# 4 Embedded Systems Code Optimization and Power Consumption

*Mostafa E.A. Ibrahim and Markus Rupp*

## CONTENTS



## 4.1 INTRODUCTION

In a growing number of complex heterogeneous embedded systems, the relevance of software components is rapidly increasing. Issues such as development time, flexibility, and reusability are, in fact, better addressed by software-based solutions. Due to the processing regularity of multimedia and DSP applications, statically scheduled devices such as VLIW processors are viable options over dynamically scheduled processors such as state-of-the-art superscalar GPPs.

The programs that run on a particular architecture will significantly affect the energy usage of a processor. The manner in which a program exercises certain parts of a processor will vary the contributions of individual structures to the total energy consumption of the processor. Minimizing power dissipation may be handled by hardware or software optimizations; in hardware through circuit design, and in software through compile-time analysis and code reshaping.

While hardware optimization techniques have been the foci of several studies and are fairly mature, software approaches to optimizing power are relatively new.

Progress in understanding the impacts of traditional compiler optimizations on power consumption and developing new power-aware compiler optimizations are important to the overall energy optimization of systems.

In this chapter, we investigate the impacts of applying high level language optimization techniques on power consumption of embedded systems. Using the example of the Code Composer Studio (CCS) C/C++ compiler, we evaluate the impact of its global performance optimization options -o0 to -o3 on power consumption. Furthermore, we explore the effects of utilizing two architectural features of the targeted processor; namely the software pipelined loop (SPLOOP) and single instruction multiple data (SIMD) capabilities from a power consumption perspective.

The currently available compiler optimization techniques target either execution time or code size and may improve power consumption only indirectly. These techniques are handicapped for power optimization due to their partial perspectives of the algorithms and their limited modifications to the data structures. Other software optimization techniques like source code transformations can exploit the full knowledge of algorithm characteristics and also modify both data structures and algorithm coding. Interprocedural optimizations are also envisioned. Hence, we investigated several loop, data, and procedural source code transformations from a power consumption perspective.

In Section 4.2, we introduce power-aware optimization methodologies for embedded systems in software. Section 4.3 provides a brief overview of the target architecture along with the experimental setup employed in our experiments. Section 4.4 describes the methodology for measuring power consumption while applying different optimization techniques. Section 4.5 presents the results of investigating global compiler optimization options, specific architectural features, and the impacts of source code transformations on power consumption. Finally, Section 4.6 summarizes the main points of this chapter.

## 4.2 POWER-AWARE OPTIMIZATION

The reduction of power consumed during the running of software on a microprocessor can be addressed at compiler, low, or high level language levels. All the mechanisms present advantages and disadvantages, depending on the target processor and architecture [1]. This section briefly presents the most recent contributions related to optimizing the power consumption of embedded systems from a software perspective.

### 4.2.1 Compiler-Based Optimization

Recently, some attempts to investigate the impact of applying standard compiler speed optimization levels to the power and energy consumptions of programmable processors have been introduced. Valluri et al. [2] evaluated some general and specific optimizations based on the power and energy consumption of the DEC Alpha 21064 processor while running SpecInt95 and SpecFp95 benchmarks [3].

Ravindran et al. [4] proposed an approach for compiler-directed dynamic placement of instructions into a low-power code cache. They showed that by

applying dynamic placement techniques, energy savings may be achieved on the WIMS microcontroller platform. Chakrapani et al. [5] also presented a study of the effect of compiler optimization on energy use by an embedded processor. Their work targeted an ARM embedded core and they used an RTL level model with a Synopsys Power Compiler to estimate power.

Seng et al. [6] reviewed the effects of the Intel compiler general and specific optimizations on energy and power consumption of a Pentium 4 processor running benchmarks extracted from Spec2000 [3]. Zambreno et al. [7] studied power consumption on portable devices. They analyzed the effects of compiler optimizations on memory energy. Based on their results, they concluded that the best optimization approach may not give the best results for power use. Also, they observed that function inlining increased power consumption of the system, unlike loop unrolling that showed a decrease in energy. It is clear that the authors interchangeably used the *power* and *energy* terms.

Casas et al. [8] studied the effects of various compiler optimizations on the energy and power use of the low power C55 DSP of Texas Instruments (TI). Their work did not consider the effects of compiler optimizations on many performance measures such as memory referencing and instructions per cycle that significantly affect power and energy use.

Esakkimuthu et al. [9] compared hardware and software optimizations analyzing cache optimization mechanisms and compiler optimization techniques to lower power consumption. Their results showed that compiler optimizations outperformed cache optimizations in terms of energy savings. Lee et al. [10] investigated compiler transformation techniques to schedule VLIW instructions, aiming to reduce the power consumption of VLIW architectures on the instruction bus. Their experiments showed a noticeable enhancement with four- and eight-way issue architectures for power consumption of the instruction bus as compared to the conventional list scheduling technique.

Kandemir et al. [11] studied the influence of five high level compiler-based optimizations such as loop unrolling and loop fusion on energy consumption using the SimplePower cycle-accurate energy simulator [12], a source-to-source code translator, and a number of benchmark codes. Mehta et al. [13] proposed a compilation technique for lower power consumption based on the energy consumed by the instruction register (IR) and register file decoder.

Leupers [14] analyzed different techniques to design power-efficient compilers and presented a set of software optimization techniques for compiler code generation. Oliver et al. [15] analyzed factors that affected power consumption at the instruction level such as cycles, branches, and instruction reordering. Their experiments demonstrated that software optimization at instruction level is a good approach to minimize energy dissipation in embedded processors.

### 4.2.2 Source-to-Source Code Optimization

Most of the software-oriented proposals for power optimization focus on instruction scheduling and code generation, possibly to minimize memory access cost [16]. As expected, standard low level compiler optimizations such as loop unrolling

and software pipelining also promote energy reduction since they reduce code execution time. However, a number of cross-related effects cannot be identified clearly and are generally difficult to apply to compilers unless some suitable source-to-source restructuring of the code is applied a priori.

Ortiz et al. [17] investigated the impacts of different code transformations—loop unrolling, function inlining, and variable type declarations—on power consumption. They chose three platforms as targets: 8- and 16-bit microcontrollers and a 32-bit ARM7TDMI processor. Their results show that loop unrolling exerted a significant impact on the consumed power with the 16- and 32-bit processors.

Azeemi et al. [18] examined the effects of loop unrolling factor, grafting depth, and blocking factor on the energy and performance of the Philips Nexperia media processor PNX1302. However, they used the *energy* and *power* terms interchangeably. Hence the improvements in energy obtained from their work are directly related to performance enhancements.

