A New Approach to Embedded Software Optimization Based on Reverse Engineering**

Nguyen Ngoc Binh†, Member, Pham Van Huong†, and Bui Ngoc Hai†, Nonmembers

SUMMARY  Optimizing embedded software is a problem having scientific and practical significance. Optimizing embedded software can be done in different phases of the software life cycle under different optimal conditions. Most studies of embedded software optimization are done in forward engineering and these studies have not given an overall model for the optimization problem of embedded software in both forward engineering and reverse engineering. Therefore, in this paper, we propose a new approach to embedded software optimization based on reverse engineering. First, we construct an overall model for the embedded software optimization in both forward engineering and reverse engineering and present a process of embedded software optimization in reverse engineering. The main idea of this approach is that decomposing executable code to source code, converting the source code to models and optimizing embedded software under different levels such as source code and model. Then, the optimal source code is recompiled. To develop this approach, we present two optimization techniques such as optimizing power consumption of assembly programs based on instruction schedule and optimizing performance based on alternating equivalent expressions.

key words: embedded software optimization, reverse engineering, power consumption, performance, instruction scheduling, genetic algorithm

1. Introduction

The optimization problem of embedded systems includes hardware optimization, software optimization and co-design - optimizing based on hardware-software partitioning. Hardware optimization can be done under four levels: system, CPU level, logic level and circuit level. Embedded software optimization is more difficult than normal software optimization because of the dependency on embedded hardware and development environment. Embedded software optimization often includes some optimal aspects such as power consumption, performance, memory. Moreover, embedded software optimization can be done in the different phases such as design, coding and executing.[1]–[4], [9], [11].

The optimization methods can be done in the embedded software development under both forward and reverse engineering. Using the optimization methods in forward engineering is to create the optimal embedded software starting from the design phase. Optimization is done in reverse engineering to improve the existing embedded software. Reverse engineering and re-engineering is a trend being studied and applied widely in software engineering [6], [10], [15]. Thus, optimizing the embedded software in reverse engineering is a new, promising approach. The essence of optimization is based on reverse engineering includes three processes such as reverse engineering, optimization and re-compiling. Reverse engineering also includes some different levels such as converting the machine code to assembly code, converting assembly code to high-level source code and converting source code to models. After each conversion level, the optimization methods in forward engineering can also be applied[8],[14]. In this research, we construct an overall model for the optimization problem and propose two optimization methods based on reverse engineering, these are optimizing the power consumption of assembly programs based on instruction scheduling and optimizing performance of high-level source code based on replacement of equivalent expressions. Our methods are applied on benchmark programs of SimpleScalar[13] for experiment.

The remaining parts of the paper are arranged as follows: Section 2 presents the overall model of the optimization problem. Sect. 3 presents the optimization method of assembly programs based on reverse engineering and instruction scheduling. Sect. 4 presents the performance optimization method based on reverse engineering and replacing equivalent expressions. Sect. 5 is the conclusion and future work.

2. An Overall Model of Embedded Software Optimization

After analyzing and evaluating related researches on embedded software optimization, we aggregate and construct an overall model for optimization problems in the embedded software development as in Fig. 1. According to this overall model, the optimization problem of embedded software is divided into two main approaches such as optimization in forward engineering and optimization in reverse engineering. In the first approach, from requirements specification, embedded software can be designed by using different models and we can select the best model based on doing the optimization methods in the design phase. In the imple-
In the implementation phase, a model can be implemented by the high-level source code being independent on CPU architecture and we can do the optimization methods on the high-level source code. The embedded software optimization in design phase and the high-level source code is similar to the conventional software optimization. After that, the high-level source code is compiled to assembly code associated with a specific embedded CPU. In the assembly code, we can apply optimization methods and optimization techniques to gain good assembly language programs. These optimization methods often have specific characters of the different CPU architecture and hardware environment of the specific embedded systems. Assembly code can be compiled and linked to create an executable file. In the executing phase, the embedded software optimization methods mostly focus on optimizing execution environments, specifying data and configuring executable code.

Based on the optimization methods in forward engineering, we proposed the optimization approach based on reverse engineering. As mentioned in the previous section, reverse engineering can be done in different levels: executable code to assembly code; assembly code to high-level
source code; high-level source code to the design model. Assembly code can also be directly converted to models without the high-level source code. For example, we can generate a task flow diagram, a dependency graph, etc. from assembly code. Output at each level in reverse engineering can be optimized under the corresponding level in forward engineering. Thus, in reverse engineering, an optimization method is a combination of a de-compiling level and a corresponding optimization method in forward engineering.

