Automatic Source Code Specialization for Energy Reduction *

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 framework to reduce the computational effort of software programs, using value profiling and partial evaluation. Our tool reduces computational effort by specializing a program for highly expected situations and such a reduction translates into both energy and performance improvement. Procedure calls executed frequently with same parameter values are defined as highly expected situations (common cases). The choice of the best transformation of common cases is achieved by solving three search problems. The first identifies effective common cases to be specialized, the second searches for an optimal solution for effective common case, and the third examines the interplay among the specialized cases. Our technique improves both energy consumption and performance of the source code up to more than twice and in average about 25% over the original program. Also, our pruning techniques reduce the searching time by 80% compared to exhaustive approach.

1. INTRODUCTION

With the widespread diffusion of processor-based embedded systems, software becomes one of the key factors to determine overall system quality. For this reason, software design for embedded systems requires aggressive optimizations to increase the code quality at the cost of increased development effort. Whereas in the past code quality was traditionally measured in terms of performance and code size, average energy of software code has become an important (if not the most important) design metric [1, 2, 3, 4].

It has been shown that high-level, architecture-independent, code transformations affect heavily both performance and energy consumption [5]. Some approaches for energy saving adopt sophisticated high-level optimization techniques. Their impact on energy consumption is assessed by instruction-level simulation to consider the underlying hardware architecture [6, 7, 8]. Also, numerous source-level transformation techniques are introduced in [9] to reduce the power consumed by memories in data-dominated applications.

From these approaches, two observations can be drawn. First, transformations effective in reducing energy consumption improve performance as well, even though the improvement ratios are different in general. Second, many high-level transformation techniques largely benefit from profiling [15].

*This work was supported in part by NSF, under grant CCR-9901190 and by ST microelectronics.

ISLPED'01, August 6-7, 2001, Huntington Beach, California, USA

Based on these facts, in [15], Chung *et al.* proposed a source code transformation technique to reduce the computational effort (the average number of executed instructions) which is a common factor for both performance and energy, using value profiling [14] and partial evaluation [12, 13].

Other approaches have also been proposed to reduce computational effort by specializing the common cases such as procedure cloning [11], redundant computation elimination using memoization [10], and hardware specialization [17].

This paper presents algorithms for the automated optimization of programs by performing code specialization, thus extending the technique proposed in [15]. In [15], the common case was specialized based on value profiling information, but the common case was intuitively selected by the user from a large set of candidates and the effectiveness of the specialization strategy was ultimately dependent on the user's ability. The proposed transformation framework overcomes these shortcomings by providing a formal way to search the common cases and their specializations for energy and performance improvement.

The major contributions of our optimization framework are three. First, for a given program, it identifies the promising candidate code fragments for which the available set of code optimizations are likely to produce non-marginal improvements. Second, it automatically explores the search space (promising candidates) with a two-phase procedure. In the first phase, architecture independent optimization is performed for each promising candidates and in the second phase, its impact on the code quality (in terms of energy or performance) is quantitatively assessed by a re-targetable architecture-sensitive measurement, *i.e.* instruction-level simulation. Third, search space pruning strategies are dynamically applied during search space exploration to direct the optimization strategy and reduce search time.

2. BASICS OF THE PROPOSED TECHNIQUE

2.1 Basic Idea and Problem Description

Figure 1 illustrates the basic idea of our approach. Consider the first call of procedure `foo` in `main`. If the first parameter `a` is 0 for most cases, this procedure can be simply reduced to `sp_foo` by partial evaluation as shown in Figure 1 (b). But, parameter `a` is not always 0. Thus, the original call is replaced by a conditional statement which selects appropriate procedure call depending on the result of *common value detection* procedure named `cvd_foo` in Figure 1 (b). We call this transformation *source code alternation.* Also, the variable whose value is often constant(*e.g.* `a`) is called *constant like argument* (CLA).

