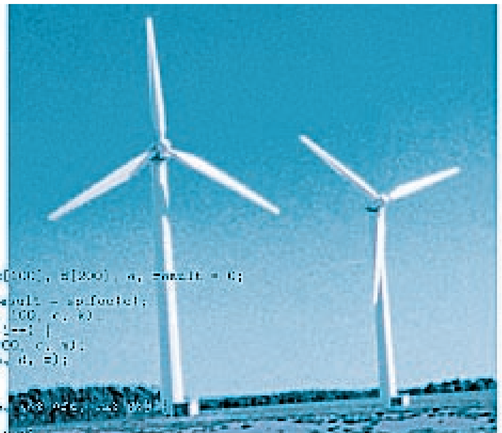
ST Journal of System Research
vol 3, number 1 April 2002

Low Power System Design

Value-based Source Code Specialization for Energy Reduction

Eui-Young Chung[(1)] - Giovanni De Micheli[(1)]
Marco Carilli[(3)]
Luca Benini[(2)] - lbenini@deis.unibo.it
Gabriele Luculli[(3)] - gabriele.lucilli@st.com

(1): CSL, Stanford University
(2): DEIS, Università di Bologna
(3): AST, STMicroelectronics



The objective of this research is to create a framework for energy optimization of software programs. In particular, this paper presents algorithms and a tool flow to reduce the computational effort of programs, using value profiling and partial evaluation. Such a reduction translates into both energy savings and performance improvement. Namely, our tool reduces computational effort by specializing a program for highly expected situations. Procedure calls which are frequently executed 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 is to identify common cases to be specialized, the second is to search for an optimal solution for effective common case, and the third is to examine the interplay among the specialized cases. Experimental results show that our technique improves both energy consumption and performance of the source code up to more than twice and in average about 35% over the original program in Lx processor environment. Also, our automatic search pruning techniques reduce the searching time by 80% compared to exhaustive approach.

1 INTRODUCTION

With the widespread diffusion of processor-based embedded systems, software design 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 code size, average energy and performance of software code has become an important (if not the most important) design metric [1, 2, 3, 4].

Most previous research on optimizing software compilation relates to the assembly and/or binary code generation steps. Indeed, this is the most appropriate level to perform energy/performance analysis while considering the underlying hardware. Numerous techniques, targeting performance and code size, have been proposed, such as instruction scheduling, code selection, register allocation and address assignment [25, 21, 22]. The optimizations proposed in [5, 7, 8] are instruction scheduling techniques especially for power reduction by minimizing switching activities.

On the other hand, it has been shown that high-level, architecture-independent, code transformations affect heavily both performance and energy consumption [6]. Classical transformation techniques for performance improvement include loop un-rolling, loop interchange, procedure in-lining and so on [23, 30].

Copyright © Stanford University, Università di Bologna, STMicroelectronics 2001

Some approaches for energy reduction also adopt sophisticated high-level optimization techniques.Their impact on energy consumption is assessed by instruction-level simulation to consider the underlying hardware architecture [9, 10, 11]. Also, numerous source-level transformation techniques are introduced in [12], with the objective of reducing the power consumed by memories in data-dominated applications.

From the approaches mentioned above, two observations can be drawn. First, the proposed high-level program transformations tend to impact both energy and performance in a similar fashion, and it appears that there is a tight relationship between these two cost metrics. In other words, 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 program profiling [18]. The basic idea of profiling-based approaches is to identify promising code fragments and optimize them. Additionally, profiling can provide useful information on the particular flavor of high-level transformation which is likely to have the largest impact on code quality. Profiling-based optimization is advantageous especially for embedded systems on which only a few programs (often only one program) are running, thus the time required for profiling is relatively small compared to general purpose systems, and profiling precision is higher.

Based on these two facts, we propose an automated source code transformation framework aiming at reducing the computational effort (i.e., the average number of executed instructions) which is a common factor for both performance and energy consumption, using value profiling [17] and partial evaluation [14]. Namely, our tool reduces computational effort by specializing a program for highly expected situations. Procedure calls which are frequently executed with same parameter values are defined as highly expected situations (common cases).

Other approaches have also been proposed to reduce computational effort by specializing the common cases. Procedure cloning [13] was proposed to specialize procedure calls which have constant parameters. In [13], instead of common case, only constant case was considered and the optimization strategy for each case was not described. On the other hand, in [20], common-case specialization was proposed for hardware synthesis. Also, in [29], redundant computation (an operation performs the same computation for the same operand) was defined and *result cache* was proposed to avoid redundant computations by reusing the result from the *result cache*.

This paper presents algorithms for the automated optimization of software programs to reduce the computational effort, which is a common factor for both energy and performance, by performing code specialization for the highly expected situations. The proposed framework provides not only a formal way to identify the highly expected situations (common cases) from a large set of candidates, but also a heuristic strategy to reduce the search time for the optimal code specialization.The overall objective of this research is to create a framework for the automated optimization of software. Thus, the techniques presented here fit within a specific optimization flow (See Figure 2). This research is complementary, and not alternative, to other techniques for software optimization, such as loop optimization, which can be easily incorporated in our framework.

The major research contributions of our optimization framework are four.

- For a given program, our tool automatically instruments the program for both execution frequency and value profiling which provides the basic information necessary to identify the common cases and provides to estimate the computational effort for each code fragment.
- For a given program, our tool automatically identifies the promising candidate code fragments for which the available set of code optimizations are likely to produce non-marginal improvements.

- Our framework 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 retargetable architecture-sensitive measurement, *i.e.* instruction-level simulation.
- Search space pruning strategies are dynamically applied during search space exploration to direct the optimization strategy and reduce search time.

In Section 2, we will demonstrate the basic idea and overall flow of the proposed technique for program specialization based on partial evaluation and value profiling. Also, search spaces to be explored are defined. In Section 3, we will present the profiling method and the computational-effort estimation technique. In Section 4, common-case selection technique for specialization based on computational-effort estimation will be described. In Section 5, specialization for each common case will be presented and the globally optimal case selection from multiple specialized cases will be discussed in Section 5. Finally, we will show the experimental result in Section 7 and conclude our work in Section 8.

2 BASIC IDEA AND OVERALL FLOW

2.1 Basic idea and problem description

The technique described in the following sections aims at reducing the computational effort of a given program by specializing it for situations that are commonly encountered during its execution. The ultimate goal of this technique is to improve energy consumption as well as performance by reducing computational effort. The specialized program requires substantially reduced computational effort in the common case, but it still behaves correctly. The "common situations" that trigger program specialization are detected by tracking the values passed to the parameters of procedures. The example in Figure 1 illustrates the basic idea.

Consider the first call of procedure `foo` in procedure `main`. Suppose the first parameter `a` is 0 for 90% of its calling frequency.

Also, suppose the same condition holds for the last parameter `k`.

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

(a) Original program

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

(b) Specialized program for the first call of foo (a=0 and k = 0)

```
main () {
  int i, a, b, k, m, c[100], d[200], e, result = 0;
  ..............
  if (cvd_foo(a)) result = sp_foo(c, k);
  else result = foo(a, 100, c, k);
  for (i = 0; i < 10; i++) {
    result += foo(i, 100, c, m);
    result += foo(b, e, d, m);
  }
}
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] + fk;
  return sum;
}
int sp_foo(int *fc, int fk) { return 50*100*fk; }
int cvd_foo(int a) {
  if (a == 0) return 1;
  return 0;
}
```

(c) specialized program for the first call of foo (a=0)

Figure 1: Example of source code transformation using the proposed technique

Using these common values, a partial evaluator can generate the specialized procedure sp_foo as shown in Figure 1 (b) which reduces the computational effort drastically.
In reality, the values of parameters a and b are not always 0. Therefore, the procedure call foo cannot be completely substituted by the new procedure sp_foo. Instead, we replace it by a conditional statement which selects appropriate procedure call depending on the result of *common value detection* (CVD) procedure named cvd_foo in Figure 1 (b). We call this transformation step *source code alternation*. Also, the variable whose value is often constant (e.g. a) is called *constant like argument* (CLA).

