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  
nann@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, L-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. Catherin 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-
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 OVER-ALL 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, L-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 substages of its working state to distinguish its triggering sources - L-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. L-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

<table>
<thead>
<tr>
<th>parameter</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_p )</td>
<td>processor cycle time</td>
</tr>
<tr>
<td>( T_m )</td>
<td>memory cycle time</td>
</tr>
<tr>
<td>( P_{ap} )</td>
<td>average active power of processor</td>
</tr>
<tr>
<td>( P_{ap} )</td>
<td>average idle power of processor</td>
</tr>
<tr>
<td>( P_{am} )</td>
<td>average active power of memory</td>
</tr>
<tr>
<td>( P_{am} )</td>
<td>average idle power of memory</td>
</tr>
</tbody>
</table>

Table 1: Hardware parameters of the system model

![Figure 1: 2D representation of Equation 5 and 6 can be simply represented as Equation 1.](image)

\[ T_{exc} = N_p \cdot T_p + N_m \cdot T_m \] (1)

Similarly, the energy consumption is:

\[ E = (P_{ap} + P_{am}) \cdot N_p \cdot T_p + (P_{ap} + P_{am}) \cdot N_m \cdot T_m \] (2)

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

\[ T(i)_{exc} = r_p(i) \cdot N_p \cdot T_p + r_m(i) \cdot N_m \cdot T_m \] (3)

\[ E(i) = (P_{ap} + P_{am}) \cdot r_p(i) \cdot N_p \cdot T_p + (P_{ap} + P_{am}) \cdot r_m(i) \cdot N_m \cdot T_m \] (4)

where, \( T(i)_{exc} \) and \( E(i) \) represent the execution time and energy consumption of the code transformed by \( T_i \).

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

\[ (r_m(i) - 1) \leq (1 - r_p(i)) \cdot \frac{N_p \cdot T_p}{N_m \cdot T_m} \] (5)

\[ (r_m(i) - 1) \leq (1 - r_p(i)) \cdot \frac{N_p \cdot T_p}{N_m \cdot T_m} + \frac{P_{ap} + P_{am}}{P_{am} + P_{dp}} \] (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 - \( T_i \) and \( T_j \) as follows:

\[
(r_m(i) - r_m(j)) \leq (r_p(j) - r_p(i)) \cdot \frac{N_p \cdot T_p}{N_m \cdot T_m} \tag{7}
\]

\[
(r_m(i) - r_m(j)) \leq (r_p(j) - r_p(i)) \cdot \frac{N_p \cdot T_p}{N_m \cdot T_m} \cdot \frac{P_{op} + P_{on}}{P_{am} + P_{op}} \tag{8}
\]

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:

\[
(r_m(i) - r_m(j)) \leq (r_p(j) - r_p(i)) \cdot \frac{N_p \cdot T_p}{N_m \cdot T_m} \cdot \sqrt{\frac{P_{op} + P_{on}}{P_{am} + P_{op}}} \tag{9}
\]

For each inequality (7, 8, 9), if the condition is satisfied, then \( T_i \) is superior to \( T_j \) for the corresponding cost metric. Otherwise, \( T_j \) improves the original program more than \( T_i \). Note that for each transformation \( T_i \) 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 simula-

Figure 3: 3D representation of critical parameters for the original source code, before transformations, to extract parameters useful for both transformation and cost estimation such as \( N_m \), \( N_m \), \( P_{op} \), \( P_p \). Currently, the WATTCH simulator [14] is used in the initial instruction-level simulation step. Also, \( P_{am} \) and \( P_{on} \) can be obtained from the data book or real measurement.

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 SU1F environment and loop blocking is performed by another SU1F package called skewed [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 \( N_m + N_{m} \) 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.
is iterated by \((TC \text{ modulo } u)\) 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-\(\text{ISS}\) 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, \(u_{\text{max}}\). \(u_{\text{max}}\) is the largest unrolling factor not to increase code size larger than I-cache size and is shown next.

\[
\frac{u_{\text{max}}}{I_{\text{org}} \cdot \text{size of instruction}} = \frac{N_{\text{org}}}{N_{\text{im}} + \left(I_{\text{org}} \cdot u_{\text{opt}}\right)}
\]

where, \(I_{\text{org}}\) is the number of instructions of the loop body and is estimated in either front-end (SUPE) or back-end part. Thus, the optimal unrolling factor \(u_{\text{opt}}\) is the greatest divisor of \(TC\) but smaller than \(u_{\text{max}}\). By finding \(u_{\text{opt}}\), we can replace iterative-\(\text{ISS}\) by single \(\text{ISS}\) to obtain \(\text{gain}(u_{\text{opt}})\).

