Source Code Transformation based on Software Cost Analysis *

Eui-Young Chung
CSL, Stanford Univ., USA
eychung@stanford.edu

Luca Benini
DEIS, Univ. di Bologna, Italy
lbenini@deis.unibo.it

Giovanni De Micheli
CSL, Stanford Univ., USA
nanni@stanford.edu

ABSTRACT

This paper presents a model and a strategy for source-code transformation applied to software application programs to reduce their energy cost. We propose a flexible performance and energy model for a processor-memory system. The benefit of the model is generality (it is not tied to a single memory and processor architecture) and effectiveness of evaluation. With this model, we first estimate the effects of source-code transformations (called transformation cost), representing the improvement ratios of processor cycles, I-cache misses, and D-cache misses. Next, we combine the transformation cost model with hardware parameters to estimate the actual effect of a transformation on performance and energy. The model can be used to guide software transformation selection for power and performance. The experimental results show that the proposed approach finds the optimal transformation in 95% of the cases, and that the penalty when the non-optimal transformation is selected is within 5%.

1. INTRODUCTION

Most electronic systems execute software programs on processor chips or cores. Energy-efficiency of the overall system depends heavily on software design [4]. Low-energy software design can be achieved in different ways, namely by energy-aware selection of the algorithms [15], code restructuring [7, 8] and instruction-level optimizations [3]. While algorithm selection has the highest potential, it is hard to automate, and its impact strongly depends on programmer's ingenuity. In contrast, instruction-level approaches can be automated (performance-oriented optimizations are available in the back-end of most compilers), but their impact on energy is local, and strongly tied to a given target architecture. Code restructuring techniques lie in between, since they can be automated to some degree [12], but they have global impact and they are not strictly architecture-dependent. This paper addresses strategies for source-code restructuring. The critical issue in code restructuring for low energy is the estimation of the impact of a given transformation. A straightforward approach (which we call *iterative-ISS*) is to compile the restructured code, generate an executable and run it either on the target hardware, or on a power-aware instruction simulator [14], to measure energy savings. This estimation flow goes through several time-consuming steps and it ultimately prevents fast, iterative exploration of many alternative transforms. Hence, more abstract and computationally-efficient energy estimation metrics are needed to support optimization.

A traditional abstract code metric is compactness. The most compact code for a program uses the least instruction memory. Moreover, if the program represents pure data flow, (i.e., no branching and iteration is involved), it executes in the shortest time and consumes the least energy. (This holds under the assumptions of constant energy cost of the instructions and if we neglect specific architectural features of processors, that may favor some instructions over some others.) This argument breaks down when considering processor-memory systems, and in particular the fact that accessing memory may consume a significant amount of energy. More refined abstract metrics rely on profiling[1, 2]. In these cases, the effects of branching/iteration may have significant impact on performance/energy. However, profiling relies on instruction simulation, which is too time consuming to be repeated for every possible transformation.

In previous work on source-code transformations, the estimation issue has been partially bypassed by focusing on hardware targets and application domains where some simplifying assumptions hold. Catthoor and coauthors [8], assume that the data memory access cost is the dominant factor for both energy and performance. Therefore, they apply extensive loop transformations for reducing data accesses and improving their locality. Other approaches [7] focused instead on the number of processor cycles, implicitly assuming that the processor is the most energy-critical system component. Thus, loop unrolling and procedure inlining were used to reduce the number of processor cycles, while data locality was improved by cache size optimization. Several papers have addressed the problem of assessing the impact of source code transformations on families of hardware architectures [5, 6, 7]. In these works, instruction-level simulation is employed to measure the effects of code transformation on energy. Without exception, these works have concluded that the optimal transformation depends on the characteristics of the processor and of the memory system.

On the other hand, pure analytical models were proposed especially for memory oriented transformations [9, 16]. These approaches estimate the effect of transformation very fast, but the accuracy is lower than the *iterative-ISS* approaches.

This paper introduces an abstract hardware model that makes it possible to take into account hardware characteristics when assessing the effectiveness of code transformations, at a fraction of the computational effort that would be required by the straightforward *iterative-ISS* approach. Bene-

*This work was supported by NSF (CCR-9901190), MARCO, ARPA, and GSRC.

