The Impact of Source Code Transformations on Software Power and Energy Consumption.

Article *in* Journal of Circuits, Systems and Computers · October 2002

DOI: 10.1142/S0218126602000586 · Source: DBLP

Carlo Brandolese
Politecnico di Milano
68 PUBLICATIONS **545** CITATIONS

SEE PROFILE

William Fornaciari
Politecnico di Milano
252 PUBLICATIONS **1,817** CITATIONS

SEE PROFILE

Fabio Salice
Politecnico di Milano
173 PUBLICATIONS **1,145** CITATIONS

SEE PROFILE

Journal of Circuits, Systems, and Computers Vol. 11, No. 5 (2002) 477–502

THE IMPACT OF SOURCE CODE TRANSFORMATIONS ON SOFTWARE POWER AND ENERGY CONSUMPTION

CARLO BRANDOLESE
Politecnico di Milano – DEI, P.za L. Da Vinci, 32,
20133 - Milano, Italy, E-mail: brandole@elet.polimi.it

WILLIAM FORNACIARI
Politecnico di Milano – DEI, P.za L. Da Vinci, 32,
20133 - Milano, Italy, E-mail: fornacia@elet.polimi.it

FABIO SALICE
Politecnico di Milano – DEI, P.za L. Da Vinci, 32,
20133 - Milano, Italy, E-mail: salice@elet.polimi.it

DONATELLA SCIUTO
Politecnico di Milano – DEI, P.za L. Da Vinci, 32,
20133 - Milano, Italy, E-mail: sciuto@elet.polimi.it

Received
Revised
Accepted

Software power consumption minimization is becoming more and more a very relevant issue in the design of embedded systems, in particular those dedicated to mobile devices. The paper aims at reviewing state of the art source code transformations in terms of their effectiveness on power and energy consumption reduction. A design framework for the C language has been set up, using the gcc compiler with SimplePower as the simulation kernel. Some new transformations have been also identified aiming at reducing the power consumption. Four classes of transformations will be considered: loop transformations, data structures transformations, inter-procedural transformations and control structure transformations. For each transformation, together with the evaluation of the energy and power consumption, some applicability criteria have been defined.

1. Introduction

In recent years, power dissipation has become one of the major concerns for the embedded systems industry. The steady shrinking of the integration scale, the large number of devices packed in a single chip coupled with high operating frequencies have led to unacceptable levels of power dissipation, in particular for battery powered systems. An effective way to cope with the requirement of lowering power consumption to uphold the increasing demands of applications, should concurrently consider the following aspects:

(i) the sources of power consumption and a set of reliable estimators;
(ii) the methodologies to reduce the power consumption, typically considering the peculiarity of the applications.

A key for the success of many solutions is, in fact, a suitable tailoring of the implementing platform to the application, in detriment of its general-purpose capability. The higher the level of abstraction for the optimization, the better energy savings can be usually achieved. In literature, these problems have been considered for a long time, initially focusing on the silicon technology then moving up to include logic design and architecture-level design[1]. Nevertheless, the presence of mixed hardware/software architectures is becoming pervasive in the embedded systems arena, with a growing importance for the software section. Many proposals take into account the environment executing the code (CPU, Memory, Operating System, etc.) as well as the impact of code organization and compiler optimizations on the energy demand of the application. Memory optimization techniques focus on reducing the energy related to memory access, exploiting the presence of multilevel memory hierarchy, possibly in conjunction with suitable encodings to reduce the bus switching activity[1]. Other software-oriented proposals focus on instruction scheduling and code generation, possibly minimizing memory access cost[6,7,8]. A number of reviews and proposals on compiler techniques for power minimization can be found in literature[11,12,13,14]. As expected, standard 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 so 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. In fact, the optimizations at compile time typically improve performance and usually 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. Another benefit of exploiting restructuring of the source code is related to portability, since the results are normally fairly general to deal with different compilers and architectures, without any intervention on existing compilers. The aim of this paper is to present a part of a more comprehensive investigation in progress within the EU-funded Esprit project called POET, where our goal is to identify a methodology to optimize the energy consumption of software for embedded applications. We set up a workbench based on the `SimplePower` and `gcc` environments, and stressed the state-of-the-art transformations, to discover and compare their effectiveness. Some new transformations have been also identified and the rest of the paper will mainly describe their characteristics[5]. The methods have been partitioned in four classes, each focusing on a specific code aspect:

(i) **Loop transformations.**
(ii) **Data Structure Transformations.**
(iii) **Inter-Procedural Transformations.**
(iv) **Operators and Control Structure Transformations.**

Due to the lack of space, despite the analysis we carried out considered a broader range of transformations[5], this paper details only some of the most innovative ones. For each method, in addition to energy saving data, some applicability criteria are reported. Some code restructuring, in fact, can produce no energy improvement but can magnify the effectiveness of other transformations applied in sequence.

The paper is organized as follows. Section 2 describes the general methodology and the environment we arranged for our investigation. Sections 3, 4, 5 and 6 are devoted to present some of the proposed source level transformations together with experimental results obtained considering ad-hoc case studies. Section 7 discusses how the different transformations can be used in practice, proposing a possible source-level design flow. Finally, Section 8 summarizes in a concise form the most relevant properties of the presented transformations and shows the results obtained on larger, real-life benchmarks.

2. Analysis Methodology

Reducing the energy consumption working at software level means to apply a set of code transformations producing a less energy hungry application. This goal is pursued by directly modifying the control structures, the data access modes, and the subprograms organization or by performing a set of source code transformations such that the resulting code can be better optimized by the compiler (e.g., copy propagation, constant propagation and common sub-expression elimination). In this paper we followed the latter approach, identifying and evaluating a set of transformations to be applied before compilation (using `gcc`). The validation tool we adopted is based on `SimplePower`[2] and to overcome its lack of support of system calls an in-house, semi-automatic simulation framework based on `SimpleScalar`[15] has been developed. The optimization framework is an algorithmic-based optimization tool, where each transformation under analysis is applied until it produces an energy improvement. The four steps composing the analysis strategy are shown in Figure 1. The C source code under analysis for the applicability of a transformation is firstly compiled, to be conservative, with the highest optimization (`gcc -O3`). Such a result constitutes the comparison term to evaluate the transformation effectiveness. The compiled code is then simulated using `SimpleScalar` and the simulation results (number of clock cycles, cache misses, RAM accesses, etc.) are gathered and post-processed to identify energy and power consumption for both the core processor and the system. Concerning the energy consumption at system level, the Shiue-Chakrabarti model has been used[3,4]. Then, the transformation is applied to the source code and, following the same previous steps, the simulation

