A Multiobjective Evolutionary Approach for Constrained Joint Source Code Optimization [1]

N. Z. Azeemi

Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms,
Institute of Communications and Radio Frequency Engineering,
University of Technology Vienna, Gusshausstrasse 25/389, A-1040 Vienna Austria
Emails: nzafar@nt.tuwien.ac.at

Abstract

The synergy of software and hardware leads to efficient application expression profile (AEP) not only in terms of execution time and energy but also optimal architecture usage. We present an architecture-based parametric optimization of 'C' source code for iterative compilation. Successive source-level, code transformations are applied in order to evaluate an application expression profile on complex multimedia processors. The proposed new code transformation methodology determines appropriate parameters for compiler optimization in order to satisfy user constraints on code size, energy, execution time and optimal target architecture usage. The optimization is based on a multicriteria, objective function. The constraints of this objective function are formulated using a penalty method; a genetic algorithm finds solutions eventually. We examined the performance improvement across typical different multimedia applications on a multimedia processor, TM1302 (Philips). Candidate applications include m100, m200, nlivq, MPEG-1, G-721 and H-264L. Experimental results show that our approach reduces cache misses by an average of 36% (max. 71%), improves typically energy dissipation up to 17% and CPU performance up to 60% for an H-264L video codec algorithm. However, the code size tends to be large, which inevitably leads to a larger memory size. The approach is general and can easily be integrated in multimedia processor compilers.

Keywords :
Iterative compilation, Low Energy, Multimedia Processor

1 Introduction

In embedded systems no matter what architecture is in use, inevitably energy efficiency of the overall system depends heavily on software applications. The energy dissipation in portable handheld embedded systems is directly linked to battery size, weight, packaging, cooling and operating time. Low energy software design can be achieved in different ways; either by energy aware selection of the algorithms [1, 2] or by code transformations [7]. It is difficult to automate an algorithm selection because of its strong dependence on the programmer's ingenuity. On the contrary, code transformation is performed at instruction level or by restructuring different code blocks. Such performance oriented code optimizations are available in the back-end of most compilers [2], but their impact on energy is strictly architecture dependent. Code transformation techniques lie in between since they can be automated to some degree, but they have global impact and they are not strongly tied to target architecture.

Though numerous tools and techniques have evolved during the last decade [5, 8] that addresses the energy reduction issue at hardware level but a significant contribution of software applications cannot be ignored. Unlike general purpose computers, relatively little efforts have been dedicated to develop efficient compilers for embedded processors. Currently, most embedded processors are programmed by expert programmers based on human intuition and skill. The traditional application compilation process is performed in two phases: front end, and back end compilation. The front end analyses the source code to build an internal representation of the program, called the intermediate representation (IR). It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope. Back end analyses uses a call graph [2] and a control flow graph for the code optimization that is later translated into the output language, usually the native machine language of the system (target architecture). This involves resources and storage decisions; such as deciding which variable to fit into register and memory and the selection and scheduling of appropriate machine instructions along with their associated address modes.

Traditional VLIW (very long instruction word) compilers generate poor machine code for handling irregular multimedia DSP processor architectures with respect to architecture usage, execution time, code size, and energy consumption since these techniques often perform a tree based code selection [12]. In contrast to that, performing a graph based code selection offers an immersed optimization potential [13]. In addition, traditional code selection techniques only achieve a restricted phase coupling. Tree based code selection techniques generate optimal code for trees but this concerns only sequential code which has

[1] This work has been funded by the Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms.

to be compacted in a separate optimization phase. In addition, the generated code does not contain spill code which is inserted by the register allocation step. Thus, energy aware optimizations made in early code generation phases are potentially nullified in subsequent optimization steps. For instance, the insertion of spill code [12] in a later phase drastically changes the dynamics successive instructions. For this reason, there is much unused optimization potential using a traditional tree based code selection technique. However, performing a graph based code selection and a phase coupled code generation means solving a more complex problem. This requires special optimization methods which are capable of finding 'good solutions' rather than optimal solutions in a huge search space in polynomial time. GA have proven to solve complex optimization problems by imitating the natural evolution process [11].

