Complex 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

Embedded software designers often use libraries that have been pre-optimized for a given processor to achieve higher code quality. However, using such libraries in legacy code optimization is nontrivial and typically requires manual intervention. This paper presents a methodology that maps algorithmic constructs of the software specification to a library of complex software elements. This library-mapping step is automated by using symbolic algebra techniques. We illustrate the advantages of our methodology by optimizing an algorithmic level description of MPEG Layer III (MP3) audio decoder for the Badge4 [2] portable embedded system. During the optimization process we use commercially available libraries with complex elements ranging from simple mathematical functions such as `exp` to the `IDCT` routine. We implemented and measured the performance and energy consumption of the MP3 decoder software on Badge4 running embedded Linux operating system. The optimized MP3 audio decoder runs 300 times faster than the original code obtained from the standards body while consuming 400 times less energy. Since our optimized MP3 decoder runs 3.5 times faster than real-time, additional energy can be saved by using processor frequency and voltage scaling.

Categories and Subject Descriptors

C.3 **[Special-Purpose and Application-Based Systems]**: *Microprocessor/microcomputer applications, Real-time and embedded systems, Signal processing systems.*

General Terms

Algorithms, Performance, Design, Experimentation, Theory.

Keywords

Embedded software optimization, Automated library mapping, Symbolic algebra, Polynomial representation, Computation intensive software.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

1. INTRODUCTION

The market demand for portable multimedia applications has exploded in the recent years. Software optimization is critical for such systems due to time-to-market pressures. The software design process consists of translating a high-level specification into the optimized machine code for the target processor, often using compilers. There has been several research projects on optimizing compilers in last few years [5]. Prototype research compilers have shown impressive results [6]. 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 optimization techniques for embedded processors [7,8], currently, most embedded processors (or DSPs) are programmed directly by expert programmers and code optimization is mostly based on human intuition and skill.

Software engineers typically start with algorithmic level C code, sometimes developed by the standards groups such as MPEG, and manually optimize it to execute on the given hardware platform. Pre-optimized libraries for embedded system design are often available. For example, Intel recently released a library targeted at multimedia developers for SA-1110 embedded processor [10], and TI has a similar library for TI'54x DSP [11]. Embedded operating systems typically provide a choice from a number of mathematical and other libraries [12,13]. In addition, a library of more complex instructions, such as those developed by Tensilica tools [4], could be used. 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. Such manual optimization is error-prone and introduces undesired delay in the overall development process.

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 [14], 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 functions 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. Clearly, the high-level arithmetic optimizations

targeted at the use of complex library elements are currently left to the designers' ingenuity.

Our methodology automates the process of identifying the code sections that benefit from complex library mapping, and then performs the mapping using symbolic techniques. Similar symbolic techniques have been used for algorithmic level synthesis of data intensive circuits [20], and for software optimization of arithmetic functions by efficient mapping into processor's instructions set [15]. To illustrate the difference, let us go back to the examples discussed above. The work presented in [20] is concerned with mapping `log` to its hardware implementation, while in [15] the focus is on representing `log` and portions of `IDCT` with polynomials and then decomposing those into complex processor instructions, such as `MAC`. In contrast, the methodology presented in this work attempts to map `log` and `IDCT` into as complex software library element as possible, without resorting to decomposition into processor instructions when not necessary. In addition, while previous work reported only simulation results, in this work we present the measurement results by running the code on a hardware platform (Badge4 [2]).

The paper is organized as follows: Section 2 describes the target software and hardware platforms used in reported measurements. Section 3 describes our methodology, and gives an overview of each of its steps. The results of MP3 decoder optimization for Badge4 are presented in Section 4. We measured a significant performance increase and energy consumption decrease over the original executable specification from the standard body. Finally, Section 5 summarizes our contributions.

