAC  R7, R1, R8, R6
```

In the MP3 decoder program, the basic block evaluating Equation 2 uses floating-point and takes 2384 cycles to run on the StrongARM SA-1110™ processor. The approximation represented in Equation 3 calculates x using floating-point and d using fixed-point arithmetic and nested MACs as

suggested by the symbolic optimization. This approximation executes in 1257 cycles. Thus, we have achieved an improvement of 47% for this simple example. ■

5. Results

We have tested the effectiveness of SymSoft using the experimental embedded system SmartBadgeIV and a wide range of code examples used in communication, digital signal processing, and streaming media. The SmartBadgeIV system and our experimental setup for hardware execution time and energy consumption measurement were described in Section 3.

The first six software examples are obtained from a DSP software benchmark suite [33]. The first two examples are software programs that perform common digital signal processing computations; discrete convolution and dot (inner) product. Convolution is the linear operator can compute the output of a linear time-invariant (LTI) system in response to an input sequence given the system impulse response sequence. The dot (inner) product of two vectors is the summation of the products of the two input sequences; i.e. $z = \sum_i x[i] \cdot y[i]$.

The next four examples are different digital filters used in digital signal processing and communication applications. The first filter is a finite impulse response (FIR) filter. The next two filters are biquad infinite impulse response (IIR) filters. A single IIR filter of arbitrary order is often decomposed into equivalent cascades of 2nd-order IIR sections known as biquads. Although the biquad cascade is analytically identical to the single filter of higher order, the biquad filter realization is more stable and less sensitive to quantization errors. The last filter is least-mean-square (LMS) FIR adaptive filter. The LMS filter is a time-varying linear system for which the filter coefficients are adjusted at each time step to minimize the error between the actual output and a given desired output.

Finally, the last example is a full MPEG Layer III (MP3) audio decoder implementation that streams MP3 encoded files from a server to a client (SmartBadgeIV).

Table 4. Results of SymSoft optimization on a set of examples

Examples	Execution time in microsecs		
	Original	SymSoft	improvement (%)
Convolution	667	627	6.01
Dot product	358	267	25.42
FIR filter	2418	1170	51.61
IIR filter (4 biquads)	5079	4355	14.25
IIR filter (1 biquad)	1396	1250	10.46
Least Mean Square	1200	1000	16.67
MP3 decoder	54700	14300	73.86

Table 4 summarizes the results of applying SymSoft tool flow to the set of examples discussed above. In each case, we start with the fixed-point implementation of the algorithm and use profiling to select the critical code sections. Optimizing a critical code section results in a noticeable improvement on any given example. Next, the critical code sections are automatically mapped to the instruction set available on the StrongARM SA-1110™ processor and Intel's integrated performance primitives (IPP) library for StrongARM SA-1110™ processor [14]. Table 4 shows the execution time of each example before and after the optimization with SymSoft. Note that the original execution time includes all optimizations that are possible with using the ARM compiler.

The improvements demonstrated in Table 4 indicate that by using SymSoft we can obtain significant execution time improvement for a range of applications over commercial compilers. The amount of improvement achieved is dependent on the number of critical blocks that are optimized and the library implementations available for the given block. Examples in Table 4 show improvements in the range of 6% to 73% with an average of 28% improvement.

In the next section, we will go through all the steps of the SymSoft flow using the MP3 decoder software as an example.

5.1 The MP3 Optimization Results

We start with an algorithmic level description of the MPEG Layer III (MP3) audio decoder obtained from the International Organization for Standardization (ISO) [3]. Our design goal is to accelerate the MP3 decoder and lower its energy consumption while keeping full compliance with the MPEG standard. The first step in decoding the MP3 stream is synchronizing the incoming bitstream and the decoder. Huffman decoding of the SubBand coefficients is performed before requantization. Stereo processing, if applicable, occurs before the inverse mapping which consists of an inverse modified discrete cosine transform (IMDCT) followed by a polyphase synthesis filterbank. During the optimization process, we used instructions available on the StrongARM SA-1110™ processor, a mathematical library available with Linux OS [17], Intel's integrated performance primitives (IPP) library for StrongARM SA-1110™ processor [14], and a library populated with in-house pre-optimized routines. The library elements ranged from simple mathematical functions such as `MAC` to as complex elements as `IMDCT` routine.