To address these diversified issues, a new profiling approach is required. This must include the information about system behavior on various levels (see Figure 1.1). The main goal of such multi-level profiling is to further improve the understanding of system behavior through correlation of profile information at different levels and hence reducing application-architecture mapping gap. In this work we demonstrate that it is necessary to use an application expression profile at all layers to understand existing performance problems such as poor architecture usage, increased execution time, and high energy consumption. This results in a huge size of the search space and its search complexity results in NP hard solution. Subject to the performance objective and architectural constraints, we consider this problem as a single task, where all desired aims have to be taken into account simultaneously. The fitness function of a genetic algorithm represents the objective function of the underlying optimization problem and thus has an essential impact on the optimization progress of the genetic algorithm. The approach leads to new energy aware source to source (sts) transformation framework that enables code restructuring according to different objectives by specifying a suitable fitness function such as minimization of execution time and energy aware optimization.

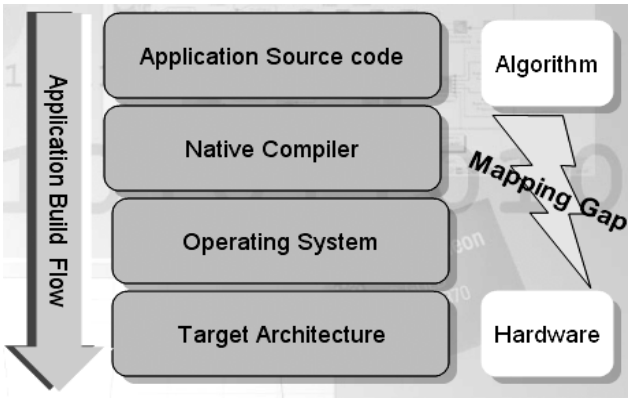

Figure 1.1. Application expression profile layers

Compared with traditional solutions [3, 5, 8, 9] of iterative compilation, our scheme has following advantages.

- **A novel profiling approach.** Vertical profiling is done for capturing and correlating performance problems across multiple execution layers.
- **Genetic algorithm (GA) is used to prune the optimization search space**
- **Penalty method is used to satisfy optimization constraints.**
- **Test results are shown for audio, video and speech codecs.**

The structure of this paper is as follows. Section 2 describes our transformation framework. Optimization model is presented in Section 3. Section 4 describes the methodology we use for our case studies and evaluates the overhead caused by our profile monitors Section 5 presents detailed case studies demonstrating how our framework can be used to better exploit the architectural benefits for cycle and energy performance of an application. Section 6 concludes the paper.

2 The Platform and Setup

We concentrate on multimedia DSP processors, following assumptions are made in our source-to-source transformation (StS) framework:

- The system has VLIW architecture whose instruction and data caches are both physically as well as logically separated.
- The associativity of the instruction cache is direct mapped or set associative with the LRU (least recently used) algorithm for replacement.
- The CPU cycle time excludes startup instruction pipeline filling time (that is filled with NOP), because until that time the actual application has not started.
- Stall cycle count is incremented by only one cycle in case of simultaneous instruction cache stall and a data cache stall occurs.

A simplified VLIW architecture is shown in Figure 2.1; it is composed of a CPU core, instruction and data cache units, memory management units and address/ data highway.

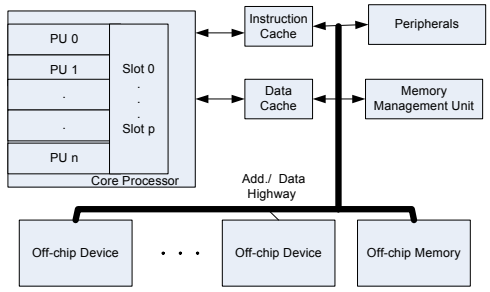

Figure 2.1: General architecture of a VLIW processor

2.1 Energy Measurement

Total power consumption for an application can be formulated as $V_{dd} I_{dd} + V_{cc} I_{cc}$. Application current consumption is recorded by measuring differential voltage drop across current sensing resistor at the processor input. Energy is measured at the target platform by measuring application current consumption and

integrating its product with core voltage over the execution time. The experiment was conducted while the processor [6] was configured to run at 200 MHz and SDRAM (32 Mbyte) was running at 166 MHz. Although this processor supports a block mode power management scheme, it was not activated throughout the experimentation.

2.2 Methodology Flow and Performance Monitors

The methodology framework is shown in Figure 2.2. The source code is processed successively in subsequent layers and a list of parameters is generated in each step during our methodology flow. These parameters are attributed to application and target processor. Intermediate trace files are generated during the code processing flow (i.e. pre/post compiler, scheduler etc.) to produce SPMs, e.g., code size, execution time number of cache miss (for both instruction and data caches), highway usage, scheduling factor, and slot utilization. After simulation these parameters are used to compute transformation control factors such as unrolling factor, grafting depth and blocking metrics (see Section 2.3).