2. BACKGROUND

The main motivation for this work is the author's experience in porting MP3 audio decoder software available from the standards body [3] first to SmartBadge [2] with Angel operating system (OS), then to SmartBadge with eCos OS, and finally to the new version of SmartBadge, Badge4, running Linux OS. The manual optimization for MP3 decode on the SmartBadge [16] required the designer first to implement a fixed-point library and replace all floating-point operations with fixed point. Then, the designer needs to fully understand the details of the SmartBadge's design, so that the critical arithmetic operations could be manually optimized with inline assembly code. Each manual optimization process lasted several days.

While hand-optimizing code may be interesting the first time, it becomes increasingly tedious in the subsequent tries. This experience is common in typical industrial settings, where the software needs to be ported and optimized to the newer versions of hardware. Thus, the goal of this work is to automate mapping of complex arithmetic functions commonly occurring in portable system designs, into a set of pre-optimized library routines.

Badge4, as shown in Figure 1, is an embedded system powered by batteries through a DC-DC converter. It consists of StrongARM-1110 processor with SA-1111 companion chip that controls peripherals, audio CODEC with microphone and speakers, Lucent's WLAN card, sensors and three types of memory: SRAM, SDRAM and FLASH. Badge4 is the SmartBadge with a new processor and the addition of SDRAM and a companion chip.

Badge4 currently runs eCos [12] and an embedded version of Linux operating system [13]. In this work we use Linux OS since the libraries available to us are implemented only for Linux. Badge4'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 MP3 files reside in the directory structure on the server and are streamed via wireless link to the Badge4 for decoding. The output can either be played back on the speakers, or streamed back and saved in a file for accuracy comparison purposes.

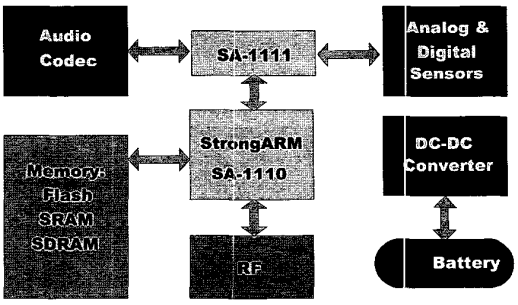

Figure 1. SmartBadge Architecture

We obtained the original MP3 audio decoder software from the International Organization for Standardization [3]. The first step in decoding 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 cosine transform (`IMDCT`) followed by a polyphase synthesis filterbank. Compliance test provided by MPEG standard [17] is used to evaluate the accuracy of the optimizations. The range of RMS error between the original code's output and the samples produced by the code under test defines the level of compliance.

The outcome of our mapping algorithm is faster than real-time MP3 decoder software for Badge4. For an MP3 player, faster than real-time execution implies that lower voltage and frequency can still meet the real-time constraint. This in turn translates into longer battery life or lighter battery requirement for the embedded system. The next section gives an overview of our methodology for mapping source code into complex software library elements.

3. COMPLEX SOFTWARE LIBRARY MAPPING METHODOLOGY

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

Our methodology automates most of the library mapping process. The mapping methodology consists of three main steps. The first step is to characterize the library elements. The characterization needs to include not only performance and energy consumption for a given 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, the critical functions are chosen via profiling. Traditional compiler techniques are used in representing the arithmetic section of the critical functions as polynomials. Finally, the target code represented by polynomials is mapped into the library elements. The mapping process uses symbolic techniques to decompose the target code into a set of library elements. The mapping process selects the solution that offers best performance with sufficient accuracy.

Our key contribution is a new method for mapping code into a library of complex software elements using symbolic polynomial manipulation. Since our methodology is compliant with other software optimization techniques, additional benefits are gained by combining it with traditional complier optimization algorithms. The next sections describe each part of our methodology in detail.

3.1 Library Characterization

The target library may consist of traditional embedded system library, such as IEEE floating-point mathematical library for Linux operating system [13], a commercial library available for the particular processor, such as Intel's integrated performance primitives library [10], and a set of in-house pre-optimized routines. 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 the polynomial representation.

