Energy Efficient Source Code Transformation based on Value Profiling

Eui-Young Chung[†] Luca Benini[‡] Giovanni De Micheli[†]

[†] {eychung,nanni}@stanford.edu
Stanford University
Computer Systems Laboratory
Stanford, CA 94305-4070, USA

[‡] lbenini@deis.unibo.it
Università di Bologna
Dip. Informatica, Elettronica, Sistemistica
40136, Bologna, ITALY

Abstract

This paper presents a source code transformation technique based on value profiling for energy-efficient system design. This technique reduces the computational effort of the program by specializing it for highly expected situations. Thus, the processor running the optimized program can be shutdown or slowed down for energy saving without performance degradation. The value profiler observes parameters passed to the procedures during profiling and detects semi-static parameters which are highly probable to have the same values whenever they are passed to the procedures. According to this information, procedures are specialized for the observed values using a partial evaluator. The corresponding procedure call in the original source code is replaced by a conditional statement which selects either the original procedure call or the specialized procedure call according to the values of the observed parameters. Experimental results show that the proposed technique improves the energy efficiency of the source code up to more than 2 times over the original code.

1 Introduction

Attention to low-power system design has been increasing due to the widespread use of portable devices [2, 3, 4]. Excessive power consumption adversely impacts several key design metrics. First, the battery life time is shortened, thus reducing the usefulness of portable device. Second, heat dissipation is increased proportionally to power consumption, thus packaging and cooling cost is also increased and the system speedup and reliability is limited by this factor. Third, the environmental impact is seriously raised due to the demand for more electricity.

Traditionally, low-power design techniques have focused on circuit and gate level design [4, 5]. Recently, the emphasis has shifted toward higher abstraction levels because power optimizations applied at the early stages of the design process can be more effective [1, 8]. Software optimization is another key issue in low-power design because most large-scale systems include processors and memories. These components are often responsible for a large fraction of system power dissipation [6, 8]. The software running on a processor (and its memory system) determines, to a large degree, its power dissipation.

Clearly, the power consumption due to software execution is tightly related to the target system architecture. For this reason, most of previous research for software optimization has focused on low-level code optimization, *i.e.* assembly or binary executable code, which is the most appropriate level to have the most accurate software analysis model with the consideration of the underlying hardware [7, 9, 10].

Compared to low-level optimization techniques, only a few high-level approaches were proposed. In [11, 12] and [13], sophisticated high-level optimization techniques for performance improvement such as loop unrolling and procedure in-lining were applied for power saving and their impact on power saving is assessed using instruction level simulation. The basic assumption of these approaches is that the positive effect on performance will also be positive to power consumption. Simunic *et al.* proposed an energy efficient source code transformation technique in [20], but their approach is restricted to a specific architecture (ARM processor) and manual code-rewriting. In [14], numerous source-level transformation techniques are introduced, aiming at reducing the power consumed by memories in data-dominated application.

The technique proposed in this paper is also applied at the source code level, and takes into account both processor and memory power. Optimization focuses on computationally-intensive procedures whose input parameters are highly probable to have the same values whenever they are called. We propose a value profiling technique which automatically identifies the procedures and parameters having the properties mentioned above. After selection, candidates are passed to a partial evaluator to be specialized for the identified parameters. Finally, the corresponding procedure calls in the original source code are replaced with conditional statements which selectively call either the original procedure or the specialized procedure depending on the value of the parameters used for the program specialization.