The SymSoft flow, as described in Section 4, consists of library characterization, target code identification, and the final library mapping step. The library characterization step uses hardware measurements for performance and simulations for energy consumption [24]. The polynomial representation is obtained either from the source code (Linux mathematical and in-house libraries), or from documentation (IPP library).

The target code identification consists of three important steps: data type conversion, code profiling, and formulating polynomials to be mapped. The first step is to check if floating-point data types are suitable for the given platform. Since SmartBadgeIV 's processor, StrongARM SA-1110™, can only

emulate the floating-point operations, there is a need for data representation transformation. The code was converted to use fixed-point arithmetic. It was verified through simulation that 27-bit precision fixed-point data-types are sufficient to meet the compliance test provided by MPEG standard [25]. Automating floating-point to fixed-point data type conversion has been targeted by the tool Fridge [4]. Profiling the original source code highlights the critical code segments. Table 5 shows the results of profiling original MP3 decoder software we obtained from the standards body. All profiling reported in Table 5 is using hardware measurements. The results are shown for one frame and represent only the most significant functions in terms of their performance impact. Next, we formulate equivalent polynomial representation of each of the critical functions shown in Table 5. We use polynomial approximations for the non-linear calculations in the critical basic blocks. Once more, we validate that these approximations satisfy the MPEG compliance test [25]. The output of the target code identification step is a set of polynomials representing the critical sections of the code.

Table 5. Profiling the Original MP3 Code

Function name	Execution time (s)	%
III_dequantize_sample	1.1754	45.33
SubBandSynthesis	0.9481	36.56
Inv_mdctL	0.3872	14.93
III_hybrid	0.0670	2.58
III_antialias	0.0131	0.51
III_stereo	0.0010	0.04
III_hufman_decode	0.0007	0.03
III_reorder	0.0005	0.02
Total for one frame	2.5931	100.00

In the first phase of optimization, the polynomial representations of the critical code sections of the first three function shown in Table 5 are mapped into the StrongARM assembly instructions by algorithm described in Section 4.3. It is important to note that StrongARM compiler was not capable of using the MAC instruction effectively. However, our symbolic algorithm was able to use this instruction

efficiently. Automatically generated inline assembly was inserted in the C code as the result of the decomposing algorithm. The results of optimizing critical functions of the MP3 code by SymSoft are compared with the original results from straightforward compilation in Table 6. The numbers reported in Table 6 are obtained using the cycle accurate energy simulator described in Section 4.2.2. As we can see, 12-70% improvement has been achieved using the SymSoft methodology. Such improvement was previously possible only thorough manual optimization with inline assembly. The automation introduced by SymSoft drastically reduces the embedded software optimization cycle.

Table 6. Comparison Between SymSoft Instruction Mapping and Commercial Compiler

	Execution time (#cycles)			Energy Consumption (mWhr)		
Function	**original**	**optimized**	**%imp**	**original**	**optimized**	**%imp**
MDCTCoeff	1454550	957051	34.2	1.051	0.922	12.2
FilterS	5263831	4196853	20.3	3.630	3.319	8.6
Power3/4	14135	5380	61.9	0.040	0.009	76.6
Dequant	650894	421976	35.2	0.940	0.747	20.5
SubBandSyn	155204	70633	54.5	1.015	0.306	69.8
MDCT	63583	31954	49.7	0.101	0.051	49.6

Next, we profile the MP3 decoder that results from this phase of optimization on the hardware and measure the execution time of each function while decoding one frame of the MP3 stream. The resulting performance profile is shown in Table 7. Although the execution time per frame is drastically reduced (by two orders of magnitude compared to Table 5), we can see that still almost 85% of the execution time is spent in the IMDCT and SubBand synthesis functions.