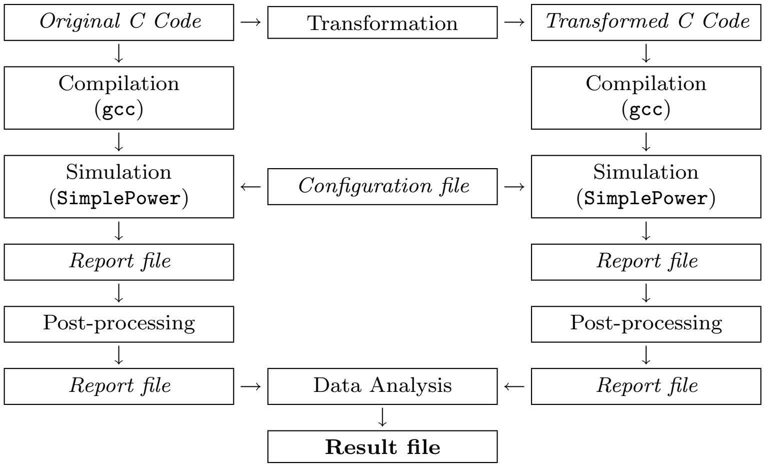


Fig. 1. The validation flow for code transformations analysis

results and the energy and power consumption are collected. Finally, the processor energies and the system energies are compared to identify the effectiveness of the transformation under analysis. It is worth noting that both simulations, with and without transformation, use the same set of values for the configuration of the simulator. In particular, a 1 KByte 2-ways set-associative unified cache has been selected. This configuration has been used to stress the analysis, since it tends to limit the benefits of the presented transformations.

3. Loop Transformations

This class includes transformations modifying either the loop body or the control structure of the loop. They are usually valuable since operate on a subset of code that is typically executed frequently: even small energy saving per cycle could strongly impact on the global energy of the application. While some loop transformations have been proposed in the field of compilers to exploit parallelism, others are specifically tailored to reduce power consumption. This type of transformations includes, among the others:

(i) **Loop unrolling.** The cycle is replicated multiple times.
(ii) **Variable expansion.** Variables like counters, accumulators, switches are identified and replicated for each code section, to reduce data dependencies and improving parallelism.
(iii) **Loop fusion.** Merges different loops to reduce control operations and D-cache misses, especially whenever they operate on the same data structures.
(iv) **Loop interchange.** Modifies the nesting ordering of the loops to change the access paths to arrays.
(v) **Loop tiling.** Increases the loop depth to reduce D-cache misses when large size arrays need to be accesses.

(vi) **Software pipelining.** Improves the instruction parallelism among the different loop iterations so that the pipeline is better exploited through stalls reduction.

(vii) **Loop unswitch.** Moves outside the loop those operations independent of the significant computation performed within the loop body.

(viii) **While-do to do-while**, **Zero output condition.** Modify the structure of the loop and the output condition to perform a zero-comparison (that is less energy expensive), respectively.

Within this class, we propose and detail the use of a transformation called Loop Distribution, that is well known in the field of parallel architectures, but its use for source level energy reduction is novel. The basic idea is to reduce the size of the loop body in order to decrease the number of I-cache misses. In particular, sections of decoupled code are distributed in disjoint loops to enable the storing of a complete loop in the cache, preventing to access the upper memory levels. Figure 2 reports an example of this transformation. Loop distribution is effective when a loop body is larger than the cache or than a given number of cache blocks and/or the cache is unified. Note that this transformation requires a reliable metric to estimate the code size at assembly level[9,10]. The proposed transformation produces positive effects in term of reduction of the number of I-cache and D-cache misses. The latter could probably occur when the original loop presents expressions with non interacting arrays so that different arrays can be distributed on disjoined loop bodies. Figure 3 reports an example where the two expressions in the loop interfere causing, probably, repeated D-cache misses since the probability of data reuse is low (b could overwrite a and viceversa). The transformed code reduces the probability of D-cache misses, this effect is amplified if the cache implements any pre-fetching mechanism. Negative consequences of this transformation could arise from both the increase of code size and the reduction of performance, due to the added control structure of loops. Table 1 collects the simulation results for the example of Figure 2, showing that both the system energy and the processor energy decrease. Note that the strong reduction of the term System Power/Processor Power indicates that the transformation affects the primary memory accesses. In

```
Original C Code                           Transformed C Code

for( i = 0; i < 80; i++ ) {               for( i = 0; i < 80; i++ ) {
   a = b[i] + i * d;                         a = b[i] + i * d;
   d = b[i+4] + d + a * (i + 1);             d = b[i+4] + d + a * (i + 1);
   a = a * d + b[i+2] * d;                   a = a * d + b[i+2] * d;
   f = i * e + c[i];                      }
   e = b[i] + i * f;                      for( i = 0; i < 80; i++ ) {
   e = e * f + b[i] * e;                     f = i * e + c[i];
}                                            e = b[i] + i * f;
                                             e = e * f + b[i] * e;
                                          }
```

Fig. 2. An example of the loop distribution transformation

Original C Code

```
for( i = 0; i < n; i++ ) {
   for( j = 0; j < n; j++ ) {
      a[j] = a[j] + c[i];
      b[j] = b[j] + c[i];
   }
}
```

Transformed C Code

```
for( i = 0; i < n; i++ ) {
   for( j = 0; j < n; j++ ) {
      a[j] = a[j] + c[i];
   }
}
for( i = 0; i < n; i++ ) {
   for( j = 0; j < n; j++ ) {
      b[j] = b[j] + c[i];
   }
}
```

Fig. 3. An example of Loop Distribution transformation with two disjoined arrays

Table 1. Power and energy data for the loop distribution

Parameter	Original	Transformed	%
Clock Cycles	14393.00	11981.00	-16.76
Processor Energy (μJ)	3.44	3.25	-5.64
Processor avg. Power (mW)	23.90	27.09	13.35
System Energy (μJ)	544.00	90.00	-83.45
System Avg Power (mW)	3778.66	751.31	-80.12
System Power/Processor Power	158.09	27.73	-82.46

particular, cache misses are considerably reduced as shown in Table 2. In fact, the original code size and the number of variables cause a set of I-cache misses at each loop iteration, which are avoided by fragmenting the computation and the variables over two loops. These effects are particularly relevant since the considered cache is unified. This transformation produces relevant energy savings at system level, also using different cache architectures. The source-level factors influencing the energy reduction are:

Table 2. D-cache and I-cache misses for the loop distribution

Parameter	Original	Transformed	%
I-Cache Misses	319.00	37.00	-88.40
D-Cache Misses	110.00	34.00	-69.10

(i) The number of loop repetitions. Often, the number of cache misses per loop is small: the greater is the number of iterations the higher the energy saved.

(ii) The amount and size of disjoined code blocks. The size of both code and data has to be close to the cache size, so that the number of cache misses is minimized. Conversely, the higher number of loops increases the control instructions affecting both the energy reduction and the performance.

Loop Distribution is a good candidate to be applied after Software Pipelining and Variable Expansion[5].