If we ignore the common value of k, the original code will be specialized as shown in Figure 1 (c). The sp_foo in Figure 1 (c) has one more multiplication than the sp_foo in Figure 1 (b), but the situation that $a = 0$ will happen more frequently than the situation that both a and k are 0. For this reason, it is not clear which specialized code is more effective to reduce the overall computational effort. This is the first search problem in our approach.

Next, consider two procedure calls inside the loop of Figure 1 with the assumption that parameter e (the second parameter of the third procedure call) has single common value, 200. Each of two procedure calls has a CLA as their second arguments, respectively.

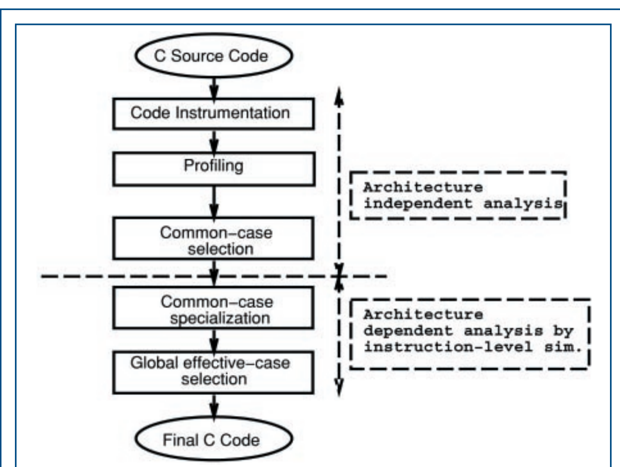


Figure 2: Overall source code transformation flow

Partial evaluation can be applied for each procedure call to reduce computational effort. However, there is not much to be done by partial evaluator except loop unrolling because all other parameters are not CLAs. The effect of loop unrolling can be either positive or negative depending on the system configuration. For this reason, it is required to find the best combination of loop unrolling for each call. In this example, there are four possible combinations for each call, but the number of combinations is exponentially proportional to the number of loops. This is the second search problem of our approach.

After each call is specialized with the best combination of loop unrolling, it is also necessary to check the interplay among the specialized calls, because both specialized calls will increase code size and they may cause cache conflict due to their alternative calling sequence. Thus, we need a method to analyze the global effect of the specialized calls caused by their interplay, which is the third problem of our approach. This example clearly shows three search problems to be addressed in this paper.

To summarize, we have three search problems to specialize a program for the highly expected situations.

1. **Common-case selection** is to find the most effective common-case among several common cases for each procedure call.
2. **Common-case specialization** is to specialize a procedure call for the given common case by controlling loop unrolling.
3. **Global effective-case selection** is to find the most effective combination of specialized calls.

We will use the term "call site" and "procedure call" interchangeably unless there is an explicit explanation.

2.2 Overall code transformation flow

The automated code transformation flow is shown in Figure 2. As shown in Figure 2, a program instrumentation and profiling is performed at the first step to collect the information of the three search problems. Also, computation

efforts of procedures and procedure calls are estimated in this step. Based on the profiling information and estimated computation efforts, each search problem is solved sequentially. As shown in Figure 2, the first three steps are architecture-independent, where as the last two steps use instruction level simulation to consider the underlying hardware architecture. Once the first three steps are performed, then any specific architecture can be considered in the last two steps.

We will briefly describe each step in this section and the details will be described in the later sections.

- **Instrumentation and profiling.** Two types of profiling are performed - *execution frequency profiling and value profiling*. Using the information from *execution frequency profiling*, the computational efforts of procedures and procedure calls are estimated. On the other hand, *value profiling* identifies CLAs and their common values by observing the parameter value changes of procedure calls.

- **Common-case selection**. Based on profiling information, all detected common cases are represented as a hierarchical tree (Section 4). To reduce the search space, *normalized computational effort* (NCE) is computed for each object in the hierarchical tree. NCE represents the relative importance of each object in terms of computational effort. By defining a user-defined constraint called *computational threshold* (CT), trivial common cases are pruned.

- **Common-case specialization**. Each case not pruned in the previous step is specialized. In our framework, specialization is performed by CMIX [15] which is a compile-time (off-line) partial evaluator. In addition to the specialized procedure, the *common value detection* (CVD) procedure is generated. Also, source code alternation is performed so that the original procedure call is replaced by a conditional statement as shown in Figure 1. For the specialized code of each common case, instruction-level simulation is performed to assess the quality of the specialization and the cases which show improvement by specialization are selected for the next step. The search space of this problem is exponentially proportional to the number of loops and the details of heuristic approaches for the search space reduction will be described in Section 5.

- **Global effective-case selection.** This step analyzes the interplay of the specialized calls chosen at the previous step and decides the specialized calls to be included for the final solution. The search space for this analysis is also exponentially proportional to the number of the specialized calls, thus a search space reduction technique based on the branch and bound algorithm is proposed.

3 PROFILING

3.1 The structure of profiler

Many profiling techniques are based on assembler or binary executable to extract more accurate architecture-dependent information such as memory address tracing and execution time estimation. Since they are designed for specific machine architectures, they have limited flexibility [16].

In our case, it is sufficient to have only relatively accurate information rather than accurate architecture-dependent profiles, while keeping source-level information. In other words, it is more important to identify which piece of code requires the largest computational effort rather than to know the exact amount of computational efforts required for its execution.

We used the SUIF compiler infrastructure [31] for source code instrumentation. The instrumentation is performed based on the abstract syntax trees (High-SUIF) which well represent the control flow of the given program in high level abstraction. In detail, a program is represented as a graph $G = \{V, E\}$, where node set V is matched to the high level code constructs such as `for-loop`, `if-then-else`, `do-white` and denoted as $v_i \in V, i = \{0, 1 \cdots N_v - 1\}$, where, N_v is the total number of nodes in a program G. Any edge $e_{ij} \in E$ connects two different nodes v_i and v_j and represents their

dependency in terms of either their execution order or nested relation. Note that v_i is hierarchical, thus each v_i can have its subgraph to represent the nested constructs.

For each v_i which is a procedure, we insert as many counters as its descendent nodes to record the visiting frequencies. And for each descendent node, SUIF instructions for incrementing the corresponding counter are inserted for *execution frequency profiling*. *Value profiling* requires additional manipulations such as type checking between formal parameters and actual parameters of procedure calls, recording the observed values and so on.

The proposed profiler has ATOM-like structure [32] in the sense that user supplied library is used for instrumentation, namely the source code is instrumented with simple counters and procedure calls. The user supplied library includes the procedures required for both *execution frequency and value profiling*. At the final stage, the instrumented source code and the user supplied library are linked to generate the binary executable for profiling.

3.2 Computational-effort estimation

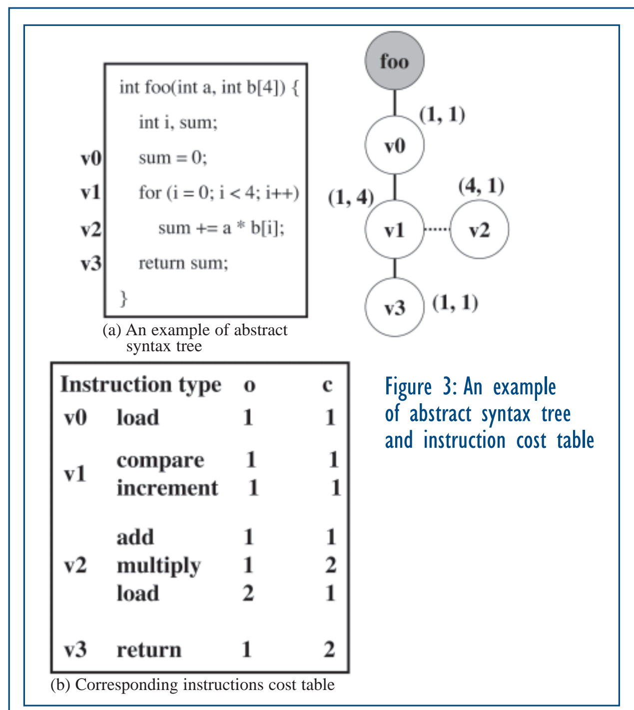
Computational kernel can be identified by execution frequency profiling and computational-effort estimation. Execution frequency profiling is a widely used technique to obtain the visiting frequency of each node (v_i in G). This information only represents how frequently each node is visited, but does not show the importance of each node in terms of computational effort.