The proposed technique is similar to procedure cloning in the sense that single procedure is replicated multiple times and each new copy is specialized for the call site environments [15]. Also, the authors of [15] proposed a methodology which selectively performs cloning among multiple cloning candidates under the code size constraint using cloning vector. However, our approach is different from the technique described in [15]. First, the cost metric in [15] is code size, but in our approach the cost metric is energy consumption for the given program. Second, the proposed technique can be applied to the call sites which have non-constant input parameters by using value profiling, while the procedure cloning technique [15] only considers constant input parameters. Third, it is possible to generate multiple clones for single call site using the proposed technique, while the procedure cloning technique considers only single clone for each call site. Finally, the approach in [15] still leaves the optmization strategy for each clone to the later stage of compilation, but in our approach, this

```
main () {
  int i, j, a, b[8], result = 0;
  for (i = 0; i < 8; i++)
    scanf(f, "%d", &b[i]);
  scanf(f, "%d", &a);
  result += foo(a, b);
  printf("result = %d", result);
}
int foo(int a, int b[8]) {
  int i, sum = 0;
  for (i = 0; i < 8; i++)
    sum += a * b[i];
  return sum;
}
```

(a) Original Program

```
main () {
  int i, j, a, b[8], result = 0;
  for (i = 0; i < 8; i++)
    scanf(f, "%d", &b[i]);
  scanf(f, "%d", &a);
  if (cvd_foo(a)) result += sp_foo(b);
  else result += foo(a, b);
  printf("result = %d", result);
}
int foo(int a, int b[8]) {
  int i, sum = 0;
  for (i = 0; i < 8; i++)
    sum += a * b[i];
  return sum;
}
int sp_foo(int b[8]) {
  return 0;
}
int cvd_foo(int a) {
  if (a == 0) return 1;
  return 0;
}
```

(b) New Specialized Program

Figure 1. Example of Source Code Transformation Using the Proposed Technique

strategy is formally described.

In Section 2, we demonstrate the basic idea and overall flow of the proposed technique for program specialization using value profiling. In Section 3, the details of profiling method will be presented. In Section 4, the source code transformation technique will be discussed. Finally, we will show the experimental result in Section 5 and conclude our work in Section 6.

2 Basics of the Proposed Technique

In this section, we present the rationale of our approach with an example and describe the overall source code transformation flow.

2.1 Basic Idea

The technique described in the following sections aims at improving the energy efficiency of a given program by specializing it for situations that are commonly encountered during its execution. 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 arguments of procedures and functions.

An example is shown in Figure 1 to illustrate the basic idea. As shown in Figure 1 (a), procedure foo in the original program has two arguments. If the value of variable `a` is 0 for most of the cases, this procedure can be simply reduced to a new procedure `sp_foo` as shown in Figure 1 (b).

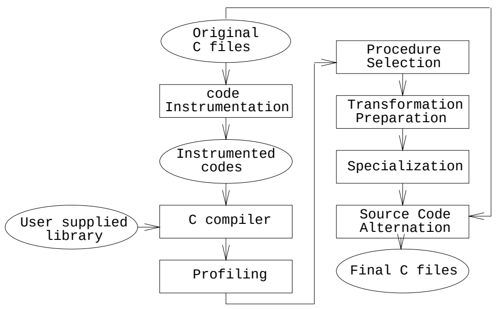

Figure 2. Overall Source Code Transformation Flow

In reality, the value of variable `a` is not always 0. For this reason, the call of procedure `foo` cannot be completely replaced by the new procedure `sp_foo`. Instead, it can be replaced 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). The example in Figure 1 handles only a single common case when $a = 0$. Our approach is not limited to single common case. If there are multiple common cases, the conditional statement is replaced with multi-way branch statement and the corresponding procedure call is selected depending on the value of conditional.

The CVD procedure becomes an overhead factor in terms of computational effort because it must always be executed to call either the original procedure or the new specialized procedure. Also, both the CVD procedure and the new specialized procedure increase the overall code size.

Thus, if the specialized procedure is only marginally simplified, or if the original procedure is more frequently selected than the new specialized procedure, the transformed program may require more computation than the original program. For this reason, our specialization technique is best suited for average energy reduction, while worst-case energy and execution time may slightly increase.

