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

Armita Peymandoust, Tajana Simunic, and Giovanni De Micheli, *Fellow, IEEE*

***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 preoptimized 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 designer's ingenuity. In this paper, we present a tool flow and a methodology, SymSoft, that automates the use of complex processor instructions and preoptimized software library routines using symbolic algebraic techniques. We use SymSoft to optimize a set of examples for the SmartBadgeIV (Maguire *et al.*, 1998) portable embedded system running the Linux embedded operating system. The results of these optimizations show that by using SymSoft we can map the critical basic blocks of the benchmark 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 *inverse discrete cosine transform* routine. Our measurements on SmartBadgeIV 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.**

***Index Terms*—Automated software library mapping, embedded systems, performance optimization, power minimization, software optimization, symbolic algebra.**

I. 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.

Preoptimized 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 preoptimized 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 preoptimized library element. Examples 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 preoptimized 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 preoptimized 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 tradeoff. 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* (*inverse discrete cosine transform*) already present in MPEG Layer III (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

Manuscript received June 24, 2002; revised December 13, 2002, February 23, 2003, and March 16, 2003. This work was supported in part by ARPA/MARCO Gigascale Research Center, in part by Hewlett Packard Labs, and in part by Synopsys Inc. This paper was recommended by Guest Editor L. Lavagno.

A. Peymandoust is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA (e-mail: armita@stanford.edu).

T. Simunic is with Hewlett Packard Labs and the Computer Systems Laboratory, Stanford University, Stanford, CA 94304 USA (e-mail: tajana@stanford.edu).

G. De Micheli is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA (e-mail: nanni@stanford.edu).

Digital Object Identifier 10.1109/TCAD.2003.814951

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

$$\begin{aligned}&\texttt{for i} = 1..3\\&\qquad \texttt{y} = \texttt{y} + \cos(\texttt{i} * \texttt{x}).\end{aligned}$$

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

$$\texttt{y} = \cos(\texttt{x}) + \cos(2 * \texttt{x}) + \cos(3 * \texttt{x}).$$

Now, assume that for a given application, $\cos(\mathrm{x})$ 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

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

The new equation can be mapped to one multiply instruction and two MACs. 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 preoptimized library elements and complex assembly instructions. Then, the performance and energy critical code sections (CCS) 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 II discusses previous work in software optimization for energy and performance. Section III describes the software and hardware platform and the measurement setup we are using in our experiments. Section IV 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 V. SymSoft lowers the execution time and energy consumption of these examples by using a preoptimized software library available for StrongARM and the StrongARM instruction set. Finally, Section VI summarizes the contributions of this work.

II. Related Work

Designers have used software performance and size optimization methodologies and tools 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. Several researchers are studying automatic code retargeting techniques for embedded processors [9], [10] using graph-covering methods. Graph covering methods have limited knowledge of algebra. Using algorithms from symbolic algebra, as explained in this paper, enables simultaneous code generation and algebraic manipulations. Currently, most embedded processors (or DSPs) are programmed directly by expert programmers and code optimization is mostly based on human intuition and skills. 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

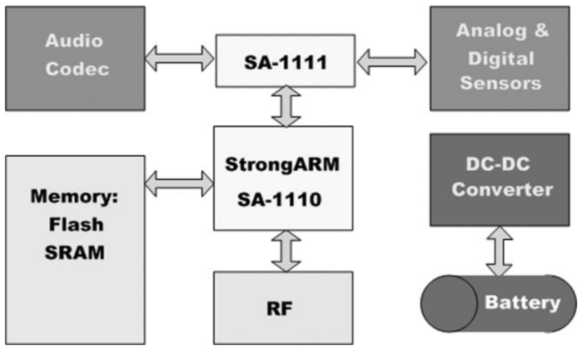


Fig. 1. SmartBadgeIV architecture.

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], [35]. 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 with the output of the commercial StrongARM compiler.

III. Experimental Setup

We used SymSoft to optimize a set of examples on the SmartBadgeIV [2]. SmartBadgeIV, as shown in Fig. 1, is an embedded system powered by batteries through a dc–dc converter. It consists of a StrongARM SA-1110 processor with a 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 the Linux operating system [17]. In this work, we use the Linux operating system since the software library available to us is implemented for Linux. SmartBadgeIV's Linux has the main parts of the operating system, including a small file system, residing in the 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.