The format of library element inputs and outputs is determined from the library include files. Techniques discussed in the next section can be used to extract the polynomial 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. But often there is not an easy way to measure processor and memory power consumption. Alternatively, a cycle-accurate energy consumption simulator [16] easily provides energy and performance estimates of library elements.

Table 1. Sample Complex Library Elements

Library Element	Execution time	Execution time ratio
float SubBandSyn	0.95	1
fixed SubBandSyn	0.01	92
IPP SubBandSyn	0.002	479
float IMDCT	0.39	1
fixed IMDCT	0.014	27
IPP IMDCT	0.0002	1898

Examples of two complex library elements, SubBand Synthesis and `IMDCT`, are shown in Table 1. 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 SA-1110 processor [10]. For each library element we have measured its performance on the Badge4 hardware. All entries in Table 1 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. Note that this is just a first order polynomial, since `cos(i,k,n)` can be calculated in advance for all `i`, `k` and `n`. Total of `n/2` windowed samples, y_k, are transformed into `n` x_i samples.

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

3.2 Target Code Identification

The first step in target code identification is to identify the energy and performance critical procedures. This step can be done with either the energy profiler simulator [16], or by profiling directly on the hardware. Once the power and performance critical procedures are identified, they are formulated as polynomials suitable for mapping into library elements.

Critical procedures calculating an arithmetic or Boolean function can be easily represented as polynomials. The polynomial representation of a procedure can be directly extracted from the C code if it calculates a linear arithmetic function. If the procedure performs a series of bit manipulations or Boolean functions, previously developed algorithms based on interpolation [22] can be used to formulate its equivalent polynomial representation. When a section of the procedure implements a nonlinear function, we use an approximation, such as the Taylor or Chebyshev series expansion, as its polynomial representation.

The goal of this step is to formulate as large polynomials as possible, so the likelihood of finding a more complex library element that matches at least a portion of the formulated polynomial is increased. This can be accomplished by using code transformation techniques such as loop unrolling, constant and variable propagation, code motion, conditional expansion and model expansion.

3.3 Library Mapping Algorithm

This step decomposes the polynomial representations of the code blocks into available complex library elements while minimizing cost. Inputs to the library-mapping algorithm are a set of polynomial representations for critical code blocks, a characterized library of complex elements, and a routine that provides accuracy and cost (e.g. performance, energy) feedback. Note that a similar algorithm was used for algorithmic level synthesis of data intensive circuits [20], and for mapping basic blocks of arithmetic functions into complex processor instructions [15] (e.g. mapping of `log` into a series of `MAC`s). The core of the

algorithm is a symbolic manipulation technique, known as *simplification modulo set of polynomials (simplify)*, based on Gröbner basis fundamentals [20]. The polynomial representations of critical code blocks are simplified modulo a subset of polynomial representations of library elements.

Symbolic computer algebra is a set of algorithms capable of algebraic manipulation of expression containing undetermined values (symbols), such as variable x in `(x+1)*(x-1)`. Several commercial symbolic computer algebra systems are available on the market; Maple [18] and Mathematica [19] are most widely used. Most interesting symbolic polynomial manipulations are based on Gröbner bases [21]. Gröbner bases also solve variable elimination in a set of polynomials and ideal membership problems, which is the core of simplification modulo set of polynomials. We use the following set of symbolic techniques: factorization, expansion, Horner transform, multivariate polynomial substitution, and variable elimination.

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)
```

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
```