Table 7. MP3 Profile After First Phase of Optimization

Function name	Execution time (s)	%
Inv_mdctL	0.0144	49.54
SubBandSynthesis	0.0103	35.30
III_dequantize_sample	0.0013	4.33
III_stereo	0.0008	2.83
III_reorder	0.0007	2.28
III_antialias	0.0006	2.15
III_hufman_decode	0.0007	2.48
III_hybrid	0.0003	1.10
Total for one frame	0.0291	100.00

In the second phase of optimization, the code is mapped to Intel's IPP library using the SymSoft methodology. Here we find two primitives that match the two critical procedures shown in Table 7. The resulting performance profile is shown in Table 8. Our method automatically uses two of the IPP routines. While the new profile shows that SubBand synthesis still takes roughly 35% of the execution time for each frame, we see that MDCT is no longer a critical portion of the code. Notice that the execution of the IPP SubBand synthesis routine is one order of magnitude faster than the previous version and the total time for decoding one frame is reduced by a factor of 5.

Table 8. MP3 Profile After Second Phase of Optimization

Function name	Execution time (s)	%
ippsSynthPQMF_MP3_32s16s	0.00176	35.242
III_dequantize_sample	0.00124	24.79
III_stereo	0.00082	16.46
III_hufman_decode	0.00067	13.416
IppsMDCTInv_MP3_32s	0.00047	9.4113
III_get_scale_factors	3.4E-05	0.6808
Total time for one frame	0.00499	100.00

Table 9 summarizes the performance and the energy results of the overall optimization process we described in this section. All measurements are performed on the SmartBadgeIV while running at maximum processing speed and voltage. We start from the original source code obtained from the standards web site that runs roughly two orders of magnitude slower than real-time playback. The next two rows show the results of mapping only into Intel's IPP library; more specifically, we are able to automatically use IPP's SubBand Synthesis and IMDCT in the original code. However, the rest of the code remains intact and still operates on floating-point data. StrongARM SA-1110™ cannot perform floating-point operations natively. As a result, the execution time of the code is still far from real-time playback.

Table 9. Execution time and Energy of Different Versions of the MP3 Decoder

Code version	Execution time (s)	Improvement factor	Energy (mWhr)	Improvement factor
Original	503.92	1.0	509.6	1.0
Original + IPP SubBand	301.43	1.7	292.5	1.7
Original + IPP SubBand & IMDCT	211.27	2.4	199.1	2.6
SymSoft first phase (FPh) optimization	5.47	92.1	4.47	114.2
FPh + IPP SubBand	3.33	151.4	2.78	182.3
SymSoft final optimization (FPh + IPP SubBand & IMDCT)	1.43	352.4	1.17	435.2
IPP MP3 (Best possible)	0.41	1240.8	0.31	1626

The fourth row corresponds to the result of the first phase of optimization using SymSoft methodology (without using the Intel library). In this phase, the target libraries used in the mapping step consist of the assembly instructions available on the StrongARM and a set of in-house fixed-point routines. As shown, we have achieved an improvement of two orders of magnitude in both performance and energy for this mapping. The improvement is because of effective use of the MAC instruction

available on StrongARM and conversion of most floating-point operations to fixed point. Fixed-point accuracy is verified through simulation.

Additional saving of a factor of four is obtained by further optimizing the code and adding Intel's IPP library to the target libraries in the mapping step. The improvement of factor of four is solely due to automatic use of complex library elements that have been pre-optimized for the given processor. Full compliance to the standard of each version of MP3 code is ensured by checking the accuracy at each mapping step with MP3 compliance test [25]. Note that even larger energy savings are possible by using processor frequency and voltage scaling, since the final MP3 code optimized by SymSoft runs almost four times faster than real-time playback.

The last row in the table, IPP MP3, represents fully hand-optimized MP3 code for StrongARM available from Intel. The final optimized version by SymSoft is a factor of 3.5-3.7 times worse than the IPP MP3. The lower bound on execution time (IPP MP3) is achieved by full manual optimization, which is an error-prone and tedious task. Our methodology reduces the manual intervention of software designers in the optimization process and its results are still faster than real-time playback. Such improvements were previously only possible by skilled designers, familiar with the hardware and software, hand optimizing the code for a given embedded system platform.