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada

fits of using this model include generality and speed of evaluation, since a drastically reduced number of instruction-level simulations is required. Thus, our work is in the middle of the previous two approaches - *iterative ISS* approaches and pure analytical approaches.

We apply our techniques to tune and select two well-known transformations, namely *loop unrolling* and *loop blocking* for several target programs. Results show that our technique is accurate in predicting the impact of code transformations. Interestingly, our analysis also demonstrates that a program should be transformed in different ways depending on the target cost metrics (energy and performance) as well as on hardware configuration (processor and memory). As a consequence, we give further evidence of the fact that energy and performance are not always optimized jointly.

2. SOFTWARE COST MODEL AND OVERALL FLOW

In this section, we introduce the abstract system performance and energy model at the basis of our approach, and we describe a computationally-efficient estimation flow for transformation analysis and exploration. It is important to stress that the abstract model should not be used for estimating power and performance of a program in an absolute sense, but it should be intended as a selection criterion to choose among alternatives. Thus, the emphasis is not on absolute accuracy, but on reliable selection guidance.

2.1 System Model

The proposed technique adopts a general system model which consists of a processor and an external memory. The processor is modeled as three major components - processor core, I-cache, and D-cache. The processor core includes all other components such as ALU, branch prediction unit, etc.

Based on this simple system model, we relate the software behavior to component usage. In other words, during software execution, the system is in one of three possible states: *i)* only processor is in working state (the processor is processing instructions and data stored in caches or registers), *ii)* only memory system is in working state (the processor stalls while external memory is accessed), *iii)* both processor and memory system are in working state (the processor does not immediately stall during external memory accesses). The third state can be omitted for simple processors which are completely stalled during memory access. For the memory system, we consider two different sub-states of its working state to distinguish its triggering sources - I-cache or D-cache misses.

2.2 Simplified Software Cost Model

For the sake of explanation, in this section we describe a simplified system model, assuming that - *i)* processor cycle and memory activity are completely non-overlapping. *ii)* the memory accesses are mostly due to D-cache miss, *i.e.* I-cache miss is negligible compared to D-cache miss. These simplifying assumptions will be removed in Section 2.4. The hardware parameters used in our cost model are shown in Table 1. For the given system, we perform instruction-level simulation for the target software to obtain the usage of each component. We denote the number of clock cycles devoted to processor as N_p and the number of accesses to memory system as N_m. Thus, execution time of the target program

parameter	meaning
T_p	processor cycle time
T_m	memory cycle time
P_{ap}	average active power of processor
P_{ip}	average idle power of processor
P_{am}	average active power of memory
P_{im}	average idle power of memory

Table 1: Hardware parameters of the system model

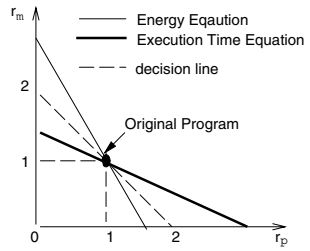

Figure 1: 2D representation of Equation 5 and 6

can be simply represented as Equation 1.

$$T_{exe} = N_p * T_p + N_m * T_m \quad (1)$$

Similarly, the energy consumption is:

$$E = (P_{ap} + P_{im}) * N_p * T_p + (P_{ip} + P_{am}) * N_m * T_m \quad (2)$$

Suppose we have a set of transformation techniques, $\mathcal{T}_i, i = 0, 1, \cdots, K-1$, and each of them changes N_p and N_m to $r_p(i) * N_p$ and $r_m(i) * N_m$, respectively. We call $r_p(i)$ ($r_m(i)$) *transformation cost of processor cycle (memory access) ratio* of transformation $\mathcal{T}_i$. Thus, Equation 1 and 2 can be rewritten as Equation 3 and 4, respectively to consider the effect of the transformation.

$$T(i)_{exe} = r_p(i) * N_p * T_p + r_m(i) * N_m * T_m \quad (3)$$

$$\begin{aligned} E(i) =& (P_{ap} + P_{im}) * r_p(i) * N_p * T_p \\ &+ (P_{ip} + P_{am}) * r_m(i) * N_m * T_m \end{aligned} \quad (4)$$

where, $T(i)_{exe}$ and $E(i)$ represent the execution time and energy consumption of the code transformed by $\mathcal{T}_i$.