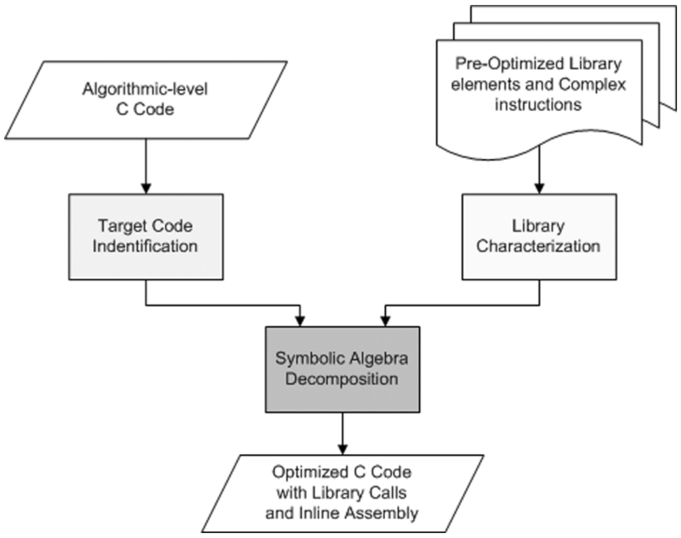

Algorithmic-level C Code
Pre-Optimized Library elements and Complex instructions
Target Code Indentification
Library Characterization
Symbolic Algebra Decomposition
Optimized C Code with Library Calls and Inline Assembly

Fig. 2. SymSoft tool flow.

All of the measurements were performed using the National Instruments Data Acquisition (DAQ) measurement system which is capable of 1.25 Msamples/s. 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 SmartBadgeIV. The execution time of the code was measured by accessing StrongARM SA-1110 on-board timer.

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

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 IV-B2, detects if data representation is an issue within

TABLE I
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

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 the 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 CCS into preoptimized 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 following sections describe each part of the SymSoft flow in detail.

A. Library Characterization

The target library consists of preoptimized software libraries and complex arithmetic instructions available for the target processor. Complex arithmetic instructions vary from the simple MAC to more complex instructions, such as those developed by Tensilica tools [6]. Preoptimized 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 preoptimized routines. Table I shows a sample of elements of the IPP library. Library characterization is done on element-by-element basis.

TABLE II
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

Each element is labeled with the type of inputs and outputs, performance, accuracy, energy consumption, and finally its polynomial representation.

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 IV-B3 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. The 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 the same functionality. Most embedded systems have operating system 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 II. The library has three different versions of each library element: the open-source floating-point version from the MP3 standards library [3], the fixed-point in-house preoptimized routine, and a version from Intel's 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 II are represented using polynomials. Since polynomials for complex library elements can be quite large, we show only a critical portion of IMDCT polynomial in (1) which 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 $\cos((\pi/2n)(2i+1+(n/2))(2k+1))$ can be calculated in advance for all i, k, and n

$$x_i = \sum_{k=0}^{(n/2)-1} y_k \cos\left(\frac{\pi}{2n}\left(2i+1+\frac{n}{2}\right)(2k+1)\right). \quad (1)$$