As it can be observed from Table 9, the reported optimization space for the MP3 decoder spans over three orders of magnitude. The major contribution of this work is to provide a semi-automated optimization flow that closely approaches the lower bound of the optimization space within the limitations of polynomial representation for code sections. Our approach is particularly suitable for data intensive algorithms such as DSP and multi-media applications, since large portions of these software codes can be easily represented by polynomials.

6. CONCLUSION

The contribution of this paper is a tool flow, SymSoft, for energy and performance optimization of algorithmic level software code to execute on a given embedded processor. There are three main steps in our methodology: library characterization, target code identification, and library mapping. Library characterization step finds a polynomial to represent the functionality of each library element and associates a set of parameters such as execution time, energy consumption, and input/output type with each library element. In the target code optimization step, our tool uses execution time and energy profiling to automatically identify need of automated data representation conversion and the critical sections of the code that would benefit most from optimization. For transcendental arithmetic functions, approximation into a polynomial representation is needed in order to enable symbolic algebra techniques. Finally, the library-mapping step uses symbolic computer algebra to automatically decompose the polynomial representations of the critical code sections into a set of library elements available for the embedded processor.

We demonstrated application of our tool, SymSoft, to the optimization of several examples on the SmartBadgeIV [2] embedded system. Using SymSoft for source code optimization, we have been able to increase performance and energy consumption of these examples dramatically while satisfying the output accuracy requirements. These improvements are achieved by the use of pre-optimized software library functions, conversion of critical floating-point operations to fixed point, and reducing the number of memory accesses and instructions executed in critical code segments. The technique presented in this paper can be easily used in conjunction with other compiler optimization techniques [7].

7. ACKNOWLEDGMENTS

This research is supported by ARPA/MARCO Gigascale Research Center, HP Labs, and Synopsys Inc. We would like to thank all organizations for their support.

8. REFERENCES

[1] P. G. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens, “Embedded software in real-time signal processing systems: application and architecture trends”, Proc. IEEE, vol. 85, no. 3, pp. 419-435, Mar. 1997.

[2] G. Q. Maguire, M. Smith, and H. W. Peter Beadle, “SmartBadges: a wearable computer and communication system”, 6th International Workshop on Hardware/Software Codesign, Invited talk, 1998.

[3] “Coded representation of audio, picture, multimedia and hypermedia information”, ISO/IEC JTC/SC 29/WG 11, Part 3, May 1993.

[4] M. Willems, H. Keding, T. Grötket, and H. Meyr, “Fridge: An interactive Fixed-Point Code Generation Environment for HW/SW CoDesign”, Proceedings of Int. Conf. On Acoustics, Speech, and Signal Processing, 1997.

[5] G. Constantinides, P. Cheung, and W. Luk, "The Multiple Wordlength Paradigm", Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, March 2001.

[6] A. Wang, E. Killian, D. Maydan, and C. Rowen, “Hardware/Software Instruction Set Configurability for System-on-Chip Processors”, Design Automation Conference, pp. 184-190, 2001.

[7] S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997.

[8] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam, “Maximizing multiprocessor performance with the SUIF compiler”, IEEE Computer, vol. 29, no. 12, pp. 84-89, Dec. 1996.

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

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

[11] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vanduoppelle, Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design, Kluwer Academic Publishers, 1998.

[12] V. Tiwari, S. Malik, A. Wolfe, and M. Lee, “Instruction Level Power Analysis and Optimization of Software”, Journal of VLSI Signal Processing Systems, vol 13, no.2-3, pp.223-238, 1996.

[13] V. Tiwari, S. Malik, and A. Wolfe, “Power Analysis of Embedded Software: A First Step Towards Software Power Minimization”, IEEE Transactions on VLSI Systems, vol. 2, no.4, pp.437-445, December 1994.