4. Data Structure Transformations

This class identifies the set of source code transformations that either modifies the data structure included in the source code or introduces new data structures or, possibly, modifies the access mode and the access paths. This class of transformations focuses on the relation with the data memory, with the aim of maximizing the exploitation of the register file (reduction of memory and cache accesses). Two sub-classes of transformations can be envisioned, focusing on arrays or scalar variables. The former is mainly constituted by innovative transformations, optimizing the array allocation and the access modes to improve cache effectiveness. The considered strategies mainly concern: **Array Declaration Sorting**, **Array Scope Modification**, **Insertion of Temporary Arrays** and **Replacement of Array Entries with Scalar Variables**.

4.1. *Arrays Declaration Sorting*

The basic idea is to modify the local array declaration ordering, so that the arrays more frequently accessed are placed on top of the stack; in such a way, the memory locations frequently used are accessed by exploiting direct access mode. The application of this transformation requires either a static estimation or a dynamic analysis of the local arrays access frequency: starting from this information, the array declarations are reorganized to place first the more frequently accessed arrays. Figure 4 shows an example where the access frequency ordering is `C[]`, `B[]` and `A[]`: the declaration order is restructured placing `C[]` in the first position, `B[]` in the second one and `A[]` at the end. This transformation is derived from both the stack allocation strategies of local arrays performed by compilers and the access mode used to access the arrays. In particular, the arrays are allocated in the stack following the order of declaration and the first array is accessed using offset addressing with constant 0, while the others use non-0 constants. Note that, in general, offset addressing with constant 0 is less energy expensive with respect to using other constants. By declaring as first the array more frequently used in the subroutine, the number of operations using offset addressing with constant 0 is maximized and, consequently, the energy consumption is reduced. Conversely, per-

Original C Code

```
void subf( int val ) {
   int A[DIM], B[DIM], C[DIM], i;
   for( i = 5; i < 3500; i += 50 )
      C[i] = val;
   for( i = 5; i < 2000; i += 100 )
      B[i] = val;
   for( i = 5; i < 1000; i += 100 )
      A[i] = val;
}
```

Transformed C Code

```
void subf( int val ) {
   int C[DIM], B[DIM], A[DIM], i;
   for( i = 5; i < 3500; i += 50 )
      C[i] = val;
   for( i = 5; i < 2000; i += 100 )
      B[i] = val;
   for( i = 5; i < 1000; i += 100 )
      A[i] = val;
}
```

Fig. 4. An example of array declaration sorting

formance does not change since the number of clock cycles is unaffected. The array size affects the energy consumption associated with the data access. In fact, when the offset exceeds a given value (depending on the Instruction Set Architecture), it can no longer be embedded in the instruction, requiring more instructions or other addressing modes (i.e. indexed addressing). For this reason, it could be convenient to place large arrays at the bottom of the declarations list, to save additional instructions for accessing small arrays. An example is reported in Figure 5, where the C code on the left-and side of the has been compiled with N=1500 and N=15000 and the relevant differences in the assembly code are highlighted with a ◇ sign. In particular, the latter imposes that the addresses for accessing B[] and C[] have to be computed at each loop iteration. In summary, the developed energy function to evaluate this transformation takes into account the array size (depending on both number and type of the entries), the execution frequency and the number of arrays. The transformation has been tested on the example of Figure 4, focusing on the

C Code

```
#define N 1500 /*15000*/
int subf( void ) {
   int A[N], B[N], C[N], i;
   for(i = 5; i < 20; i++) {
      A[i] = 5;
      B[i] = A[i] + 7;
      C[i] = B[i] + 6;
   }
   return C[10];
}
```

Assembly code for N=1500

```
subf:
.frame $sp, 18000, $31
.mask 0x00000000, 0
.fmask 0x00000000, 0
  subu  $sp, $sp, 18000
  li    $4, 0x00000001
  li    $7, 0x00000005
  li    $6, 0x0000000c
  li    $5, 0x00000012
  addu  $3, $sp, 4
$L11:
  sw    $7, 0($3)
  sw    $6, 6000($3)
  sw    $5, 12000($3)
  addu  $3, $3, 4
  addu  $4, $4, 1
  slt   $2, $4, 20
  bne   $2, $0, $L11
.set noreorder
  lw    $2, 12040($sp)
.set reorder
  addu  $sp, $sp, 18000
  j     $31
.end subf
```

Assembly code for N=15000

```
subf:
.frame $sp, 18000, $31
  li    $8, 0x0002bf20
  subu  $sp, $sp, $8
  li    $5, 0x00000001
  li    $10,0x00000005
◇ li    $9, 0x0000ea60
  li    $8, 0x0000000c
◇ li    $6, 0x00010000
◇ ori   $6, $6, 0xd4c0
  li    $7, 0x00000012
  addu  $4, $sp, 4
$L11:
◇ addu  $2, $4, $9
◇ addu  $3, $4, $6
  sw    $10, 0($4)
  addu  $4, $4, 4
  addu  $5, $5, 1
  sw    $8, 0($2)
  slt   $2, $5, 20
  sw    $7, 0($3)
  bne   $2, $0, $L11
◇ li    $2, 0x00010000
◇ ori   $2, $2, 0x8000
◇ addu  $2, $sp, $2
.set noreorder
  lw    $2, 21736($2)
.set reorder
◇ li    $8, 0x0002bf20
  addu  $sp, $sp, $8
  j     $31
.end subf
```

Fig. 5. A sample code for the analysis of the relation between energy consumption and array size

sorting effect. Table 3 gathers the simulation results: it is clear that the energy saving is confined to the processor while the performance and the system level energy are both unchanged.

Table 3. Power and energy data for array declaration sorting transformation

Parameter	Original	Transformed	%
Clock Cycles	13185.00	13185.00	0.00
Processor Energy (μJ)	0.96	0.93	-2.69
Processor avg. Power (mW)	7.29	7.09	-2.69
System Energy (μJ)	388.00	388.00	0.00
System Avg Power (mW)	2941.04	2941.04	0.00
System Power/Processor Power	403.41	414.56	+2.76

4.2. *Array Scope Modification: local to global*

This method converts local arrays into global arrays to store them within data memory rather than on the stack. In this case the array allocation address can be determined at compile time. Conversely, if the array is declared locally, the allocation address can only be determined when the subprogram is called and it depends on the stack pointer value. As a consequence, the global arrays are accessed with offset addressing mode with constant 0 while local arrays, excluding the first, are accessed with constant offset different from 0: an energy reduction is thus achieved. Figure 6 shows an example of this code transformation. Unfortunately, since global declarations are memory consuming, only a subset of the arrays can actually take advantage of such a scope modification. A specific cost function we developed considers this feature, apart from other relevant aspects, suggesting a scope modification for arrays frequently accessed. The transformation has been analyzed on the

Original C Code

```
main() {
   int i;
   int b[50];
   int a[50];
   for( i = 0; i < 48; i++ ) {
      b[i] = i;
      a[i] = i;
   }
   for( i = 2; i <= 40; i++ ) {
      b[i] = ( a[i] + b[i] ) / 2;
      if( i % 2 ) == 0 ) {
         a[i] = a[i-2] + 1;
      } else {
         a[i] = a[i-1] - 1;
      }
   }
}
```

Transformed C Code

```
int b[50];
int a[50];
main() {
   int i;
   for( i = 0; i < 48; i++ ) {
      b[i] = i;
      a[i] = i;
   }
   for( i = 2; i <= 40; i++ ) {
      b[i] = ( a[i] + b[i] ) / 2;
      if( i % 2 == 0 ) {
         a[i] = a[i-2] + 1;
      } else {
         a[i] = a[i-1] - 1;
      }
   }
}
```

Fig. 6. An example of array scope modification

example of Figure 6. As predictable for this simple case study, the influence is in terms of processor energy (Table 4) since the transformation affects a specific characteristic related to the offset addressing. However, this transformation has other valuable side-effects, related to the probability of not exceeding the offset during access to locally declared arrays. In such a way, more energy can be saved since the negative influence of high instruction numbers (cache misses and performance degradation) is mitigated.

