Code Transformations and SIMD Impact on Embedded Software Energy/Power Consumption

Mostafa E. A. Ibrahim[1+2], Markus Rupp[2], and Hossam A. H. Fahmy[3]
[1]Electrical Engineering Department High Institute of Technology-Benha University
[2]Institute of Communications and RF Engineering-Vienna University of Technology, Austria
[3] Electronics and Communication Department Faculty of Engineering-Cairo University, Egypt
Email: mibrahim@ieee.org, mrupp@nt.tuwien.ac.at, hfahmy@arithmetic.stanford.edu

***Abstract*— The increasing demand for portable computing has elevated power consumption to be one of the most critical embedded systems design parameters. In this paper, we present a qualitative study wherein we examine the impact of code transformations on the energy and power consumption. Three main categories of code transformations are investigated, namely data, loop and procedural oriented transformations. Moreover, we evaluate the influence of employing Single Instruction Multiple Data (SIMD) on energy and power dissipation via the utilization of compiler intrinsic C-functions. Results show that a trade-off between power and performance can be achieved by employing the intrinsic C-functions in conjunction with some transformations such as loop unrolling and procedure integration.**

I. Introduction

In a growing number of complex heterogeneous embedded systems the relevance of the software component is rapidly increasing. Issues such as development time, flexibility and reusability are, in fact, better addressed by software based solutions. Given a particular architecture, the programs that run on it will have a significant effect on the energy usage of the processor. The manner in which a program exercises particular parts of the processor will vary the contribution of individual structures to total energy consumption.

Reducing energy and power dissipation of an embedded system have become optimization goals in their own right, no longer considered a side-effect of traditional performance optimizations which mainly try to reduce program execution time. Power and energy optimizations can be implemented in hardware through circuit design, and by the compiler through compile-time analysis and code reshaping. While hardware optimizations has been the focus of several studies and are fairly mature, software approaches to optimizing power are relatively new. Progress in understanding the impact of traditional compiler optimizations on the power consumption and developing new power-aware compiler optimizations are important to the overall energy optimization of the system.

The optimizations at compile time typically improve performance and occasionally the power consumption, with the main limitations of having a partial perspective of the algorithms and without the possibility of introducing significant modifications to the data structures. On the contrary, source code transformations can exploit full knowledge of the algorithm characteristics, with the capability of modifying both data structures and algorithm coding. Furthermore, inter-procedural optimizations can be envisioned.

In this paper, we evaluate the effect of applying source code transformations on the energy and power consumption of the targeted architecture. Three different categories of source code transformations namely data, loop and procedural oriented transformations are investigated. Finally, We assess the impact of utilizing the SIMD, via the use of the available intrinsic C-functions, on the power consumption as well as the performance.

The rest of the paper is organized as follows: Section II describes prior research efforts related to this work. Section III presents the methodology and a general overview of the target architecture along with the experimental measurement setup. Section IV illustrates the achieved results of applying the source code transformations and its impact on power, energy and performance. Moreover, it evaluates the impact of the SIMD via the employment of the intrinsic C-functions. Finally, Section V summarizes the main contributions of this paper.

II. Previous work

Most of the software-oriented proposals for power optimization focus on instruction scheduling and code generation, possibly minimizing memory access cost [1]. As expected, standard low level compiler optimizations, such as loop unrolling or software pipelining, are also beneficial to energy reduction since they reduce the code execution time. However, there are a number of cross-related effects that cannot be clearly identified and, in general, are hard to be applied by compilers, unless some suitable source-to-source restructuring of the code is a priori applied.

Ortiz et al. [2] investigated the impact of three different code transformations namely, loop unrolling, function inlining and variable types declaration on the power consumption. They choose three platforms as the target for their work, 8-bit and 16-bit micro-controllers and the 32-bit ARM7TDMI processor. Their results show that loop unrolling has a significant impact on the consumed power in case of using the 16-bit and 32-bit processors.