[14] www.intel.com, ”Integrated Performance Primitives for the Intel StrongARM SA-1110 Microprocessor”, 2000.

[15] Texas Instruments, “TI’54x DSP Library”, 2000.

[16] Cygnus Solutions, “eCos™ Reference Manual”, 1999.

[17] RedHat, “Linux-arm math library reference manual”.

[18] J. Crenshaw, *Math Toolkit for Real-Time Programming*, CMP Books, Kansas, 2000.

[19] H. Mehta, R.M. Owens, M.J. Irvin, R. Chen, and D. Ghosh, “Techniques for Low Energy Software”, International Symposium on Low Power Electronics and Design, pp. 72-75, 1997.

[20] Y. Li and J. Henkel, “A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems”, Design Automation Conference, pp.188-193, 1998.

[21] H. Tomyiama, H., T. Ishihara, A. Inoue, and H. Yasuura, “Instruction scheduling for power reduction in processor-based system design”, Design, Automation and Test in Europe, pp. 23-26, Feb. 1998.

[22] M. Kandemir, N. Vijaykrishnan, M. Irwin, and W. Ye, “Influence of Compiler Optimizations on System Power”, The 27th International Symposium on Computer Architecture, pp.35-41, 2000.

[23] Advanced RISC Machines Ltd (ARM), ARM Software Development Toolkit Version 2.11, 1996.

[24] T. Simunic, L. Benini, and G. De Micheli, “Energy-Efficient Design of Battery-Powered Embedded Systems”, Special Issue of IEEE Transactions on VLSI, pp. 18-28, May 2001.

[25] ISO/IEC JTC 1/SC 29/WG 11 13818-4, “Information Technology, Generic Coding of Moving Pictures and Associated Audio: Conformance”, International Organization for Standardization, 1996.

[26] Maple, Waterloo Maple Inc., www.maplesoft.com, 1988.

[27] Mathematica, Wolfram Research Inc., www.wri.com, 1987.

[28] A. Peymandoust and G. De Micheli, “Symbolic Algebra and Timing Driven Data-flow Synthesis”, Proceedings of the International Conference on Computer Aided Design, 2001.

[29] A. Peymandoust and G. De Micheli, “Using Symbolic Algebra in Algorithmic Level DSP Synthesis”, *Proceedings of the Design Automation Conference*, pp. 277-282, 2001.

[30] T. Becker and V. Weispfenning, Gröbner Bases, Springer-Verlag, New York, NY, 1993.

[31] J. Smith and G. De Micheli, “Polynomial Methods for Component Matching and Verification”, Proceedings of the International Conference on Computer Aided Design, 1998.

[32] J. Smith and G. De Micheli, “Polynomial circuit models for component matching in high-level synthesis”, *IEEE Trans. on VLSI*, Vol. 9, Issue 6, Dec. 2001, pp. 783 –800.

[33] V. Zivojnovic, J. Martinez, C. Schläger and H. Meyr, “DSPstone: A DSP-Oriented Benchmarking Methodology”, Proceedings of the International Conference on Signal Processing Applications and Technology, Dallas, TX, 1994.

[34] T. Simunic, L. Benini, G. De Micheli, and M.Hans, “Source Code Optimization and Profiling of Energy Consumption in Embedded Systems”, Proceedings of the International Symposium on Systems Synthesis, September 2000, pp. 193–198.

Response to the Associate Editor

Only a few minor changes were requested. I will check them myself when you resubmit the new version.

Response to Review Number 1

Most of the concerns of the original paper have been taken into account. A few comments still remain:

- the fact that some transformations are semi-automatic or even manual is mentioned in the response to the reviewers comments, but the same clarity is missing in the paper itself. I would expect such a clarification, for example, at the end of section 4.1.

A sentence has been added to the second paragraph of Section 4.1 to address this concern.