Table 4. Power and energy data for array scope modification transformation

Parameter	Original	Transformed	%
Clock Cycles	8311.00	8310.00	-0.01
Processor Energy (μJ)	2.70	2.08	-22.82
Processor avg. Power (mW)	32.48	25.07	-22.81
System Energy (μJ)	15.20	15.20	0.00
System Avg Power (mW)	183.03	183.05	+0.01
System Power/Processor Power	5.63	7.30	29.57

4.3. *Array Resizing: temporary array insertion*

This transformation introduces a small temporary array where to store a subset of the elements of a larger array. The candidates are those elements accessed more frequently, e.g. identified via profiling or static considerations, which could be accessed without any data cache miss. Note that the application of this transformation could be not significant for small arrays, if they can be totally contained in the cache and they are not in conflict, in terms of memory resources, with other data. Practical application of the transformation requires, once the subset of elements is identified, their copy in a temporary array (resizing) to be used instead of the original array and a copy back (restoring) of the final results. An adaptation of the indexes used in the original array towards the resized one is also necessary. Figure 7 shows an example of the application of this transformation on the arrays `a[]` and `b[]`. In the case the candidate subset of elements changes dynamically, not to vanish the transformation, the temporary array identification has to be computed every time the considered elements are not part of the temporary array. The increase of control instructions related to both the array initialization and, possibly, the data restoring, increases the processor energy. However, it has been noticed that when the ratio between the size of the initial array and the temporary one is roughly more than 10, the system energy reduction is significant and largely compensates the increase of the processor energy. The results for the example of Figure 7 are reported in Table 5, showing a processor energy rising due to the adding of two cycles (array initialization and data restoring) and a significant system energy reduction mainly related to the data cache misses improvement (Table 6). Conversely, performance is significantly reduced, so that particular attention has to be paid during the application of this transformation in timing sensitive systems.

Original C Code

```
#define DIM1 250
main() {
   int a[DIM1], b[DIM1];
   int c, i, j = 40;
   while( j-- > 0 ) {
      for( i = 0; i < 100; i += 10 ) {
         a[i] = b[i] + 5;
         c = a[i] + b[i] * 15;
   }
}
```

Transformed C Code

```
#define DIM1 250
#define DIM2 15
main() {
   int a[DIM1], b[DIM1];
   int ta[DIM2], tb[DIM2];
   int c, i, j = 40;
   for( i = 0; i < 10; i++ ) {
      ta[i] = a[i*10];
      tb[i] = b[i*10];
   }
   while( j-- > 0 ) {
      for( i = 0; i < 10; i++ ) {
         ta[i] = tb[i] + 5;
         c = ta[i] + tb[i] * 15;
      }
   }
   for( i = 0; i < 10; i++ ) {
      a[i*10] = ta[i];
      b[i*10] = tb[i];
   }
}
```

Fig. 7. An example of temporary array insertion

Table 5. Power and energy data for array resizing transformation

Parameter	Original	Transformed	%
Clock Cycles	4427.00	6369.00	+43.87
Processor Energy (μJ)	0.47	1.94	+7.71
Processor avg. Power (mW)	10.73	30.42	+83.39
System Energy (μJ)	186.00	63.40	-65.99
System Avg Power (mW)	4209.71	995.25	-76.36
System Power/Processor Power	392.13	32.71	-91.66

Table 6. D-cache and I-cache misses for array resizing

Parameter	Original	Transformed	%
I-Cache Misses	19.00	15.00	-21.05
D-Cache Misses	128.00	35.00	-72.66

4.4. *Scalarization of Array Elements*

This transformation introduces a set of temporary variables as a substitute of the more frequently used elements of an array. It allows the compiler to optimize the computation by using the CPU registers avoiding repeated memory accesses. The application of this transformation requires gathering information (statically, by analyzing the code or dynamically) to identify the subset of the more frequently used array elements or to identify references that can be tightly independent of the iteration index. Figure 8 shows an example of the proposed transformation where the introduction of the scalar variables `t1` and `t2` eliminates the accesses to the array elements `a[i-1]` and `a[i-2]` while memory accesses to `a[i]` are reduced by introducing `t0`. Table 7 shows the results concerning the example of Figure 8.

Original C Code

```
main() {
   int i, b[50], a[50];
   for( i = 0; i < 48; i++ ) {
      b[i] = i;
      a[i] = i;
   }
   for( i = 2; i <= 40; i++ ) {
      b[i] = ( a[i] + b[i] ) / 2;
      if( i % 2 ) == 0 ) {
         a[i] = a[i-2] + 1;
      } else {
         a[i] =a [i-1] - 1;
      }
   }
}
```

Transformed C Code

```
main() {
   int i, b[50], a[50];
   int t2, t1, t0;
   for( i = 0; i < 48; i++ ) {
      b[i] = i;
      a[i] = i;
   }
   t2 = a[0]; t1 = a[1];
   for( i = 2; i <= 40; i++ ) {
      t0 = a[i];
      b[i] = ( t0 + b[i] ) / 2;
      if( i % 2 ) == 0 ) {
         t0 = t2 + 1;
      } else {
         t0 = t1 - 1;
      }
      a[i] = t0;
      t2 = t1; t1 = t0;
   }
}
```

Fig. 8. An example of scalar variables introduction

Processor energy and power consumption are improved thanks to the local accesses to the register file. At system level, the impact of the transformation on energy and power is less evident due to the specific characteristics of the test-bench. However, the reduction of the memory accesses helps the cache misses reduction.

Table 7. Power and energy data for scalarization of array elements

Parameter	Original	Transformed	%
Clock Cycles	1775.00	1077.00	-39.32
Processor Energy (μJ)	0.38	1.59	-58.66
Processor avg. Power (mW)	21.67	14.76	-31.87
System Energy (μJ)	33.00	31.70	-3.85
System Avg Power (mW)	1856.71	2942.39	+58.46
System Power/Processor Power	859.91	1993.18	+131.68

Table 8 shows this effect where the cache hit reduction clearly indicates an intensive use of the register file and a relevant decreasing of cache access operations.