2.2 Overall Source Code Transformation Flow

The simplified overall source code transformation flow is illustrated in Figure 2. As shown in Figure 2, the first step is profiling to identify the procedures to be transformed. Two different kinds of profiling information are important for the identification. First, it is important to know how much each procedure contributes to the overall computational effort. Second, for the computation-intensive procedures, it is necessary to detect the procedures which have Constant-Like Arguments (CLA), *i.e.* the arguments which are highly probable to have the same values. When a procedure satisfies these two conditions, it can be specialized as mentioned in Section 2.1. Profiling for detecting computational kernels, *i.e. execution frequency profiling* is commonly done. *Value profiling* is a less-developed discipline, which has started to attract the attention of the research community only in the last few years [21]. Our approach to value profiling is described in detail in Section 3.3.

These two kinds of profiling can be staged in two different ways. Value profiling could be performed after execution frequency profiling. Alternatively, execution frequency and value profiling could be carried out concurrently. In the first case, value profiling can be applied only to the computation intensive procedures, thus the time required for value profiling can be significantly reduced. But serialization can cause longer profiling time, whereas in second case, profiling is done in one pass, but the amount of work in the single pass is increased.

In our profiler, we opted for the second approach. Instrumentation of the original C source code is performed in a preliminary step (code instrumentation) and the instrumented code is compiled with specialized libraries for profiling. During profiling, both execution frequency and value information for each procedure are obtained by executing the binary code generated by the compilation step. Based on profiling information, the candidate procedures which are computationally intensive and have CLAs are selected for specialization in procedure selection step. In transformation preparation step, those procedures are copied into a new file with assigning values to CLAs.

The newly generated file is fed into the partial evaluator for specialization. In our current framework, specialization step is performed by CMIX [17] which is a compile-time (off-line) program specializer, and the interface from profiling to CMIX is automated. Our framework is not restricted to a specific partial evaluator. Any other partial evaluator can replace CMIX. In addition to the specialized procedure, the CVD (Common Value Detection) procedure is also automatically generated in this step. In the final step (source code alternation), the original source code is automatically transformed into a new specialized code by replacing the procedure call with the conditional statement as shown in Figure 1. The current framework is semi-automated in the sense that the tool provides the information needed to make the choice and performs a tentative selection, but the candidate procedure selection can be overridden by the user and the final decision on which procedure calls should be replaced is also left to the user.

3 Profiling

3.1 The Structure of Profiler

Most profiling techniques are aimed at low level abstraction of programming language, *i.e.* 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 [18].

For the technique proposed in this paper, 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 [22] for source code instrumentation because it provides essential core routines for program analysis and manipulation. The instrumentation for profiling 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 semantics such as `for-loop`, `if-then-else`, `do-while` constructs 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 their execution order. Note that node v_i is hierarchical, thus each node v_i can have its subgraph to represent the nested constructs. For each node v_i which is a procedure, we insert counters as many 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 [19] in the sense that user supplied library is used for instrumentation, namely the source code is instrumented with simple counters and procedure calls which are defined in user supplied library as shown in Figure 2. The user supplied library includes the procedures required for both execution frequency profiling 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 Kernel Identification

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

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

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 for-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 efforts estimation. As mentioned in 3.1, the node v_i is hierarchical. Thus, the cumulated computational efforts for each node v_i can be estimated by the sum of current level computational effort and the computational effort of its descendent nodes.

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 that 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.

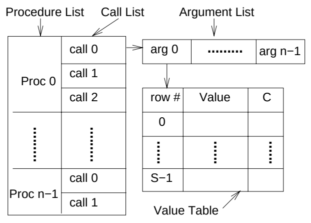

Figure 3. Internal Date Structure of Value Profiling

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 3 shows the internal data structure of value profiling system. As shown in Figure 3, 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.

$$or_i = c_i / \sum_{j=0}^{S-1} c_j \qquad (2)$$

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 [21]. 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.

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

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 $r_j \rightarrow r_{j-1}$, $j \in \{i + 1, \cdots S-1\}$ if $j < S-1$. Finally, the new value is stored to a new row r_{S-1}.