Obviously, $\mathcal{T}_i$ is only effective when $T_{exe} > T(\mathcal{T}_i)_{exe}$ if the target objective is performance. Similarly, it is only effective when $E > E(\mathcal{T}_i)$ if the target objective is energy. These two inequalities can be rearranged with respect to r_p and r_m as shown in next.

$$(r_m(i) - 1) \leq (1 - r_p(i)) * \frac{N_p * T_p}{N_m * T_m} \quad (5)$$

$$(r_m(i) - 1) \leq (1 - r_p(i)) * \frac{N_p * T_p}{N_m * T_m} * \frac{P_{ap} + P_{im}}{P_{am} + P_{ip}} \quad (6)$$

These two equations characterize the software cost (energy and performance) relations in terms of r_m and r_p when a general transformation technique is applied. Each equation is represented as a line in a two-dimensional space, as shown in Figure 1. First, each equation defines the boundary conditions for r_p and r_m that can be acceptable after the transformation. Only when r_p and r_m are in the region below the line (the triangle formed by two axes and the line itself), the transformation is effective for the given cost metric. We call this region *effective region*.

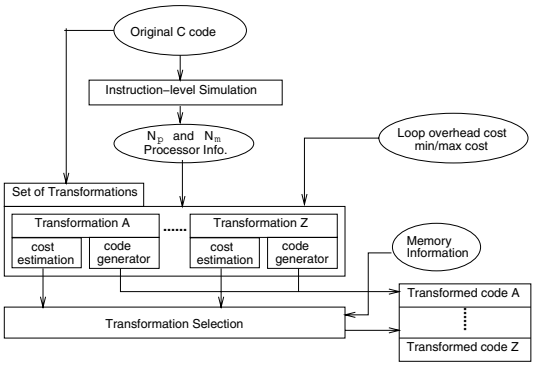

Figure 2: Overall transformation flow

Second, the slope of the line indicates the relative importance of r_p and r_m. In detail, r_p (r_m) is more important than r_m (r_p) if the slope is greater (smaller) than 1 (the slope of decision line) because the given cost metric is more heavily affected by $r_p(r_m)$. Thus, performance and energy may require different transformations.

Both equations can be generalized to directly compare two arbitrary transformations - $\mathcal{T}_i$ and $\mathcal{T}_j$ as follows.

$$(r_m(i) - r_m(j)) \leq (r_p(j) - r_p(i)) * \frac{N_p * T_p}{N_m * T_m} \quad (7)$$

$$(r_m(i) - r_m(j)) \leq (r_p(j) - r_p(i)) * \frac{N_p * T_p}{N_m * T_m} * \frac{P_{ap} + P_{im}}{P_{am} + P_{ip}} \quad (8)$$

The condition which minimizes the energy-delay product can be simply obtained by multiplying Inequalities 7 and 8 because each of them is the condition for execution time and energy, respectively. Also, notice that the left hand side terms of Inequalities 7 and 8 are identical, thus they always have the same polarity and the right hand side terms of these two equations are always positive. Thus, the inequality for energy-delay product is:

$$(r_m(i) - r_m(j)) \leq (r_p(j) - r_p(i)) * \frac{N_p * T_p}{N_m * T_m} * \sqrt{\frac{P_{ap} + P_{im}}{P_{am} + P_{ip}}} \quad (9)$$

For each inequality (7, 8, 9), if the condition is satisfied, then $\mathcal{T}_i$ is superior to $\mathcal{T}_j$ for the corresponding cost metric. Otherwise, $\mathcal{T}_j$ improves the original program more than $\mathcal{T}_i$. Note that for each transformation $\mathcal{T}_i$ we need to provide a value for the transformation cost $r_p(i)$ and $r_m(i)$. Estimation of transformation cost is discussed in Section 3.

We can use these inequalities for two different purposes. First, for a system with a fixed external memory, we use these inequalities to find the optimal transformation among a set of transformations by estimating $r_m(i)$ and $r_p(i)$ of each transformation. Second, when there exist multiple choices of external memories, we can find the optimal transformation for various external memory configurations. Then, the impact of external memory selection and optimal transformation can be evaluated by Equation 3 and 4, thus optimal pair of external memory and transformation can be obtained.