Next, consider two procedure calls inside the loop of Figure 1 with the assumption that parameter `e` has single common value, 200. Each procedure has a CLA as its second argument, and partial evaluation can be applied for each of them. However, there is not much to be done by partial

```
main () {
  int i, a, b, c[100], d[200], e, result = 0;
  ..............
  result = foo(a, 100, c);
  for (i = 0; i < 10; i++) {
    result += foo(i, 100, c);
    result += foo(b, e, d);
  }
}
int foo(int fa, int fb, int *fc) {
  int i, sum = 0;
  for (i = 0; i < fb; i++)
    for(j = 0; j < fb/2; j++)
      sum += fa * fc[i];
  return sum;
}
```

(a) Original program

```
main () {
  int i, a, b, c[100], d[200], e, result = 0;
  ..............
  if (cvd_foo(a)) result += sp_foo(b);
  else result += foo(a, 100, c);
  for (i = 0; i < 10; i++) {
    result += foo(i, 100, c);
    result += foo(b, e, d);
  }
}
int foo(int fa, int fb, int *fc) {
  int i, sum = 0;
  for (i = 0; i < fb; i++)
    for(j = 0; j < fb/2; j++)
      sum += fa * fc[i];
  return sum;
}
int sp_foo(int *c) {return 0; }
int cvd_foo(int a) {
  if (a == 0) return 1; else return 0; }
```

(b) Specialized program for the first call of `foo` with a=0

Figure 1: Example of source code transformation

evaluator except loop unrolling. The effect of loop unrolling can be either positive or negative depending on the system configuration (*e.g.* cache size). For this reason, it is first required to evaluate the effectiveness of partial evaluation for each call. And then, it is also necessary to check the combined effect of the two procedure calls because both specialized calls will increase code size and they may cause cache conflict due to their alternative calling sequence.

2.2 Search Space and Transformation Flow

In Section 2.1, three search problems are addressed (Figure 1). The first problem searches the common cases to be specialized, the second problem searches the optimal solution for the given common case and call site, and the third problem examines the interplay among the specialized calls.

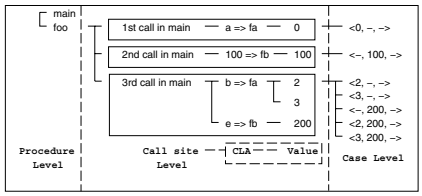

Figure 2: Hierarchical tree of common cases

For the first problem, we represent the common cases as a hierarchical tree. An example for Figure 1 is shown in Figure 2. We assume that variable `b` (the first parameter of the third call) has two common values - 2 and 3. *Call site level* has two-level sub-hierarchies: *CLA level* represents the mapping relation between CLA and its corresponding formal parameter and *value level* is used for common values of CLAs. In *case level*, common values are related to each formal parameter by positional mapping and "-" means *don't care* - the parameter value in that position is not considered.

The overall search space for the first problem is the total number of common cases shown at *case level*.

The second problem is to explore the loop combinations to be unrolled. Thus, the size of search space for each case specialization is simply 2^n, where n is the number of loops of the corresponding procedure.

The third search space size is also exponentially proportional to the number of specialized calls.

The ultimate goal of our approach is to solve these three problems sequentially with appropriate pruning techniques to find the solution with reasonable searching time.

The overall code transformation flow is as follows.

1. **Profiling:** By performing execution frequency and value profiling, the hierarchical tree is constructed and the computation ratio of each procedure is estimated. The details of profiling are described in [15].
2. **Effective common case selection:** The search space for common cases is explored (Section 3.1).
3. **Common case specialization:** The search space for single case specialization is explored (Section 3.2).
4. **Globally effective case selection:** The search space supporting global effect analysis for specialized calls is explored (Section 3.3).

We introduce some notations for the hierarchical tree which will be used in Section 3.1.

We denote the procedure level as P, each procedure as $p_i \in P$, the call site level as C_i (which is a set of procedure calls for p_i) and each procedure call as $c_{ij} \in C_i$. Similarly, CLAs of each c_{ij} are denoted as $a_{ijk} \in A_{ij}$ and common values of a_{ijk} are denoted as v_{ijkl}.

Finally, the set for case level is denoted as $B_{ij} = \{b_{ijm}, m = 0, 1, \cdots, |B_{ij}| - 1\}$. The vector of common values is defined as $b_{ijm} =< cv_0, cv_1, \cdots, cv_k, \cdots, cv_{A_{ij}-1} >$, where cv_k is $v_{ijkl}, l \in \{0, 1, \cdots, |V_{ijk}| - 1\}$ or "-" as in Figure 2.

3. CODE TRANSFORMATION

3.1 Effective Common Case Selection

Effective common cases are selected based on *normalized computational effort* (NCE) which is the computational effort of the given object normalized to the total computational effort. The computational effort of each procedure is obtained using the technique proposed in [15]. Based on this, NCE of each object can be estimated in a hierarchical order. A user constraint called *computational threshold* (CT) is defined in terms of NCE and any object whose NCE is smaller than CT is pruned out.