Zafar et al. [3] examined the effect of loop unrolling factor, grafting depth and blocking factor on the energy and performance for the Philips Nexperia media processor PNX1302. But, they interchangeably use the term energy and power

for the same meaning. Hence the improvement in energy is directly related to the performance enhancement.

Brandolese et al. [4] stressed the state-of-the-art source to source transformations, to discover and compare their effectiveness from power and energy perspective. The data structure, loop and inter-procedural transformations were investigated with the aid of GCC compiler. The compiled software codes were then simulated with a framework based on the SimpleScaler [5]. The simulation framework was configured with a 1-KByte 2-ways set-associative unified cache.

Catthoor et al. [6] showed the crucial role of source-to-source code transformations in the solution of the data-transfer and storage bottleneck in modern processor architectures. They survey many transformations that are mainely aiming to enhance the data locality and reuse.

Kulkarni et al. [7] improved the software controlled cache utilization, in order to achieve lower power requirements for multi-media and signal processing applications. Their methodology took into account many program parameters like the locality of data, size of data structures, access structures of large array variables, regularity of loop nests and the size and type of cache with the objective of improving the cache performance for lower power. The targeted platform for their research were the embedded multi-media and DSP processors. In the same way McKinley et al. [8] investigated the impact of loop transformations on the data locality. Yang et al. [9] studied the impact of loop optimizations in terms of performance and power tradeoffs, with the aid of 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. Both, low-level loop optimizations at code generation (back-end) phase, (loop unrolling and software pipelining) and high-level loop optimizations at program analysis and transformation phase (frontend), (loop permutation and tiling) are studied.

III. Methodology

First of all, we decide the source code transformations to be investigated. We prepare a suitable software kernel that allows the employment of each code transformation. The functionality of both the original and transformed kernels are tested and we verify that they give the same result. Next, both the original and transformed kernels are compiled and profiled. The performance, power and energy results of both the original and transformed kernels are compared to analyze the impact of the applied transformation.

Several code transformations were investigated, but due to space constraints, we focus on the results for array declaration sorting, loop peeling and procedure integration.

A. Measurement Setup

The targeted architecture in this work is the TMS320C6416T fixed-point VLIW DSP from Texas Instruments (TI). All the power measurements are carried out on the DSP Starter Kit (DSK) of the TMS320C6416T manufactured by Spectrum Digital Inc. Although the targeted architecture operating frequency ranges from 600 MHz to 1200 MHz, in our setup, the operating frequency is adjusted to 1000MHz and the DSP core voltage is 1.2V. The Agilent 34410A $6\frac{1}{2}$-digit Digital Multi-Meter (DMM) is used for measuring the current drawn by the DSP core. As shown in Fig. 1 the current is captured in term of differential voltage drop across a 0.025Ω sense resistor inserted, by the DSK manufacturer, in the current path of the DSP core.

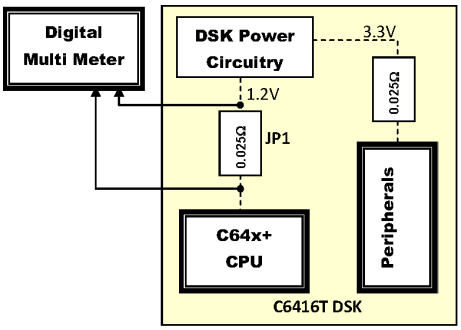

Fig. 1. CPU current measurement setup.

Code Composer Studio (CCS3.1), the Texas Instruments C/C++ compiler Ver.6.0.1, is utilized to produce the code binaries and to profile the targeted source code transformations.

IV. Experimental Results

In this section, experimental results concerning the influence of employing source code transformation to the power, energy and performance are presented. Next, we present the results of utilizing SIMD, via intrinsic C-functions, and its impact on the power, energy and performance are presented.

A. Influence of Source code Transformation

First, we present an example of the data oriented transformation, array declaration sorting. Figure 2 shows an example where the array access frequency ordering is C[], B[] and A[]. The declaration order, in the original code A[], B[], and C[], is restructured placing C[] in the first position, B[] in the second one and A[] at the end. This declaration reordering is employed to assure that the frequently accessed arrays are placed on top of the stack; in such a way, the memory locations frequently used are accessed by exploiting direct access mode.