Brandolese et al. [19] stressed the state-of-the-art source-to-source transformations to discover and compare their effectiveness from power and energy perspectives. The data structure, loop and inter-procedural transformations were investigated with the aid of a GCC compiler. The compiled software codes were then simulated with a framework based on the SimpleScaler [20]. The simulation framework was configured with a 1-kByte two-way set-associative unified cache.

Catthoor et al. [21] showed the crucial role of source-to-source code transformations in solving the data transfer and storage bottlenecks in modern processor architectures. They surveyed many transformations aimed at enhancing data locality and reuse.

Kulkarni et al. [22] improved software-controlled cache utilization to achieve lower power requirements for multimedia and signal processing applications. Their methodology took into account many program parameters such as data locality, sizes of data structures, access structures of large array variables, regularity of loop nests, and the size and type of cache. Their objective was to improve cache performance at lower power. The targeted platforms for their research were embedded multimedia and DSP processors.

In the same way, McKinley et al. [23] investigated the impacts of loop transformations on data locality. Yang et al. [24] studied the impacts of loop optimizations on performance and power tradeoffs. They utilized the Delaware Power-Aware Compilation Testbed (Del-PACT), an integrated framework consisting of a modern industry-strength compiler infrastructure and a state-of-the-art micro-architecture level power analysis platform. Low level loop optimizations at the code generation (back end) phase (loop unrolling and software pipelining) and high level loop optimizations at the program analysis and transformation phase (front end, loop permutation and tiling) were studied.

## 4.3 SOFTWARE AND HARDWARE PLATFORMS

The hardware architecture utilized in this work was the TMS320C6416T fixed-point VLIW DSP from Texas Instruments (TI). This DSP is considered one of the highest performance fixed-point devices [25]. Although the targeted architecture operating

**TABLE 4.1**
**Features of Global Performance Optimization Options**

| Optimization | Features |
|---|---|
| -o0 | Performs control-flow-graph simplification, allocates variables to registers, performs loop rotation, eliminates unused code, simplifies expressions and statements, expands calls to functions declared inline |
| -o1 | All -o0 optimizations; also performs local copy/constant propagation, removes unused assignments, eliminates local common expressions |
| -o2 | All -o1 optimizations; also performs software pipelining and loop optimizations, eliminates global common sub-expressions and unused assignments, converts array references in loops to incremented pointer form, performs loop unrolling |
| -o3 | All -o2 optimizations; also removes all functions that are never called, simplifies functions with return values that are never used, inline calls to small functions, reorders function declarations so that attributes of called functions are known when caller is optimized, propagates arguments into function bodies when all calls pass same value in same argument position, identifies file-level variable characteristics |

frequency ranged from 500 to 1200 MHz, in our set-up, the operating frequency was adjusted to 1000 MHz and the DSP core voltage was 1.2 V.

All measurements were carried out on the TMS320C6416T DSP Starter Kit (DSK) manufactured by Spectrum Digital Inc. The three power test points on this DSK are DSP I/O current, DSP core current, and system current [26]. The current drawn by the DSP core while running an algorithm was captured by the Agilent 34410A 6.5-digit digital multi-meter (DMM). This meter features very high DC basic accuracy (0.003% of the reading plus 0.003% of the range) [27].

The current is captured as a differential voltage drop across a 0.025 Ω sense resistor inserted by the manufacturer in the current path of the DSP core. The input differential voltage drop is divided by 0.025 Ω to determine the current drawn.

The software platform for generating the software binaries to be loaded to the DSP was the embedded C/C++ compiler (Version 6.0.1) in the Code Composer Studio (CCS 3.1) from TI. This compiler accepts C and C++ code meeting the International Organization for Standardization (ISO) standards for these languages and produces assembly language source code for the C6416T device. The compiler supports the 1989 version of the C language.

This compiler features many levels of optimization mainly tuned for speed and/or code size as shown in Table 4.1. A user can invoke the -o0, -o1, -o2, and -o3 options for global speed optimization [28].

## 4.4 METHODOLOGY AND APPLICATIONS

Let a program $X$ run for $T$ seconds to achieve its goal. $V_{CC}$ is the supply voltage of the system, and $I$ denotes the average current (amperes) drawn from the power source for $T$ seconds. Consequently, $T$ can be rewritten as $T = N \times \tau$ where $N$ is the number of clock cycles and $\tau$ is the clock period. The power consumed by running program

$X$ is calculated as $P = V_{CC} \times I$. The amount of energy consumed by $X$ to achieve its goal is given as $E = P \times N \times \tau$ joules.

Since $V_{CC}$ and $\tau$ are fixed for specific hardware, $E \propto I \times N$. However, at the application level, it is more meaningful to consider $T$ instead of $N$, and therefore energy is expressed as $E \propto I \times T$. This expression shows the main idea in the design of energy-efficient software: reduction of both $T$ and $I$. From running time (average case) of an algorithm, a measure of $T$ is achieved. However, to compute $I$, one must consider the average current drawn during execution of a program.

In this chapter, several image and signal processing benchmarks were used to investigate the impacts of invoking different compiler optimization levels on power and energy. The CCS3.1 profiler was employed to profile the benchmarks.

For assessing the effects of SIMD instructions on energy and power consumption, we prepare two versions of the inverse discrete cosine transform (IDCT) algorithm, the discrete cosine transform (DCT) algorithm, and the median filter with a $3 \times 3$ window. The first version was implemented without using any SIMD instructions. The second was implemented with all possible SIMD instructions. The functionalities of both versions were tested and verified to yield the same results.

To study the impacts of applying different source code transformations, we first decided the source code transformations to be investigated. Second, we prepared the suitable software kernel to allow the employment of each code transformation. The functionality agreements between the original and transformed kernels were verified. Then the performance, power, and energy results of the original and transformed kernels were compared to analyze the impacts of the transformations. Several code transformations were investigated. Due to space constraints, we focus on the results for array declaration sorting, loop peeling, and procedure integration.

## 4.5 CODE OPTIMIZATION IMPACT ON POWER CONSUMPTION

In this section, we explore and analyze the experimental results of evaluating the impacts of various optimization techniques on power consumption.

### 4.5.1 Impact of Global Compiler Optimization

This section discusses the different compiler optimization levels offered by the CCS3.1 along with the individual optimization tasks performed at each level. The targeted C/C++ compiler features many levels of optimization tuned to achieve speed by invoking the -o0, -o1, -o2, and -o3 options for global speed optimization [28].