B. 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 CCS that

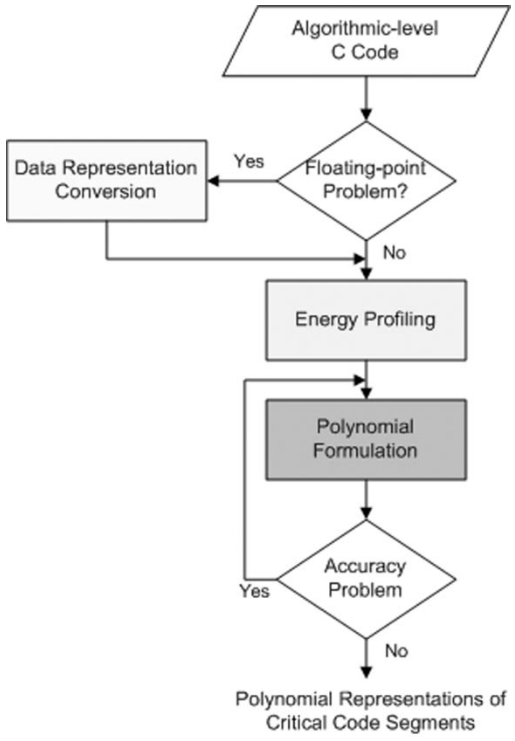
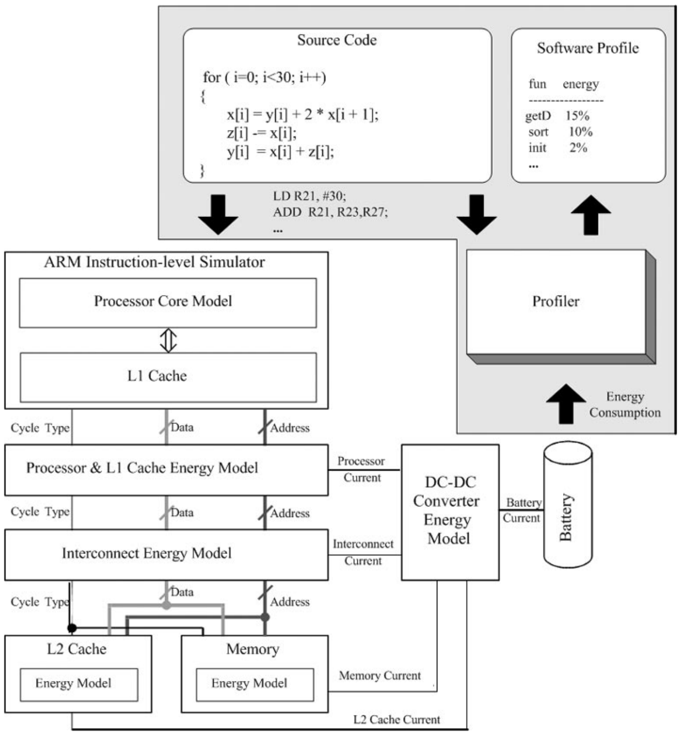


Fig. 3. Target code identification.



```
for ( i=0; i<30; i++)
{
    x[i] = y[i] + 2 * x[i + 1];
    z[i] -= x[i];
    y[i]  = x[i] + z[i];
}
LD R21, #30;
ADD  R21, R23,R27;
...
```

Fig. 4. Profiler architecture.

would benefit most from mapping to complex instruction and preoptimized library elements. Target code identification consists of three stages as shown in Fig. 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.

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 semiautomating 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 (also known as CoCentric fixed-point designer) to automate this stage of optimization.

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 [24] is shown in Fig. 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.

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 CCS 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.

3) Polynomial Formulation: Our goal is to automatically map the CCS selected by the profiler into preoptimized library elements or complex assembly instructions such that optimum execution time and power consumption are achieved. The symbolic mapping algorithm, described in Section IV-C, takes as input the polynomial representations of the CCS and the polynomial equivalence of complex arithmetic assembly instructions and preoptimized 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 CCS. The second set of inputs is calculated in the library characterization step as described in Section IV-A.

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 and constant and variable propagation to form larger basic blocks.

C. Symbolic Mapping Algorithm

The symbolic mapping algorithm requires two sets of inputs: a set of polynomials representing the CCS and another set of polynomials representing the preoptimized 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 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 IV-A. 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 $\mathtt{x}$ in $(\mathtt{x}+1)*(\mathtt{x}-1)$. 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], [35]. 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 nontrivial and important task, especially since different

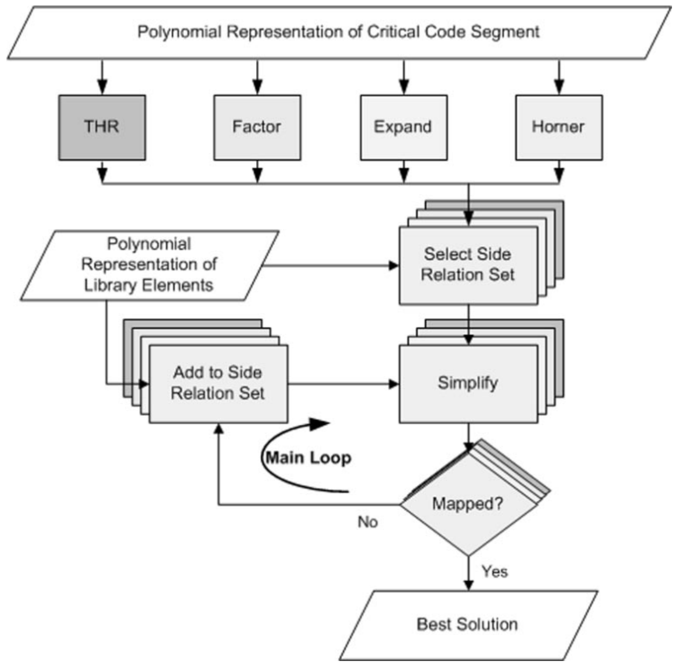


Fig. 5. Overview of the library mapping algorithm.