Moreover, the model in Fig. 1 is only the general model built to provide the completed view of the optimization problem of embedded software. Therefore, each optimization method can be done under a part of this model. For example, we can transform the high-level source code to a class diagram and optimize under the class diagram to gain the better structure.

In the next two sections, we develop two optimization methods in reverse engineering that is optimizing the power consumption of the assembly program based on instruction scheduling and optimizing the performance based on replacing equivalent expressions. In the first method, we only optimize at the assembly code level as follows: disassembling machine code to assembly code of the MIPS processor of RISC architecture; scheduling assembly instruction to optimize power consumption. In the second method, we only decompile machine code to C code and optimize at the high-level source code.

3. Optimizing the Power Consumption Based on Reverse Engineering and Instruction Scheduling

3.1 Idea and the Deployment Process of the Optimization Method

Software controls most activities of hardware in the systems, therefore, it can have a significant effect on the power dissipation of a system, and power optimization can be achieved by software techniques and instruction scheduling is an effective software approach. Our method based on reverse engineering and instruction scheduling to reduce power consumption of embedded software. This method is applied to the processors having RISC architecture. Moreover, in this method, we suppose that embedded software does not require time constraint.

Scheduling is an NP-hard problem; the main difficulty is that the search space of the possible instruction orders is very large. When finding a good schedule, we usually stuck to the constraints of the data flow graph, i.e. when we create a new order of instructions, we are not sure whether it satisfies the data flow graph or not. Here, our approach uses the genetic algorithm with a chromosome encoding that solves the data dependency problems better. This method was introduced in [16], the authors proposed the method and used it to solve TSP; we use their idea to apply to the scheduling problem for reducing energy consumption. This algorithm has the advantage of avoiding the local optimum. The flow of our method is shown in Fig. 2.

Fig. 2  Power optimization based on reverse engineering and instruction scheduling.

For finding solutions in the large search space, we use a heuristic table, called Power Dissipation Table (PDT), which is generated by power simulations. A PDT for an instruction set with n instructions is a \((n \times n)\) matrix, where each entry \(PDT(i, j)\) is the power cost that consumed in the execution of instruction \(i\) followed by instruction \(j\), each entry is used as overhead cost between \(i\) and \(j\), and this table is used for evaluating the solutions. Original assembly programs are divided into basic blocks, then a Data Flow Graph (DFG) is constructed for each basic block, this is a directed graph that presents the data dependencies of instructions in a basic block. Our algorithm is applied for each basic block of an assembly program; it takes as input a data flow graph of a given basic block and the power dissipation table and output the low power instruction sequence. For experiments, we use two open source simulation tools that are SimpleScalar Tool Set \([13]\) and SimplePower \([5]\). The sub set of SimpleScalar Instruction Set is considered and SimplePower is used to simulate the power consumption, the algorithm is applied to assembly programs of SimpleScalar ISA, then these programs are compiled and then have their power consumptions measured by SimplePower for visual observation.

3.2 Genetic Algorithm for Low Power Instruction Scheduling

Genetic algorithm is very effective in problems which have
Fig. 4  Basic blocks and assembly code of a decompiled program.

Table 1  Experimental results of GA scheduling.

| No. | Benchmark    | Power Unscheduled (pF) | Power Scheduled (pF) | Power Reduction (%) |
|-----|--------------|------------------------|----------------------|---------------------|
| 1   | Example1     | 18387.7622             | 14467.3678           | 21.32               |
| 2   | Example2     | 19304.6189             | 15315.5843           | 20.66               |
| 3   | Example3     | 13951.9957             | 11371.1628           | 20.50               |
| 4   | Example4     | 12115.3119             | 10861.9029           | 18.85               |
| 5   | Quick Sort   | 2077771.2516           | 1928151.7644         | 7.29                |
| 6   | Bubble Sort  | 12365628.8359          | 11355245.3421        | 8.33                |
| 7   | Binary Search| 11500.0162             | 11483.8485           | 0.14                |
| 8   | Hanoi        | 6341659.6373           | 6176948.4327         | 2.60                |
| 9   | Heap Sort    | 3598279.6097           | 3526183.7633         | 2.00                |
| 10  | Permutation  | 24001185.1845          | 234057651.7652       | -6.24               |
| 11  | Matrix Mul   | 110399.0015            | 107020.6322          | 2.98                |
| 12  | Fast Fourier | 15510.3116             | 15600.4884           | -0.58               |
| 13  | RSA Encryption| 23137.0363             | 22616.5602           | 2.25                |
|     | Average      |                        |                      | 7.39                |