2.3 Overall Transformation Flow

The proposed transformation flow is illustrated in Figure 2. The flow requires a single instruction-level simulation for the original source code, before transformations, to extract parameters useful for both transformation and cost estimation such as N_p, N_m, P_{ap}, P_{ip}. Currently, the WATTCH simulator [14] is used in the initial instruction-level simulation step. Also, P_{am} and P_{im} can be obtained from the data book or real measurement.

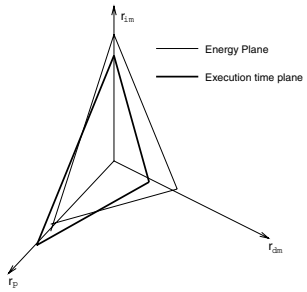

Figure 3: 3D representation of critical parameters

The transformation costs (r_p and r_m) are estimated by the dedicated cost estimators of each transformation. Each cost estimator requires additional information such as loop overhead and min/max operation cost which are independent of the characteristics of each program. Techniques for cost estimation, and their computational cost, are described in Section 3.

In the transformation selection phase, we evaluate two well-known high-level transformation techniques - *loop unrolling* and *loop blocking*. *Loop unrolling* is implemented under *SUIF* environment and *loop blocking* is performed by another *SUIF* package called *skweel* [13]. Notice that our framework can encapsulate any other class of transformations, provided that a cost estimator is available. In the next section, we illustrate the cost estimators we developed for *loop unrolling* and *loop blocking*.

2.4 General Cost Model

In Section 2.2, we ignored the effect of I-cache. Also, we assumed external memory accesses are never overlapped with processor cycles. We now extend the simplified model to consider these effects.

I-cache effects can be simply considered by breaking N_m into $N_{dm} + N_{im}$ which are number of memory accesses due to D-cache miss and I-cache miss respectively. The graph shown in Figure 1 becomes 3-dimensional as shown in Figure 3. We can still use the inequalities introduced in Section 2.2 by choosing the most critical plane. The most critical plane has the smallest *effective region* and the parameters consisting of the plane replace the parameters appearing in each equation.

Finally, the overlapping between memory access and processor operation for processors with non-blocking caches is considered by scaling the memory cycle time. The scaling factor can be achieved by performing instruction-level simulation twice. The first simulation is performed with zero-latency (ideal) memory system and the second simulation is performed with non-ideal memory system. The scaling factor is the difference in execution time divided by the product of the latency of the non-ideal memory system and N_m. It is a measure of the aggressiveness of the non-blocking cache implementation, rather than a property of software execution. Thus, we compute the factor for a set of programs and use the average value for all other programs.

```
for (i = 0; i < TC; i++)
  X[i] = Y[i]*Z[i];
```

(a) Original version

```
for (i = 0; i < TC-1; i++) {
  X[i] = Y[i]*Z[i];
  i++;
  X[i] = Y[i]*Z[i];
}
for (; i < TC; i++)
  X[i] = Y[i]*Z[i];
```

(b) Unrolled version (u = 2)

Figure 4: An example of loop unrolling

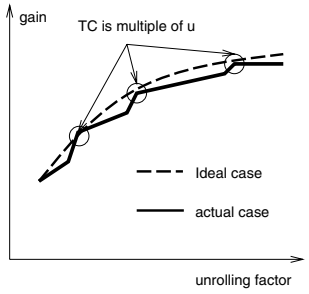

Figure 5: Gain variation with unrolling factor

3. TRANSFORMATION COST ANALYSIS

In this section, we describe the cost estimators of the transformation techniques which estimate r_p and r_m. We consider two well-known transformation techniques - *loop unrolling* and *loop blocking(tiling)*. *Loop unrolling* aims at reducing the number of processor cycles by eliminating the loop overheads. Also, it provides a better starting point to the conventional optimization techniques by enlarging the basic block size. This technique improves the number of processor cycles (N_p), but it also increases I-cache misses (N_{im}). N_{dm} is rarely sensitive to this technique because it does not change the data access behavior.