Successively, after each cycle, all these parameters are computed again and are compared to constraints mentioned in the user constraint file (UCF). This file contains user constraints, to be used in maximizing objective function.

2.3 Transformation Steering Factors and Their Interpretations

In this section, we describe the control factors that maneuver the code transformation which determines when to cease iterations in the transformation engine shown in Fig. 2.2. We consider the following transformation techniques:

- loop unrolling,
- decision tree grafting and
- loop tiling.

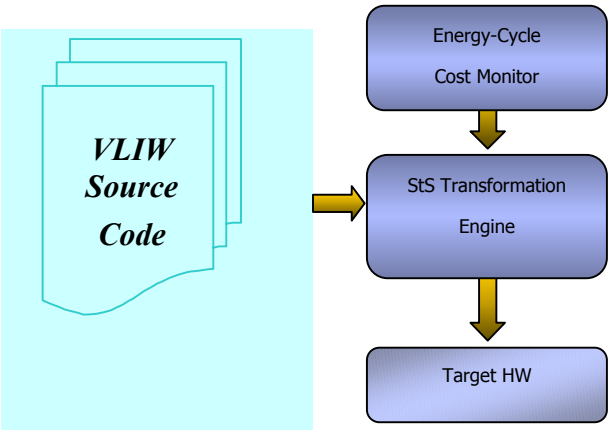

Figure 2.2. Transformation methodology

Loop unrolling is performed to exploit the degree of parallelism available in architecture and is controlled by a preset unrolling factor (*K)*. We compute *K* in each successive transformation based on measured application expression profile (AEP). Thus, to find an optimal *K*, typically iterative simulations are required by modifying the unrolling factor and hence finding an optimum is a complex and time consuming task. Further details are shown in [10].

Decision tree grafting replaces a particular exit of a decision tree with a copy of the destination decision tree. This eliminates a potential branch operation and increases the available parallelism of the current decision tree, at the cost of increased code size.

Block algorithms have better foreground memory such as register and cache reuse; they also provide better cache locality. The tile size is chosen using block algorithms as proposed in [10].

3 Optimization

In this section we define our optimization criteria. Section 3.1 summarizes traditional compilation phases. Section 3.2 introduces the fundamental problems in two most common compilation techniques leading to poor machine code generation.

3.1 Optimization Algorithm

We have two objects for the optimization: cycles per instruction and energy dissipation. For every measure point of the transformation space, two conditions must be satisfied.

First, the successive architecture utilization (in terms of functional units, internal register usage, best cache fit) must be greater than a predefined, system dependent limit (execution cycle and energy threshold).

Second, the predecessor transformation scheme must overlap the successor in order to follow a smooth optimization. The smooth optimization over two samples of code is defined by minimum and maximum limits of transformed code. If the output profile of code is between these limits, this point must lie on smooth curve for optimization.

Further, two properties regarding search space and objective function are demanded: firstly, every point of the search space must be able to be coded as finite length string; secondly, every point of the search space must have a positive fitness described by the objective function. The problem is now to find that number k, k < n, of n transformation possibilities and their yielded code profile (both static and runtime) that maximizes our two targets.

We formulate the above optimization problem as a joint optimization of energy dissipation and cycle per instruction:

$$\begin{aligned} &\min \mathbb{C}(i) \\ &s.t. \Psi_l^{fs}(i) \le \Psi^f(i), \end{aligned} \tag{1}$$

Where $\mathbb{C}(i)$ is the cost of the selected full scheme, and i is the application. Let $q \in_{\|} i$ denote that transformation agent q appears in the previous transformation iteration $\Psi^f(i)$. The cost of transformation is computed as:

$$\mathbb{C}(i) = \sum_{l \in \| p \wedge l \in \Im} \mathbb{C}(l) + \lambda \sum_{r \in \| p \wedge r \in \Re} \mathbb{C}(r) \tag{2}$$

the parameter λ measures the relative cost of transformation versus energy-cycle gain. We use the genetic algorithm (GA) and consider *i* as an individual. The concept of the GA [11, 12, 13] allows for working parallel with many feasible solutions (individuals) by operating between these solutions. Because of

working with many solutions in parallel, it is improbable that the genetic algorithm stalls in any local optimum and thus likely that it finds the global solution. An individual has a certain structure, containing the information of transformation space and previous iteration. The constraints are modeled as a penalty term of the fitness function $\mathbb{C}(i)$. The first term of the fitness function, $0 \le \mathbb{C}(l) \le 1$, denoted the achieved fraction of the CPI[2] for total transformation space. The second term $0 \le \mathbb{C}(r) \le 1$, denotes the fraction of points where the energy dissipation is fulfilled. Coefficient λ is weight factors to the criteria and they define the importance of different criteria with respect to each other, e.g. if λ=1 the method optimizes only energy dissipation.