A topological sort is an ordering of vertices in a directed graph, such that if there is a path from vertex \( v_i \) to vertex \( v_j \), then \( v_j \) appears after \( v_i \) in the ordering. In the topological sorting procedure, in each step, select any vertex without incoming edges and then store the vertex and its position. Then, the vertex and all the arcs from this vertex are removed from the graph. Scheduling is similar to Topological sorting; from the Data Flow Graph, we have to choose an order that satisfies the constraints of the graph. Our scheduling problem is to find a topological order so that the total cost through all vertices is the smallest or smaller the original’s one, the cost between two vertices in a sequence is the overhead cost between two instructions. However, there are more than single sequence of vertices can be derived from a large search space; this is a flexible algorithm and can be applied in many different areas such as engineering, computer science, biology, etc. We use the genetic algorithm (GA) approach based on topological sorting for low power scheduling problem.

Table 2  Experimental results of list scheduling.

| No. | Benchmark    | Power Unscheduled (pF) | Power Scheduled (pF) | Power Reduction (%) |
|-----|--------------|------------------------|----------------------|---------------------|
| 1   | Example1     | 18387.7622             | 14467.3678           | 21.32               |
| 2   | Example2     | 19304.6189             | 15315.5843           | 20.66               |
| 3   | Example3     | 13951.9957             | 11371.1628           | 20.50               |
| 4   | Example4     | 12115.3119             | 10861.9029           | 18.85               |
| 5   | Quick Sort   | 2077771.2516           | 1928151.7644         | 7.29                |
| 6   | Bubble Sort  | 12365628.8359          | 11355245.3421        | 8.33                |
| 7   | Binary Search| 11500.0162             | 11483.8485           | 0.14                |
| 8   | Hanoi        | 6341659.6373           | 6176948.4327         | 2.60                |
| 9   | Heap Sort    | 3598279.6097           | 3526183.7633         | 2.00                |
| 10  | Permutation  | 24001185.1845          | 234057651.7652       | -6.24               |
| 11  | Matrix Mul   | 110399.0015            | 107020.6322          | 2.98                |
| 12  | Fast Fourier | 15510.3116             | 15600.4884           | -0.58               |
| 13  | RSA Encryption| 23137.0363             | 22616.5602           | 2.25                |
|     | Average      |                        |                      | 7.39                |

Fig. 5  The process of optimizing performance based on replacing equivalent expressions.
Fig. 6 Steps to prove two equivalent expressions.

| Step | Expression 1          | Expression tree | Expression 2          |
|------|-----------------------|-----------------|-----------------------|
| 1    | \((a+b)\cdot c-d/e\) | ![Expression Tree 1](image) | \((c\cdot(b+a))-(d/e)\) |
| 2    | \(((a+b)\cdot c)-(d/e)\) | ![Expression Tree 2](image) |                       |
| 3    | \((a+b)\)\n\((d/e)\) | ![Expression Tree 3](image) | \((b+a)\)\n\((d/e)\) |
| 4    | \((a+b)\cdot c\) | ![Expression Tree 4](image) | \(c\cdot(a+b)\) |
| 5    | \(((a+b)\cdot c)-(d/e)\) | ![Expression Tree 5](image) | \((c\cdot(b+a))-(d/e)\) |

Fig. 7 A DAG of a basic block.

directed graph using this topological sorting procedure. To overcome this issue, and to obtain a feasible complete path from a directed graph, an ordering technique using the topological sort and random assignment of priority is used [16].

A random priority assignment technique is used to randomly assign a different priority to each vertex in the graph. For a graph, each priority sequence can derive a unique sequence of vertices by apply the topological sort. Therefore, a string of priorities can represent a feasible path.

Instead of using a sequence of vertices of the graph, we can use a string of priorities to represent chromosome, each string of priorities will represent one individual in the population. Figure 3 shows an example of chromosome representation.