We start by evaluating the effects of the four global compiler optimization levels (-o0, -o1, -o2, -and o3) on the energy and power consumption of the targeted processor. Moreover, we analyzed the effects on other performance measures such as memory references and cache miss rates. Figure 4.1 presents the measured power consumptions and averages at the four global performance optimization levels for different signal and image processing benchmarks. It is obvious from the figure that these optimization options generally increase power consumption.

The most aggressive optimizations they may lead to minimum execution times but do not necessarily achieve the best code for minimal power consumption.

The highest optimization level (-o3) increased power consumption on average by 30.3% compared to the no-optimization option [29]. This percentage reached 45% for two individual benchmarks (FDCT8×8_i and correlation_3×3). On average, invoking -o2 or -o3 led to greater power consumption than invoking -o0 or -o1.

The software pipelining loop feature enabled with -o2 and -o3 allowed better instruction parallelization. As we will explain later, this feature had a significant impact on power consumption.

Although the results in Figure 4.1 demonstrate that invoking global optimization levels increases power consumption on average, the energy significantly decreased. Table 4.2 demonstrates that there is a strong correlation between execution time and energy consumption.

The most aggressive speed optimization level (-o3) reduced execution time on average by 96.2% compared to the no-optimization option. While –o3 reduces the energy on average by 94.8%, it is obvious that invoking -o2 and -o3 saves more energy than invoking -o0 or -o1. This can be explained by the enabling of the Software Pipelined Loop (SPLoop) at -o2 and -o3 that leads to far greater reduction

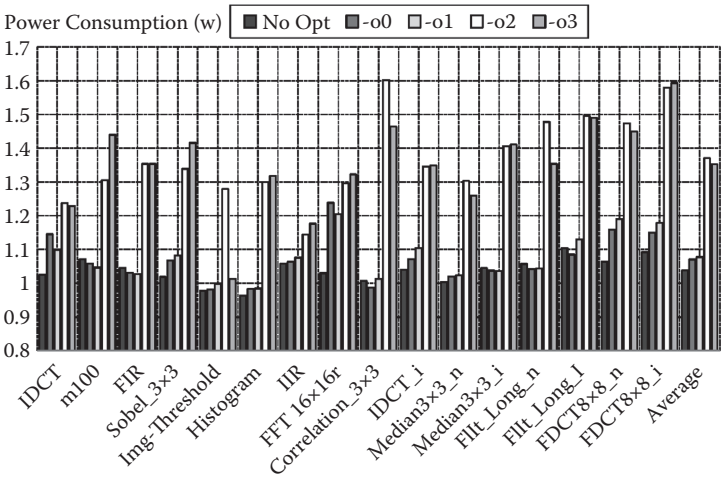


**FIGURE 4.1** Power consumption of C6416T while running different benchmarks.

**TABLE 4.2**
**Effects of Employing Global Optimization Options on Power, Energy and Execution Time**

| Optimization | Power (%) | Execution Time (%) | Energy (%) |
|---|---|---|---|
| -o0 | 3.11 | –27.35 | –27.99 |
| -o1 | 3.841 | –22.32 | –23.58 |
| -o2 | 32.16 | –91.17 | –89.06 |
| -o3 | 30.36 | –96.20 | –94.85 |

in execution time and hence in energy use. The effect of SPLoop will be covered later in this chapter.

Based on the results shown in Table 4.2 we can distinguish two groups of optimization levels (-o0, -o1) improved by register allocation, and (-o2, -o3) discriminated by the use of SPLoop). In the (-o0, -o1) group, data were fetched simultaneously with the instruction execution, saving CPU cycles, thus consuming less energy. However, in the case of no optimization, data were pre-fetched from memory before execution of the instruction that required these data. If a program utilizes many variables, the power consumption increases while the energy decreases. Conversely, if a program utilizes few variables it retains an energy decrease through unchanged power consumption because the registers are loaded less frequently and reused more often.

The (-o2,- o3) group is characterized mainly by software pipelined loops. SPLoops save a fixed number of cycles per loop iteration, mostly by avoiding pipeline stalls. Since stalls consume less power than normal instruction executions, shortening the programs this way actually increases power consumption. However, the magnitude of the increase depends on how long the loop kernel is and the instructions it carries. To a lesser extent, the elimination of global common subexpressions further reduces execution time and power consumption.

To analyze the previous results, we found it worthwhile to study the effects of compiler optimizations on four important execution characteristics: L1 data cache misses, memory references, instructions per cycle (IPCs), and CPU stall cycles. Table 4.3 illustrates the effects of different optimization levels on the four characteristics. All the percentage changes in Table 4.3 are obtained w.r.t the case of no optimization.

AU: Define w.r.t. in next sentence.

The L1D cache misses decreased on average by almost 69% when -o3 was invoked. The L1D cache misses required the access of the L2D cache/SRAM which in turn required additional power consumption. The CPU stall cycles decreased by 78% when -o3 was invoked. Several reasons such as cache misses, resource conflicts, and memory bank conflicts can cause a CPU to stall. Although one data cache miss caused at least six CPU stall cycles, two features of the C6416T CPU are expected to decrease cache miss penalties.

The L1D cache of the C6416T DSP pipelines the L1D cache read misses. A single L1D read miss takes six cycles when serviced from L2/SRAM and eight cycles when serviced from the L2 cache. Pipelining of cache misses can hide many miss penalties

**TABLE 4.3**
**Effects of Employing Global Optimization Options on Execution Characteristics**

| Optimization | L1D Cache Misses (%) | CPU Stall Cycles (%) | IPCs (%) | Memory References (%) |
|---|---|---|---|---|
| -o0 | 0.0 | 2.09 | 59.28 | –81.89 |
| -o1 | –1.12 | –1.29 | 59.14 | –81.91 |
| -o2 | –1.12 | –45.05 | 256.5 | –82.69 |
| -o3 | –69.27 | –78.75 | 269.95 | –94.12 |

(CPU stall cycles) by overlapping the processing of several cache misses. The miss overhead can be expressed as [4 + (2 × *M*)] when serviced from L2/SRAM or as [6 + (2 × *M*)] when serviced from L2 (*M* is the number of cache misses) [30]. The pipelining of cache misses significantly reduced execution time and energy but exerted no effect on power consumption.

The write cache miss feature does not directly stall the CPU because of the use of L1D write buffer [30]. This feature also affects execution time but has no effect on power consumption especially when the write buffer is not full.