Loop blocking breaks large arrays into several pieces (tiles smaller than cache size) and reuses each piece without self-interference. Thus N_{dm} can be effectively reduced. But it may cause significant increase of N_p because the depth of loop nesting is increased and min/max operations are required to compute the lower and upper bounds of the blocked loop. The variation of N_{im} can be ignored in this approach because the code size increase is marginal.

3.1 Loop Unrolling

Loop unrolling is controlled by the *unrolling factor*, u which is the number of duplications of the body statements inside the loop. If $u = 1$, the original loop is kept without any change. Even though it is well-known that *loop unrolling* is effective to reduce N_p, there is no obvious method to determine this factor. Thus, typically *iterative-ISS* is required with the change of the unrolling factor to find the optimal one. We propose a new unrolling strategy to find the optimal unrolling factor with a single *ISS*.

An example of *loop unrolling* is shown in Figure 4 ($u = 2$). The gain of loop unrolling is achieved by the first loop of Figure 4 (b), while the second loop is same to the original loop and its number of iterations is (TC *modulo* u).

We denote the gain of loop unrolling as $gain(u)$ when the unrolling factor is u. The $gain(u)$ can be graphically represented as shown in Figure 5. In Figure 5, the solid line (actual case) is when the number of loop iterations is limited to integer numbers (the second loop in Figure 4 (b) is iterated by (TC *modulo* u) times), while the dotted line (ideal case) is when the number of loop iterations can be non-integer numbers, which is not possible in practice (the second loop in Figure 4 (b) is never executed) . Thus, when u is selected as a multiple of TC, the unrolling effect becomes as same as the ideal case. Also, u should be selected as large as possible to maximize the gain. But, as u increases, the I-cache miss rate also increases and will eventually decrease the gain achieved by loop unrolling.

Without the estimations of both gain and I-cache miss rate, it is unavoidable to perform *iterative-ISS* with the change of u.

Instead of measuring I-cache miss rate directly, we estimate the code size increased by loop unrolling. Based on the code size estimation, we decide the maximum unrolling factor, u_{max}. u_{max} is the largest unrolling factor not to increase code size larger than I-cache size and is shown next.

$$u_{max} = \frac{size\ of\ instruction\ cache}{I_{org} * size\ of\ instruction} \tag{10}$$

where, I_{org} is the number of instructions of the loop body and is estimated in either front-end (SUIF) or back-end part.

Thus, the optimal unrolling factor u_{opt} is the greatest divisor of TC, but smaller than u_{max}. By finding u_{opt}, we can replace *iterative-ISS* by single ISS to obtain $gain(u_{opt})$.

The transformation costs of loop unrolling are computed using u_{opt} and shown next.

$$\begin{aligned} r_p &= (N_p - gain(u_{opt}))/N_p \\ r_{im} &= (N_{im} + (I_{org} * u_{opt}))/N_{im} \\ r_{dm} &= 1 \end{aligned} \tag{11}$$

Notice that r_{dm} is set to 1 without loss of generality because its effect on data access behavior is relatively small.

3.2 Loop Blocking

This technique is very effective to reduce the number of D-cache misses, but it often increases N_p largely due to the overhead of loop bound decision. To estimate r_{dm}, we propose an estimation technique for the number of misses caused by *loop blocking*. An example of loop blocking is shown in Figure 6. Two innermost loops (loop k and j) in the original version are blocked to avoid self-interference and the graphical representation is shown in Figure 7. One tile of array `Z` (BxB words) is fully used during the iteration of two innermost loops in the tiled version (loop k and j), whereas only B words of array `Y` and `X` are used, respectively. Also, loop i uses the same tile of array `Z` used in two innermost loops, while each iteration uses different B words of array `X` and `Y`. Therefore, unavoidable misses to complete one iteration of loop i, namely intrinsic misses are:

$$M_{intrinsic} = \frac{B^2}{CL}(array\ \texttt{Z}) + 2\frac{N * B}{CL}(array\ \texttt{X}\ and\ \texttt{Y}) \tag{12}$$

where CL is the cache line size in terms of words.

Notice that the tile size B can be chosen such that there is no self-interference using the tile size selection algorithms presented in [9, 10, 11]. Among them, we use the algorithm proposed in [10].