The cross over operator generates a new chromosome from two old chromosomes. For constructing the cross over operator, we base on the operator called Moon Cross Over introduced in [16] and modify it. Moon Cross Over creates a substring of a chromosome and constructs a new one from it; our cross over operator does the same but reverses the substring before constructs the new chromosome. In the mutation operation, any two genes will be randomly selected and swapped. The fitness function is the total cost from the first vertex to the last vertex of the sequence of vertices which have been sorted. In our problem, the path between each pair of vertices is the corresponding PDT value when switching between two instructions that correspond to these vertices. Our goal is to select the sequence that has the smallest fitness function value.

### Table 3 Common subexpression elimination.

| Original code | Code after removing sub-expressions |
|---------------|-------------------------------------|
| \(x = (a + b \cdot c) \cdot 10\) | \(\text{temp} = a + b \cdot c\) |
| \(y = a + b \cdot c - 100 + d\) | \(x = \text{temp}\) |
| \(z = a + c \cdot b + e \cdot f\) | \(y = \text{temp} - 100 + d\) |
| \(t = b \cdot c + a - d \cdot e \cdot f\) | \(z = \text{temp} + e \cdot f\) |
| ... | \(t = \text{temp} - d \cdot e \cdot f\) |

3.3 Experiment and Evaluation

As mentioned in previous sections, in experiments, we use two simulation tools - SimpleScalar [13] and SimplePower [5]. The compiler ssbig-na-strix-gcc of SimpleScalar is used to compile an assembly program into benchmarks. From assembly programs, we create bench-
### Fig. 8
Comparison of the C source code and the decompiled code.

| Origin source code | Decompiled source code |
|--------------------|-----------------------|
| int n, first = 0, second = 1, next, j, i=0; int exp1, x=10, y=25; int main() |
| { |
| n=40; exp1 = x*y; |
| for(j=0; j<1000; j++) |
| { |
| printf("\nFibonacci series from 0-\%d are:\n\%d\n\t",n, first); first=0; second=1; exp1 = x*y; i=0; while (i < 40) |
| { |
| next = first + second; first = second; second = next; |
| } |
| } |
| int i; int first; int second = 1; int y = 25; int x = 10; int j; int exp1; int n; int next; int main(int argc, char **argv, char **envp) |
| { |
| int local6; // first(95) int local7; // second(94) int local8; // second(107) int local9; // first(108) n = 40; exp1 = x * y; j = 0; local8 = second; local9 = first; second = local8; first = local9; while (j <= 999) |
| { |
| printf("\nFibonacci series from 0-\%d are:\n\%d\n\t", 40, first); first = 0; second = 1; |
| }

### Fig. 9
Experimental process.

1. **Compile with GCC CSE**
   - **P1’**
   - **P1”**

2. **Compile with GCC -o0**
   - **P1**

   1. **Compile with GCC -o2**
      - **P2’**
      - **P2”**

   2. **Compile with GCC -o3**
      - **P3’**
      - **P3”**

   3. **Compile with GCC CSE**
      - **P4’**
      - **P4”**

   4. **Use Boomerang to decompile**
      - **P4**

   **Evaluation**
marks and execute them on SimplePower to measure power consumption. Then, we apply scheduling algorithm to these programs, create benchmarks and measure with SimplePower for comparison. All modules of our work were written in C. Our algorithm parameters are selected as follow: population size \( \text{pop\_size} = 100 \), max generation \( \text{max\_gen} = 200 \), probability of cross over \( pc = 1 \), probability of mutation \( pm = 0.5 \). We also implement the list scheduling algorithm - a greedy algorithm that used by Tiwari [12], then running both list scheduling and our GA scheduling with the same benchmark set and compare the results of them. We created 4 assembly programs manually that named Example1, Example2, Example3 and Example4. Each consists of 20 repetitions of a basic block of random instructions. They have a feature that their instructions are less interdependent than the available sample program. Our benchmark set include these 4 programs, 7 programs that were selected from the sample benchmarks of SimplePower, and 2 common programs of embedded systems that are Fast Fourier Transform and RSA encryption.

To do this experiment, the first we use objdump, part of GNU Binutils, to disassemble the executable files formatted by elf to MIPS assembly files. Figure 4 shows basic blocks and assembly code of the assembly program decompiled from an executable file. After that, we do our scheduling program mentioned above to optimize power consumption of the programs. The experimental results are shown in Table 1 and Table 2, where \( pF \) is picofarad.