First, procedure level pruning is performed by removing every p_i whose NCE is smaller than CT. For each remaining p_i's, NCE of c_{ij} is calculated using Equation 1.

$$NCE(c_{ij}) = NCE(p_i) * f_{ij} / \sum_{j=0}^{|C_i|-1} f_{ij} \qquad (1)$$

where, f_{ij} is the execution frequency of c_{ij} (from profiling) and c_{ij} whose NCE is smaller than CT is eliminated. The same pruning is performed at *CLA level* and *value level*.

Finally, NCE of each case is obtained by multiplying NCE of common values which are involved in forming the case and represented as Equation 2 ($NCE(-) = 1$).

$$NCE(b_{ijm}) = \prod_{k=0}^{|B_{ij}|-1} NCE(cv_k) \qquad (2)$$

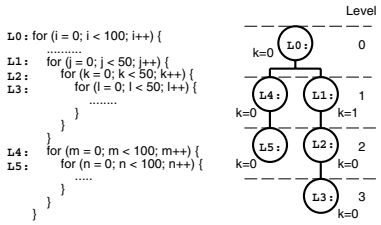

Figure 3: An example of loop graph

To reduce the search space further, we exploit *dominated cases* that can be eliminated from the search space. Vector b_{ijm} is dominated by b_{ijt} if all common values of b_{ijm} appear in b_{ijt} and $NCE(b_{ijm})$ is less than or equal to $NCE(b_{ijt})$.

3.2 Common Case Specialization

3.2.1 Semi-Exhaustive Approach

First, the entire loop structure inside a procedure is levelized. The outermost loop is assigned to *level* 0 and the next outermost loop is assigned to *level* 1 and so on. After levelization, a loop graph is constructed as in Figure 3. Each loop is represented as a node and the nested relation between two loops is represented by an edge connecting two nodes. If a loop has multiple nested loops, the connecting edges are identified as a branch. The basic rationale of this approach is to find the best solution for each branch in the order of computational effort, with the code size constraint to consider the cache size of the target architecture.

An example is shown in Figure 3. The example has 64 (2^6) combinations of loop unrolling. Suppose the subgraph on the right branch (L1) has higher computational effort than the one on the left branch (L4). And the size constraint is initially set to the cache size of the target architecture. Thus, the best combination for the right subgraph is searched first. Because the right subgraph is a three-level loop nest(L1, L2, and L3), there are eight combinations of loop unrolling and the best combination is found by exhaustive search. Next, for the best combination, the unrolled code size of the subgraph is estimated using profiling information (average number of iterations for each node) and number of instructions for each node. And the estimated code size is subtracted from the size constraint because the loop unrolling strategy for this subgraph is already decided and gives a tighter constraint to the next subgraph. Next, the left descendent (L4) and the top node(L0) are examined in the same way, but with different size constraint.

3.2.2 One-Shot Approach

This approach is close to *semi-exhaustive approach*, but differs because the choice of the best combination for each subgraph depends on just code size estimation instead of exhaustive search. The code size estimation is performed in *depth first search* fashion for each subgraph.

As in Figure 3, the subgraph (L1) is visited first due to the same reason in *semi-exhaustive approach*. Unrolled code size is estimated from the lowest level (L3) to the highest level (L1). If the estimated code size is smaller than the size constraint, the current node is unrolled and the node in the next level is visited. Otherwise, the estimation for the given subgraph is terminated. For example, If the estimated code size is larger than the size constraint at L1, L2 and L3 are unrolled. Using the same technique, loop unrolling strategy for the entire graph is decided.

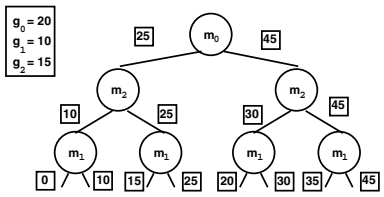

Figure 4: An example of binary tree for M

C programs	Pruning Ratio			
	procedure	call site	case	dominated case
Compress	0.75	0.00	0.00	0.75
Expand	0.60	0.00	0.00	0.83
Edetect	0.67	0.00	0.00	0.88
FFT	0.50	0.00	0.00	0.00
g721 encode	0.88	0.88	0.00	0.67
Convolve	0.50	0.00	0.00	0.98
Average	0.65	0.15	0.00	0.69

Table 1: Pruning by effective common case selection

3.3 Globally Effective Case Selection

Multiple candidates from common case specialization are evaluated for the final solution. Each candidate is denoted as $m_k \in M, k = \{0, 1, \cdots, |M| - 1\}$. candidate m_k has an attribute called *gain* denoted as g_k, which is the improvement amount in terms of the given cost metric (energy or performance) and obtained at the previous step. The purpose of this step is to select m_k's for maximally improving the quality of the specialized code. Thus, the search space in this step is again exponentially proportional to $|M|$ and the *branch and bound* algorithm is used for the search space reduction.