The array declaration sorting reduces the execution time by 1.95% and consequently saves the energy by 2.19%. The power consumption is almost not affected, hence this transformation is not a power hungry transformation.

Second, we assess the effect of applying loop peeling transformation. This transformation, also called loop splitting, attempts to eliminate or reduce the loop dependencies introduced by the first or last few iterations by splitting these iterations from the loop and perform them outside the loop. Thus, it enables better instructions parallelization. Moreover, this transformation can be used to match the iteration control of adjacent loops allowing the two loops to be fused together. Figure 3 shows an example of loop peeling transformation. In

Original Code

```
int A[DIM], B[DIM], C[DIM], i;
    for (i = 5; i < 3500; i+=5)
        C[i] = val;
    for (i = 5; i < 2000; i+=10)
        B[i] = val;
    for (i = 5; i < 1000; i+=10)
        A[i] = val;
}
```

Transformed Code

```
int C[DIM], B[DIM], A[DIM], i;
    for (i = 5; i < 3500; i+=5)
        C[i] = val;
    for (i = 5; i < 2000; i+=10)
        B[i] = val;
    for (i = 5; i < 1000; i+=10)
        A[i] = val;
```

Fig. 2. Array declaration sorting transformation.

Original Code

```
int p = 10;
for (i=0; i<N; ++i)
{
    y[i] = x[i] + x[p];
    p = i;
}
```

Transformed Code

```
y[0] = x[0] + x[10];
for (i=1; i<N; ++i)
{
    y[i] = x[i] + x[i-1];
}
```

Fig. 3. loop peeling transformation.

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

Table I shows the impact of applying the loop peeling transformation on the 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% maintaining the same number of L1D cache misses. The instructions parallelization, expressed by Instruction Per Cycle (IPC), improved by 4.6%. Hence, the execution time and the power consumption are enhanced by 11.5% and 2.78% respectively leading to an energy saving of 13.97%.

TABLE I

INFLUENCE OF LOOP PEELING TRANSFORMATION ON THE ENERGY AND POWER CONSUMPTION.

	Original	Transformed	%
Exec. Cycles	2 808	2 485	−11.5
Power (W)	1.034	1.006	−2.78
Energy (mJ)	0.0029	0.0025	−13.97
IPC	0.919	0.962	4.6
Memory References	802	499	−37.78

Third, we study the impact of the procedure integration. Procedure integration, also called procedure inlining, replaces calls to procedures with copies of their bodies [10]. It can be a very useful optimization, because it changes calls from opaque objects that may have unknown effects on aliased variables and parameters to local code that not only exposes its effects but that can be optimized as part of the calling procedure [11]. Although procedure integration removes the cost of the procedure call and return instructions, these are often small savings. The major savings often come 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 having procedure calls embedded in loops or if it turns a loop that calls a procedure, whose body is itself a loop, into a nested loop [11].

Ordinarily, when a function is invoked, control is transferred to its definition by a branch or call instruction. With procedure integration, control flows directly to the code for the function, without a branch or call instruction. Moreover, the stack frames for the caller and callee are allocated together. Procedure integration may make the generated code slower as well; for instance, by decreasing locality of reference.
Figure 4 shows an example of the use of procedure integration. In this example the function pred(int) is integrated in the function f(int).

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

Finally, Fig. 5 summarizes the results of applying different code transformations on the power, execution time, and energy. In Fig. 5 the original code represents the 100%. Hence, any deviation above or under 100% is related to the applied code transformation.