Elimination theory based on the Gröbner basis formalizes 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
```

Since evaluating all subsets of the library is exponentially expensive, the library-mapping algorithm uses the branch-and-bound method with performance and energy consumption as bounding functions to prune the search space. All previously described symbolic manipulation except *simplify* are used as guidelines in formulating different side relation sets to speed up the mapping algorithm. The symbolic manipulations result in various equivalent polynomials and thus provide more options to the mapping algorithm. Therefore, the speed and likelihood of finding the right match will increase. As with all branch-and-bound algorithms, in spite of using heuristics to speed up the mapping process, this algorithms worst-case complexity still remains exponential.

Table 2 shows the pseudo-code of the library-mapping algorithm. The target code to be mapped into a library *L* is represented with *S*. Mapping *S* into *L* is equivalent to simplifying *S* modulo elements of the library *L* as side relations (*sr*). Decision tree (*solution_tree*) implements the branch-and-bound algorithm. The results of each *simplify* step are also saved in the tree data structure. When a simplification result is within an acceptable tolerance of the polynomial representation of a library element, a possible solution is found and the corresponding tree node is marked accordingly. The algorithm also applies tree-height reduction, factorization, substitution, expansion, and Horner-based transform on *S*. As a result, there are several polynomials representing the target code (*exp_tree*), which can used to guide the initial side relation selection process. When all initial side relations are used and the result of simplify is not a library element, we decompose the result without further guidance from the expression tree. The algorithm is implemented in C with calls to Maple V for the symbolic manipulations. Typically, the algorithm takes only a few minutes to execute since we use heuristics to speed up the mapping process. However, its worst-case complexity is still exponential.

Table 2. Library Mapping Algorithm

```
function Decompose (exp_tree, boundVal) {
  #initialize a solution tree
  solution_tree ← tree (exp_tree)
  Depth ← 0
  Bound ← boundVal
  for all n ∈ tree with Depth {
    if Depth == 0
      choose sr ∈ L to preserve the exp_tree structure
    else for all sr ∈ L {
      result = simplify (n,sr);
      Add_child (n,result)  #make result a child of node n
      Depth ← Depth + 1
      if result ∈ L
      { # solution is found
        Bound = cost of node result; } } }
  return the best solution with sufficient accuracy
end Decompose

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

4. RESULTS

We illustrate the advantages of our methodology by performing library mapping on an algorithmic level description of the MPEG Layer III (MP3) audio decoder we obtained from the standards body [3]. The optimization target is the Badge4 portable embedded system shown in Figure 1. Badge4 currently runs an embedded version of Linux operating system [13]. During the optimization process we used a mathematical library available with Linux OS [13], Intel's integrated performance primitives (IPP) library for SA-1110 processor [10], and a library populated with in-house pre-optimized routines. The library

elements ranged from simple mathematical functions such as `exp` to as complex elements as `IMDCT` routine.

Our library mapping methodology, as described in Section 3, 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 [16]. 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 two important steps: profiling the code and formulating polynomials to be mapped. All profiling is done using hardware measurements. Table 3 shows the results of profiling original MP3 decoder software we obtained from the standards body. The results are shown for one frame and represent only the most significant functions in terms of their performance impact.

Table 3. Original MP3 Profile

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

The next step is to represent portions of these functions as polynomials and then map them into available libraries. For the first mapping step we selected Linux mathematical (LM) and in-house libraries (IH). The resulting performance profile is shown in Table 4. Although the performance per frame is drastically reduced (by two orders of magnitude), we can see that still almost 85% of the execution time is spent in the `IMDCT` and subband synthesis functions.

Table 4. MP3 Profile after LM & IH mapping

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

The next step is to map to Intel's IPP library to further optimize the code. Here we find two primitives that match the two critical procedures shown in Table 4. The new performance profile is shown in Table 5. As shown, our method automatically uses two of the IPP routines. These two routines correspond to the two most time consuming sections of the code as shown in Table 4. 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 50.

Table 5. MP3 Profile after LM & IH & IPP mapping

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 6 summarizes the performance and the energy results of the overall mapping process. Both measurements were performed on the Badge4 while running at maximum processing speed and voltage. We started from the original source code that runs roughly two orders of magnitude slower than real time. The next two entries show the results of mapping only into Intel's IPP library; more specifically, we were 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-1110 cannot perform floating-point operations natively; therefore the code is still far from real-time execution. IH library entry represents the mapping from the original code, into Linux mathematical library and our in-house pre-optimized library. We save two orders of magnitude in both performance and energy for this mapping. This is due to changing 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 mapping to all three libraries: Linux mathematical, in-house and Intel's IPP library. The factor of four improvement is achieved