For this purpose, m_k's are sorted in descending order of g_k. Then, set M is represented as a binary tree as in Figure 4 and the node with the largest gain is placed at *level* 0. Each node has two descendents except the leaf nodes, but the leaf nodes still keep their edges. The right edge of each node represents that the node is included in the solution and the left edge has the opposite meaning. Each edge has an attribute called expected gain, which is the expected gain maximally obtained from its descendents (boxes in Figure 4).

The search starts from the rightmost path of the tree (maximum expected gain). When the right edge is selected and the simulation result is better than the expected gain of left edge, the left descendent is pruned because it is obvious that its maximal gain is smaller than the simulation result. Otherwise, the path through the left edge is simulated and either right or left edge is selected depending on the simulation result. This procedure is recursively applied until the traversal reaches the root node.

4. EXPERIMENTAL RESULTS

We have chosen Smart Badge, an ARM based portable device, as the target architecture with the cycle accurate energy simulator proposed in [16]. And we applied the proposed technique to six DSP C programs - `Compress`, `Expand`, `Edetect`, and `Convolve` [18], `g721 encode` [19], and `FFT` [20].

The experiment was conducted for two assessments - search

C programs	common case		global
	semi-exhaustive	one-shot	
Compress	0.78	0.98	0.00
Expand	0.75	0.98	0.33
Edetect	0.87	0.98	0.86
FFT	0.13	0.88	0.00
g721 encode	0.25	0.50	0.00
Convolve	0.75	0.94	0.86
Average	0.59	0.87	0.35

Table 2: Search space reduction ratio in common case and globally effective case selection step

C programs	Code Quality								
	exhaustive			semi-exhaustive			one-shot		
	energy	performance	code size	energy	performance	code size	energy	performance	code size
Compress	0.91	0.91	1.01	0.91	0.91	1.01	0.93	0.93	1.15
Expand	0.84	0.83	1.15	0.84	0.83	1.15	0.90	0.90	1.12
Edetect	0.44	0.37	1.20	0.44	0.37	1.20	0.44	0.37	1.20
FFT	0.86	0.86	1.16	0.86	0.86	1.16	0.86	0.86	1.16
g721 encode	0.88	0.88	1.04	0.88	0.88	1.04	0.88	0.88	1.04
Convolve	0.54	0.48	1.18	0.54	0.48	1.18	0.54	0.48	1.18
Average	0.74	0.72	1.12	0.74	0.72	1.12	0.76	0.74	1.14

Table 3: Quality of the code transformed with different approaches (normalized to original code)

space reduction and the code quality. Each program was profiled to collect computational effort and common values. There exist two important parameters in value profiling [15]. First, OR(*Observed Ratio*) is the ratio of the observation frequency of a specific value over the total call site visiting frequency for a given parameter. Second, OT(*Observed Threshold*) is a threshold value to cut off values whose OR is lower than OT. In this experiment, OT was set to 0.5.

Table 1 shows the pruning ratio achieved by each step with $CT = 0.1$. The procedure pruning was always important to reduce the search space, but call site pruning showed large variation depending on the property of the programs. This is because the computational kernels of some programs such as `compress` and `FFT` were called only once while the kernel of `g721 encode` was called several times in different sites with different calling frequencies. Thus, this step is useful for the kernels called at different sites with different frequencies.

In some cases, the case pruning step was not effective due to high OT which was set to 0.5 in value profiling. Under this OT, the OR of each common value is usually large enough not to be pruned out due to small CT. It is interesting that dominated case pruning was effective for most of application programs because at least one of the CLAs per each program had a common value with $OR = 1.0$.