Table 4.3 indicates that IPCs increased by about 269% when -o3 was invoked compared to the case when all optimization options are disabled. This surely decreases execution time and energy because more overlapping in the execution of the instructions per cycle is achieved. However, the resulting higher parallelization increases the power consumption.

Although Table 4.3 indicates that the memory references decreased by 94% when -o3 was invoked (and expected to save power), we found that the power use increased. This emphasizes our results [31,32] demonstrating that the instruction management unit (IMU) responsible for fetching and dispatching instructions dominates the memory referencing contribution.

### 4.5.2 Effects of Specific Architectural Features

In this section, we explore the impacts of two special architectural features of the targeted DSP: the SPLoop and SIMD on power consumption.

#### 4.5.2.1 Impact of Software Pipelined Loop

The SPLoop, also called the hardware zero overhead loop (ZOL) is a type of instruction scheduling that exploits instruction level parallelism (ILP) across loop iterations. SPLoop is a specific architectural optimization feature of the C64x+ CPU; the C64x CPU does not support SPLoop. This feature allows the CPU to store a single iteration of a loop in a specialized buffer containing hardware that will selectively overlay copies of the single iteration in a software pipeline to construct an optimized execution of the loop [33].

Modulo scheduling is a form of SPLoop that initiates loop iterations at a constant rate called the iteration interval (ii). To construct a modulo scheduled loop, a single loop iteration is divided into a sequence of stages, each with length ii. In the steady state of the execution of the SPLoop, each stage executes in parallel. The instruction schedule for a modulo scheduled loop has three components: a prolog, a kernel, and an epilog. The kernel is the instruction schedule that executes the pipeline steady state. The prolog and epilog are instruction schedules that set up and drain the execution of the loop kernel [33].

This section evaluates the impact of software pipelining on power and energy consumption. The SPLoop feature is implicitly enabled with global optimization options -o2 and -o3. However, we overrode this by invoking the -mu option in conjunction with -o2 or -o3 to disable only the SPLoop. To distinguish whether software pipelining is enabled or disabled, we used -o2-mu and -o3-mu to indicate disabling of the SPLoop.

**TABLE 4.4**
**Average Power, Execution Time, and Energy for Investigated Benchmarks**

| Benchmark | Power (W) | Execution Time (ms) | Energy (mJ) |
|---|---|---|---|
| -o2 | 1.371 | 0.168 | 0.221 |
| -o2-mu | 1.109 | 0.703 | 0.766 |
| -o3 | 1.352 | 0.072 | 0.104 |
| -o3-mu | 1.117 | 0.16 | 0.204 |

Table 4.4 summarizes the average power, execution time, and energy for different signal and image processing benchmarks when -o2, -o2-mu, -o3, and -o3-mu were invoked. The table clearly illustrates the strong impact of the SPLoop on execution time. When the -o2-mu was invoked, the execution time increased by 317.3% relative to -o2. Execution time increased by 120.8% when -o3-mu was invoked relative to -o3. Clearly the impact of disabling the SPLoop on execution cycles was greater with -o2 than with -o3. The -o3 option includes all the individual optimizations in -o2 along with more performance-oriented optimizations intended to reduce execution cycles through reductions in memory references.

Despite the increase in the execution cycles when SPLoop was disabled, the power consumption decreased on average by 19.1 and 17.4%, respectively, when -o2-mu and -o3-mu were invoked. Table 4.4 demonstrates that the energy increases when SPLoop is disabled and the increase relates directly to the increase in execution time.

Power consumption increased on average by 7.67% when -o3-mu was invoked compared to the case of no optimization option. The SPLoop contributed 70.3% to the total power increase when -o3 was invoked. More attention should focus on the design of specialized hardware for software pipelining to achieve a compromise of performance and power trade-offs for the C6416T.

Figure 4.2 summarizes the effects of the SPLoop on power consumption. It is pretty clear that invoking -o3-mu results in a trade-off between performance and power consumption.

#### 4.5.2.2 Impact of SIMD

The C6000 compiler recognizes a number of intrinsic C-functions. Intrinsics allow a programmer to express the meanings of certain assembly statements that would otherwise be cumbersome or inexpressible in C/C++. Most intrinsic functions utilize the SIMD capabilities of the C6416T. Intrinsics are used like functions. A programmer can use C/C++ variables with these intrinsics, just as he or she would with any normal function. The intrinsics are specified with a leading underscore and are accessed by calling them as function. For example: int X1, X2, Y; Y = _sadd(X1, X2). For a complete list of the C6000 and specific C64x+ intrinsic functions readers are encouraged to review Reference [28].

To assess the effects of utilizing SIMD instructions on energy and power consumption, we prepared two versions of the inverse discrete cosine transform (IDCT) algorithm as a case study. The first version was implemented without SIMD