The misses due to cross interference can be estimated using the `footprint` of arrays in loop k. The ratio of the space occupied by array `Z` over the cache size, CS is $B * B/CS$.

```
for (i = 0; i < N; i++)
  for (k = 0; k < N; k++)
    r0 = Y[i][k];
    for (j = 0; k < N; k++)
      X[i][j] += r0*Z[k][j];
```

(a) Original version

```
for (kk = 0; kk < N; kk+=B)
  for (jj = 0; jj < N; jj+=B)
    for (i = 0; i < N; i++)
      for (k = max(0, kk) < min(N, kk+B-1); k++)
        if (jj = 0)
          r0 = Y[i][k];
        for (j = max(0, jj); j < min(N, jj+B-1); j++)
          X[i][j] += r0*Z[k][j];
```

(b) Tiled version

Figure 6: An example of loop blocking (Tiling)

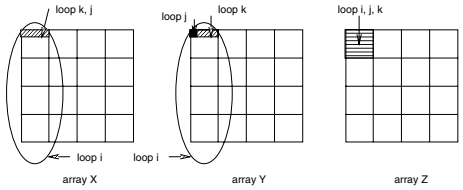

Figure 7: Data access pattern of loop blocking

Similarly, the ratio for array `X` and `Y` over the cache size is commonly B/CS. Thus, the probability that two or more references will access the same cache line is:

$$P_{cross} = \frac{2 * B^2}{CS} * \frac{B}{CS} + (\frac{B}{CS})^2 + \frac{B^2}{CS} * (\frac{B}{CS})^2 \qquad (13)$$

and total cross-interference occurred per iteration of *loop i* is $M_{cross} = P_{cross} * B^2$, where B is the trip count of loop k and j. Therefore, total misses to complete the whole loop nest is shown next and r_{dm} is simply M_{total_block}/N_{dm}.

$$M_{total\_block} = (M_{intrinsic} + M_{cross} * N) * (\frac{N}{B})^2 \qquad (14)$$

On the other hand, r_{im} can be set to 1 because the code size increase is trivial, but r_p should be carefully analyzed.

$$N_p(blocking) = N_p + \sum_{i=0}^{N} (\prod_{l=0}^{k} TC_l * T_{mo}) \qquad (15)$$

where, N is the total number of *min* and *max* operations appearing in the tiled version and k is the loop level that each *min* or *max* operation can be moved without destroying the dependency. Also, T_{mo} is the cost of *min* or *max* operation. Notice that the cost of *min/max* operation can be characterized by simulating a simple program which only includes *min/max* operation and the cost can be generally used for all other programs.

In Figure 6 the *min* and *max* operations in *level j* can be moved up to *level jj*, thus its impact on N_p is marginal. But if the tiled version is skewed or has triangle-shape tile, these operations cannot be moved, thus its impact cannot be neglected. Finally, r_p is simply $N_p(blocking)/N_p$.

4. EXPERIMENTAL RESULTS

The experiment was conducted based on WATTCH simulator. The processor was configured such that it had one arithmetic unit for both integer and floating point operation without L2-cache. Also, both D-cache and I-cache were 4K and their associativity was 2.