Table 8. D-cache and I-cache misses for scalarization

Parameter	Original	Transformed	%
I-Cache Misses	12.00	11.00	-8.33
D-Cache Misses	14.00	14.00	0.00
I-Cache Hits	1650.00	962.00	-41.70
D-Cache Hits	256.00	116.00	-54.69

It is worth noting that the application of this transformation enhances its effects if it is applied after the loop unrolling transformation (see Figures 9 and 10).

Original C Code

```
main() {
   int x[80], y[80], a[80][80];
   int i, j, n = 40;
   for( i = 1; i <= n; i++ ) {
      x[i] = 0;
      for( j = 1; j < n; j++ ) {
         x[i] = x[i] + a[i][j] * y[j];
      }
   }
}
```

Fig. 9. An example of scalar variables introduction

Unrolled C Code

```
main() {
   int x[80], y[80], a[80][80];
   int i, j, n = 40;
   for( i = 1; i <= n; i += 3 ) {
      x[i] = 0;
      for( j = 1; j < n; j++ ) {
         x[i] = x[i]+a[i][j]*y[j];
         x[i+1] = x[i+1]+a[i+1][j]*y[j];
         x[i+2] = x[i+2]+a[i+2][j]*y[j];
      }
   }
}
```

Final C Code

```
main() {
   int x[80], y[80], a[80][80];
   int i, j, n = 40;
   int t0, t1, t2, t4;
   for( i = 1; i <= n; i += 3 ) {
      t0 = t1 = t2 = 0;
      for( j = 1; j <n; j++ ) {
         t4 = y[j];
         t0 = t0 + a[i][j] * t4;
         t1 = t1 + a[i+1][j] * t4;
         t2 = t2 + a[i+2][j] * t4;
      }
      x[i] = t0;
      x[i+1] = t1;
      x[i+2] = t2;
   }
}
```

Fig. 10. An example of the combination of loop unrolling and scalarization transformations

In addition to the above transformations, other energy optimization strategies can be envisioned:

(i) **Local copy of global variable.** This strategy increases the possibility for the variables to be stored in registers whose access is less energy demanding.

(ii) **Pointer-chain reduction.** Multiple indirect addressing are replaced with a temporary variable storing the actual address of the data structure.

(iii) **Global variable inizialization.** Assignment is carried out during declaration. This solution allows the first procedure executing the initialization to skip the allocation in memory of the values.

Transformations operating on scalar variables (e.g. modifying their scope) are not discussed here for the sake of conciseness.

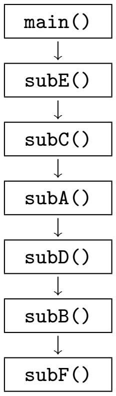
5. Inter-subroutine Transformations

This class of transformations includes the set of source code manipulations operating at subroutine level, typically not considered by compilers, analyzing whether or not it is convenient to modify the subroutine interface (i.e. parameters passing strategy, data types, etc.), the subroutine declaration and/or the subroutine call. The most known of these transformations is **Function Inlining** whose benefits are to reduce context switch due to the call itself and to enable more aggressive compiler optimization by eliminating function boundaries. In addition, we propose two new transformations: **Subroutines Queuing Reordering** and **Scope reduction of by-address parameters** whose details are give in the following.

5.1. *Subroutines Queuing Reordering*

Usually, compilers produce object code by queuing the subroutines imitating the source code structure. Based on this peculiarity, this transformation sorts the subroutines declarations according to the subroutine call graph (possibly annotated with dynamic information) in order to reduce the I-cache misses. Let us consider the first call of a sub-program. When such a subroutine is called, its code, or part of it, has to be fetched in cache. Consequently, if the object code of the called function is adjacent to the calling subroutine it is probable that it (or part of it) has been already loaded in cache during the execution of the caller sub-program. This effect is magnified if the cache adopts a pre-fetching policy where some cache blocks following the required block (cache miss) are automatically fetched. Conversely, if the two subprograms are not adjacent, a double penalty is introduced, since one or more instruction cache blocks could be uselessly. In order to take full advantage of this transformation, the call graph should be annotated with information, dynamically extracted, for subprograms called in more than one subroutine and for those calling more than one subroutine. Such dynamic indications improve the transformation effectiveness with respect to the simple static analysis of the call graph which, however, it is an improvement over the typical random subroutine queuing. The transformation has been experimented with the simple example reported in Figure 11 and the obtained results are gathered in Table 9. The main impact is on the system level energy though it can be identified a contribute to the reduction of the number of clock cycles; these effects are both originated from the reduction of instruction cache misses consequent to the implicit optimization of the cache blocks contents. Figure 11 only shows the relevant portion of the different routines, i.e. the calls. The results reported in Table 9 refer to subroutines composed of 5 to 10 lines of code each, mostly performing integer arithmetic operations on scalar variables. Different factors can influence the effectiveness of this transformation. Subroutines with high call frequency are good candidates, and the lower the fan out (fan in) of the call graph, the higher the probability to exploit the shared cache block. Similarly, the higher the subroutines code size, the lower the probability to reuse the shared cache block is. This factor depends on the cache block substitution policy,

Call graph



Original C Code

```
main () {
   res = subE();
}

int subA() { res = subD(); }

int subB() { res = subF(); }

int subC() { res = subA(); }

int subD() { res = subB(); }

int subE() { res = subC(); }

int subF() { ...  }
```

Transformed C Code

```
main () {
   res = subE();
}

int subE() { res = subC(); }

int subC() { res = subA(); }

int subA() { res = subD(); }

int subD() { res = subB(); }

int subB() { res = subF(); }

int subF() { ...  }
```

Fig. 11. An example of subroutine code sorting: call graph and transformation

Table 9. Power and energy data for subroutine code sorting

Parameter	Original	Transformed	%
Clock Cycles	1943.00	1931.00	-0.62
Processor Energy (μJ)	0.35	0.35	-0.00
Processor avg. Power (mW)	18.12	18.16	+0.22
System Energy (μJ)	156.00	153.00	-1.62
System Avg Power (mW)	8034.96	7953,48	-1.01
System Power/Processor Power	443.43	437,84	-1.26

the cache architecture and the cache blocks number. Other aspects to be considered are the position of the call inside the calling subroutine, with respect to the cache substitution policy and the distance between two consequent calls: the higher is their distance (in terms of code) the lower is probability to reuse the shared cache block.