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 CCS of software by mapping to preoptimized 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.

Fig. 5 gives an overview of the mapping algorithm. Inputs to the algorithm are the polynomial representations of the 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 Fig. 5 as *main loop*, is implemented using branch-and-bound algorithms.

Table III shows the pseudocode of the library-mapping algorithm. Inputs to this algorithm are the polynomial representation of the critical-code sections and the polynomial representations

TABLE III
PSEUDOCODE 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
```

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.

The branch-and-bound algorithm in Table III is applicable to most practical problems and its runtime is a 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

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

which is approximated using Pade approximation to the polynomials shown in (3) in the previous step of the SymSoft flow as described in Section IV-B3

$$\begin{aligned} x =& \frac{\pi}{72}\left(2p+1+\frac{N}{2}\right)(2m+1) \\ d \cong& \frac{1-\frac{3665}{7788}x^2+\frac{711}{25\,960}x^4-\frac{2923}{7\,850\,304}x^6}{1+\frac{229}{7788}x^2+\frac{1}{2360}x^4+\frac{127}{39\,251\,520}x^6}. \end{aligned} \tag{3}$$

The simplification modulo set of polynomials routine can be used to map the numerator and denominator of (3) to the available instruction set. Let dn be the numerator of (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 (3) can be mapped to one square and three MAC's 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 (2) uses floating-point and takes 2124 cycles to run on the StrongARM SA-1110 processor. The approximation represented in (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 901 cycles. Thus, we have achieved an improvement of 57% for this simple example. The fixed-point version with no symbolic optimization executes in 1367 cycles. Thus, approximately 50% of the improvement achieved is due to use of fixed-point arithmetic and 50% is due to smarter use of processor instructions. ■

V. 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 III.

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 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 second-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 a least-mean-square (LMS) FIR adaptive filter. The LMS

TABLE IV
RESULTS OF SYMSOFT OPTIMIZATION ON A SET OF EXAMPLES

	Execution time in microsecs		
Examples	**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	5470000	1430000	73.86

TABLE V
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

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 MP3 audio decoder implementation that streams MP3 encoded files from a server to a client (SmartBadgeIV).

Table IV 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 CCS results in noticeable improvement on any given example. Next, the CCS are automatically mapped to the instruction set available on the StrongARM SA-1110 processor and Intel's IPP library for StrongARM SA-1110 processor [14]. Table IV shows the execution time of each example before and after the optimization with SymSoft. Note that the original execution time column reports the execution time of the examples when all possible optimizations available with the ARM compiler are used.

The improvements demonstrated in Table IV 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 IV 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.

A. MP3 Optimization Results

We start with an algorithmic level description of the MP3 audio decoder obtained from the International Organization for Standardization [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 the Linux operating system [17], Intel's IPP library for StrongARM SA-1110 processor [14], and a library populated with in-house preoptimized 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 IV, 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 CCS. Table V shows the results of profiling original MP3 decoder software we obtained from the standards body. All profiling reported in Table V 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 V. We use polynomial approximations for the nonlinear 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.

In the first phase of optimization, the polynomial representations of the critical-code sections of the first three function shown in Table V are mapped into the StrongARM assembly instructions by algorithm described in Section IV-C. 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

TABLE VI
COMPARISON BETWEEN SYMSOFT INSTRUCTION MAPPING AND COMMERCIAL COMPILER

	Execution time (#cycles)			Energy Consumption (mWhr)		
Function	**original**	**optimized**	**%imp**	**Original**	**optimized**	**%imp**
III_dequantize_sample	650894	421976	35.2	0.940	0.747	20.5