B. Impact of SIMD Employment

The C6000 CCS compiler recognizes a number of intrinsic C-functions. Intrinsics allow the programmer to express the

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;
}
```

Fig. 4. Procedure integration transformation.

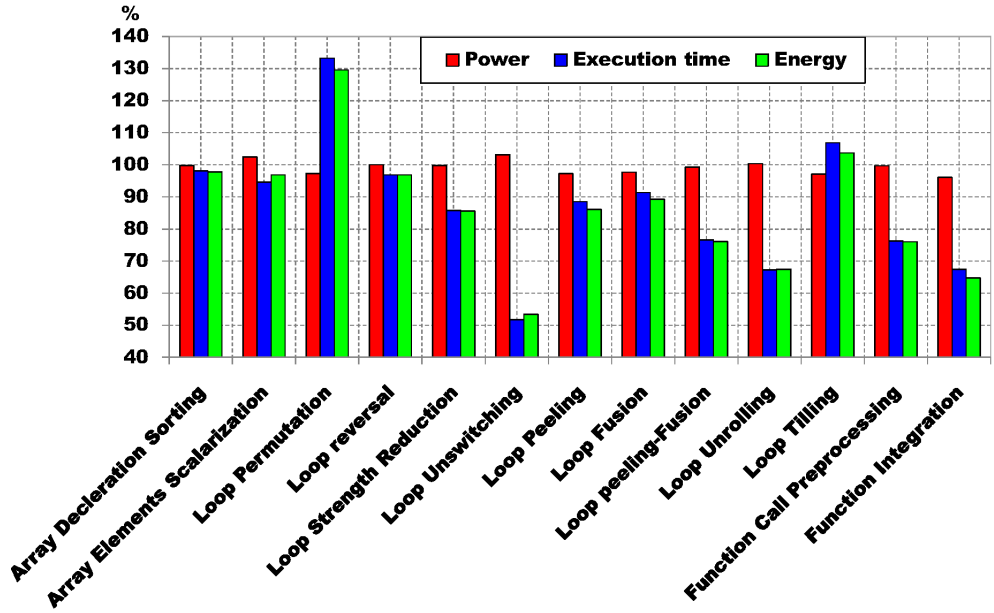

Fig. 5. Impact of applying code transformations on power, execution time and energy.

TABLE II

INFLUENCE OF PROCEDURE INTEGRATION TRANSFORMATIONS ON THE ENERGY AND POWER CONSUMPTION.

	Original	Transformed	%
Exec. Cycles	3 218	2 168	−32.63
Power (W)	1.039	0.998	−3.93
Energy (mJ)	0.0033	0.0022	−35.27
IPC	0.983	0.859	−12.59
Memory References	804	704	−12.44
Executed Instructions	3 162	1 862	−41.11

meaning of certain assembly statements that would otherwise be cumbersome or inexpressible in C/C++. Most of the intrinsic functions make use of the Single Instruction Multiple Data (SIMD) capabilities of the TMS320C6416T. The intrinsics are specified with a leading underscore, and are accessed by calling them as done with usual C/C++ functions [12]. For example:

```
int X1, X2, Y;
Y = _add4(X1, X2)
```

In order to assess the effect of utilizing SIMD instructions on the energy and power consumption, with the aid of the Texas Instrument host intrinsics package Ver.0.72 [13], we prepare two instances from the Inverse Discrete Cosine Transform (IDCT) algorithm as a case study. The first is implemented without using any of the SIMD instructions while the other

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    = _dotpsu2(F17, C71);
        Q0    = _dotpsu2(F53, C35);
        S0    = _dotpsu2 (F53, C53);
        S1    = _dotpsu2 (F17, C17);

        SOQ0  = _pack2(S0, Q0);
        S1Q1  = _pack2(S1, Q1);
```

Fig. 6. An example of the IDCT code with and without intrinsic C-functions.

is implemented with the aid of all possible SIMD instructions as shown in Fig. 6. The functionality of the two instances are tested and verified to give the same result. We study the effect of employing SIMD instructions with each of the compiler performance optimization levels (-o0 to -o3).

To study the effect of utilizing the SIMD isolated from the effect of the Software Pipelined Loop (SPLOOP) we compile and optimize the two versions of IDCT with -o3-mu (-mu: disables the software pipelined loop). Table III demonstrates that the SIMD version of the IDCT compiled and optimized by invoking -o3-mu achieves 3.96% power saving while it achieves 25.4% and 28.35% reduction in the execution time and the energy respectively. The achieved power saving is mainly caused by the reduction of the IPC by 20.86% while the enhancement in the execution time is derived by the significant memory references reduction, by more than 62%.