5.2. *Substitution of a variable passed as an address with a local variable*

This transformation replaces a routine argument passed as an address with a local copy of a variable; the substitution is performed immediately before the subroutine is called while the returned values are restored in the initial variable immediately after the call. Typically, compilers tend to store in memory a variable used as subroutine argument passed as an address, so that using such a variable inside the calling routine is energy-expensive, especially if it is intensively used. This transformation drives the compiler in the use of registers, to minimize the energy necessary to access such data. Unfortunately, the insertion of temporal variables has a price to be paid: it adds instructions to perform copies and to restore values that could make ineffective the transformation, if the enlarged code increases the instruction cache misses. To cope with this risk, the approach can be applied only to the subset of variables involved (in the caller subroutine) in intensive computations. The higher is the number of variables involved in the transformation the higher is the probability to take advantage of this transformation. However, the variable number is constrained by both the amount of registers and the number of parallel variables concurrently active. In particular, the latter influences the spilling effect where some load/store operations are forced to cope with the limited number of registers. The analysis performed on the example reported in Figure 12 has produced the results collected in Table 10. The transformation mainly affects the system level energy consumption. This effect is due to the confinement of the computation inside the CPU reducing cache hits. Furthermore, the decreasing of code size resulting from the elimination of same load/store operations introduces an extra energy improvement since it affects the instruction cache misses.

Original C Code

```
main() {
   int a, b = 436, d = 52;
   int e = 362, f = 783, h = 35;
   int r = 13, ind;
   for( ind = 1; ind < 200; ind++ ) {
      e = d + ind * f;
      f = h * e + d - r * ind;
      h = h + ind;
      b++; f++; d++;
   }
   for( a = 1; a < 10; a++ ) {
      for( b = 1; b < 14; b++ ) {
         d++; e++; f++; h++; r++;
         res = fc( &a, &b, &d, &e,
                   &f, &h, &r );
      }
   }
}

int fc( int *a, int *b, int *d,
   int *e, int *f, int *h, int *r) {
   int cont, result;
   for( cont = 1; cont < 5; cont++ ) {
      *d = *a + *b;
      *f = *b + *e;
      result = *d + *f + *h + *r;
   }
   return result;
}
```

Final C Code

```
int a, b = 436, d = 52;
int e = 362, f = 783, h = 35;
int r = 13, ind;
main() {
   int ta, tb, td, te, tf, th, tr;
   for( ind = 1; ind < 200; ind++ ) {
      e = d + ind * f;
      f = h * e + d - r * ind;
      h = h + ind;
      b++; f}++; d++;
   for( a = 1; a < 10; a++ ) {
      for( b = 1; b < 14; b++ ) {
         d++; e++; f++; h++; r++;
         ta = a; tb = b; td = d;
         te = e; tf = f; th = h;
         tr = r;
         res = fc( &a, &b, &d, &e,
                   &f, &h, &r );
         a = ta; b = tb; d = td;
         e = te; f = tf; h = th;
         r = tr;
      }
   }
}

int fc( int *a, int *b, int *d,
   int *e, int *f, int *h, int *r) {
   int cont, result;
   for( cont = 1; cont < 5; cont++ ) {
      *d = *a + *b;
      *f = *b + *e;
      result = *d + *f + *h + *r;
   }
   return result;
}
```

Fig. 12. An example of substitution of a variable passed as an address with a local variable

Table 10. Power and energy data for local variable substitution

Parameter	Original	Transformed	%
Clock Cycles	12298.00	12278.00	-0,16
Processor Energy (μJ)	4.25	4.22	-0,86
Processor avg. Power (mW)	34.57	34.33	-0,70
System Energy (μJ)	34.30	31.80	-7,41
System Avg Power (mW)	279.10	258.83	-7,26
System Power/Processor Power	8.07	7.54	-6.60

6. Operators And Control Structure Transformations

This class gathers source code transformations optimizing either specific operations or control structures. Since compilers, directly perform many optimizations belonging to this class (e.g. when the maximum optimization level is selected by `gcc`) we focus the attention on those typically not directly provided. The section discusses two transformations: **Conditional Expression Reordering** that rearranges the conditional sub-expression of a test condition and **Function Call Preprocessing** that wraps library function calls with macros in order to to eliminate the call when the result can be a-priori determined.

6.1. *Conditional Expression Reordering*

This transformation analyzes a complex conditional expressions by rearranging the sub-expressions set in order to save energy by exploiting implicit shortcuts operations. The proposed transformation reassembles the sub-expressions by following, recursively, this criterion: two sub-conditions AND–connected (OR–Connected) are reordered by placing after (before) the sub-condition whose probability to be true is higher. Figure 13 shows an example. The application of such a transformation

Original C Code

```
int a = 1, b = 1, d = -2, e = 22;
int f = 1, res = 0;
main() {
   int b = 12, i;
   for( i = 1; i < 400; i++ ) {
      a++; b++; f++; e++;
      res = funct( a, b, f );
   }
}

int funct( int a, int b, int c ) {
   if( a > 1 && b > 40 && c > 90 ) {
      a = b * c * e;
      b = a * c + e;
   }
   return b;
}
```

Final C Code

```
int a = 1, b = 1, d = -2, e = 22;
int f = 1, res = 0;
main() {
   int b = 12, i;
   for( i = 1; i < 400; i++ ) {
      a++; b++; f++; e++;
      res = funct( a, b, f );
   }
}

int funct( int a, int b, int c ) {
   if( c > 90 && b > 40 && a > 1 ) {
      a = b * c * e;
      b = a * c + e;
   }
   return b;
}
```

Fig. 13. An example of substitution of a variable passed as an address with a local variable

requires a dynamic analysis of the conditional expression (and/or some designer directives) since information concerning the probability to be true or false are crucial to optimize the sub-expressions reorganization. Such statistics are computed by combining iteratively the probabilities of the involved sub-expressions. In particular, by representing with P_a and P_b the two sub-conditions probabilities, the probability that `a && b` is true is $P_a \cdot P_b$ while the probability that `a || b` is true is $P_a + P_b - P_a \cdot P_b$. For example, by considering the conditional expression `a && (b || c) && e || d` where $P_a = 0.3$, $P_b = 0.2$, $P_c = 0.5$, $P_d = 0.5$; $P_e = 0.1$, `(b || c)` is reorganized by swapping the position of `b` and `c`; then, since $P_{b||c} = 0.2+0.5-0.1 = 0.6$, the sub-expression `a && (c || b) && e` is reorganized obtaining `e && a && (c || b)` that induces, since $P_{e\&\&a\&\&(c||b)} = 0.1 \cdot 0.3 \cdot 0.6 = 0.018$, the final reorganization `d || e && a && (c || b)`. The higher is the number of sub-expressions and their relative difference of probability, the higher is the effectiveness of this optimization strategy. The proposed transformation reduces the energy consumption due to control operations, but a complete analysis requires considering the energy consumption of the involved arithmetic operators (if any exist); in particular, the operations complexity (comparison, sums, products, etc.) could induce a reordering modification with respect to the simple probability-based approach. For this reason, a more general $F(energy, probability)$ cost function has been introduced. Consequently, in a 2 AND–connected conditional expression, a low-energy high-probability sub-condition could be placed before a high-energy low-probability sub-condition while in a 2 OR–connected conditional expression, a low-energy low-probability sub-condition could be placed before a high-energy high-probability sub-condition. The analysis performed on the example reported in Figure 13 has produced the set of data summarized in Table 11.