The individual sample points in transformation space are chosen with a uniform probability distribution. They are profiled later by evaluation the application expression at the target architecture. The selected individual transformations are updated based on their success, i.e. CPI and energy saving factor of the sequence as a whole. Transformations contributing to better performance are rewarded while those resulting in performance losses are penalized. Thus, future sample points are more likely to include previously successful transformations more frequently and search their neighborhood more intensively.

4 Methodology

This section quantifies the overhead introduced by our application expression profiler. The goal of profiling is to find cause-effect relations between performance phenomena and finally generating an architecture efficient code. Since this approach depends on the instrumentation of code on various levels, we need to verify that the phenomena we observer are not caused by our instrumentation. Section 4.1 describes the impact of monitor instrumentations. The code perturbation caused by the introduction of monitor marker is discussed in Section 4.2

4.1 Profile Monitoring Overheads

This section demonstrates that our implementation of the application expression profile is fast enough to be useful. Figure 4.1, summarizes profiling overhead. For each benchmark, the best of ten runs was chosen. All times are in seconds. The column labeled "Baseline code" lists the total time needed to execute the benchmark without any profiling. The two columns denoted "Profiling ON" show the total time needed to execute the benchmark with profiling, and the overhead as a percentage of the baseline run. The overhead for profiling ranges from as little as 1.4% for m100 to as much as 6.2% for MPEG-1 application.

During this measurement all software performance monitors were enabled. Often a user might be interested in only a small subset of the monitoring results, and would thus be able to reduce overhead even more.

[2] CPI is obtained from number of instruction executed and total number of execution cycles.

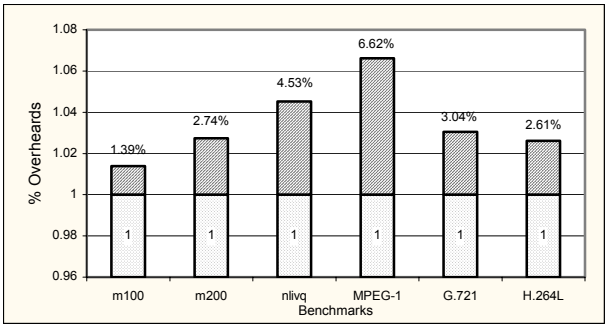

Figure 4.1. Application Expression Profile Overhead

4.2 Application Perturbation Analysis

The inevitable instrumentation of SPMs to the application can give rise to perturbation, the very behavior that it is trying to understand. E.g., instrumentation for collecting data on a software performance monitor may change the cache behavior of the application. The purpose of this perturbation analysis is to assure that the data collection is not perturbing the behavior of interest. The instrumentation required to capture SPMs may also perturb the system behavior. To verify that the mechanism for recording SPMs is not perturbing HPMs, we visually compare the HPM signals collected with and without SPM tracing enabled. If the signals visually correlate, then we have some confidence that SPMs are not changing HPMs in a significant way.

4.3 Benchmarks

Table 1, presents our benchmark suite, which consists of a modified version of the MediaBench suite [14]. Our set of applications contains computer-intensive DSP kernels as well as applications composed of more complex algorithms and data structures.

Table 1. DSP benchmark applications

Applications	Description
m100	Matrix 100x100 multiplications
m200	Matrix 200x200 multiplications
nlivq	Non linear interpolative vector quantization
MPEG-1	MPEG video transcodec
G.721	ADPCM Speech codec
H.264L	Mpeg-4 H.264L Video compression codec

5 Results and Discussion

We have integrated our version of the transformation methodology into our native compiler environment [8] and have evaluated its ability to restructure applications compiled with aggressive transformations. We use benchmark applications as enlisted in Table 1.

Table 2. H-264L , average static code metrics [5]

Metrics	Values
Files	145
Functions	929
Lines of code	50275
Average Cyclomatic complexity	40
Average Modified complexity	40
Average Strict complexity	42
Average Essential complexity	5

Table 3. H.264L profile for successive transformations.