For this reason, we used a simple estimation technique of computational efforts for each basic unit using the number of instructions of each basic unit, where the instruction set used is the builtin instructions defined in SUIF framework. Due to the lack of specification of a target architecture, it is assumed that all the instructions require same computational effort. But we provide a way to distinguish the cost of each instruction when the target architecture is determined using an instruction cost table. Each SUIF instruction is defined with its cost in the instruction cost table, thus the execution time of each node v_i of graph G can be calculated as follows.

$$ce_i = f_i * i_i \sum_{j=0}^{N-1} (o_{ij} * c_j) \quad (1)$$

where, ce_i is the estimated computational effort of node v_i, f_i is the execution frequency of node v_i from execution frequency profiling, i_i is the average number of iterations for each visit of node v_i, o_{ij} is the number of instruction j observed in node v_i, c_j is the cost of instruction j, and N is the total number of instructions defined in SUIF. Note that the basic unit of our approach includes loop and do-while constructs. For this reason i_i is considered in Equation 1. It is also worthwhile to mention that the Equation 1 represents the single level computational-effort estimation. As mentioned in Section 3.1, the node v_i is hierarchical. Thus, the cumulative computational efforts for each node v_i can be estimated the sum of current level computational effort and the effort of its descendent nodes.

An example of abstract syntax tree is shown in Figure 3 (a), where a solid edge represents the dependency of two nodes and a dotted edge represents their nested relation and its corresponding instruction cost table is shown in Figure 3 (b). A pair of numbers assigned to each node is (f_i, i_i) which is obtained from the *execution frequency profiling*.



(a) An example of abstract syntax tree

	Instruction type	o	c
v0	load	1	1
v1	compare	1	1
	increment	1	1
v2	add	1	1
	multiply	1	2
	load	2	1
v3	return	1	2

(b) Corresponding instructions cost table

Figure 3: An example of abstract syntax tree and instruction cost table

Example: Consider node v_2 in Figure 3 (a). f_2 and i_2 are 4 and 1 as shown in the graph. Also, from Figure 3 (b), v_2 has 1 addition, 1 multiplication, and 2 load instructions. Among them, only multiplication has cost twice higher than the other two instruction. By substituting these values into Equation 1,
ce_2 = 4 x 1 x (1 x 1 + 1 x 2 + 2 x 1) = 20. Similarly, ce_0 = 1, ce_1 = 8, and ce_4 = 2. Therefore the computational effort of procedure `foo` is 31 by summing these values.

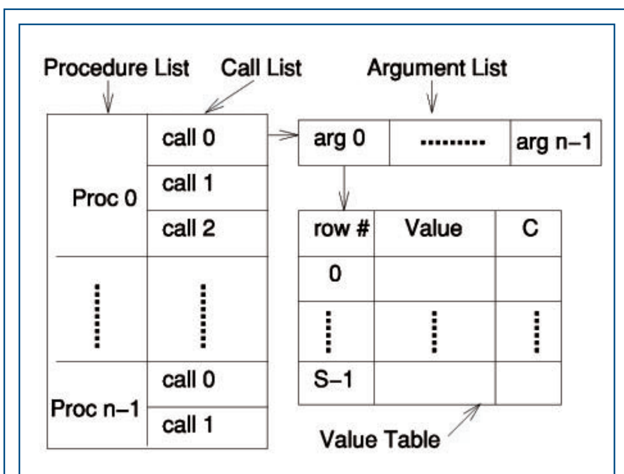


Figure 4: Internal data structure of value profiling

3.3 Value profiling

As mentioned in Section 2.1, value profiling is performed at the procedure level. In other words, each procedure call is profiled, because single procedure can be called in many different places with different argument values. The reason why we chose value profiling instead of value tracing is to avoid huge disk space and disk accesses required for value tracing.

One of the difficulties in value profiling occurs when the argument size is dynamic. For example, any size of one-dimensional integer array can be passed to an integer type pointer argument whenever the corresponding procedure is called. Another difficulty occurs when the argument has complex data type because complex data type requires hierarchical traversal for value profiling. For this reason, currently value profiling in our work is restricted to elementary type scalar and array variables. Note that this restriction is not applied to the arguments defined at each procedure, but to the variables passed as arguments for each procedure call. When a procedure call has both types of variables as arguments, only the variables which violate this restriction are excluded from profiling. Pointers to procedures are not considered in our approach due to its dynamic nature.

Figure 4 shows the internal data structure of value profiling system. As shown in Figure 4, each procedure has a list of procedure calls which are activated inside the procedure. Each procedure call in the list has a list of arguments and each argument in this list satisfies the type constraint mentioned above and has its own fixed size value table to record the values observed and their frequencies. Each row in the value table consists of three fields - index field, value field and count (*C*) field.

The index field represents not only the index of the row, but also the chronological order of the row in terms of the updated time relative to other rows. Thus, the larger the index is, the more recently the corresponding row is updated. In our representation, each row is denoted as $r_i, i \in \{0, 1, \cdots S, -1\}$, where *S* denotes the size of value table, *i.e.* the number of the rows in the table. The value field is used to store the observed value, and the c_i field in r_i counts the number of observations of the corresponding value. The table is continuously updated whenever the corresponding procedure call is executed. At the end of profiling, each argument of the value table is examined to find the values which are frequently observed and only the argument-value pairs which satisfy user defined constraint called *OT* (Observed Threshold) are reported to the user. For this purpose, OR_i (Observed Ratio) is calculated for each r_i in the value table as follows.

$$OR_i = c_i / f \tag{2}$$

where, *f* is the visiting frequency of this call site. The larger OR_i is, the more frequently the value is observed. When OR_i is smaller than *OT*, the value in r_i is disregarded.

The key feature of value profiling is the value table replacement policy [18]. As mentioned above, the size of each value table is fixed to save memory space and table update time. c_i of each value table is initialized to 0. Thus if a new value is observed and at least one of c_i is 0, the new value is recorded in r_i which has the smallest index among these rows. On the other hand, when the table is full (there is no c_i which is 0), the following formula is used to select the row which is to be replaced.

$$rfi = W * i + (1 - W) * c_i \qquad (3)$$

where, rf_i, $i \in \{0, 1 \cdots S-1\}$ is replacement factor which is the metric to decide which row is to be replaced. The smaller rf_i is, the more likely r_i will be selected for replacement. The weighting factor W is used to specify the importance of the chronological order relative to observed count c_i. The selected r_i which has the smallest rf_i is deleted from the table and $r_i \rightarrow r_j - 1, j \in \{i+1 \cdots S-1\}$ if $_j < S-_1$.
Finally, the new value is stored to a new row $r_s - 1$.

4 COMMON-CASE SELECTION

As shown in the example of Section 2.1, every procedure call which has CLAs can be specialized. Some procedure calls can be effectively specialized, while others may not show significant improvement. Also, some CLAs are not useful for specialization. Thus, it is necessary to search all procedure calls which can be effectively specialized by using their common values.

Due to the large search space, we represent all possible common cases as a hierarchical tree based on profiling information and prune out the cases which are expected to show only marginal improvement even after specialization.

4.1 Common case representation

Figure 5 shows the hierarchical tree for the example shown in Figure 1 based on the profiling information. For the sake of the simplicity, we ignore the parameter `fk` which is the fourth parameter of procedure `foo`.

We assume that variable `b` (the first parameter of the third call) has two common values - 2 and 3.

In Figure 5, *call-site level* has two-level sub-hierarchies to represent the CLAs and their common values. *CLA level* represents the mapping relation between CLA parameter 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 "-" represents *don't care* - the parameter value in that position is not considered in this case.