external memory	latency	power
M1	30	0.1
M2	50	0.1
M3	30	0.5
M4	50	0.5

Table 2: Four different memory configurations

M	pgm	cost metrics					
		performance		energy		perf. * energy	
		unroll	block	unroll	block	unroll	block
1	m100	0.82	0.75*	0.80*	0.91	0.66	0.68+
1	m200	0.85	0.74*	0.80*	0.90	0.68	0.67*
1	l100	0.82*	0.89	0.81*	1.00	0.66*	0.89
1	l200	0.84*	0.90	0.84*	0.96	0.73*	0.86
1	sub	0.88*	1.02	0.77*	1.02	0.68*	1.04
1	idct	0.82*	1.06	0.69*	1.06	0.56*	1.12
2	m100	0.85	0.63*	0.82	0.82*	0.70	0.52*
2	m200	0.85	0.62*	0.82	0.82*	0.70	0.51*
2	l100	0.85	0.80*	0.82*	0.93	0.70*	0.74
2	l200	0.89	0.83*	0.87*	0.96	0.77*	0.80
2	sub	0.91*	1.01	0.80*	1.00	0.73*	1.01
2	idct	0.85*	1.07	0.72*	1.07	0.61*	1.12
3	m100	0.82	0.75*	0.83	0.78*	0.71	0.59*
3	m200	0.85	0.74*	0.83	0.78*	0.71	0.58*
3	l100	0.82*	0.89	0.84*	0.95	0.70*	0.76
3	l200	0.84*	0.90	0.85*	0.93	0.72*	0.84
3	sub	0.88*	1.02	0.82*	1.02	0.72*	1.04
3	idct	0.82*	1.06	0.72*	1.06	0.59*	1.12
4	m100	0.85	0.63*	0.86	0.67*	0.73	0.42*
4	m200	0.85	0.62*	0.86	0.67*	0.73	0.42*
4	l100	0.85	0.80*	0.85	0.84*	0.72	0.67*
4	l200	0.86	0.83*	0.87	0.85*	0.75	0.71*
4	sub	0.91*	1.01	0.85*	1.00	0.77*	1.01
4	idct	0.85*	1.07	0.77*	1.01	0.77*	1.08

Table 3: Decision accuracy of the proposed approach

We used four different external memory configurations shown in Table 2. Note that latency is in terms of processor cycle and the power consumption is normalized to the average power consumption of processor.

We applied our technique to two well-known programs used for loop-blocking - matrix multiplication (m100, m200) and LU-decomposition (lu100, lu200). Also, it was applied to two kernels of mp3 decoder - subbandSynthesis (sub) and inverse discrete cosine transform(idct) [15].

First, instruction-level simulation was performed for each program to extract necessary parameters - N_p, N_m, P_{ap}, and P_{ip}. Then, each program was transformed into two versions (unrolled version and tiled version) and transformation cost was estimated without considering specific memory configuration. Next, we performed the transformation selection step for each program by applying a set of Inequalities(7, 8, 9) to decide the most effective transformation. This step was repeated four times with four different different external memories shown in Table 2. To measure the accuracy of the selection step, the optimal transformation of each program was searched by performing the simulation for each transformed version. Table 3 shows the comparison between the decisions made by our technique and simulation results.

The numbers in Table 3 were obtained from the simulation and normalized to the original program. Notice that we marked our decisions with "*" and "+" in Table 3. "*" represents the correct decision, while "+" represents the wrong decision. The proposed technique achieves about 95% accuracy and the penalty due to wrong decisions was less than 5% over the correct decisions (For example, a wrong decision marked by a + sign was the case for the program $m100$ with memory configuration $M1$ for energy-delay product). Furthermore, our approach found the optimal transformation, even when the best execution time and best energy consumption were found in different versions ($m100$ and $m200$ in $M1$, $l100$ and $l200$ in $M2$).

Notice that this situation usually happens when the power consumption ratio of processor over the external memory is large. In this case, the slope difference of energy and execution time equations in Figure 1 becomes larger and each equation has different dominant factor.

Two kernels from mp3 decoders were not suitable for *loop blocking* due to small array size and complex array indexing, even though program *sub* suffers from data cache misses. Thus, more aggressive and general transformations for data locality improvement are required. It is also worthwhile to mention that the instruction-level simulation was performed twice (one for parameter extraction, one for loop unrolling) for each program to make a decision.

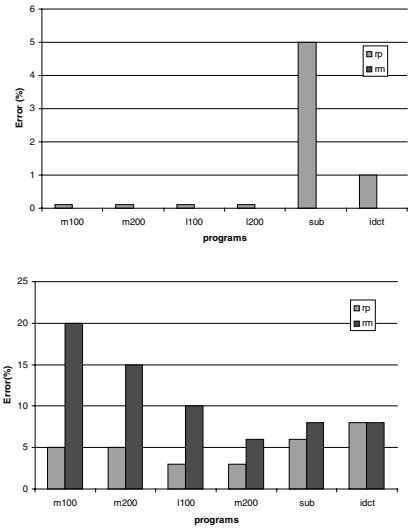

Figure 8: Transformation cost estimation error of loop unrolling (top) and loop blocking(bottom)

We also show the transformation cost estimation error of each transformation in Figure 8. On average, the estimation accuracy of *loop unrolling* is about 95% for r_p and 99% for r_m (almost invisible in Figure 8 (top) because the error is too small), while the estimation error of *loop blocking* is about 11.5% for r_{dm} and 4% for r_p.