TABLE III

INTRINSICS EFFECTS WHEN -O3-PM-MU OPTIMIZATION OPTIONS ARE INVOKED.

	Original	with Intrinsics	%
Exec. Cycles	3 319	2 476	−25.4
Power (W)	1.091	1.048	−3.96
Energy (mJ)	0.00362	0.00259	−28.35
IPC	2.416	1.913	−20.86
CPU Stall Cycles	96	0	−100
Memory References	1 536	576	−62.5

To summarize the effect of utilizing the SIMD on the power consumption, energy and the execution time we investigate two more case studies, the Discrete Cosine Transform (DCT) and the Median filter with a 3x3 window in the same manner as the investigation of the IDCT. Figure 7, 8 and 9 represent a comparison between the power consumption, energy and the execution cycles with/without SIMD at various performance optimization options.

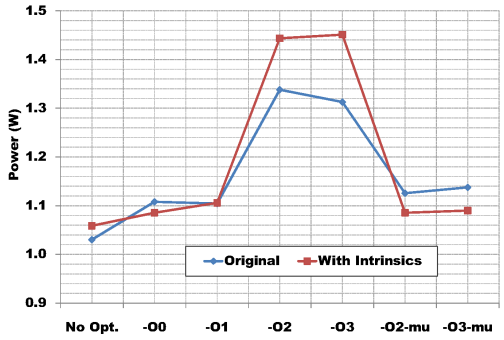

Fig. 7. Power consumption w/o SIMD utilization vs. various optimization options.

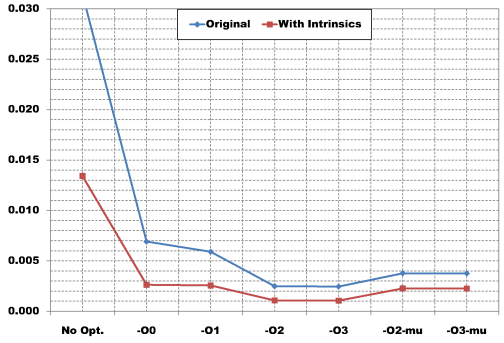

Fig. 8. Energy w/o SIMD employment vs. various optimization options.

In general employing the SIMD significantly enhance the performance and the energy saving. SPLOOP feature is the main basis for the significant improvement in the performance when -o2 or -o3 is invoked [14]. Hence, by disabling the SPLOOP feature, -o2-mu or -o3-mu, the utilization of SIMD instructions result in a comparable performance enhancement with -o2 or -o3 with the great advantage of an average power saving of 18.83% and 17% respectively.
Thus, it is pretty clear that rewriting the algorithm to maximally utilize SIMD instructions, while invoking the optimization options -o3-mu, is the best choice from the power consumption and performance perspective. Therefore, it can be considered as a trade-off between the power consumption from one side and the execution time and the energy from the other side.

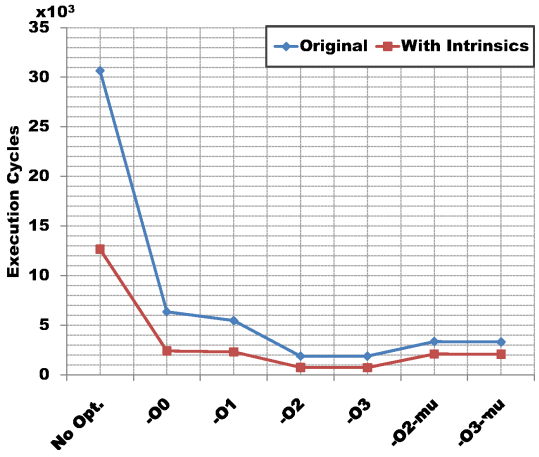

Fig. 9. Execution cycles w/o SIMD employment use vs. various optimization options.