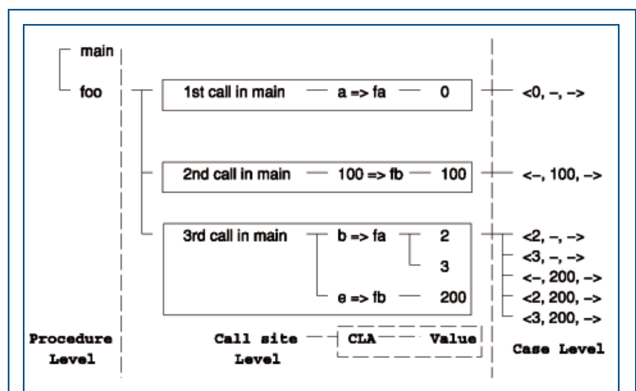


Figure 5: Hierarchical tree representation of common cases

level	set	element
procedure	P	p_i
call site	C_i	c_{ij}
CLA	A_{ij}	a_{ijk}
value	V_{ijk}	v_{ijkl}
case	B	$b_{ijm} = < cv_0, cv_1, \cdots, cv_k, \cdots, cv_{A_{ij}}-1 >$

Table 1: Notations for a hierarchical tree

There are seven possible cases, even though the number of call sites are only three. There is nothing to be examined for procedure `main` because it does not have any CLA.
We introduce some notations for convenience to indicate each level and object in a hierarchical tree as shown in Table 1.

As shown in Table 1, *procedure level* is denoted as P which is a set of procedures denoted as p_i . Each procedure p_i has a set C_i which is a set of procedure calls denoted as c_{ij}. And the same rule is applied to CLA *level* and *value level*. Each common case of c_{ij} is denoted as $b_{i\,jm}$ which is a vector and

each element of $b_{i\,jm}$ (cv_k) is one of the common values of c_{ij} or *don't care*, namely $cv_k = v_{i\,jkl}, l \in \{0, 1, \cdots, |V_{i\,jk}|-1\}$ or "-".

The overall size of the search space to find common cases is calculated by summing the size of search space for each call site. And the number of cases to be examined at each call site is one less than the product of the number of common values for all CLAs (the case that all CLAs are ignored is not considered). Based on the notation in Table 1, the size of the overall search space is denoted as S is shown in Equation 4.

$$|B_{ij}| = \prod_{k=0}^{|A_{ij}|-1} (|V_{ijk}|+1) - 1$$

$$S = \sum_{k=0}^{|P|-1} \sum_{j=0}^{|Ci|-1} |B_{ij}| \qquad (4)$$

4.2 Pruning trivial cases

Due to the large size of the common case set, it is necessary to reduce the search space without missing promising candidates. We define *common cases* which are the cases to be included in the search space after search space reduction.

p i			c ij				a i jk			
i	proc	NCE	j	site	freq.	NCE	k	var	value	freq.
0	main	5%	0	-	1	5%	-	-	-	-
1	foo	95%	0	1st	100	8%	0	a	0	100
			1	2nd	10000	29%	1	100	100	10000
			2	3rd	10000	54%	0	b	2	1000
									3	8000
							1	e	200	10000

Table 2: Profiling information for the hierarchical tree shown in Figure 2

The search space reduction is performed based on *normalized computational effort* (NCE). The computational effort of each procedure is obtained from execution frequency profiling and computational-effort estimation technique described in Section 3. Based on this, NCE of each common case can be estimated in a hierarchical order. In other words, NCE of each procedure is estimated first and then NCE of each call site is calculated and so forth.

NCE in a hierarchical tree represents the maximum degree of improvement to be obtained by specializing all cases belonging to the given node. For pruning purpose, a user constraint called *computational threshold* (CT) is defined in terms of NCE. We will assume *CT* = 0.1 for all examples illustrated in this section.
Usually, maximizing the usage of common values is considered to be better because more information is provided to the optimizer. But in our case, maximizing the usage of common values is not always advantageous (cases for the third call).

Example: Consider two common cases <2,200,–> and <–, 200,–> for the third call of procedure `foo`. The profiling information is shown in Table 2 which is a sample profiling information used for all examples in this section. From Table 2, *b* = 2 with the probability of 0.1 and *e* = 200 with the probability of` 1.0. Then, the probability that case <2,200,–> will happen is 0.1, while that of case <–, 200,–> is 1.0. Thus, the specialized code for case <2,200,–> is useful only when it reduces the computational effort 10 times more than the specialized code for case <–, 200,–>.

The cases like case <2,200,–> is pruned out before progressing to the next step - *common-case specialization* for the sake of the computation efficiency. Pruning is not limited only to *case level*, but also performed at any other level based on NCE. We will describe NCE computation and pruning at each level in the next subsections.

4.2.1 Procedure level pruning

NCE of each procedure is obtained by normalizing its computational effort to the total computational effort. Because NCE of procedure `main` is lower than CT, it is eliminated from the hierarchical tree. Also, the procedure which doesn't have any descendant is eliminated. The pruning at this level has the largest impact on reducing the search space.

4.2.2 Call site level pruning

A similar pruning is performed in this level. The profiler described in Section 3 can estimate the computational

effort of each procedure as well as each procedure call. Thus, NCE of each procedure can be computed in the same way as NCE of each procedure computed. In Table 2, the first call of procedure `foo` will be out because its NCE is less than CT.

We also consider NCE for two sub-hierarchies in call-site level. NCE of each CLA is calculated by weighting the NCE of the corresponding procedure call (c_{ij}) by its *observed ratio* (OR_i) and can be represented as Equation 5. Also, NCE of each common value ($v_{i\,jkl}$) also can be computed similarly.

$$NCE(a_{ijk}) = NCE(c_{ij}) * \sum_{k=0}^{|Aij-1|} OR_k \qquad (5)$$

Example: Let us consider the third call of procedure `foo`, where a_{120} is variable as shown in Table 2. a_{120} has two common values - 2 (V_{1200}) and 3(V_{1201}). Also, from Equation 2, $OR(v_{1200})$ = 100/10000 = 0.1 and $OR(v_{1201})$ = 8000/10000 = 0.8. Thus, NCE (a_{120}) = 0.54 x (0.1+0.8) = 0.486 which is larger than CT, thus, a_{120} is not pruned at *CLA level*. At *value level*, *NCE* (v_{1200}) = *NCE* (a_{120}) x $OR(v_{1200})$ = 0.486 x 0.1 = 0.0486 which is smaller than CT and v_{1200} is pruned out, whereas v_{1201} is not eliminated because its NCE is larger CT.

4.2.3 Case Level Pruning

NCE of each case can be calculated using NCE of common values.

But NCE at this level cannot be obtained in the same way used in other levels because each case may depend on multiple common values such as case <2,200,–>. Thus, NCE of each case is obtained by multiplying NCE of common values which are involved in forming the case and represented as Equation 6.

$$NCE(b_{ijm}) = \prod_{k=0}^{|Bij|-1} NCE(cv_k) \qquad (6)$$

Remember that cv_k is $v_{i\,jkl}$, $l \in \{0, 1, \cdots, |v_{i\,jk}|-1\}$ or "-" and NCE(-) is defined as 1.

Example: Let us consider case b_{120} = <2,200,–> which is a child of c_{12} (third call in `main`) and c_{12} is also a child of p_1 (procedure `foo`). From the example in Section 4.2.2, $NCE(v_{1200})$ = 0.0486. Similarly, $NCE(v_{1210})$ = 0.54 x 10000/10000 = 0.54. From Equation 6, $NCE(b_{120})$ = 0.0486 x 0.54 = 0.027, thus b_{120} is dropped from the search space. But this pruning does not happen in practice because v_{1200} is already pruned out at *value level*. Also, notice that case <–, 200,–> which has less information than case <2, 200,–> (from the view-point of a specializer in the next step) is still in the tree due to its high NCE (0.54).