Table 11. Power and energy data for conditional sub-expression reordering

Parameter	Original	Transformed	%
Clock Cycles	14759.00	14309.00	-3.05
Processor Energy (μJ)	4.35	4.25	-2.35
Processor avg. Power (mW)	29.50	29.72	-0.72

6.2. *Function Call Preprocessing*

This transformation associates with a specific function a proper set of macros that will substitute a function call with either an equivalent but low energy function call or a specific result; in short, the transformation skips a function call, or reduces its impact, when its actual parameters allow to directly identify either the returned value or another equivalent function. Figure 14 shows a simple but meaningful example. This transformation presents some potential drawbacks depending on the application. In particular, both the increase of code size and the possible insertion of some control conditions have to be justified by a significant probability to exploit the power saving of the transformation. By considering, as an example, the function acos(x), the substitution saves energy since the implementation of such a function does not include a pre-computed value for -1. Consequently, the advantage of the substitution is twofold: no energy is used to call the function and no energy is used to compute the corresponding value.

Original C Code

```
#include <math.h>
main() {
   int i, val;
   float r1, r2;
   for( i = 1; i < 200; i++ ) {
      val = i % 5;
      r1 = sqrt( val );
   }
   for( i = -20; i < 60; i++ ) {
      r2 = fabs( val );
   }
}
```

Final C Code

```
#include <math.h>
#define sqrt(x) ((x==0)?0:(x==1)?1:sqrt(x))
#define fabs(x) ((x>=0)?x:-x)
main() {
   float r1, r2;
   for( i = 1; i < 200; i++ ) {
      val = i % 5;
      r1 = sqrt( val );
   }
   for( i = -20; i < 60; i++ ) {
      r2 = fabs( val );
   }
}
```

Fig. 14. An example of function call preprocessing

The analysis of the example of Figure 14 led to the results gathered in Table 12.

Table 12. Power and energy data for function call preprocessing

Parameter	Original	Transformed	%
Clock Cycles	15202.00	14818.00	-2.52
Cache Misses	834.00	823.00	-1.32
Cache Hit	17885.00	17316.00	-3.18

The macros listed in Table 13 have been considered taking into account their frequency within a representative benchmark set. Obviously, the same approach applies to other libraries. Do note that these transformations have some side effects

Table 13. Macro definition for functions in math.h

Macro name	Macro definition
acos(x)	((x==-1) ? 3.141592653589793238462643383 : acos(x))
asin(x)	((x==0) ? 0 : asin(x))
atan(x)	((x==0) ? 0 : atan(x))
cos(x)	((x==0) ? 1 : cos(x))
sin(x)	((x==0) ? 0 : sin(x))
tan(x)	((x==0) ? 0 : tan(x))
exp(x)	((x==0) ? 1 : (x==1) ? 2.718281828459045235360287471 : exp(x))
log(x)	((x==1) ? 0 : log(x))
log10(x)	((x==1) ? 0 : log10(x))
pow(x,y)	((y==1) ? x : pow(x,y))
sqrt(x)	((x==0) ? 0 : (x==1) ? x : sqrt(x))
fabs(x)	((x>=0) ? x : -x)

related to the application of the expansion. For example, the expanded function `fabs(i++)` causes a double increment of the variable `i` making the transformation inconsistent. In such a case, two alternative solutions are possible: the first is to prevent the macro expansion (i.e. to use the original function) by enclosing the function name in parentheses, i.e. `(fabs)(i++)`; the second solution consists in postponing the increment after the function evaluation, i.e. `fabs(i); i++`. Note that the second solution only exploits the advantages of the transformation. Other proposals that can be found in literature consider the **Optimization of Modulo-Division** by substituting the modulo operator with a sequence of cascaded conditions in order to avoid expensive ALU operations.

7. Design Methodology: A Case Study

The huge number of available transformations faces the designer with the following critical issues. First of all, it must be noted that transformations are not decoupled. In fact, the effectiveness of a given transformation is strongly influenced by the structure of the code it operates on. Hence, particular attention has to be devoted to identify a proper order of application of different transformations. As a consequence, the design space grows exponentially with the number of envisioned transformations. For this reason, it is valuable to identify some criteria to restrict the analysis—at each refinement step—only to those transformations expected to be more promising. To fit industrial environment needs, this methodology should be supported by a proper tool chain, possibly integrated with widespread development frameworks. In particular, the availability of a user-driven, automated tool to perform the source code transformation is crucial to reduce the error-proneness of the manual intervention and to speed-up the design-space exploration. Moreover, a retargetable analysis tool to evaluate the actual benefits deriving from the application of the selected transformations, at each refinement step, is also essential.

7.1. *Design flow*

Based on the above considerations, a design flow such as that of Figure 15 can be envisioned. Starting from the characteristics of the source code, a set of applicable transformations is selected among all those available. A second step of analysis is devoted to determine which of these transformations are expected to produce better improvements. The transformations selected at the end of these two steps are then applied producing different source codes, whose power and energy properties are then evaluated and compared. The code of the best candidate becomes the input for a new iteration of the optimization flow. A realistic design flow requires human intervention at least in the steps involving transformation selection. Heuristic techniques can, in fact, only support the experience of a designer but cannot completely replace it. A typical situation that requires user suggestions is when a transformation does not produce any improvement by itself but enables the application of a further transformation, possibly magnifying its benefits. Under a more

Fig. 15. Design flow

general perspective, this corresponds to perform a branch-and-bound exploration of the design space where part of the bounding is performed capitalizing user's expertise, relieving him/her from the tedious decisions that can be automated thanks to reliable heuristic criteria. Branching, on the other hand, can be automated by means of language parsing and manipulation tools, such as SUIF2[16].

7.2. *A case study: IIR Filter*

In this section the methodology is applied to the case study of an IIR filter. The first selection step led to the following candidate transformations:

(i) Loop unrolling
(ii) Variable expansion
(iii) Software pipelining
(iv) Exit-condition modification
(v) Scalarization of array elements

Based on previous experience, transformations (i), (ii) and (iii) have been grouped in sequence to take full advantage of their mutual synergies. The new set of transformations is thus:

(a) Loop unrolling → Variable expansion → Software pipelining
(b) Exit-condition modification
(c) Scalarization of array elements

Transformation (a), (b) and (c) have then been repeatedly applied to the source code. The sequences of transformations are depicted in Figure 16 where a solid outline of the boxes indicates that the corresponding transformation is accepted while a dashed outline means that the transformation is rejected. In this study, the selection criterion has been the reduction of the total (system) energy. The analysis of the tree of Figure 16 reveals that only three different paths are obtained. The quality of these solutions are summarized in Table 14. In conclusion the best results are obtained by first applying transformation (a) and then transformation (b).