PowThreeFourth	14135	5380	61.9	0.040	0.009	76.6
SubBandSynthesis	155204	70633	54.5	1.015	0.306	69.8
generateFilterS	5263831	4196853	20.3	3.630	3.319	8.6
Inv_mdctL	63583	31954	49.7	0.101	0.051	49.6
generateMDCTTable	1454550	957051	34.2	1.051	0.922	12.2

TABLE VII
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

TABLE VIII
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

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 VI. The numbers reported in Table VI are obtained using the cycle accurate energy simulator described in Section IV-B2. The first, third, and fifth rows in Table VI correspond to the first three rows of Table V. The second, fourth, and sixth rows in Table VI are functions related to the function in the previous row. 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.

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 VII. Although the execution time per frame is drastically reduced (by two orders of magnitude compared with Table V), we can see that still almost 85% of the execution time is spent in the IMDCT and SubBand synthesis functions.

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 VII. The resulting performance profile is shown in Table VIII. 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 five.

Table IX 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 website 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.

The fourth row corresponds to the results 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 savings 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 a factor of

TABLE IX
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

four is solely due to automatic use of complex library elements that have been preoptimized 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× 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 IX, 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 multimedia applications, since large portions of these software codes can be easily represented by polynomials.

VI. 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 CCS 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 preoptimized software library functions, conversion of critical floating-point operations to fixed point, and reducing the number of memory accesses and instructions executed in CCS. The technique presented in this paper can be easily used in conjunction with other compiler-optimization techniques [7].

ACKNOWLEDGMENT

The authors would like to thank the Gigascale Research Center, Hewlett Packard Labs, and Synopsys Inc. for their support.

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, pp. 419–435, Mar. 1997.
[2] G. Q. Maguire, M. Smith, and H. W. P. Beadle, "SmartBadges: A wearable computer and communication system," in Proc. 6th Int. Workshop on Hardware/Software Codesign, 1998, pp. 10–16.
[3] "Coded Representation of Audio, Picture, Multimedia and Hypermedia Information,", ISO/IEC JTC/SC 29/WG 11, pt. 3, 1993.
[4] M. Willems, H. Keding, T. Grötket, and H. Meyr, "Fridge: An interactive fixed-point code generation environment for HW/SW codesign," in *Proc. Int. Conf. Acoustics, Speech, Sig. Process.*, 1997, pp. 687–690.
[5] G. Constantinides, P. Cheung, and W. Luk, "The multiple wordlength paradigm," in *Proc. IEEE Symp. Field-Programmable Custom Comput. Mach.*, Mar. 2001.
[6] A. Wang, E. Killian, D. Maydan, and C. Rowen, "Hardware/software instruction set configurability for system-on-Chip processors," in *Proc. Design Automation Conf.*, 2001, pp. 184–190.
[7] S. Muchnick, *Advanced Compiler Design and Implementation*. San Mateo, CA: 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 Comput.*, vol. 29, pp. 84–89, Dec. 1996.
[9] P. Marwedel and G. Goossens, *Code Generation for Embedded Processors*. Norwell, MA: Kluwer, 1995.
[10] R. Leupers, *Retargetable Code Generation for Digital Signal Processors*. Norwell, MA: Kluwer, 1997.

[11] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vanduoppelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Norwell, MA: Kluwer, 1998.
[12] V. Tiwari, S. Malik, A. Wolfe, and M. Lee, "Instruction level power analysis and optimization of software," *J. VLSI Signal Process. Syst.*, 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 toward software power minimization," *IEEE Trans. VLSI Syst.*, vol. 2, pp. 437–445, Dec. 1994.
[14] Integrated Performance Primitives for the Intel StrongARM SA-1110 Microprocessor (2000). [Online]. Available: www.intel.com
[15] *TI'54x DSP Library*, Texas Instruments, 2000.
[16] *eCos™ Reference Manual*, Cygnus Solutions, 1999.
[17] *Linux-Arm Math Library Reference Manual*, RedHat, Inc., 2000.
[18] J. Crenshaw, *Math Toolkit for Real-Time Programming*. Lawrence, KS: CMP Books, 2000.
[19] H. Mehta, R. M. Owens, M. J. Irvin, R. Chen, and D. Ghosh, "Techniques for low energy software," in *Proc. Int. Symp. Low-Power Electron. Design*, 1997, pp. 72–75.