To reduce the search space further, we define *dominated cases* that can be eliminated from the search space. We say that $b_{i\,jm}$ is dominated by $b_{i\,jt}$ if all common values of $b_{i\,jm}$ appear in $b_{i\,jt}$ and *NCE* ($b_{i\,jt}$) is greater than or equal to *NCE* ($b_{i\,jm}$).

$$NCE\ (b_{i\,jm}) \le NCE\ (b_{i\,jt}) \quad \forall\ cv_k \text{ in } b_{i\,jm} \in cv_k \text{ in } b_{i\,jt} \qquad (7)$$

where, $a \in b$ is defined as *true* when $a = b$ or $a = -$. For example, b_{121} is dominated by b_{124}. A dominated case needs not to be specialized because it has less information and is less important in terms of NCE than dominant case.

To summarize, pruning is performed at each level, but higher level pruning is more effective because its all descendants are removed. Also, notice that pruning sacrifices the amount of the information useful in the specialization step by increasing the possibility that the common situation occurs more frequently (*e.g.* case <2, 200,–> is pruned, but case <–, 200,–> is not). This trade-off is controlled by pruning based on the metric - NCE.

5 COMMON-CASE SPECIALIZATION

5.1 Overview

After having pruned out trivial common cases (which show marginal improvement, even when they are specialized), we have only common cases (expected to show non-marginal improvement by specialization) left in the hierarchical tree. For each remaining case in the hierarchical tree, we perform

the specialization using partial evaluation.The common values of each case are used by partial evaluator for - *i*) simplifying control flow (pre-computing *if* test or unrolling loops), *ii*) constant folding and propagation, *iii*) pre-computing well-known functions calls such as trigonometric functions and so on. These optimizations are not performed independently. Indeed, applying one optimization technique can provide a better chance to other techniques to succeed. For example, loop unrolling can provide better chance to constant propagation/folding by simplifying control dependency and enlarging basic blocks.

Due to such combined effects, it is not easy to estimate the quality of the specialized code analytically. For this reason, this step uses instruction-set level simulator for the purpose of code quality assessment with the consideration of the underlying hardware architecture. It differs from the common-case selection step which performs architecture-independent analysis.Thus, this step takes much longer time than effective case selection step due to specialization and instruction-set level simulation.

Among the techniques mentioned above, loop unrolling should be used most carefully because its side effect (code size increase) can severely degrade both performance and energy consumption. But in traditional applications of partial evaluation, this fact is not deeply studied, based on the assumption that taking more space will reduce computational effort [14].This assumption may be true for general systems such as workstations, but may not be true for the resource limited systems such as embedded systems.Therefore, we need to address our second search problem by exploring various loop combinations for unrolling. The size of search space for each case specialization is simply 2^n, where n is the number of loops inside procedure p_i.

In case of exhaustive search, the specialization of each case is iteratively performed for the overall search space and each iteration requires instruction-set level simulation to assess the specialized code quality. In our framework, loop unrolling can be suppressed by declaring the corresponding loop index variable as a residual variable. It means that the residual variable will not be specialized, henceforth the corresponding loop construct will not be affected by specialization either. Because the search space is exponentially proportional to the number of loops, two heuristic approaches are proposed in this section.These two approaches may provide lower quality of specialization over the exponential approach, but reduce the search space (both specializations and instruction-set level simulations) drastically.

5.2 Semi-exhaustive approach

Unlike pure exhaustive search, semi-exhaustive approach performs an exhaustive search for each loop nest rather than for the entire set of loops. Thus, pure exhaustive search guarantees a globally optimal solution, while semi-exhaustive approach can provide a sub-optimal solution. This is the trade-off between the searching time and the code quality. The trade-off efficiency will be shown in the experimental part (Section 7).

For this purpose, we represent the entire loop structure inside a procedure as a loop graph and an example of loop graph is shown in Figure 6. To construct such loop graph, we first levelize the loop structure.The outermost loop is assigned to *level 0* and the next outermost loop is assigned to *level 1* and so on. Next we represent each loop as a node and place each

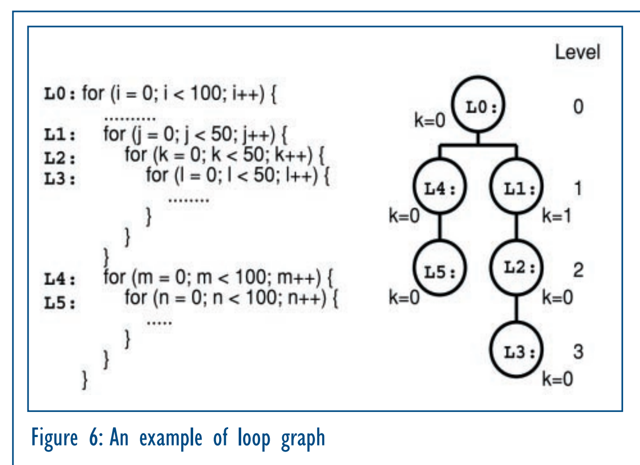


Figure 6: An example of loop graph

node to its assigned level. Finally, we represent the nested relation between two nodes as an edge connecting these two nodes. Notice that if a loop has multiple loop nests, the connecting edges are identified as a *branch* which has as many subgraphs as the number of loop nests. For example, the edge between *L0* and *L4* and the edge between *L0* and *L1* forms a *branch*. Each node is represented as $v_i(k)$, where i is the level to which the node belongs and k is the index of the nodes that have the same parent. Thus, if a node is not connected to a branch, k is always 0.

After constructing a loop graph, the best loop combination for unrolling is searched for each subgraph in a bottom up fashion (*i.e.* the *branch* in the lower level is visited first). For a given branch, we visit the subgraphs in the order of their computational efforts.

While searching the best solution of each subgraph, we exclude the loop combinations which are expected to increase the code size drastically, because such loop combinations increase specialization, compilation, and simulation time drastically. Furthermore, such combinations provide very low quality of specialized code due to the high instruction cache misses. To identify such undesirable cases, we use a code size constraint and a code size estimation technique. The code size constraint is set to the cache size of the target architecture because the code size larger than the cache size will increase the instruction cache miss drastically. Also, the code size is estimated as shown in Equation 8.

$$cs_i(k) = \left(\sum_{j=0}^{|Ki+1|-1} cs_i + 1(j) + NI_i(k) \right) * I_i(k) / U_i(k) \quad (8)$$

where, $cs_i(k)$ the cumulated code size of the descendent nodes of node $v_i(k)$ in addition to the code size of $v_i(k)$ itself. Also, $NI_i(k)$ represents the number of instructions of node $v_i(k)$, $I_i(k)$ represents the average number of iterations per each visiting of node $v_i(k)$. Finally, $U_i(k)$ returns 1 when node $v_i(k)$ is unrolled, and $I_i(k)$ when $v_i(k)$ is not unrolled. In other words, we estimate the code size to be linearly increased by a factor of $I_i(k)$ when $v_i(k)$ is unrolled. Notice that $I_i(k)$ and $NI_i(k)$ are available from the profiler in Section 3.

Example: Consider the loop graph shown in Figure 6, and suppose the subgraph on the right *branch* (L1) has higher computational effort than the one on the left *branch* (L4). In case of pure exhaustive approach, there are 64 (2^6) combinations of loop unrolling, thus the given case should be specialized and simulated 64 times to find the best combination. In case of semi-exhaustive approach, we first visit the right subgraph (L4) because it has higher computational effort. Because the right subgraph is a three-level loop nest (L1, L2, and L3), there are eight combinations of loop unrolling and all combinations are examined to find the best loop combination for the subgraph. While examining these eight combinations, the code size of each combination is estimated using Equation 8. If the estimated code size is larger than the code size constraint, the combination is excluded from the specialization.

After finding the best combination for the right subgraph (*L1*), we visit the left subgraph (*L4*) which has four possible loop combinations and find the best solution in the same way.