From the results in Table 1 and Table 2, we note that applying our approach made lower power consumption in all cases; up to 20% of power consumption reduction have been obtained. We also can see that when the instructions are less interdependent, the algorithm perform more effectively. The experimental results showed that the genetic algorithm is a good approach for the problem of scheduling for low power, with a large search space and a difficulty to find the optimal solution. In general, using genetic algorithm has an advantage compared with other greedy scheduling algorithms; this is, greedy algorithms often fall into a local optimum, and GA can avoid it [2].

### 4. Optimizing the Performance Based on Reverse Engineering and Replacing Equivalent Expressions

#### 4.1 Idea and the Deployment Process of the Optimization Method

In this section, we propose an improvement of the local optimization method for high-level source code based on reverse engineering and replacing equivalent expressions. This improvement is not dependent on processors so it can be applied for all processors, general software, and embedded software. The main idea of this method is that decompiling an executable file to a high-level source code, analyzing and replacing equivalent expressions. Because all equivalent expressions in a program are replaced by an expression, it only needs to calculate equivalent expression once. Therefore, the executing time of the program is decremented. The deployment process of optimization is shown in Fig. 5.

#### 4.2 Develop the Optimization Method

- Identifying and replacing equivalent expressions

The important point of our improvement is the need to determine equivalent expressions based on the mathematical properties. To prove automatically that two expressions are equivalent, we need to construct the expression trees. Figure 6 illustrates the steps to analyze and construct tree expressions. In step 1, the original expression is added some parentheses to distinguish operations by priority. Base on the single operands which are constants, variables that are analyzed in step 2, the sub-trees allow the highest priority operations be constructed in step 3. The process continues for lower operations as in step 4 and 5. In each step, we can indicate that two expressions are equivalent by the commutative property. Two expressions are equivalent if they are equivalent in every step.

Base on the method for determining two equivalent expressions as described in the previous section, in this section, we will mention about replacing an expression with an equivalent representative expression in a basic block. This replacement procedure is done before applying optimization techniques to improve the quality of optimum while removing common expressions. First, we browse each statement in a basic block to determine expressions and sub-expressions [7], [8]. Then, we identify equivalent expression from those. Finally, we replace the equivalent expressions by a representative one to obtain a code block with some same expressions. These same expressions will be removed by the optimization technique that is presented in the next section.

- Constructing DAG of a basic block

Local optimization methods in a basic block usually use a directed acyclic graph, which represents this basic block. Base on DAG, we apply optimization techniques such as removing common sub-expressions; removing dead code. A DAG of a basic block is described as follows: Leaf nodes are labeled with unique identifiers that are variables or constants; internal nodes are labeled with operation symbols; each node may have a list of labels; arcs represent relationships between operations and operand; internal nodes represent computed values hold by the identifiers of these nodes.

To construct a DAG of a basic block, we use the algorithm introduced in [8]. An example of constructing DAG is illustrated in Fig. 7.

- Local optimization by removing common sub-expressions

When a sub-expression is contained in other expressions, we need to compute its values and replace the result in every expression that contains it. This procedure can improve performance, reduce size and power. Table 3 shows...
the technique for removing common sub-expressions. After constructing a DAG of a basic block, optimal operations are performed on the DAG by removing redundant identifiers on the label list.

4.3 Experiment and Evaluation

In experiment, we implemented a program that analyzes and replaces equivalent expressions to improve the optimization technique based on common subexpression elimination (CSE). In this experiment, we test on five different programs. The experimental process is shown in Fig. 9. Versions of benchmarks are described in Table 6. Compilations in GCC with CSE options are shown in Fig. 11. We use the command `time` on operating system Ubuntu to measure CPU execution time of each program, repeat 10 times for each one and calculate the average execution time. Figure 8 shows a comparison of the origin source code and the decompiled code of an experimented program. As illustrated in Table 4 and Fig. 10, the average of time reduction is 9.56%. Moreover, Table 5 and Fig. 12 also show that using GCC associated with our improvement, the average of time reduction is 7.716%.