V. Conclusion

Previous work [15] has shown that the most aggressive performance optimization level -o3 increases the power consumption by 25%. The CCS allows a very limited programmer's control over the individual optimizations within any optimization level. Thus, in this paper we evaluate the impact of several source code transformations from power, energy and performance perspective. The applied code transformations can be classified into three main categories: Data structures, loop and procedural transformations.
The results show that several code transformations have good impact on power consumption, energy and performance such as loop peeling, loop fusion and procedure integration while other transformations improve the power consumption on the account of the performance such as loop permutation and loop tilling. The results also show that some transformations have no impact on the power consumption but they improve the performance and energy. This type of transformations is not power hungry such as loop reversal, loop strength reduction and array declaration sorting.
Moreover, we investigated the influence of a powerful capability of the TMS320C6416T which is the ability to execute SIMD instructions. The CCS3.1 recognizes a number of intrinsic C-functions. Most of these intrinsic functions make use of the SIMD capabilities of the TMS320C6416T. Hence, we prepare two versions for each of the three different benchmarks namely DCT, IDCT and median filter. The first version is not employing any of the SIMD instructions, while the second utilizes all of the possible SIMD instructions.
The SPLOOP feature is the major contributor to the power increase when optimization level -o3 is invoked [14]. But, also it is the main reason for the performance enhancement when -o3 is invoked. Therefore, we investigate the effect of utilizing SIMD while enabling/disabling SPLOOP. The results show that by disabling the SPLOOP feature, -o2-mu or -o3-mu, the utilization of SIMD instructions result in a comparable performance enhancement with -o2 or -o3 with the great advantage of, on average, 18.83% and 17% power saving, respectively.

References

[1] M. T.-C. Lee, M. Fujita, V. Tiwari, and S. Malik, "Power analysis and minimization techniques for embedded dsp software," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 5, no. 1, pp. 123–135, 1997.

[2] D. Ortiz and N. Santiago, "Impact of Source Code Optimizations on Power Consumption of Embedded Systems," June 2008, pp. 133–136.

[3] Z. N. Azeemi and M. Rupp, "Energy-Aware Source-to-Source Transformations for a VLIW DSP Processor," in *proceedings of the 17th ICM'05*, Islamabad, Pakistan, December 2005, pp. 133–138.

[4] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "The Impact of Source Code Transformations on Software Power and Energy Consumption," *Journal of Circuits, Systems, and Computers*, vol. 11, no. 5, pp. 477–502, 2002.

[5] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, 1997.

[6] F. Catthoor, K. Danckaert, S. Wuytack, and N. D. Dutt, "Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors," *IEEE Design and Test of Computers*, vol. 18, no. 3, pp. 70–82, 2001.

[7] C. Kulkarni, F. Catthoory, and H. De Man, "Code Transformations for Low Power Caching in Embedded Multimedia Processors," in *proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium (IPPS'98)*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 292–297.

[8] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 4, pp. 424–453, 1996.

[9] H. Yang, G. R. Gao, A. Marquez, G. Cai, and Z. Hu, "Power and Energy Impact by Loop Transformations," in *proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'01)*, 2001.

[10] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 421–461, 1994.

[11] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[12] T. Instruments, *TMS320C6416T, Fixed Point Digital Signal Processor, Optimizing Compiler User Guide*, May 2004, spru187l. [Online]. Available: www.ti.com

[13] *Texas Instruments Inc., C6000 Host Intrinsics*, January 2009. [Online]. Available: www.tiexpressdsp.com

[14] M. E. A. Ibrahim, M. Rupp, and S. E.-D. Habib, "Performance and Power Consumption Trade-offs for a VLIW DSP," in *proceedings of the IEEE International Symposium on Signals, Circuits and Systems (ISSCS'09)*. IEEE, 2009, pp. 197–200.

[15] ——, "Compiler-Based Optimizations Impact on Embedded Software Power Consumption," in *proceedings of the IEEE joint conference NEWCAS-TAISA'09)*. IEEE, 2009, pp. 247–250.