After loop unrolling for both subgraphs is decided, we move to the top node (*L0*). There are only two combinations for this node because loop unrolling for all its descendent nodes is already decided. Thus, we need to examine total 14 loop combinations using semi-exhaustive approach.

```
main() {
  int *a, b, *c;
  ........
  for (i = 0; i < 100; i++) {
    ........
    if (a[i] > 0)
      foo(a, b);
    ........
    bar(a, c);
  }
}
bar(int *fa, int *fc) {
  bar2(fa, fa[0]);
  ........
  bar2(fc, fc[0]);
}
```

Figure 7: A more complex example for global effective-case selection

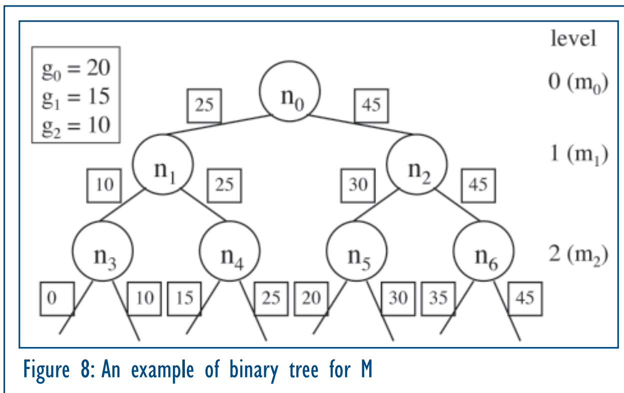


Figure 8: An example of binary tree for M

5.3 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. We will illustrate this approach using the following example.

Example: Let us consider the loop graph shown in Figure 6. The subgraph (*L1*) is visited first due to the same reason in exhaustive approach (higher computational effort). Initially, all are assumed not to be unrolled. However, at this time, all 8 possible combinations are not examined. Instead, unrolled code size is estimated in *depth first order* (from the lowest level (*L3*) to the highest level (*L1*)). First, *L3* is visited and the unrolled code size is estimated. If the unrolled code size is larger than the code size constraint, the code estimation procedure is terminated and the node is decided not to be unrolled. Also, all nodes in the higher level of this subgraph are decided not to be unrolled.

Otherwise (estimated code size is smaller than code size constraint), we decide to unroll this node and move up to node *L2*. The same procedure is repeated until it reaches to the top of the subgraph. After all nodes in the right graph are traversed, we move to the left graph and the same decision procedure is applied. Finally, we move up to the top node and the same procedure is repeated.

To summarize, this approach requires only single specialization and simulation, but it is more limited in improving the quality of partial evaluation.

6 GLOBAL EFFECTIVE-CASE SELECTION

The last search problem is to analyze the interplay among the specialized calls to maximize the specialization effect in a global perspective. We already described this problem in Section 2 using simple example in Figure 1.

Let us consider a more complex situation in Figure 7. Suppose that the call of procedure `foo` and both calls of procedure `bar2` inside procedure `bar` are computationally expensive and have common cases. Then, all three call sites are specialized independently in the common-case specialization step. If we analyze their interplay in a local scope (intra-procedural analysis), two calls inside procedure `bar` will interfere with each other marginally. Furthermore, the interplay between procedure call `bar2` and procedure `foo` is not detected because their interplay occurs in inter-procedure level, even though they may affect to each other severely. Thus, the interplay among the specialized calls should be analyzed in a global scope (inter-procedural analysis). Such an analysis may reveal that the combination of multiple specialized calls may yield a gain inferior to the sum of the gains of the individual specialized calls, because of mutual interference such as I-cache conflict. Also, it is not obvious to estimate their interference analytically. For this reason, each combination should be assessed by instruction-set level simulation and the best combination is chosen for the final solution.

We represent each specialized call $m_k \in M, k = \{0, 1, \cdots, |M|-1\}$. Each m_k has an attribute called gain, g_k which is the amount of improvement in terms of the given cost metric (either energy consumption or performance) and obtained when each call is specialized at the common-case specialization step. We always sort m_k's in descending order for g_k, i.e. $g_k \geq g_{k+1}$. And we deonte a combination of the specialized calls as $c_i \in C, i = \{0, 1, \cdots, |C|-1\}$ and $|C| = 2^{|M|}$, thus the search space is exponentially large. Each c_i is a binary vector to represent

which specialized calls are included in this combination. For example, $c_0 = <1,1,1>$ means m_0, m_1, and m_2 are included in the combination c_0. Also, $c_1 = <1,1,0>$ means only m_0 and m_1 are included in the combination c_1. Each c_i has two gain attributes *ideal_g*$_i$ and *actual_g*$_i$ which are *ideal gain* and *actual gain*, respectively.

- **ideal gain** (*ideal_g*$_i$) is the sum of gains of the individual specialized calls in each combination by assuming that there is no interference with each other. Thus, this is the maximum gain that can be achieved for the given combination.

- **actual gain** (*actual_g*$_i$) is the sum of gains of specialized calls in each combination with the consideration of their interference. Thus, it is always less than or equal to (when there is no interference) the ideal gain and can be obtained by instruction-set level simulation.

We represent each combination c_1 as a path in a binary tree as shown in Figure 8. The rightmost path represents $c_0 = <1,1,1>$ and the second rightmost path represents $c_1 = <1,1,0>$ and so on. Each level of the tree corresponds to each element of the vector c_1 and the right edge and the left edge correspond to "1" and "0", respectively. Thus, the number of levels in the binary tree is always $|M|$. Each edge $e_i(l)$, $i = \{0, 1, \cdots, 2^{(l+1)} - 1\}$ and $l = \{0, 1, \cdots, |M| - 1\}$, also has a gain attribute, $g_i(l)$. Where, l is the level to which the edge belongs and i is the index of an edge in level l (from left to right).
Initially, $g_i(|M|-1)$ (the gain of each edge connected to the leaf nodes) is set to *ideal_gi*. And $g_i(l)$ is set to max $(g_{2i}(l+1), g_{2i+1}(l+1))$ namely the edges above than leaf-level inherit the maximum gain of their children. After the gain initialization as shown in Figure 8, we perform the search procedure based on *branch and bound* algorithm in Figure 9. We will illustrate the how the procedure works using the following example.

Example: $g_7(2)$ (the gain for the right edge of n_6) is initially set to 45 (*ideal_g*$_0$) because this path corresponds to $c_0 = <1,1,1>$ which means m_0, m_1, and m_2 are included in the combination c_0. Similarly, $g_6(2)$ (the gain for the left edge of n_6) is set to 35

```
search_solution (binary_tree) {
  initialze_gain;
  solution = traverse_tree(root_node_of_binary_tree);
  return solution;
}
traverse_tree(binary_tree_node) {
  if (binary_tree_node == NULL) {
    simulate the selected case;
    return sim.  result;
  }
  if (right_node_visited == FALSE) {
    select right_edge;
    gain_right_edge = traverse_tree(right_node);
  }
  if (gain_right_edge >= gain_left_edge) {
    delete left descendent; /* pruning */
    if (solution == NULL)
      save current case to solution;
    return gain_right_edge;
  }
  if(left_node_visited)
    return gain_left_edge;
  else {
    select left_edge;
    gain_left_edge = traverse_tree(left_node);
    if (gain_right_edge >= gain_left_edge) {
      select right_edge;
      save this case to solution
      delete left descendent;
      return gain_right_edge;
    }
    else {
      delete right descendent;
      save current case to solution;
      return gain_left_edge;
    }
  }
}
```

Figure 9: Search procedure for the given binary tree

(corresponds to $c_1 = <1,1,0>$). Also, $g_3(1) = \max(g_6(2), g_7(2)) = 45$ and the gains of other edges are also decided in the same way. Next, we apply the procedure in Figure 9. First, we visit the rightmost path (c_0). For c_0, we perform instruction-set level simulation to get *actual_* g_0 and $g_7(2)$ is updated to $_g_0$. We compare $g_7(2)$ to $g_6(2)$ which is the maximum gain that can be achieved by combination c_1. If $g_7(2) \geq g_6(2)$, it is obvious that c_0 is better than c_1, thus we eliminate the left edge of n_6 (identical to eliminate c_1). On the other hand, if $g_7(2) < g_6(2)$, c_1 can be better than c_0. Thus, we perform instruction-set level simulation for c_1 and update $g_6(2)$ with *actual_g*$_1$. Then, we can decide which combination is better and prune out the worse combination. Next, we move to node n_2 in the next level by updating $g_6(1)$ to *max* $(g_6(2), g_7(2))$ without simulation because we already selected either c_0 or c_1 in level 2.
If $g_3(1) \geq g_2(1)$, we can prune out the left descendent of n_2 (c_2 and c_3) due to the same reason. But, if $g_3(1) < g_2(1)$, we visit node n_5 to choose the better combination from c_2 and c_3 by performing the same procedure as we did for c_0 and c_1. After choosing either c_2 or c_3, we compare two edges of node n_2 and select better one.
We repeat the same procedure until there remains only one path in the binary tree.