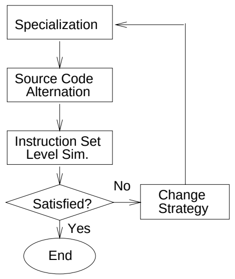

Figure 4. Iterative Procedure for Source Code Transformation

4 Source Code Transformation

From the profiling information, the procedure which is identified as a computational kernel and has CLA is selected for code transformation. The code transformation in our approach depends on the existing partial evaluator called CMIX, but it can be replaced by any other specializer. Even though the selected procedure is specialized by external software, it is possible to control specialization by using a control file which is fed to the specializer. This is one of the reasons that the interface from profiling to specialization is semi-automated, namely there is a chance for user to control the specialization. The input C-code file to specializer is automatically prepared from the user script which specifies the target procedure and the values of arguments to be declared as static.

The partial evaluator performs optimization for the given code using the information on static data. Thus, constant-related optimizations such as constant propagation and constant folding are the important optimization features in partial evaluation. Also, other optimization techniques such as loop unrolling, simple algebraic simplification, procedure in-lining, and procedure cloning are performed.

One of the side effects of partial evaluation is that taking more space for programs and data may produce less computations, while taking less space for those may produce more computations [16]. For this reason, the code size of the specialized procedure is usually larger than the size of original procedure. Increased code size changes the memory access pattern for both data and instruction, therefore system performance and power consumption after specialization are hard to predict.

To overcome this problem, we adopt an iterative procedure as shown in Figure 4. Whenever specialized code is generated, instruction-set level simulation is performed to see the impact of specialization. While the iterative specialization procedure is performed, user can change the specialization strategy as follows.

- **Loop unrolling suppression:** The loop which would cause significant code size increase when it is unrolled can be suppressed not to be unrolled.
- **Static data declaration change:** When the argument defined as static does not have much impact on the improvement, it can be changed to dynamic argument. This technique can be effective when several arguments are declared as static for a single procedure call.

- **Canceling specialization:** When the specialization for some procedure call is not so effective on the improvement, the corresponding call can be canceled for the specialization.
- **CVD procedure elimination:** When it is possible to identify that a static argument always has the same value, the CVD procedure is not required. In this case, the overhead from CVD procedure can be eliminated.

5 Experimental Results

We applied the proposed technique to the ARM software development environment [23], with a target hardware architecture which consists of ARM SA-110 processor, and 1MB burst SRAM main memory. Under the ARMulator based environment. ARM SA-110 has both 16KB I-cache and 16KB D-cache and its clock speed is set to 99.4MHz. The power consumption of SA-110 is assumed as 450mW in active state and 20mW in idle state. Also, the memory access time is assumed as 90ns for non-sequential access and 45ns for sequential access and its power consumption is 305mW in active state and 1.7mW in idle state. All these values are adopted from the data book. Under the system configuration mentioned above, 6 DSP application C programs were applied for validation purpose. The metrics measured in this experiment can be summarized as follows.

- **Computational kernel identification:** The proposed frequency execution profiling with high-level computational effort estimation technique is compared to the profiler provided by ARMulator.
- **Value profiling correctness:** We compared the CLA obtained from the proposed value profiling with the result of value tracing to validate the proposed value table replacement policy.
- **Code size:** The impact on the code size due to source code transformation is assessed.
- **Speedup:** The effect of the proposed technique for performance is also evaluated.
- **Energy saving:** The energy consumption of the given program is estimated by the sum of energy consumption of both processor and memory and it can be represented as follows.

$$ENERGY_{total} = \sum_{i=0}^{1}\sum_{j=0}^{1} P_{ij} * N_{ij} * T_{cycle} \qquad (4)$$

  where, $i \in \{0,1\}$ and 0 represents processor and 1 represents memory, respectively. Similarly, $j \in \{0,1\}$, where 0 means active state and 1 means idle state, respectively. Finally, P_{ij} represents average power consumption of resource i in state j, N_{ij} represents the number of cycles that resource i is in state j, and T_{cycle} is the cycle time.