- the request for more experiments has not been taken into account as expected. In their response, the authors mention 6 new experiments. However, these "experiments" are just additional results from the MPEG example and not really referring to the symbolic algebra mapping. I would have expected results from a different application, showing the power of symbolic algebra mapping

The new experiments are not additional results from the MPEG example. As described in the second paragraph of Section 5, they are from a DSP benchmark suite (DSPstone). A reference has been added to avoid this confusion.

- the English could still be improved by adding some articles here and there (for example, the 4th line in section 4.1 should read

.... include traditional software librarIES, such as THE IEEE floating point .....

Same for 4th line of 5.1, add THE in front of MP3

Corrected.

- the 6th row of table 9 should explicitly include "IMDCT" in column 1, since the use of that function seems to be the main change, compared to row 5.

Added.

Response to Review Number 3

I think this is a largely improved version of the paper. However, I would recommend that more polishing be done.

- In the Introduction you say "Our methodology automates the process of identifying the code sections that benefit from complex library mapping", but in fact this process seems only semi-automated. I would clarify that statement.

The identification and mapping processes are automated as mentioned in the response to the previous version of the paper. Therefore, the sentence was not changed.

- In the related work section you say "In the previous work, MP3 audio...".

Which previous work? Is this documented? Was it part of your work?

A reference was added for clarification.

- How does your methodology compare to [9] and [10]?

Previous works use traditional graph covering methods for code generation. Our methodology uses algorithms from symbolic algebra that are capable of mapping and algebraic manipulations simultaneously. Graph covering techniques do not have knowledge of laws of algebra.

- The introductory text of section 4 can be shortened considerably. One paragraph should be enough to introduce the section, and another to describe the flow in general (e.g. three lines per step); this could avoid some redundancy with the later detailed sub-sections.

This section is summarizing the flow for readers who may not be interested in all the details.

- The result of the optimization depends on the characterization of the library elements. What kinds of input sequences are used for profiling?

We have used the same input to the MP3 decoder for all profiling and characterization purposes.

- The architecture of the profiler detailed in Figure 4 is not explained.

Tajana can you take care of this, or should we refer him to the publication?

- It's clear that you need to profile the library elements for performance and energy. But why do you need to profile the source code during the code identification process? Why don't you simply apply your polynomial formulation everywhere in the code and then optimize? Or is the information gathered by the profiler used in any way by the mapping process?

We use the profiling information to select sections of the code that benefit most from further optimization. In most cases, only few sections of the code benefit from the mapping process and profiling helps select these sections.

- At the end of section 4.2.3, you say that you look for "large" polynomials. What exactly is a "large" polynomial? Does it mean that it covers more of the source code? Why does a large polynomial increase the likelihood of finding a more complex library element?

The sentence has been reworded and clarified.

- In Section 4.3 you say that the mapping algorithm introduced in [28] could find implementations with the minimal critical path delay. Is there a similar property in the new context? Why is this also appropriate for software? Could optimizing a different metric provide better results in your case?

Yes, the similar property is execution time. The sentence following the one quoted here has been reworded for clarification. Software execution time is directly proportional to the energy consumed by system executing the given software. Other metrics could also be used in the branch and bound algorithm. However, for our example system, execution time of the software is the only controllable parameter.

*- On page 19 you say: "The goal of the symbolic algebra mapping…". I would move this sentence and the one that follows to the beginning of section 4.3 so that it is immediately clear what the goal of this step is. These two sentences also make it clear what the inputs and the outputs are, so that the current first paragraph of the section can be removed. On page 19 you are actually discussing the *conditioning* of the input to the procedure.*

The two sentences have been moved.

- On page 21-22, starting from "The speed of this algorithm…". Here you discuss again the operations that you apply to the input that you discussed on page 19-20. I would make only one discussion of this on page 19-20, perhaps with more details (see below).

- *Speaking about the initial operations applied to the input. Could you hint at how the different operations influence the choice of the side relation set?*

This has been explained in the paragraph before Example 5.

- *Why is your algorithm breadth-first and not depth-first? Is one better than the other?*

It is more likely that the best solution uses few number of library elements to calculate a critical basic block. Therefore, breadth first would potentially find the best solution faster.