Fig. 16. Transformation application sequences

Table 14. Comparison of the results obtained for the IIR filter optimization

Sequence	Processor Energy	System Energy	Total Energy
(a), (b)	-5.5 %	-5.7 %	-5.8 %
(a), (c)	-5.4 %	-4.6 %	-4.7 %
(a)	-2.8 %	-3.6 %	-3.7 %

8. Concluding Remarks

Our preliminary software power optimization framework has been tested considering several full-size benchmarks. Different source level transformations have been applied, including those proposed in this paper. Table 15 shows the measured energy reduction of the transformed code with respect to the original one. Note that the reported energy reductions are an underestimate of the actual ones, since the

Table 15. Processor energy improvements for some benchmarks

Application	Total Energy
CRC-16	-5.6 %
WAVE	-9.6 %
HASH	-4.0 %
Bubble Sort	-3.7 %
Matrix Multiplication	-6.2 %
IIR Filter	-5.8 %

constant cost of application start/exit has a strong influence for these small benchmarks.Current effort is devoted to analyze the impact of different cache/memory configurations, less conservative than the one considered in this paper.

The systematic analysis of source-to-source transformations, part of which has been described in this paper, allowed to highlight potential benefits and possible side-effects, as summarized in Tables 16, 17, 18 and 19.

Table 16. Loop transformations

ID	Transformation	Advantages	Side-effects
L1	Loop unrolling	↑ Loop count ↑ Parallelism Enables compiler optimizations Enables (L2) (D4)	↑ Code size ↑ I-cache miss ↑ I-cache miss
L2	Variable expansion	↑ Parallelism ↓ Data-dependent stalls	↑ Loop body code size
L3	Loop distribution	↓ Loop body code size ↓ I-cache miss	↑ Branching ↓ Timing performance
L4	Exit-condition	↓ Condition complexity	
L5	Loop fusion	↓ Branching ↓ D-cache miss Enables (L1), (L8)	↑ Loop body code size
L6	Loop interchange	↑ Parallelism ↑ Array access efficiency ↑ Loop-invariant expression #	↑ Memory access efficiency
L7	Loop tiling	↓ D-cache miss ↑ Array access efficiency	↑ Code size ↑ Branching
L8	Software pipelining	↓ Pipeline stalls ↑ Parallelism	↑ I-cache miss
L9	Loop unswitch	↓ I-cache miss ↓ Branching	↑ Code size

Table 17. Data transformations

ID	Transformation	Advantages	Side-effects
D1	Array declaration sorting	↑ Use of low-power addressing modes	N/A
D2	Array scope modification	↑ Use of low-power addressing modes	↑ Memory usage
D3	Temporary array insertion	↓ D-cache miss	↑ Branching
D4	Scalarization of array elements	↓ D-cache access ↑ Register usage efficiency	↑ Code size
D5	Local copy of global variable	↓ D-cache access ↑ Register usage efficiency	↑ Code size
D6	Global variable initialization	↓ Memory access	N/A
D7	Pointer-chain reduction	↓ Memory access ↓ Executed instructions	N/A

Table 18. Inter-procedural transformations

ID	Transformation	Advantages	Side-effects
S1	Function inlining	↓ Context switching Enables compiler optimizations	↑ Code size
S2	Scope reduction of by-address parameters	Enables compiler optimizations	↑ Code size

Table 19. Operator and control flow transformations

ID	Transformation	Advantages	Side-effects
C1	Modulo division elimination	↓ ALU usage	↑ Code size
C2	Conditional expression reordering	Enables shortcuts	N/A
C3	Function call preprocessing	↓ Context switching	↑ Code size ↑ Branching

References

1. E. Macii, M. Pedram and F. Somenzi, “High-level power modeling, estimation, and optimization,” *IEEE Trans. On CAD,* Vol. 17, N. 11 (1998) 1061–1079.
2. W. Ye, N. Vijaykrishnan, M. Kandemir and M.J. Irwin, “The design and use of SimplePower: a cycle-accurate energy estimation tool,” *Proc. of IEEE Design Automation Conference,* (2000) 340–345.
3. W.–T Shiue and C. Chakrabarti, “Memory exploration for low power embedded systems,” *Proc of IEEE Symp. on Circuits and Systems,* Vol. 1 (1999) 250–253.
4. M.B. Kamble and K. Ghosse, “Analytical energy dissipation models for low power caches,” *Proc of IEEE Symp. Low Power Electronics and Design,* (1997) 143–148.
5. C. Brandolese, W. Fornaciari and F. Salice, “Code-level transformations for software power optimization,” *CEFRIEL, Tech. Rep. N. RT-02-004,* (2002).
6. S. Gupta, M. Miranda, F. Catthoor and R. Gupta, “Analysis of high-level address code transformations for programmable processors,” *Proc. of IEEE Design Automation Conference in Europe,* (2000) 9–13.
7. C. Hulkarni, F. Catthoor and H. De Man, “Code transformations for low power caching in embedded multimedia processors,” *Proc. of IPPS/SPDP,* (1998) 292–297.
8. E. De Greef, F. Catthoor and H. De Man, “Program transformation strategies for memory size and power reduction of pseudoregular multimedia subsystems,” *IEEE Trans. on Circuits and Systems for Video Technology,* Vol. 8, N. 6 (1998) 719–733.
9. C. Brandolese, W. Fornaciari, L. Pomante, F. Salice and D. Sciuto, “A multi-level strategy for software power estimation,” *Proc. of IEEE Intnl. Symp. on System Synthesis,* (2000) 187–192.
10. C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, “Source-Level Execution Time Estimation of C Programs,” *Proc. of IEEE Hardware/Software Codesign Workshop,* (2001) 98–103.
11. N. Bellas, I.N. Hajj, C.D. Polychronopoulos and G. Stamoulis, ”Architectural and compiler techniques for energy reduction in high-performance microprocessors,” *IEEE Trans. on VLSI,* Vol. 8, N. 3 (2000) 317–326.
12. V. Tiwari, S. Malik and A. Wolfe, “Compilation techniques for low energy: an overview,”

Proc. of IEEE Intnl. Symp. on Low Power Electronics, Digest of Technical Papers, (1994) 38–39.

13. L. Benini and G. De Micheli, "System-level power optimization: Techniques and tools," *ACM Trans. on Design Automation of Electronic Systems,* Vol. 5 (2000) 115–192.
14. R. Mehta, R.M. Owens, M.J. Irwin, R. Chen and D. Ghosh, "Techniques for low energy software," *Proc. of Intnl. Symp. on Low Power Electronics and Design,* (1997) 72–75.
15. D. Burger and T.M. Austin, "The SimpleScalar Tool Set, version 2.0," *University of Wisconsin-Madison, Comp. Sci. Dept., Tech. Rep. N. 1342,* `http://www.cs.wisc.edu/∼mscalar/simplescalar.html` (1997).
16. The Stanford SUIF Compiler Group , "SUIF Compiler System," *Stanford University,* `http://suif.stanford.edu/` (1999).