Relative Measures	Iter-1	Iter-2	Iter-3	Iter-4	Iter-5
codeSize	14%	25%	12%	24%	65%
executionTime	-5%	-13%	-17%	-45%	-60%
EnergyConsumption	-1%	-7%	-4%	-13%	-17%
slotUtilization	15%	17%	49%	41%	67%
schedulingFactor	4%	4%	9%	15%	33%
highwayUsage	85%	165%	201%	290%	297%
instructionCacheMiss	-3%	-8%	-8%	10%	-10%
dataCacheMiss	-1%	-12%	-18%	-18%	-36%
dataCacheConflict	5%	25%	27%	21%	17%

To evaluate performance, we first compiled the benchmark with the native compiler with all optimizations turned on. This baseline code is further examined for successive transformation. We obtained the application expression profile during different stages. Table 2 and Table 3 show such profile monitors (both HPMs and SPMs) for example H.264L application. Note that our focus in this paper is on optimizing the application execution time and dissipated energy simultaneously. Each transformation iteration (iter-1 to iter-5) shown in the table corresponds to percent relative change to the original profile for the baseline version of H-264L. E.g., each successive iteration give rise to code size from 14% to 65%, while first iteration has reduced the execution time by 5% (see -5% in Iter-1 column). Similarly energy consumption is also decreased by 17 %.

We observe that the performance improvement in terms of timing is correlated positively with the energy reduction. From Table 3 we see the variation of slot utilization and scheduling factor causes the similar variation in execution time. Similarly instruction and data cache miss leads to variation in energy consumption. Loop unrolling improves the program execution by increasing instructions level parallelism thus increasing size correspondingly. For H.264L example, execution time is reduced by 60% with a significant decrease in energy consumption, i.e., 17%.

Functional unit utilization exploits parallelism and is shown here as slot utilization. While each loop tiling has increased misses in both cache, and increases data cache conflicts (iter-3 to iter-4). Moreover an implicit improvement in energy can be observed due to the impact of decision block grafting made on the scheduling factor. An optimal grafting depth (see Section 2.3) can cause the scheduling factor to grow higher with a benefit of better data/address highway usage and slot utilization as shown in iter-4 to iter-5.

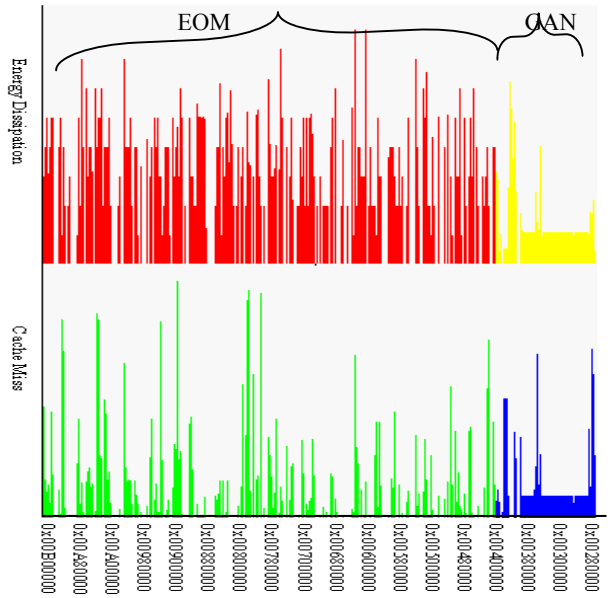

Figure 5.1.(a) Cache miss (bottom) and corresponding energy dissipation (top) – before transformation

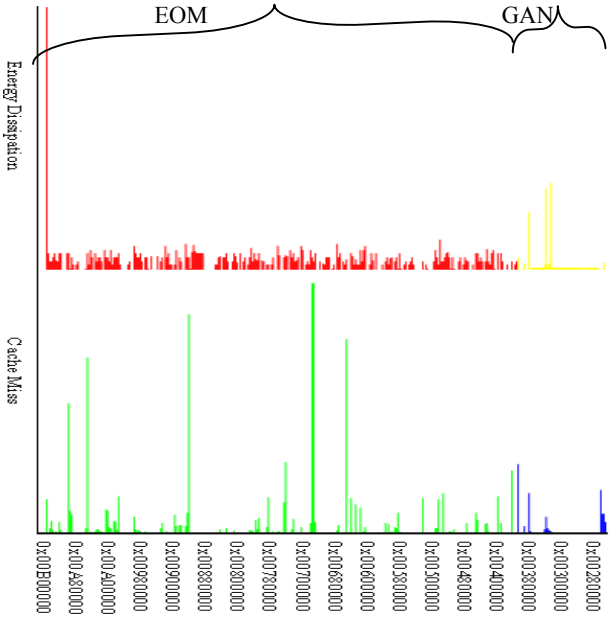

Figure 5.1.(b) Cache miss (bottom) and corresponding energy dissipation (top) – after transformation

Taking the fact into account that off-chip memory traffic leads to higher energy consumption, we developed a tool to visualize the energy consumption corresponding to each cache miss at function level. This helps to identify hot spots in a function that are memory hogging. Figure 5.1, depicts such relation in two consecutive functions in H.264L application during execution.