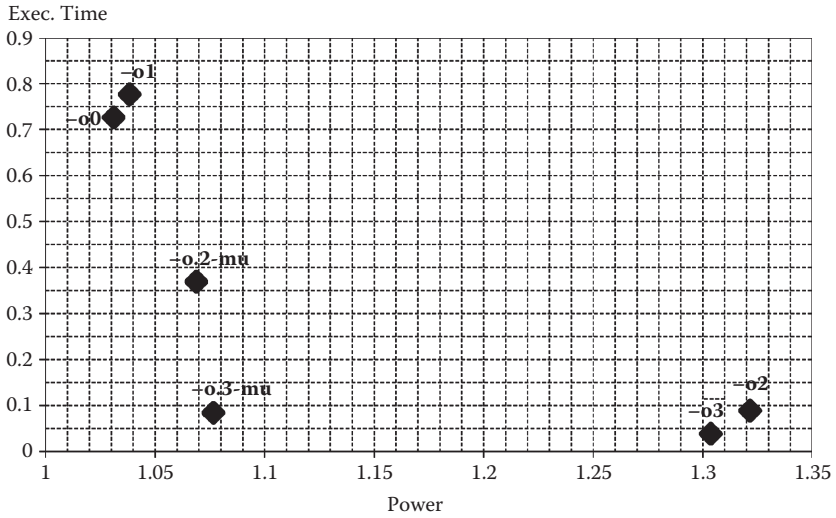


**FIGURE 4.2** Execution time versus power consumption at various optimization levels.

Original Code

```
#include "idct_8x8_c.h"
#pragma CODE_SECTION(idct_8x8_cn,   ".text:ansi");
void idct_8x8_cn(short *idct_data, unsigned num_idcts)
{
    _nassert((int) idct % 8 == 0);
    _nassert(num_idcts >= 1);
    for (i = 0; i < num_idcts; i++)
    {
        for (j = 0; j < 8; j++)
        {
            F0 = idct[i][0][j];
            F1 = idct[i][1][j];
            F2 = idct[i][2][j];
            F3 = idct[i][3][j];
            F4 = idct[i][4][j];
            F5 = idct[i][5][j];
            F6 = idct[i][6][j];
            F7 = idct[i][7][j];

            P0 = F0;                P1 = F4;
            R1 = F2;                R0 = F6;

            Q1 = (F1*C7 - F7*C1 + 0x8000) >> 16;
            Q0 = (F5*C3 - F3*C5 + 0x8000) >> 16;
            S0 = (F5*C5 + F3*C3 + 0x8000) >> 16;
            S1 = (F1*C1 + F7*C7 + 0x8000) >> 16;

            p0 = ((int)P0 + (int)P1 + 1 ) >> 1;
            p1 = ((int)P0 - (int)P1     ) >> 1;
            r1 = (R1*C6 - R0*C2 + 0x8000) >> 16;
            r0 = (R1*C2 + R0*C6 + 0x8000) >> 16;
```

Code with Intrinsics

```
#include "idct_8x8_i.h"
#pragma CODE_SECTION(idct_8x8_c,   ".text:intrinsic");
void idct_8x8_c(short *idct_data, unsigned num_idcts)
{
    _nassert((unsigned)idct_data % 8 == 0);
    #pragma MUST_ITERATE(4,,4);
    for (i = j0 = j1 = 0; i < num_idcts * 4; i++)
    {
        F00   = _amem4(&_ptr[ 0 + 2*j0]);
        F11   = _amem4(&_ptr[ 8 + 2*j0]);
        F22   = _amem4(&_ptr[16 + 2*j0]);
        F33   = _amem4(&_ptr[24 + 2*j0]);
        F44   = _amem4(&_ptr[32 + 2*j0]);
        F55   = _amem4(&_ptr[40 + 2*j0]);
        F66   = _amem4(&_ptr[48 + 2*j0]);
        F77   = _amem4(&_ptr[56 + 2*j0]);

        if (++j0 == 4) { j0 = 0; i_ptr += 64; }

        F17   = _pack2(F11, F77);
        F53   = _pack2(F55, F33);
        F26   = _pack2(F22, F66);
        F04   = _pack2(F00, F44);

        Q1    = _dotprsu2(F17, C71);
        Q0    = _dotprsu2(F53, C35);
        S0    = _dotprsu2 (F53, C53);
        S1    = _dotprsu2 (F17, C17);

        SQQ0  = _pack2(S0, Q0);
        S1Q1  = _pack2(S1, Q1);
```

**FIGURE 4.3** Example of IDCT kernel with and without SIMD utilization.

instructions; the second was implemented with all possible SIMD instructions as shown in Figure 4.3. The functionalities of both versions were tested and verified to yield the same results.

We studied the effect of the employing SIMD instructions isolated from the effects of the SPLoop feature by compiling two versions with -o2-mu and -o3-mu

(-mu disables the SPLoop feature). It is worth mentioning that invoking -o0 or -o1 will not enable the SPLoop.

Table 4.5 demonstrates that the SIMD version of the IDCT compiled and optimized with -o3-mu achieved a 3.96% power saving along with 25.4 and 28.35% reductions in execution time and energy, respectively. The achieved power saving is mainly the result of reduction of the IPC by 20.86%. The enhancement of execution time was derived by a significant reduction (more than 62%) on memory references.

To summarize the effects of utilizing SIMD on power and energy consumption, we conducted two more case studies utilizing a discrete cosine transform (DCT) and a median filter with a 3 × 3 window in the same manner as the IDCT investigation. Figures 4.4 and 4.5 represent comparisons of power consumption and energy with and without SIMD at various performance optimization levels.

In general, employing SIMD significantly enhanced performance and energy saving. The SPLoop was the main basis for the significant improvement in performance when -o2 or -o3 was invoked [35]. By disabling the SPLoop feature, -o2-mu or -o3-mu, the utilization of SIMD produced a comparable performance enhancement

**TABLE 4.5**
**Effects of Employing SIMD While Invoking -o3-mu Optimization**

| | Original | With Intrinsics | % |
|---|---|---|---|
| Execution cycles | 3319 | 2476 | −25.4 |
| Power (W) | 1.091 | 1.048 | −3.96 |
| Energy (mJ) | 0.00362 | 0.00259 | −28.35 |
| IPCs | 2.416 | 1.913 | −20.86 |
| CPU stall cycles | 96 | 0 | −100 |
| Memory references | 1536 | 576 | −62.5 |

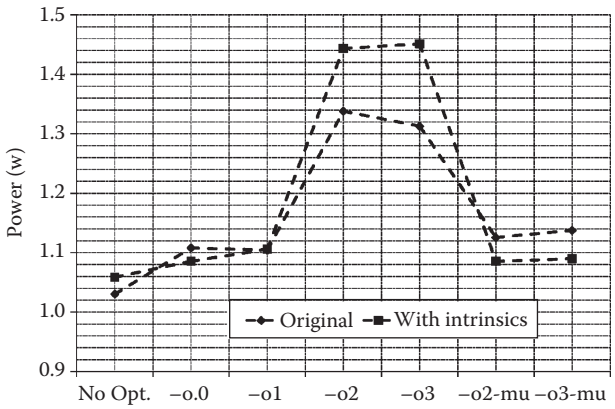


**FIGURE 4.4** Power consumption with and without SIMD utilization versus various optimization options.

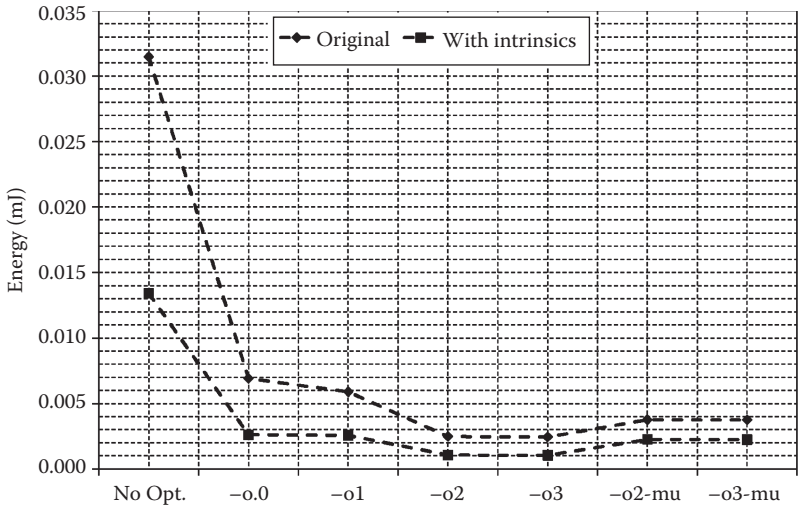


**FIGURE 4.5** Energy with and without SIMD utilization versus various optimization options.

with -o2 or -o3 along with the great advantage of average power savings of 18.83 and 17%, respectively.