7 EXPERIMENTAL RESULTS

Even though source code transformations are applicable to a wide set of architecture, we consider now two specific hardware platforms to be able to quantify the results. The Smart Badge, an ARM processor based portable device [19] and Lx processor developed by STMicroelectronics and Hewlett-Packard [33] were selected as the target architectures. For these target architectures, applied the proposed technique to seven DSP application C programs - `Compress`, `Expand`, `Edetect`, and `Convolve` from [27], `g721 encode` from [26], and `FFT` from [28], `FIR` [34], and `turbo code`.

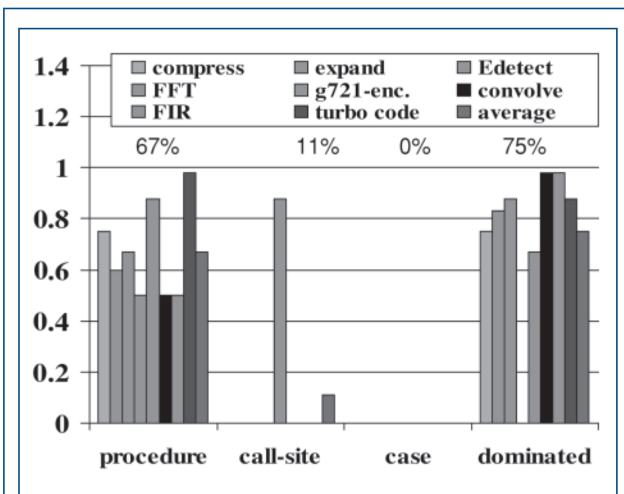


Figure 10: Search space reduction using common-case selection

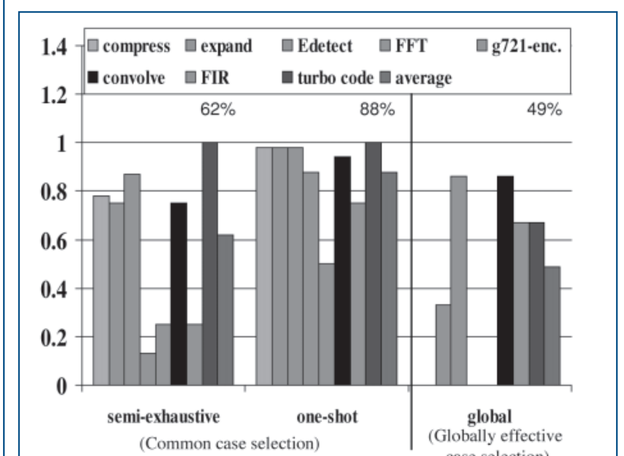


Figure 11: Search space reduction ratio in common-case and global effective-case selection step

`Compress` compresses a pixel image by a factor of 4:1 while preserving its information content using DCT and `Expand` performs the reverse process using IDCT. `Edetect` detects the edges in a 256 gray-level pixel image using Sobel operators and `Convolve` convolves an image relying on 2D-convolution routine.

`g721` is CCITT ADPCM encoder. `FFT` performs FFT using Duhamel-Hollman method for floating-point type complex numbers (16-point). Finally, `turbo code` is iterative (de)coding algorithm of two-dimensional systematic convolutional codes using log-likelihood algebra.

The experiment was conducted for two aspects - search space reduction and quality of the transformed code.

The quality of transformed code was analyzed in terms of energy saving, performance improvement, and code size increase. Each application program was profiled to collect computational effort and CLAs with their common values. There exist two important parameters in value profiling as described in Section 3.3. 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 and the value of which OR is lower than OT is disregarded from the common case values. In this experiment, OT was set to 0. 5.

First, we analyzed the effectiveness of the proposed search space reduction techniques. Figure 10 shows the pruning ratio achieved by each step with computation threshold, *CT*=0.1. Notice that this step is architecture-independent as shown in Figure 2, thus Figure 10 is common to both SmartBadge and Lx processors.

The procedure pruning step always plays an important role to reduce the search space, but call-site pruning step shows large variation depending on the property of the application programs. This is because the computational kernels of some programs such as `compress` and `FFT` were called only once, while the kernel of `g712 encode` was called several times in different sites with different calling frequencies. Thus, this step is useful for the kernels called frequently in different sites with different frequencies.

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
FIR	0.53	0.53	1.12	0.53	0.53	1.12	0.53	0.53	1.12
turbo code	-	-	-	0.89	0.90	1.22	0.89	0.90	1.22
Average	0.71	0.69	1.12	0.74	0.72	1.13	0.75	0.73	1.15

(a) Specialized code quality in SmartBadge environment

C programs	Code Quality								
	exhaustive			semi-exhaustive			one-shot		
	energy	performance	code size	energy	performance	code size	energy	performance	code size
Compress	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Expand	0.94	0.95	1.08	0.94	0.95	1.08	1.00	1.00	1.00
Edetect	0.27	0.26	1.04	0.27	0.26	1.04	0.27	0.26	1.04
FFT	0.18	0.19	1.14	0.18	0.19	1.14	0.18	0.19	1.14
g721 encode	0.91	0.95	1.01	0.91	0.95	1.01	0.91	0.95	1.01
Convolve	0.65	0.68	1.04	0.65	0.68	1.04	0.65	0.68	1.04
FIR	0.38	0.35	1.06	0.38	0.35	1.06	0.38	0.35	1.06
turbo code	-	-	-	0.82	0.82	1.23	0.82	0.82	1.23
Average	0.62	0.62	1.05	0.65	0.65	1.07	0.66	0.66	1.06

(b) Specialized code quality in Lx processor environment

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

The ineffectiveness of the case pruning step was due to high OT which was set to 0.5 for 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 methods used in common-case specialization and global effective-case selection were evaluated. Figure 11 shows the pruning ratios of these two steps for SmartBadge environment. Our technique in Lx processor environment also showed the similar results. As shown in Figure 11, both semi-exhaustive and one-shot approach drastically reduced the search space by 57% and 86%, respectively. Also, pruning technique in global effective-case selection step showed 46% of search space reduction and large variation of pruning ratio depending on the property of application 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.

Next, both one-shot and semi-exhaustive approaches were compared to exhaustive approach in terms of code quality and specialization time. Common-case selection step was commonly used for each approach to avoid large search space. Also, global effective-case selection step was used in all three specializations because it always guaranteed the optimal solution and its worst case run time was same to exhaustive approach. As expected, the one-shot approach showed the smallest running time and semi-exhaustive approach was ranked at second. In average, both one-shot approach and semi-exhaustive approach are about 8.3 (8.0) times and 2.7 (2.5) times faster than exhaustive approach in Smart-Badge (Lx processor) environment, respectively. Notice that Figure 11 only shows the reduction ratio of the search space, which is different from the specialization time

in the sense that search space reduction ratio only implies the reduction ratio of the number of specializations, while the specialization time includes partial evaluation, compilation, and instruction-set level simulation.
It is interesting that exhaustive approach often generated a huge size of code which is one of the main problems in partial evaluation.

For the code, compilation or simulation was not terminated within a few hours, which is a bottleneck for automation. For this reason, we adopted time-out approach especially for the exhaustive approach by assuming that the code requiring long simulation time would be very huge and require large energy consumption.