[20] Y. Li and J. Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in *Proc. Design Automation Conf.*, 1998, pp. 188–193.
[21] H. Tomyiama, H. T. Ishihara, A. Inoue, and H. Yasuura, "Instruction scheduling for power reduction in processor-based system design," *Design, Automation, Test Eur.*, pp. 23–26, Feb. 1998.
[22] M. Kandemir, N. Vijaykrishnan, M. Irwin, and W. Ye, "Influence of compiler optimizations on system power," in *Proc. 27th Int. Symp. Comput. Architecture*, 2000, pp. 35–41.
[23] *ARM Software Development Toolkit Version 2.11*, Advanced RISC Machines Ltd (ARM), 1996.
[24] T. Simunic, L. Benini, and G. De Micheli, "Energy-efficient design of battery-powered embedded systems," *IEEE Trans. VLSI Syst.*, vol. 9, pp. 18–28, May 2001.
[25] *Information Technology, Generic Coding of Moving Pictures and Associated Audio: Conformance*, ISO/IEC JTC 1/SC 29/WG 11 13 818–4, 1996.
[26] (1988) Maple. Waterloo Maple Inc. [Online]. Available: www.maplesoft.com
[27] (1987) Mathematica. Wolfram Research Inc. [Online]. Available: www.wri.com
[28] A. Peymandoust and G. De Micheli, "Symbolic algebra and timing driven data-flow synthesis," in *Proc. Int Conf. Computer-Aided Design*, 2001, pp. 300–305.
[29] ——, "Using symbolic algebra in algorithmic level DSP synthesis," in *Proc. Design Automation Conf.*, 2001, pp. 277–282.
[30] T. Becker and V. Weispfenning, *Gröbner Bases*. New York: Springer-Verlag, 1993.
[31] J. Smith and G. De Micheli, "Polynomial methods for component matching and verification," in *Proc. Int. Conf. Computer-Aided Design*, 1998, pp. 678–685.
[32] ——, "Polynomial circuit models for component matching in high-level synthesis," *IEEE Trans. VLSI Syst.*, vol. 9, pp. 783–800, Dec. 2001.
[33] V. Zivojnovic, J. Martinez, C. Schläger, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," in Proc. Int. Conf. Sig. Process. Applicat. Technol., 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," in *Proc. Int. Symp. Syst. Synthesis*, Sept. 2000, pp. 193–198.
[35] A. Peymandoust and G. De Micheli, "Application of symbolic computer algebra in high-level data-flow synthesis," *IEEE Trans. Computer-Aided Design*, vol. 22, Sept. 2003, to be published.

Armita Peymandoust received the B.S. degree in electrical and computer engineering from University of Tehran, Tehran, Iran, the M.S. degree in electrical and computer engineering from Northeastern University, Boston, MA, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, 2003.

Previously, she was a Design Engineer on the IA-64 product line with the Intel Corporation, Santa Clara, CA. She is currently with Synopsys Inc., Mountain View, CA. Her research interests include embedded systems, system-level design and synthesis, hardware/software codesign, and computer architecture.

Tajana Simunic received the M.S. degree in electrical engineering from the University of Arizona, Tucson and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, 2001.

She is currently a Research Scientist with Hewlett Packard Labs, Palo Alto, CA, and a Research Affiliate in the Computer Systems Lab, Stanford University. Formerly, she was a Senior Design Engineer with Altera Corporation, San Jose, CA. Her interests are in low-power system design, embedded systems, and wireless system design.

Giovanni De Micheli (S'79–M'79–SM'80–F'94) received the nuclear engineer degree from Politecnico di Milano, in 1979 and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1980 and 1983, respectively.

He is a Professor of Electrical Engineering and, by courtesy, of Computer Science at Stanford University, Stanford, CA. His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on synthesis, system-level design, hardware/software co-design and low-power design. He is author of *Synthesis and Optimization of Digital Circuits* (New York: McGraw–Hill, 1994) and co-author and/or co-editor of five other books and of over 250 technical articles.

Dr. De Micheli is a Fellow of ACM. He received the 2003 IEEE Emanuel Priore Award for contributions to computer-aided synthesis of digital systems. He received the Golden Jubilee Medal for outstanding contributions to the IEEE CAS Society in 2000. He received the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS Best Paper Award and two Best Paper Awards at the Design Automation Conference, in 1983 and 1993. He is President of the IEEE CAS Society. He was Editor in Chief of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS from 1987 to 2001. He was the Program and General Chair of the Design Automation Conference (DAC) from 1996 to 1997 and 2000, respectively. He was the Program and General Chair of the International Conference on Computer Design (ICCD) in 1988 and 1989, respectively. He was a founding member of the ALaRI institute at Universita' della Svizzera Italiana (USI), in Lugano, Switzerland, where he is currently scientific counselor.