It is clear that rewriting the algorithm to yield maximum utilization of SIMD instructions while invoking the –o3 optimization options was the best choice from power consumption and performance perspectives. This choice may be considered a trade-off between power consumption on one side and execution time and energy use on the other.

### 4.5.3 Influence of Source-to-Source Code Transformations

Code reshaping techniques, also known as source-to-source code transformations, consist of loop and data flow transformations. They are essential to modern optimizing and parallelizing compilers. They enhance temporal and spatial locality for cache performance and expose an algorithm's inherent parallelism to loop nests [21].

In this section, we assess the impacts of applying source to source code transformations from power and energy perspectives. The source code transformations presented in this chapter are classified into loop, data, and procedural transformation groups.

To evaluate the effectiveness of the applied transformations, we compiled the original and transformed versions of each program on the target architecture. We recorded the current drawn (consumed power) from the core CPU. With the aid of a CCS3.1 profiler, we recorded execution times and other execution characteristics such as memory references, L1D cache misses, and others. To obtain reliable and precise information, we repeated the measuring procedure several times for each transformation [32].

First we assessed the effects of loop peeling transformations. This type of transformation, also called loop splitting, attempts to eliminate or reduce loop

dependencies introduced in a few first or last iterations by splitting them from the loop and performing them outside the loop to allow better instruction parallelization. Moreover, the transformation can be used to match the iteration control of adjacent loops, allowing two loops to be fused together.

Figure 4.6 shows an example of loop peeling transformation. In the original code of this example, the first iteration makes use only of the variable $p = 10$; for all other iterations, the variable is $p = i - 1$. Therefore, the first iteration in the transformed code is moved outside the loop and the loop iteration control is modified.

Table 4.6 shows the impact of applying the loop peeling transformation on power, energy, and execution time. Because of splitting the first iteration from the loop's body and performing it outside the loop, the memory references decreased by 37.78% while maintaining the same number of L1D cache misses. The instruction parallelization expressed as IPCs, improved by 4.6%. The execution time and power consumption were enhanced by 11.5 and 2.78%, respectively, leading to an energy saving of 13.97%.

AU: [] symbols okay?

Second, we present array declaration sorting transformations as examples of data-oriented transformations. Figure 4.7 shows an example in which the array access frequency ordering is C[], B[] and A[]. The A[], B[], and C[], declaration order in the original code is restructured placing C[] in the first position, B[] in the second, and A[] at the end. This declaration reordering is employed to assure that frequently accessed arrays are placed on top of the stack. In this way, the frequently used memory locations are accessed by exploiting direct access mode.

| Original Code | Transformed Code |
|---|---|
| `int p = 10;`<br>`for (i=0; i<N; ++i)`<br>`{`<br>`  y[i] = x[i] + x[p];`<br>`  p = i;`<br>`}` | `y[0] = x[0] + x[10];`<br>`for (i=1; i<N; ++i)`<br>`{`<br>`  y[i] = x[i] + x[i-1];`<br>`}` |

**FIGURE 4.6** Loop peeling transformation.

**TABLE 4.6**
**Effects of Loop Peeling Transformation on Energy and Power Consumption**

| | Original | Transformed | % |
|---|---|---|---|
| Execution cycles | 2808 | 2485 | −11.5 |
| Power (W) | 1.034 | 1.006 | −2.78 |
| Energy (mJ) | 0.0029 | 0.0025 | −13.97 |
| IPCs | 0.919 | 0.962 | 4.6 |
| Memory references | 802 | 499 | −37.78 |

| Original Code | Transformed Code |
|---|---|
| `int A[DIM], B[DIM], C[DIM], i;`<br>`  for (i = 5; i < 3500; i+=5)`<br>`  C[i] = val;`<br>`  for (i = 5; i < 2000; i+=10)`<br>`  B[i] = val;`<br>`  for (i = 5; i < 1000; i+=10)`<br>`  A[i] = val;`<br>`}` | `int C[DIM], B[DIM], A[DIM], i;`<br>`  for (i = 5; i < 3500; i+=5)`<br>`  C[i] = val;`<br>`  for (i = 5; i < 2000; i+=10)`<br>`  B[i] = val;`<br>`  for (i = 5; i < 1000; i+=10)`<br>`  A[i] = val;` |

**FIGURE 4.7** Array declaration sorting transformation.

The array declaration sorting reduces execution time by 1.95% and consequently saves energy by 2.19%. Power consumption is little affected so this transformation is not power hungry.

Third, we studied the impact of procedure integration, also known as procedure inlining. Procedure integration replaces calls to procedures with copies of their bodies [36]. It can be a very useful optimization because it changes calls from opaque objects that may exert unknown effects on aliased variables and parameters to local code that exposes its effects and may be optimized as part of the calling procedure [37].

Although procedure integration removes the costs of the procedure call and return instructions, the savings are often small. The major savings arise from the additional optimizations that become possible on the integrated procedure body. For example, a constant passed as an argument can often be propagated to all instances of the matching parameter. Moreover, the opportunity to optimize integrated procedure bodies can be especially valuable if it enables loop transformations that were originally inhibited by embedding procedure calls in loops or turning a loop that calls a procedure (whose body is itself a loop) into a nested loop [37].

Ordinarily, when a function is invoked, the control is transferred to its definition by a branch or call instruction. With procedure integration, the control flows directly to the function code without a branch or call instruction. Moreover, the stack frames for the caller and called are allocated together. Procedure integration may make the generated code slower as well, for example, by decreasing the locality of the reference. Figure 4.8 shows an example of procedure integration. The *pred(int)* function is integrated into the *f(int)* function.