Table 4  Summary of experimental results.

| Benchmarks   | P1 | P2 | P3 | P4 | P1' | P1'' | P2' | P2'' | P3' | P3'' | P4' | P4'' |
|--------------|----|----|----|----|-----|------|-----|------|-----|------|-----|------|
| sumNnumber   | 341| 317| 294| 338| 340 | 273  | 325 | 267  | 299 | 261  | 341 | 275  |
| (ms) (%)     |    |    |    |    |     |      |     |      |     |      |     |      |
| Fibonacci    | 586| 491| 487| 503| 493 | 495  | 489 | 492  | 483 | 499  | 492 | 492  |
| sumPrimes    | 150| 142| 139| 146| 145 | 139  | 143 | 137  | 140 | 135  | 147 | 139  |
| FFT (Fast Fourier Transform) | 892| 859| 835| 871| 873 | 845  | 863 | 823  | 848 | 803  | 869 | 847  |
| RSA Encryption | 1228| 1195| 1187| 1205| 1201| 1072 | 1194 | 1068 | 1193 | 1083 | 1208 | 1087 |
| Average      | 59 | 9.56|

Table 5  Comparison of execution time when compiling by GCC and GCC associated our improvement.

| Benchmarks   | Case -o0 | Case -o2 | Case -o3 | Case CSE |
|--------------|-----------|-----------|-----------|-----------|
| sumNnumber   | 67        | 58        | 38        | 66        |
| (P1' - P1'') | 19.71     | 17.85     | 12.71     | 19.35     |
| Fibonacci    | 16        | 6         | 9         | 7         |
| (P2' - P2'') | 1.988     | 1.212     | 1.829     | 1.403     |
| sumPrimes    | 6         | 6         | 5         | 8         |
| (P3' - P3'') | 4.138     | 4.196     | 3.571     | 5.442     |
| FFT          | 28        | 40        | 45        | 22        |
| (P4' - P4'') | 3.267     | 4.635     | 5.397     | 2.532     |
| RSA Encryption | 129    | 109        | 108        | 119        |
| (Average)    | 48        | 43.8       | 41        | 44.4       |

Table 6  Description of benchmark versions.

| Versions   | Description                                                                 |
|------------|-----------------------------------------------------------------------------|
| P1         | Is compiled from origin C source code with GCC without optimization (-o0)   |
| P2         | Is compiled from origin C source code with GCC with -o2 optimization option |
| P3         | Is compiled from origin C source code with GCC with -o3 optimization option |
| P4         | Is compiled from origin C source code with GCC with optimization options CSE|
| P1'        | Is decompiled from decompilation code of P1 by GCC with CSE optimization options without my improvement |
| P1''       | Is decompiled from decompilation code of P1 by GCC with CSE optimization options associated my improvement |
| P2'        | Is decompiled from decompilation code of P2 by GCC with -o2 optimization options without my improvement |
| P2''       | Is decompiled from decompilation code of P2 by GCC with -o2 optimization options associated my improvement |
| P3'        | Is decompiled from decompilation code of P3 by GCC with -o3 optimization options without my improvement |
| P3''       | Is decompiled from decompilation code of P3 by GCC with -o3 optimization options associated my improvement |
| P4'        | Is decompiled from decompilation code of P4 by GCC with CSE optimization options without my improvement |
| P4''       | Is decompiled from decompilation code of P4 by GCC with CSE optimization options associated my improvement |
Fig. 12 The chart of execution time when compiling by GCC and GCC associated our improvement.

5. Conclusion and Future Work

In this paper, we have proposed and developed a new approach to the embedded software optimization problem based on reverse engineering. Theoretically, we constructed an overall model for the optimization problem of embedded software in both forward and reverse engineering. Along with the overall model, we have developed two optimization methods based on reverse engineering such as optimizing the power consumption of assembly programs based on instruction scheduling, and optimizing performance of high-level source code based on replacement of equivalent expressions. Experimentally, we have implemented the program finding the topological string having the lowest power consumption based on the genetic algorithm and implemented the program replacing equivalent expressions to optimize the performance of embedded software. The researches in this paper can also be used as a foundation to do further researches such as the optimization of embedded software in the design based on reverse engineering, improved optimization options in GCC and research other optimization issues in the overall model. Semilar researches for other CPU architectures of embedded systems are also our future work.

Acknowledgments

This research is partly supported by a VNU scientific project # QGTD.12.20 for 2013-2014.