5. CONCLUSION AND FUTURE WORK

We proposed an abstract software performance and energy estimation when multiple transformations are available. This approach greatly reduces the time required for code quality assessment compared to traditional approaches by avoiding *iterative-ISS*. The proposed approach found the optimal transformation with 95% accuracy and the penalty when the non-optimal transformation is selected is within 5%. Also, the proposed technique can easily reselect the optimal transformation when the memory configuration is changed. With this strategy, we were able to reduce the energy consumption in average by 20% and also improve the performance in average by 21%. It was also shown that both optimal performance and optimal energy consumption were not always found by single transformation because their improvement ratio is different depending on the power consumption ratio between the processor and memory.

Finally, we would like to address three limitations of our approach which will be enhanced in the future. First, our technique is limited to single-stage optimization, namely each optimization is considered independently. The reason is that the present tool has as a main objective the comparisons of transformations for different hardware parameters. We will extend our technique to support multi-stage optimization which may provide better quality of transformed code. Second, there are only two transformations are available, but we will encapsulate more high-level transformations such as *in-lining* in our framework. Third, the metric for the transformation is two-dimensional *i.e.* processor cycles and external memory cycles which are known as the most dominant factors in performance and energy consumption. But our appoarch can be extended to consider another metric such as on-chip memory cycles (by considering caches as separate components) using the similar concept.

6. REFERENCES

[1] T. Ball and J. Larus, "Optimally Profiling and Tracing Programs", Proceedings of the 19th Annual Symposium on Principles of Programming Languages, Jan., 1992

[2] E.-Y. Chung, L. Benini, and G. De Micheli, "Energy Efficient Source Code Transformation based on Value Profiling", *1st workshop for Compilers and Operating Systems for Low-Power*, pp. D-1-D.7, Philadelphia, PA, 2000

[3] V. Tiwari, S. Malik, A. Wolfe, "Instruction Level Power Analysis and Optimization of Software", *Journal of VLSI Signal Processing Systems*, vol. 13, pp.223-233, 1996

[4] L. Benini and G. De Micheli, "System-Level Power Optimization Techniques and Tools", *ACM TODAES*, vol. 5, issue 2, pp.115-192, Apr. 2000

[5] H. Mehta, R. Owens, M. Irwin, R. Chen, and D. Ghosh, "Techniques for Low Energy Software", *ISLPED*, pp.72-75, 1997

[6] G. Esakkimuthu, N. Vijaykrishnan, M. Kandemir, M. Irwin, "Memory system energy: influence of hardware-software optimizations," *ISLPED*, pp. 244-246, 2000.

[7] Y. Li and J. Henkel, "A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems", *Design Automation Conference*, pp.188-193, 1997

[8] F. Cathoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*, Kluwer, 1998

[9] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996

[10] M.S. Lam, E.E. Rothberg, and M.E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms", *ASPLOS*, pp. 63-74, 1991

[11] P.Panda, H. Nakamura, N. Dutt, and A. Nicolau, "Augmenting Loop Tiling with Data Alignment for Improved Cache Performance", *IEEE trans. on Computers*, vol. 48, No. 2, pp. 142-148, 1999

[12] D. Bacon, S, Graham, and O. Sharp, "Compiler Transformation for High-Perforamnce Computing", *ACM Computing Surveys*, pp.345-420, vol26, No. 4, Dec. 1994

[13] Stanford Compiler Group, The SUIF Library: A set of core routines for manipulating SUIF data structures, Stanford University, 1994

[14] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations", ISCA, pp. 83-94, 2000

[15] T. Simunic, L. Benini, G. De Micheli, and M. Hans, "Source Code Optimization and Profiling of Energy Consumption in Embedded Systems", *ISSS*, pp. 193-198, 2000.

[16] P. R. Panda, N. Dutt, and A. Nicolau. Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration. Kluwer, 1999.