Table 4.7 shows the impacts of applying procedure integration transformations on power, energy, and execution time. As noted earlier, procedure integration eliminates call overhead and consequently reduces the memory references in the proposed example by 12.44%. Moreover, procedure integration reduces the executed instructions by 41.11% and IPCs by 12.59%. Thus, power consumption and execution time are reduced by 3.93 and 32.63%, respectively.

Finally, Figure 4.9 summarizes the results of applying different code transformations to power, execution time, and energy. The original code represents 100%. Any deviation above or under 100% is related to the applied code transformation.

Original Code

```
main()
{
int i,res[DIM] , val = 7;
    for( i = 0; i < N; i++ )
    {
        res[i] = f(val);
        val += 5;
    }
}

int f(int y)
{
    return pred(y) + pred(0)
           + pred(y+1);
}

int pred(int x)
{
    if (x == 0)
        return 0;
    else
        return x-1;
}
```

Transformed Code

```
main()
{
int i,res[DIM] , val = 7;
    for( i = 0; i < N; i++ )
    {
        res[i] = f(val);
        val += 5;
    }
}

int f(int y)
{
    int temp = 0;
    if (y == 0) temp += 0;
    else temp += y - 1;
    if (0 == 0) temp += 0;
    else temp += 0 - 1;
    if (y+1 == 0) temp += 0;
    else temp += (y+1) - 1;
    return temp;
}
```

**FIGURE 4.8** Procedure integration transformation.

**TABLE 4.7**
**Effects of Procedure Integration Transformations on Energy and Power Consumption**

| | Original | Transformed | % |
|---|---|---|---|
| Execution cycles | 3218 | 2168 | −32.63 |
| Power (W) | 1.039 | 0.998 | −3.93 |
| Energy (mJ) | 0.0033 | 0.0022 | −35.27 |
| IPCs | 0.983 | 0.859 | −12.59 |
| Memory references | 804 | 704 | −12.44 |
| Executed Instructions | 3162 | 1862 | −41.11 |

The results show that several code transformations such as loop peeling, loop fusion, and procedure integration produce good impacts on power consumption, energy, and performance. Other transformations such as loop permutation and loop tiling improve power consumption through performance. The results also show that some transformations have no impacts on power consumption but improve performance and energy. These loop reversal, loop strength reduction, and array declaration sorting transformations are not power hungry. The last group of code transformations (loop unswitching, loop normalization, fusion, and scalarization of array elements) improve the performance through power consumption.

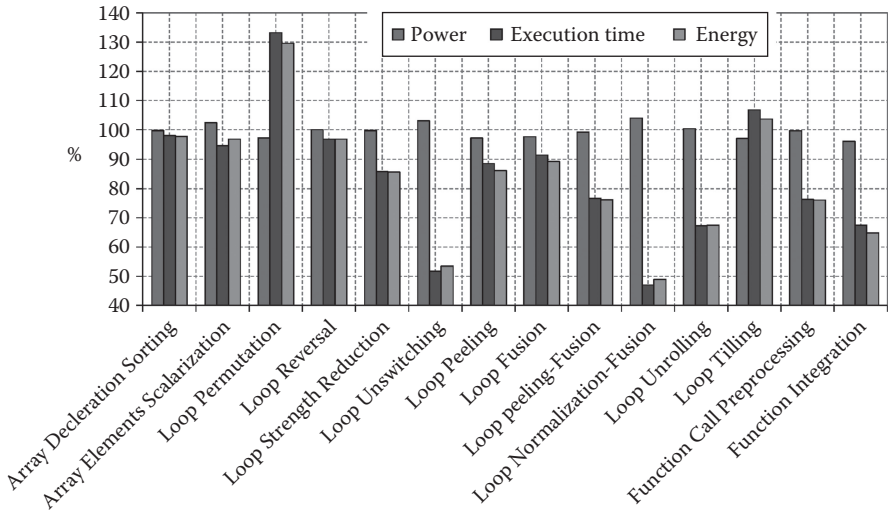


**FIGURE 4.9** Impacts of applying code transformations on power, execution time, and energy.

## 4.6 CONCLUSIONS

Reducing energy and power dissipation of an embedded system have become optimization goals in their own right. They are no longer considered side effects of traditional performance optimizations that attempt to reduce program execution times. In this chapter, we investigated the impacts of a set of high level language optimization techniques on the power consumption levels of embedded system architectures.

As a specific example, we considered the powerful Texas Instruments C6416T DSP processor. We presented a qualitative study that evaluated global compiler-based optimization levels (-o0, -o1, -o2, and -o3), certain architectural features, and several source code transformations from energy and power consumption perspectives.

First, we precisely measured the impacts of applying global execution time optimization levels on power consumption. The results indicated that the most aggressive execution time optimization level -o3 increased consumption by 30.3%. Meanwhile, energy decreased by 94.8%. This was mainly due to aggressive utilization of instruction execution overlapping that reduced execution time and energy but not power consumption.

Second, we assessed the effect of the C64x+ architectural feature known as SPLoop on energy and power consumption. The results show that the software loop pipelining feature contributed on average 69.7% to the total power consumption increase if -o3 was invoked.

We also investigated the influence of the powerful capability of the TMS320C6416T to execute SIMD instructions. We investigated the effect of SIMD while enabling and disabling SPLoop. By disabling the SPLoop, -o2-mu, or -o3-mu, SIMD instruction use generated comparable performance enhancement with -o2 or -o3 along with great advantages of 18.83 and 17% average power savings, respectively.

Finally, we examined the impacts of applying source code transformations on energy and power consumption. Three main categories of code transformations (data, loop, and procedural oriented) were investigated. Several code transformations (loop peeling, loop fusion, and procedure integration) demonstrated good impacts on power consumption, energy, and performance.

Loop permutation and loop tiling improved power consumption on account of performance. The results also indicated that some transformations produced no impacts on power consumption but they improved performance and decreased energy consumption. Loop reversal, loop strength reduction, and array declaration sorting transformations are not power hungry.