Table 6. Performance and Energy for MP3 library mapping

Code version	Perf (s)	Factor	Energy (J)	Factor
Original	503.92	1.0	509.6	1.0
IPP SubBand	301.43	1.7	292.5	1.7
IPP SubBand & IMDCT	211.27	2.4	199.1	2.6
IH Library	5.47	92.1	4.47	114.2
IH + IPP SubBand	3.33	151.4	2.78	182.3
IH + IPP SubBand & IMDCT	1.43	352.4	1.17	435.2
IPP MP3	0.41	1240.8	0.31	1626.

solely based on automatic use of complex library elements that have been pre-optimized for the given processor. Such optimization was not possible in previous work [15], since mapping was limited to the complex instructions available on the target processor (MAC). 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 [17].

The last table entry, IPP MP3, represents fully hand-optimized code available for MP3 from Intel. Our most optimized version (IH+IPP SubBand & IMDCT) is a factor of five worse than hand-optimized IPP MP3. Our approach only automates the process of mapping code that can be represented with polynomials into complex library elements. Since large portions of MP3 decoder software can be represented with polynomials, we were able to measure savings of a factor of 350 in performance and 435 in energy over the original source code obtained from the standards organization. Even larger energy savings are possible by using processor frequency and voltage scaling, because our most optimized MP3 code runs almost four times faster than real time. One should note that the improvement factors reported are algorithmic floating-point measurements divided by optimized code measurements. Improvements of this order are realistic when skilled designers hand optimize code for a given embedded system, as shown in the last column of Table 6. Our contribution is to automate most of this optimization process.

5. CONCLUSION

The contribution of this paper is a methodology for automatic mapping of critical code sections into complex library elements. Since our mapping methodology uses symbolic algebra methods, we focus only on code sections that can be represented with polynomials. There are three main steps in our methodology: library characterization, target code identification, and library mapping.

We have tested our methodology by mapping critical code sections of MP3 audio decoder into libraries we had available for the Badge4 portable embedded system. We have measured savings of a factor of 350 in performance and 435 in energy over the original specification while still keeping in full compliance with the MPEG standard. Currently, skilled software designers hand optimize code to achieve such improvement gains. Our proposed methodology automates this tedious process. Additional energy savings are possible by using frequency and voltage scaling since the final MP3 code runs a factor of four faster than real-time.

6. ACKNOWLEDGMENTS

This research is supported by ARPA/MARCO Gigascale Research Center, HP Labs, and Synopsys Inc. We would like to thank all preceding organizations for their support. We would also like to thank Dr. Yung-Hsiang Lu for his help with the data acquisition.

7. 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, 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] Albert Wang, Earl Killian, Dror Maydan, Chris Rowen, "Hardware/Software Instruction Set Configurability for System-on-Chip Processors", *Proceedings of the Design Automation Conference*, pp. 184-190, June 2001.

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

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

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

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

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

[10] Intel, "Integrated Performance Primitives for the Intel StrongARM SA-1110 Microprocessor", 2000.

[11] Texas Instruments, "TI'54x DSP Library", 2000.

[12] Cygnus Solutions, "eCos™ Reference Manual", 1999.

[13] RedHat, "Linux-arm math library reference manual".

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

[15] A. Peymandoust, T. Simunic, and G. De Micheli, "Low Power Embedded Software Optimization using Symbolic Algebra", *Proceedings of the Design Automation and Test in Europe Conference*, pp. 1052-1058, March 2002.

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

[17] 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.

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

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

[20] A. Peymandoust and G. De Micheli, "Symbolic Algebra and Timing Driven Data-flow Synthesis", *Proceedings of the International Conference on Computer Aided Design*, pp. 300-305, November 2001.

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

[22] J. Smith and G. De Micheli, "High-Level Synthesis and Design Reuse using Polynomial Circuit Models", *IEEE Transactions on VLSI*, Vol. 9, No. 6, pp. 783-800, December 2001.