Next, the pruning in common case specialization and globally effective case selection were evaluated. In Table 2, both semi-exhaustive and one-shot approach drastically reduced the search space by 59% and 87%, respectively. Also, pruning technique in globally effective case selection step showed 35% of search space reduction with the large variation depending on the programs. There was nothing to be pruned for `Compress`, `FFT` and `g721 encode` programs because only one case was passed from common case specialization step. But, it was effective when multiple cases were passed.

We also compared the total transformation time of two approaches to the exhaustive approach. Notice that effective common case selection and globally effective case selection were commonly applied with all three approaches to avoid huge search space. As expected, one-shot approach showed the shortest running time (12% of exhaustive approach) and semi-exhaustive approach was ranked at second (36% of exhaustive approach). Exhaustive approach often generated a huge size of code which was one of the main problems in partial evaluation. In this case, compilation or simulation was not terminated in a few hours. To avoid this problem, we adopted time-out policy for the exhaustive approach.

Table 3 shows the quality of the transformed programs. Semi-exhaustive approach is comparable to exhaustive approach with less computation time (2.8 times faster). One-shot solution is also useful by trading off its code quality and computation time. (8.3 times faster and 3% more energy). It is also shown that the deviation of improvement is depending on the nature of the programs. For the best case, the improvement is more than twice(`Edetect`), but for the worst case, about 10% is improved (`Compress`).

5. CONCLUSION

We presented algorithms and an automated tool flow to reduce the computational effort of software programs, by using value profiling and partial evaluation.

Within our approach, a first tool performs program instrumentation and profiling to collect information for transformations, such as execution frequency and common values at each call site. Using the information, another tool selects effective common cases based on the estimated computational effort. Each selected case is specialized independently using a partial evaluator and code explosion due to loop unrolling - which may hamper partial evaluation - is avoided by code size estimation technique and pruning. Finally, the interplay among the multiple specialized cases is analyzed based on instruction-set level simulation. The overall transformation time is 8 times faster than the exhaustive approach and the transformed code shows in average 24% energy saving and 26% in average performance improvement with 14% code size increase (one-shot approach).

6. REFERENCES

[1] J.R. Lorch, A.J. Smith, "Software Strategies for Portable Computer Energy Management", *IEEE Personal Communications*, vol. 5, issue 3, pp.60-73, Jun. 1998

[2] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye, "Energy-driven Integrated Hardware-Software Optimizations using SimplePower", *ISCA*, pp.95-106

[3] V. Tiwari, S. Malik, and A. Wolfe, "Compilation Techniques for Low Energy: An Overview", IEEE Symposium on Low Power Electronics, pp. 38-39, 1994

[4] J. Rabaey and M. Pedram, *Low-Power Design Methodologies*. Kluwer, 1996.

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

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

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

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

[9] F. Catthoor, *et. al*, "System-Level Transformation for Low Power Data Transfer and Storage", A. Chandrakasan, R. Brodersen eds. Low-Power CMOS Design, IEEE Press, 1998

[10] S. Richardson, "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation", *Tech. report, Sun Microsystems Lab.*, 1992

[11] K. Cooper, M. Hall, and K. Kennedy, "A Methodology for Procedure Cloning", Computer Languages, Vol. 19, No. 2, pp 105-117, April, 1993

[12] C. Consel and O. Denvy, "Tutorial Notes on Partial Evaluation", ACM Symposium on Principles of Programming Languages, pp.493-501, 1993

[13] L. O. Andersen, Program Analysis and Specialization for the C Programming Language, PhD thesis. DIKU, Univ. of Copenhagen, 1994.

[14] B. Calder, P. Feller, and A. Eustace, "Value Profiling and Optimization", *Journal of Instruction-Level Parallelism*, vol. 1, Mar. 1999

[15] 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

[16] T. Simunic, L. Benini, and G. De Micheli, "Cycle Accurate Simulation of Energy Consumption in Embedded Systems", *DAC*, pp.867-872, 1999

[17] G. Lakshminarayana, A. Raghunathan, K. Khouri, K. Jha, and S. Dey, "Common-Case Computation: A High-Level Technique for Power and Performance Optimization", *DAC*, pp.56-61, 1999

[18] http://www.eecg.toronto.edu/ stoodla/benchmarks/benchmarks.html

[19] http://www.cs.ucla.edu/ leec/mediabench

[20] P. Duhamel and H. Hollman, "Split-Radix FFT Algorithm", *Electronics Letters*, vol. 20, no. 1, pp.14-16, Jan. 5, 1984