- *On page 23, "In order to comply with Maple terminology...". You've been already using the terms "simplify" and "side relations" many times, it's odd that you say it only now that you call them that way.*

Sentence was removed.

- *How do you get the side relations in the example on page 23? Why isn't, for example, y = x^2 included in the set? (since you say you consider all permutations of the variables...).*

We have shown an instance of what siderel can be. In another instance, siderel would include y=x^2 as well.

- *In the example at the top of page 24, how much of the improvement is due to the use of fixed-point arithmetic, and how much is due to a smarter mapping?*

I don't have this info!

- *Section 5, the beginning: you don't test the "efficiency" of SymSoft, but rather its "effectiveness", or alternatively the "efficiency of the resulting code".*

Corrected.

- *Page 25: the audio decoder "steams" the encoded file. Perhaps it's "streams"?*

Corrected.

- I can't find a match between the data on the MP3 decoder in Table 4 and the data on any version of the MP3 decoder in Table 9. Have you used different input streams?

Yes.

- Is the data reported in Table 5 before or after the conversion from floating-point to fixed-point? Is it before of after the approximation to polynomial? I ask because the fixed-point conversion comes earlier in the flow, but you say that you profile the "original" code.

The results are from the original floating point software.

- What is the relationship between the entries in Table 5 and the entries in Table 6?

The first sentence of the paragraph following Table 5 was changed to clarify the relation.

- Similar to before, it is unclear to me whether the entry labeled "Original" in Table 9 is before or after the conversion to fixed-point and/or the conversion to polynomial representation. In other words, suppose you take the code from the standard body, perform the fixed-point conversion, then the approximation to polynomial representation, and finally compile with the standard ARM compiler. What do you get? How does it compare with your mapping algorithm? This comparison would isolate the contribution of the mapping alone. This might be exactly what you have done, but I don't find it very clear from the text.

The original line the code obtained from the standards web site. That was added to the text for clarification. The number he is asking for should be in Tajana's previous paper.

- It would be nice in the conclusions to have your perspective on possible different applications of your algorithm and on avenues of future research. For example, would it have been possible to map the whole MP3 decoder directly to Intel's manually optimized procedure? Then, if you add that element to the library, your mapping would find the best solution!

The granularity of the library functions currently available is less that the suggestion proposed.

- I find that the paper is well written. However, the use of articles should be improved. Every time you refer to an object in particular you should use an article. There are too many instances to report them individually.

Response to Review Number 4

In my previous comments/questions:

(A)

(2) The algorithm in Table 3 (p.21) is unclear:
- How about the case when "solution is not found" for all nodes in the tree ?
You answered:
"Since we start with a polynomial and we always have ADD and MULT instruction on our processor, we are guaranteed a solution".
It is not a proof. How about other instructions, e.g. DIV or SHIFT, if they exist in an application? Please discuss more formally and clearly in the paper.

As mentioned in the previous response, we are starting with a polynomial. A polynomial by definition does not include division or shift operation. A polynomial contains only add and multiply operations.

(B)

(3) The paper did not show a mathematical proof of
(a) Existence of a polynomial approximation (with a given accuracy) for any basic block of the C code ?
(b) Existence of a library's element (in mapping process) for any polynomial ?
You answered:
"Our target applications are multimedia and DSP …"
This is should be seen in the title of the paper. For example, the title maybe: "Complex Instruction and Software Library Mapping for Multimedia and/or DSP Embedded Software Using Symbolic Algebra"
Then you discuss on the restrictions of the domain applications, etc.

We have decided to keep the title intact, since only Reviewer 4 suggested a title revision. On the other had, we are willing to change the title if the associate editor finds it necessary. The application domain of the work has been emphasized in the introduction and abstract of the paper.

(C

5

(j) Please re-arrange the references according to the publisher's rule. Writing of Ref. 8, 12, 13 needs further revisions.

Corrected.

Congratulation on your successes!

Best,

Reviewer #3