Function getAffNeighbour(GAN) ends execution at 0x0038000, and next function encode_one _macroblock (EOM) starts executing.. The codec was tested for video segmentation 'flowergarden'. Higher spikes can be observed in the address range 0x00600000 to 0x00700000 due to cache misses. A higher miss rate increase off-chip traffic which leads to a significant increase in energy dissipation and is depicted in Figure 5.1(a).

6 Conclusions

We strongly believe that architecture aware application level vertical profiling is strictly required for efficient embedded applications implementation. In this article we introduced an architecture-based parametric optimization for iterative compilation. Our approach captures behavioral information about multiple layers of a system and correlates that information to find the optimal code scheme. We use a GA to prune the optimization space. We examined the performance improvement across typical MediaBench applications, candidate applications include m100, m200, nlivq, MPEG-1, G-721 and H-264L. Experimental results are important for developing a general methodology for energy-aware embedded DSP software since low power is critical to complex DSP applications in many cost sensitive markets.

Acknowledgement

The author is grateful to Prof. Dr. Atta-ur-Rahman FRS, Prof. Dr. Markus Rupp, Prof. Dr. Arpad Scholtz and Christian Doppler laboratory at Institute of Communication and Radio-Frequency Engineering, Vienna University of Technology for their financial support and kind input on the subject of this work.

References

[1] V. Zivojnovic, J. Velarde, C. Schlager, H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," Proc. of the International Conference on Signal Processing Applications & Technology (ICSPAT '94), Dallas, TX, USA, pp. 715–720, Oct. 1994.

[2] G. Fursin, M. O'Boyle, P. Knijnenburg, "Evaluating iterative compilation," Proc. of Languages and Compilers for Parallel Computers (LCPC'02), College Park, MD, USA, 2002.

[3] N. Zafar Azeemi, "Power Aware Framework for Dense Matrix Operations in Multimedia Processors," Proc. Of the IEEE 9th International Multi-topic Conference, Dec. 2005.

[4] http://hissa.ncsl.nist.gov/sw_assurance/strtest.html

[5] W. Ye, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, ''The design and use of SimplePower: A cycle-accurate energy estimation tool,'' Proc. of the Annual ACM IEEE Design Automation Conference, pp. 340-345, June 2000.

[6] TM1300 Data Book, Philips Electronic, North America Corporation, pp. 3.1-3.16, Oct 1999.

[7] David B. Loveman, "Program improvement by source to source transformation," Proc. of the Annual ACM SIGACT-SIGPLAN symposium on Principles on programming languages, pp.140-152, 1976.

[8] C. H. Gebotys, R. J. Gebotys, "An empirical comparison of algorithmic, instruction, and architectural power prediction models for high performance embedded DSP processors," Proc. of the International Symposium on Low Power Electronics and Design, 1998.

[9] V.Tiwari, S.Malik, A.Wolfe, "Compilation techniques for low energy," Proc. of the ISLPED, Oct 1994.

[10] N. Zafar Azeemi, "A Framework for Architecture Based Energy-Aware Code Transformations in VLIW Processors," Proc. Of the IEEE International Symposium on Telecommunications (IST 2005) pp.393-398. Sep. 2005.

[11] T. Baeck. Evolutionary Algorithms in Theory and Practice. Oxford University Press, 1996.

[12] S. Bashford and R. Leupers, "Constraint driven Code Selection for Fixed-Point DSPs," Proc. of the 36th Design Automation Conference (DAC), Nov. 1999.

[13] M. Lorenz, T. Draeger, R. Leupers, P. Marwedel, G. P. Fettweis, "Low-Energy DSP Code Generation Using a Genetic Algorithm," Proc. of the IEEE International Conference on Computer Design 2001, Austin, Texas, Jan. 2001.

[14] http://cares.icsl.ucla.edu/MediaBench