Table 3 shows the quality of transformed code in terms of energy, performance, and code size for the three approaches. As shown in Table 3, semi-exhaustive approach is comparable to exhaustive approach in terms of transformed code quality with much less computation time (63% for SmartBadge and 60% for Lx processor). One-shot solution is also useful by trading off its code quality and computation time. (About 8.0 times faster and 2% consumes more energy compared to exhaustive approach). We could not perform exhaustive approach for `turbo code` because its computational kernel had too many loops (18) which yielded a huge number of loop combinations (2^{18} = 524288). It is also worthwhile to mention that the deviation of improvement is largely 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% (0%) is improved (`Compress`) in SmartBadge (Lx processor) environment.

It is interesting that our tool specialized `Compress` and `Expand` in different ways depending on the target architecture. `Compress` and Expand show non-marginal improvement in SmartBadge environment, whereas their improvement ratio in Lx processor is marginal. Also, the improvement ratio of `FFT` is much larger in Lx processor environment than in SmartBadge environment, even though the specialized programs for both architectures are identical.

programs	SmartBadge		Lx processor	
	energy	perf.	energy	perf.
Compress float	0.91	0.91	1.0	1.0
Compress fixed	0.80	0.79	0.91	0.91
Expand float	0.84	0.83	0.94	0.95
Expand fixed	0.55	0.53	0.73	0.76

Table 4: Quality improvement ratio of floating-point versions and fixed-point versions by semi-exhaustive approach

The common feature of these programs is that the computational kernels of all three programs have floating-point operations which are not directly supported by the hardware in both architectures, but they are handled by floating-point emulation.

From the careful analysis of these programs, we found two reasons for this fact. First, the computation cost of floating-point emulation in Lx processor is much more expensive than in Smart-Badge environment (relative to their integer operations). Notice that floating-point emulation is performed by the built-in library functions which is out of the scope in our technique. Second, the loop overhead in SmartBadge is larger than in Lx processor.

The results in Table 4 support this claim. `Compress_float` and `Expand_float` are the floating-point versions used in Table 3 and `Compress_fixed` and `Expand_fixed` are their fixed-point versions, respectively. Notice that the improvement by the specialization is mainly due to loop unrolling for both versions of two programs. As shown in Table 4, the improvement ratio using our technique is about 2.5 times larger for the fixed-point version compared to the floating-point version in SmartBadge environment. On the other hand, it is about 5 times larger in Lx processor environment. It means that the relative cost of floating-point emulation in Lx processor environment is twice larger than that in SmartBadge environment. But, the improvement ratio using our technique in SmartBadge environment is still larger than in Lx processor environment. It implies that the loop overhead elimination by our technique is more effective (about twice) in SmartBadge environment rather than in Lx processor environment.

In case of `FFT`, the specialization step eliminates trigonometric functions such as `cos`. The computation cost of `cos` function is four times expensive in Lx processor environment than in Smart-Badge environment in terms of number of clock cycles (measured by simulators). Thus, the elimination of such functions is more advantageous in Lx processor than in SmartBadge environment.

In summary, our technique is more effective in fixed-point arithmetic programs, therefore it is desirable to apply our technique after transforming the floating-point arithmetic programs into the fixed-point arithmetic programs as proposed in [35]. Also, the computation cost of the built-in functions such as trigonometric functions is architecture dependent, thus the impact of the specialization varies largely depending on the underlying hardware architecture.

As a final remark, the run time of the optimization flow depends on the two user-defined constraints *CT* and *OT* that drive the pruning. Also, program size and loop depth are critical factors in specialization step, because our approach uses instruction set-level simulation. Nevertheless, it is important to remember that low energy and fast execution of the target code is the overall objective, which can be achieved at the expense of longer optimization time for large programs.

8 CONCLUSION

We presented algorithms and a tool flow to reduce the computational effort of software programs, by using value profiling and partial evaluation. We showed that the average energy and run time of the optimized programs is drastically reduced. The main contribution of this work is the automation of an optimization flow for software programs. Such a flow operates at the source level, and is compatible with other software optimization techniques, e.g., loop optimizations and procedure in-lining.

Within our approach, a first tool performs program instrumentation and profiling to collect useful information for transformations, such as execution frequency and commonly-observed values at each call site. Using the profiling information, another tool selects common cases based on the estimated computational effort.
Each selected case is specialized independently using a partial evaluator. In the selection step, code explosion due to loop unrolling - which may hamper partial evaluation - is avoided by code size estimation technique and pruning. Finally, the inter-play among the multiple specialized cases is analyzed based on instruction-set level simulation.
The transformed code shows in average 35% (26%) energy saving and 38%(31%) in average performance improvement with 7% (13%) code size increase in Lx processor (SmartBadge) environment.

Currently, our approach has two limitations. First, the common cases only in procedure level are considered. Second, complex data type and/or pointer type parameters are not supported due to their dynamic nature. However, we believe that the first problem can be solved by extending our technique to the lower level common cases (i.e. loop level) which may provide better quality of code specialization, while the second problem still remains as a challenging topic. Also, the specialization technique can be extended to consider more architecture dependent characteristics.

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. M. Rabaey and M. Pedram (editors), Low-Power Design Methodologies. Kluwer, 1996.

[5] V. Tiwari, S. Malik, A. Wolfe, "*Instruction Level Power Analysis and Optimization of Software*", Journal of VLSI Signal Processing Systems, vol. 13, no.1-2, pp.223-233, 1996

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

[7] H. Tomiyama, H. Ishihara, A. Inoue, and H. Yasuura, "*Instruction Scheduling for Power Reduction in Processor-Based System Design*", Design Automation and Test in Europe, pp.855-860, 1998

[8] C.L. Su, C.Y. Tsui, and A.M. Despain, "*Saving Power in the Control Path of Embedded Processors*", IEEE Design and Test of Computers, vol. 11, no. 4, pp.24-30, Winter 1994

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

[10] G. Esakkimuthu, N. Vijaykrishnan, M. Kandemir, M. Irwin, "*Memory system energy: influence of hardware-software optimizations,*" International Symposium on Low Power Electronics and Design, pp. 244-246, 2000.

[11] 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

[12] F. Catthoor, S. Wuytack, E. De Greef, L. Nachtergaele, and H. De Man, "*System-Level Transformation for Low Power Data Transfer and Storage*", A. Chandrakasan, R. Brodersen eds. Low-Power CMOS Design, IEEE Press, 1998

[13] K. Cooper, M. Hall, and K. Kennedy, "*A Methodology for Procedure Cloning*", Computer Languages, vol. 19, no. 2, pp 105-117, Apr., 1993

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

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

[16] J. Pierce, M. Smith, and T. Mudge. Instrumentation tools. in Fast Simulation of Computer Architectures (T. Conte and C. Gimarc, eds.), Kluwer, Boston, MA, 1995, pp. 47-86.

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

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

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

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

[21] C. Liem, T. May, and P. Paulin, "*Instruction-Set Matching and Selection for DSP and ASIP Code Generation*", European Design and Test Conference, pp. 31-37, 1994

[22] G. Araujo and S. Malik, "*Optimal Code Generation for Embedded Memory Non-Homogeneous Resister Architectures*", Intl. Symposium on System Synthesis, pp. 36-41, 1995

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

[24] B. Rau, M. Lee, P. Tirumalai, and M. Schlarsker, "*Register Allocation for Modulo Scheduled Loops: Strategies, algorithms, and Heuristics*", PLDI, pp.283-299, 1992

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

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

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

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

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

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

[31] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Programming Analysis Tools", Proceedings of the SIG-PLAN 1994 Conference on PLDI, pp.196-205, Jun. 1994

[32] Lx Architecture Manual, Version 0.92, August, 17, 1998

[33] Lx Programming and Benchmarking Guide, Version 1.00, 2000

[34] 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.

[35] J. Hagenauer, E. Offer, and L. Papke, "*Iterative Decoding of Binary Block and Convolutional Codes*", IEEE trans. on Information Theory, vol. 42, no.2, March, 1996

ACKNOWLEDGMENT

The authors would like to thank D. Mestdagh, T. B. Ismail, A. Avenel and J. M. Brossier who provided us the turbo code benchmark program.