For the comparison shown in Table 1, it is assumed that the cost of each instruction is identical, *i.e.* 1 for computational effort estimation. The value table size is fixed as 10 and the weighting factor W used for replacement factor calculation is set to 0.5. Finally, the OT (Observed Threshold) is also set to 0.5.

Even though the instruction cost is assumed to be identical, the relative importance of each procedure evaluated by the proposed profiling is exactly same as the result from the ARMulator. The absolute importance of each procedure is also evaluated with 90% accuracy. It is also verified that CLA detection using the proposed value profiling is exactly performed with reduced computation time and disk space.

C Programs	comp. kernel		# of CLA	
	proposed	ARMulator	proposed	trace
Compress	block_dct	block_dct	2	2
Expand	block_idct	block_idct	2	2
Edetect	convolve2d	convolve2d	1	1
FFT	fft	fft	1	1
g721 encode	quan	quan	2	2
Composite	nv_composit	nv_composit	5	5

Table 1. Proposed Profiling Technique Validation

C Programs	code size	execution time	energy
Compress	1.30	0.93	0.94
Expand	1.18	0.86	0.86
Edetect	1.25	0.42	0.39
FFT	1.16	0.83	0.80
g721 encode	1.04	1.00	1.01
Composite	1.17	1.01	1.02

Table 2. Comparison between Transformed Code and Original Code (normalized to original code)

Table 2 shows the improvement achieved by the proposed technique over the original C programs.

For each program, different specialization strategy is applied due to the different nature of each program. The strategy for each program is summarized as follows.

- **compress:** The computational kernel, block_dct procedure has two three-level nested for-loops. Thus, the specialization causes drastic code size explosion. Code explosion is a serious problem in embedded system environment due to the small size of cache. For this reason, one out of three levels for each nested loop is not unrolled.
- **expand:** expand is similar to compress program. Thus, same strategy is applied. Nevertheless it shows better improvement than compress program because the portion of computational effort devoted to each computational kernel is different. The computational kernel of expand requires higher computational effort than that of compress program.

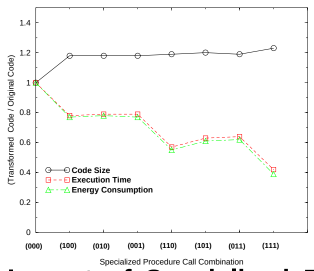

Figure 5. Impact of Specialized Procedure Call Combination (edetect program)

- **edetect:** edetect shows the best improvement in terms of both performance and energy. It is interesting that the computational kernel is called in three different locations with different values passed to CLA. For this reason, the computational kernel is specialized three times for the common CLA with three different values. Also, the computational kernel includes four-level nested for-loop, thus two outer loops are not unrolled. It is also worthwhile to mention that simple algebraic reduction can be performed because there are several multiplications with 0 or 1 in the unrolled code. For this reason, drastic performance and energy improvement can be achieved. Figure 5 shows the variation of improvement with the change of the combinations of calls specialized. Each tuple index on X-axis represents which of the three calls are selected for specialization. For example, (100) means only the first procedure call is specialized and (111) means all three procedure calls are specialized. As shown in Figure 5, the performance increase and energy reduction ratio show more steep slope than code size increase. Thus, it is better choice to select all the procedure calls for specialization.
- **FFT:** The FFT program is specialized for 16 points. It is interesting that the trigonometric functions which are provided by the C standard math library has constant arguments when they are specialized. For this reason, the specialized program does not contain the expensive trigonometric functions, but has the pre-computed their constant outputs.
- **g721:** The computational kernel has two CLAs - one is constant array and the other is an integer which represents the array size. At first, we specialized the program for both CLAs, but the result is worse than the original program due to the overhead of CVD procedure. For this reason, only the integer CLA is used for specialization and there is little improvement as shown in Table 2. But it is found that the array CLA is true constant which is defined with its declaration. Therefore, it is possible to eliminate the CVD procedure. With this modification, it is possible to achieve 24.2% performance improvement and 22.3% energy saving over the original program. The kernel of this program is not so computationally intensive, but it is called so frequently. Thus, the effect of CVD procedure is significant because CVD procedure is evaluated at each procedure call.
- **composite:** Even though the computational kernel of this program has five CLAs, it is shown that there is little benefit to specialize the given program for those CLAs. In this case, the performance and energy consumption can be worse than the original program due to the overhead of CVD procedure.