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

\[
\begin{align*}
r_p &= \frac{N_p - \text{gain}(u_{\text{opt}})}{N_p} \\
r_{\text{im}} &= \frac{N_{\text{im}} + (I_{\text{org}} \cdot u_{\text{opt}})}{N_{\text{im}}} \\
r_{\text{dm}} &= 1
\end{align*}
\]

Notice that \(r_{\text{dm}}\) 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 \(r_{\text{dm}}\), 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 \(Z\) (\(BxG\) 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 \(Y\) at every tile 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:

\[
M_{\text{intrins}} = \frac{B^2}{CL} (\text{array } Z) + \frac{2N \cdot B}{CL} (\text{array } X \text{ and } Y)
\]

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 \(B \cdot B / CS\).
for (i = 0; i < N; i++)
  for (k = 0; k < N; k++)
    R0 = V(i,k);
  for (j = 0; j < N; j++)
    X[i][j] = r0*Z[i][j];

(a) Original version

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    X[i][j] = \text{r0}\times Z[i][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/C. Thus, the probability that two or more references will access the same cache line is

\[ P_{cross} = \frac{B}{C} \times B^2 \times \frac{B}{C} + \left( \frac{B}{C} \right)^2 \times \frac{B}{C} \times \left( \frac{B}{C} \right)^2 \]

and total cross-interference occurred per iteration of loop \( j \) is \( M_{cross} = P_{cross} \times B^2 \), where \( B \) is the trip count of loop \( k \) and \( j \). Therefore, total misses to complete the whole loop nest is shown next and \( \text{rmin} \) is simply \( M_{\text{total block}} / \text{Nmin} \).

\[ M_{\text{total block}} = (M_{\text{intrinsic}} + M_{\text{cross}} \times N) \times \left( \frac{N}{B} \right)^2 \]  

On the other hand, \( \text{rmin} \) can be set to 1 because the code size increase is trivial, but \( \text{r}_p \) should be carefully analyzed.

\[ N_p(\text{blocking}) = N_p + \sum_{i=0}^{N} \prod_{j=0}^{k} (T_c \times T_{i,j}) \]

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, \( T_{i,j} \) 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, \( \text{r}_p \) is simply \( N_p(\text{blocking}) / N_p \).

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 L-cache were 4K and their associativity was 2.

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 - \( N_p \), \( N_{min} \), \( P_{unroll} \) and \( P_{idct} \). Then, each program was transformed into two versions (unrolled 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.

Table 2: Four different memory configurations

<table>
<thead>
<tr>
<th>Memory</th>
<th>Latency</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>30</td>
<td>0.1</td>
</tr>
<tr>
<td>M2</td>
<td>50</td>
<td>0.1</td>
</tr>
<tr>
<td>M3</td>
<td>30</td>
<td>0.5</td>
</tr>
<tr>
<td>M4</td>
<td>50</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 3: Decision accuracy of the proposed approach