References

[1] A. Fog, “Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms,” Technical Report, http://www.agner.org/optimize/optimizing_cpp.pdf, Technical University of Denmark, 2014.
[2] C. Erbas, S. Cerav-Erbas, and A.D. Pimentel, “Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design,” Trans. Evol. Comp., vol.10, no.3, pp.358–374, 2006.
[3] C.F. Goss, “Machine Code Optimization — Improving Executable Object Code,” Ph.D. dissertation, Technical Report #246, Courant Institute of Mathematical Sciences, New York University, June 1986, revised Aug. 22, 2013.
[4] C. Galuzzi and K. Bertels, “The instruction-set extension problem: A survey,” ACM Trans. Reconfigurable Technol. Syst., pp.1–28, 2011.
[5] C.L. Su, C.Y. Tsui, and A.M. Despain, “Low power architecture design and compilation techniques for high-performance processors,” Technical Report ACAL-TR-94-01, University of Southern California, ACAL, Feb. 1994.
[6] E.J. Chikofsky and J.H. Cross, “Reverse engineering and design recovery: A taxonomy,” IEEE Softw., vol.7, no.1, pp.13–17, 1990.
[7] H. Falk and P. Marwedel, Source Code Optimization Techniques for Data Flow Dominated Embedded Software, pp.36–45, Springer, 2012.
[8] H. Lee, J. Kim, S. Hong, and S. Lee, “Task scheduling using a block dependency DAG for block-oriented sparse cholesky factorization,” Proc. 14th ACM Symposium on Applied Computing, pp.641–648, 2000.
[9] J. Dean, D. Grove, and C. Chambers, Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis, pp.77–101, Springer-Verlag, 1995.
[10] M. Bajec and D. Vavpoti, “A framework and tool-support for reengineering software development methods,” Informatica, vol.19, no.3, pp.321–344, 2008.
[11] M. Harman and L. Tratt, “Pareto optimal search based refactoring at the design level,” in GECCO, ed. H. Lipson, pp.1106–1113, ACM, 2007.
[12] M.T.C. Lee, V. Tiwari, S. Malik, and M. Fujita, “Power analysis and minimization techniques for embedded DSP software,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.5, no.1, pp.122–135, March 1997.
[13] P. Dongi, “Force-directed instruction scheduling for low power,” theses and Dissertations. Paper 1357, 2003.
[14] S. Pinnepalli, J. Hong, J. Ramanujam, and D.L. Carver, “Code size optimization for embedded processors using commutative transformations,” 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2007, pp.409–416, Daegu, Korea, 2007.
[15] S. Valenti, Successful Software Reengineering, IGI Publishing, Hershey, PA, USA, 2002.
[16] C. Moon, J. Kim, G. Choi, and Y. Seo, “An efficient genetic algorithm for the traveling salesman problem with precedence constraints,” European Journal of Operational Research, vol.140, no.3, pp.606–617, 2002.
Nguyen Ngoc Binh received the B.S. degree in Applied Mathematics from Kishinev University, Moldova (in Former USSR) in 1981, the MSc degree in Information and Computer Sciences from Toyohashi University of Technology, Japan in 1995 and the PhD in Information and Computer Sciences from Osaka University, Japan in 1998. Dr. Binh also received the Honorary Doctor from Toyohashi University of Technology, Japan in 2011. His research interests include SE, KDD, EDA, Embedded Systems, Embedded SE. Dr. Binh now works in the VNU University of Engineering and Technology and the International Francophone Institute, Hanoi. He is a member of societies: IEEE and Computer Society; Institute of Electronics, Information and Communication Engineers (IEICE) of Japan. He is the President of the Radio and Electronics association of Vietnam (REV), the Chairman of IEICE Vietnam Section, and the Chairman of Asia Professional Education Network (APEN), Vietnam Chapter. He is also an International Adviser to the President of the National Institute of Information and Communication Technology (NICT), Japan.

Pham Van Huong is a Ph.D. student in Information Technology at the VNU University of Engineering and Technology (VNU-UET), Hanoi since 2009. He received B.S. degree in Information Technology from the VNU-UET in 2006 and the MSc degree in Information Technology from the same university in 2008. He is now a lecturer in the Academy of Cryptography Techniques of Vietnam. His research interests include embedded software, embedded systems, information security, ubiquitous computing, cloud computing, image recognition, data mining and knowledge engineering.

Bui Ngoc Hai is a researcher at University of Engineering and Technology (VNU - UET), Hanoi. He received B.S. degree and Master’s degree in Information Technology from UET in 2010 and 2014, respectively. His research interests are embedded systems, embedded software, and optimization problems in EDA.