In average, the proposed technique improves the energy efficiency of the source code by 20% (25% with the consideration of CVD elimination for g721 encode) with 18% code size increase for six DSP application programs. It is observed that the deviation of improvement is very large depending on the nature of the programs. For the best case, the improvement is more than twice, but for the worst case, the transformed code can be worse than the original code due to the overhead of the CVD procedure.

6 Conclusion

In this paper, we presented an energy efficient source code transformation technique based on value profiling and partial evaluation. The proposed high-level execution frequency profiling and computational effort estimation technique for computational kernel identification is validated by comparing it to a commercial instruction-level profiler. Also, the value profiling technique is verified by the comparison with value tracing which requires more computation time and disk space.

Based on the profiling information, a source code transformation technique using partial evaluation is proposed and its impact on code size, performance, and energy consumption is evaluated. The proposed technique reduces energy consumption (as well as average-case performance), but it also increases program size. For some application programs, code size increase can be serious. To avoid this code size explosion, a careful specialization strategy unrolling suppression is required.

The experimental results show that the transformed source code can reduce processor and memory energy consumption by more than a factor of two compared to original program, but the effectiveness of the proposed technique depends on the property of the given application program. It is also shown that it is possible to apply the proposed technique to tradeoff performance, power, and code size (edetect program). Finally, we show a possibility to improve the effectiveness of the proposed technique by eliminating the CVD procedure overhead when the CLA is true constant(g721 encode program).

Acknowledgement

This research has been supported in part by ST Microelectronics and by NSF under grant CCR-9901190. Also, the authors would like to thank Dr. G. Luculli for several comments and suggestions.

References

[1] L. Benini and G. De Micheli, *Dynamic Power Management of Circuits and Systems: Design Techniques and CAD Tools*, Kluwer, 1997.

[2] A. Chandrakasan and R. Brodersen, *Low-Power Digital CMOS Design*. Kluwer, 1995.

[3] W. Nebel and J. Mermet (Eds.), *Low-Power Design in Deep Submicron Electronics*. Kluwer, 1997.

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

[5] M. Pedram, “Power minimization in IC design: principles and applications”, *ACM TODAES*, vol. 1, issue 1, pp.3-56, Jan. 1996

[6] V. Tiwari, S. Malik, and A. Wolfe, “Power Analysis of Embedded Software: A First Step Towards Software Power Minimization”, *IEEE Transactions on VLSI*, vol. 2 no. 4, pp437-445, Dec. 1994

[7] 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

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

[9] H. Tomiyama, H. T. 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, Feb. 1998

[10] 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

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

[12] 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.

[13] 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

[14] 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

[15] K. Cooper, M. Hall, and K. Kennedy, "A Methodology for Procedure Cloning", Computer Languages, Volume 19, Number 2, pp 105-117, April, 1993

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

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

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

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

[20] T. Simunic, L. Benini, and G. De Micheli, "Energy-Efficient Design of Battery-Powered Embedded Systems", *Proceedings of ISLPED*, pp.212-217, 1999

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

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

[23] Advanced RISC Machines Ltd (ARM), ARM Software Development Toolkit Version 2.11, 1996