## REFERENCES

1. D. Ortiz and N. Santiago. 2007. High level optimization for low power consumption on microprocessor-based systems. In *Proceedings of 50th IEEE Midwest Symposium on Circuits and Systems,* pp. 1265–1268.
2. M. Valluri and L. John. 2001. Is compiling for performance compiling for power? In *Proceedings of 5th Workshop on Interaction between Compilers and Computer Architectures,* Monterrey, Mexico.
3. Standard Performance Evaluation Corporation (SPEC). n.d. *SPEC Benchmark Suite*. http://www.spec.org
4. R.A. Ravindran, P.D. Nagarkar, G.S. Dasika et al. 2005. Compiler managed dynamic instruction placement in a low-power code cache. In *Proceedings of International Symposium on Code Generation and Optimization.* Washington: IEEE, pp. 179–190. http://dx.doi.org/10.1109/CGO.2005.13
5. L.N. Chakrapani. 2001. The emerging power crisis in embedded processors: what can a poor compiler do? In *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Emdedded Systems.*
6. J.S. Seng and D.M. Tullsen. 2003. The effect of compiler optimizations on Pentium 4 power consumption. In *Proceedings of 7th Workshop on Interaction between Compilers and Computer Architectures.* Washington: IEEE, pp. 51–56.
7. J. Zambreno, M.T. Kandemir, and A.N. Choudhary. 2002. Enhancing compiler techniques for memory energy optimizations. In *Proceedings of 2nd International Conference on Embedded Software.* London: Springer, pp. 364–381. http://dl.acm.org/citation.cfm?id = 646788.703900
8. M. Casas-Sanchez, J. Rizo-Morente, C. Bleakley et al. 2007. Effect of compiler optimizations on DSP processor power and energy consumption. In *Proceedings of Conference on Design of Circuits and Integrated Systems*.
9. G. Esakkimuthu, N. Vijaykrishnan, M. Kandemir et al. 2000. Memory system energy: influence of hardware–software optimizations. In *Proceedings of International Symposium on Low Power Electronics and Design.* New York, ACM, pp. 244–246.
10. C. Lee, J. K. Lee, T. Hwang et al. 2000. Compiler optimization on instruction scheduling for low power. In *Proceedings of 13th International Symposium on System Synthesis.* Washington: IEEE, pp. 55–60. http://dx.doi.org/10.1145/501790.501803
11. M. Kandemir, N. Vijaykrishnan, and M.J. Irwin. 2002. Compiler optimizations for low power systems. In R. Graybill and R. Melhem, Eds., *Power-Aware Computing,* pp. 191–210.
12. W. Ye, N. Vijaykrishnan, M. Kandemir et al. 2000. The design and use of SimplePower: a cycle-accurate energy estimation tool. In *Proceedings of 37th Conference on Design Automation.* New York: ACM, pp. 340–345.

13. H. Mehta, R.M. Owens, M.J. Irwin et al. 1997. Techniques for low energy software. In *Proceedings of International Symposium on Low Power Electronics and Design.* New York: ACM, pp. 72–75. http://doi.acm.org/10.1145/263272.263286
14. R. Leupers. 2000. Code generation for embedded processors. In *Proceedings of IEEE 13th International Symposium on System Synthesis*. New York: ACM, pp. 173–178.
15. J. Oliver, O. Mocanu, and C. Ferrer. 2003. Energy awareness through software optimization as a performance estimate case study of the MC68HC908GP32 microcontroller. In *Proceedings of 4th International Workshop on Microprocessor Testing and Verification.* Washington: IEEE, pp. 111–116.
16. M.T.C. Lee, M. Fujita, V. Tiwari et al. 1997. Power analysis and minimization techniques for embedded DSP software. *IEEE Transactions on VLSI Systems*, 5, 123–135.
17. D. Ortiz and N. Santiago. 2008. Impact of source code optimizations on power consumption of embedded systems, June, 133–136.
18. Z.N. Azeemi and M. Rupp. 2005. Energy-aware source-to-source transformations for a VLIW DSP processor. In *Proceedings of 17th ICM*, pp. 133–138.
19. C. Brandolese, W. Fornaciari, F. Salice et al. 2002. The impact of source code transformations on software power and energy consumption. *Journal of Circuits, Systems, and Computers*, 11, 477–502.
20. D. Burger and T.M. Austin. 1997. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, 25, 13–25.
21. F. Catthoor, K. Danckaert, S. Wuytack et al. 2001. Code transformations for data transfer and storage exploration preprocessing in multimedia processors. *IEEE Design and Test of Computers*, 18,70–82.
22. C. Kulkarni, F. Catthoory, and H. De Man. 1998. Code transformations for low power caching in embedded multimedia processors. In *Proceedings of 12th. International Parallel Processing Symposium.* Washington: IEEE, pp. 292–297.
23. K.S. McKinley, S. Carr, and C.W. Tseng. 1996. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18, 424–453.
24. H. Yang, G.R. Gao, A. Marquez et al. 2001. Power and energy impact by loop transformations. In *Proceedings of Workshop on Compilers and Operating Systems for Low Power: Parallel Architecture and Compilation Techniques*.
25. Texas Instruments Inc. 2003. *TMS320C6416T Fixed Point Digital Signal Processor Datasheet*, www.ti.com
26. Spectrum Digital. 2004. *TMS320C6416T,DSK Technical Reference.* http://c6000.spectrumdigital.com/dsk6416/
27. Agilent Technologies Inc. 2007. *Agilent 34410A Digital Multimeter Datasheet.* http://www.home.agilent.com/agilent/product.jspx?pn = 34410A
28. Texas Instruments Inc. 2004. *TMS320C6416T Fixed Point Digital Signal Processor Optimizing Compiler User Guide.* www.ti.com
29. M.E.A. Ibrahim, M. Rupp, and S. E.D. Habib. 2009. Compiler-based optimizations impact on embedded software power consumption. In *Proceedings of IEEE Joint Conference,* pp. 247–250.
30. Texas Instruments Inc. 2006. *TMS320C6416T DSP Two-Level Internal Memory Reference Guide.* www.ti.com
31. M.E.A. Ibrahim, M. Rupp, and H.A.H. Fahmy. 2008. Power estimation methodology for VLIW digital signal processor. In *Proceedings of Conference on Signals, Systems and Computers.* Washington: IEEE, pp. 1840–1844.
32. M.E.A. Ibrahim, M. Rupp, and H. A. Fahmy. 2011. A precise high-level power consumption model for embedded systems software. *EURASIP Journal on Embedded Systems*, 14.
33. Texas Instruments Inc. 2007. *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide.* www.ti.com

34. Texas Instruments Inc. 2009. *C6000 Host Intrinsics.* www.tiexpressdsp.com
35. M.E.A. Ibrahim, M. Rupp, and S.E.D. Habib. 2009. Performance and power consumption trade-offs for a VLIW DSP. In *Proceedings of IEEE International Symposium on Signals, Circuits and Systems.* Washington: IEEE, pp. 197–200.
36. D.F. Bacon, S.L. Graham, and O.J. Sharp. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26, 421–461.
37. S.S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. San Francisco: Morgan Kaufmann.