<table>
<thead>
<tr>
<th>Program</th>
<th>( M_p )</th>
<th>( M_{unroll} )</th>
<th>( M_{block} )</th>
<th>( M_{perf} )</th>
<th>( M_{energy} )</th>
<th>( M_{block} )</th>
<th>( M_{energy} )</th>
<th>( M_{block} )</th>
<th>( M_{energy} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>m100</td>
<td>0.82</td>
<td>0.75</td>
<td>0.80</td>
<td>0.91</td>
<td>0.66</td>
<td>0.68</td>
<td>0.70</td>
<td>0.52</td>
<td>0.70</td>
</tr>
<tr>
<td>m200</td>
<td>0.85</td>
<td>0.74</td>
<td>0.80</td>
<td>0.90</td>
<td>0.68</td>
<td>0.67</td>
<td>0.70</td>
<td>0.51</td>
<td>0.70</td>
</tr>
<tr>
<td>l100</td>
<td>0.85</td>
<td>0.62</td>
<td>0.82</td>
<td>0.93</td>
<td>0.67</td>
<td>0.70</td>
<td>0.70</td>
<td>0.74</td>
<td>0.70</td>
</tr>
<tr>
<td>l200</td>
<td>0.89</td>
<td>0.60</td>
<td>0.87</td>
<td>0.96</td>
<td>0.72</td>
<td>0.74</td>
<td>0.72</td>
<td>0.74</td>
<td>0.72</td>
</tr>
<tr>
<td>sub</td>
<td>0.91</td>
<td>1.01</td>
<td>0.90</td>
<td>1.00</td>
<td>0.73</td>
<td>0.75</td>
<td>0.73</td>
<td>1.01</td>
<td>0.73</td>
</tr>
<tr>
<td>idct</td>
<td>0.85</td>
<td>1.07</td>
<td>0.72</td>
<td>1.06</td>
<td>0.56</td>
<td>1.12</td>
<td>0.56</td>
<td>1.12</td>
<td>0.56</td>
</tr>
<tr>
<td>m100</td>
<td>0.82</td>
<td>0.75</td>
<td>0.80</td>
<td>0.78</td>
<td>0.71</td>
<td>0.59</td>
<td>0.71</td>
<td>0.59</td>
<td>0.71</td>
</tr>
<tr>
<td>m200</td>
<td>0.85</td>
<td>0.74</td>
<td>0.83</td>
<td>0.78</td>
<td>0.71</td>
<td>0.58</td>
<td>0.71</td>
<td>0.58</td>
<td>0.71</td>
</tr>
<tr>
<td>l100</td>
<td>0.82</td>
<td>0.80</td>
<td>0.86</td>
<td>0.96</td>
<td>0.70</td>
<td>0.76</td>
<td>0.70</td>
<td>0.76</td>
<td>0.70</td>
</tr>
<tr>
<td>l200</td>
<td>0.84</td>
<td>0.90</td>
<td>0.85</td>
<td>0.89</td>
<td>0.72</td>
<td>0.84</td>
<td>0.72</td>
<td>0.84</td>
<td>0.72</td>
</tr>
<tr>
<td>sub</td>
<td>0.88</td>
<td>1.02</td>
<td>0.82</td>
<td>1.02</td>
<td>0.72</td>
<td>1.04</td>
<td>0.72</td>
<td>1.04</td>
<td>0.72</td>
</tr>
<tr>
<td>idct</td>
<td>0.82</td>
<td>1.06</td>
<td>0.72</td>
<td>1.06</td>
<td>0.59</td>
<td>1.12</td>
<td>0.59</td>
<td>1.12</td>
<td>0.59</td>
</tr>
<tr>
<td>m100</td>
<td>0.85</td>
<td>0.85</td>
<td>0.86</td>
<td>0.67</td>
<td>0.73</td>
<td>0.42</td>
<td>0.73</td>
<td>0.42</td>
<td>0.73</td>
</tr>
<tr>
<td>m200</td>
<td>0.85</td>
<td>0.80</td>
<td>0.86</td>
<td>0.67</td>
<td>0.73</td>
<td>0.42</td>
<td>0.73</td>
<td>0.42</td>
<td>0.73</td>
</tr>
<tr>
<td>l100</td>
<td>0.85</td>
<td>0.81</td>
<td>0.85</td>
<td>0.84</td>
<td>0.72</td>
<td>0.67</td>
<td>0.72</td>
<td>0.67</td>
<td>0.72</td>
</tr>
<tr>
<td>l200</td>
<td>0.86</td>
<td>0.83</td>
<td>0.87</td>
<td>0.85</td>
<td>0.75</td>
<td>0.71</td>
<td>0.75</td>
<td>0.71</td>
<td>0.75</td>
</tr>
<tr>
<td>sub</td>
<td>0.91</td>
<td>1.01</td>
<td>0.85</td>
<td>1.00</td>
<td>0.77</td>
<td>1.01</td>
<td>0.77</td>
<td>1.01</td>
<td>0.77</td>
</tr>
<tr>
<td>idct</td>
<td>0.85</td>
<td>1.07</td>
<td>0.77</td>
<td>1.01</td>
<td>0.77</td>
<td>1.08</td>
<td>0.77</td>
<td>1.08</td>
<td>0.77</td>
</tr>
</tbody>
</table>
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 presented 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, M100 and M200 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)](image)

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 \( T_n \) and 99% for \( T_m \) (almost invisible in Figure 8 (top)) because the error is too small, while the estimation error of loop blocking is about 11.5% for \( T_{nm} \) and 4% for \( T_m \).

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 unfolding 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 approach 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


