Reducing Engergy Consumption in Programs Using Cohesion Technique

Nattachart Ia-Manee and Peraphon Sophatsathit

***Abstract*—The objective of this work is to reduce energy consumption of source programs written in C. The underlying technique employs code transformation which focuses on cohesion. Four classes of transformations will be considered: function, loop optimization, control structure, and operator. Code transformation is evaluated by effectiveness, efficiency, space complexity, number of instructions executed, number of pages, size of memory page allocated, and energy consumption. The results suggest that different cohesion level will affect the energy consumption. Moreover, different types of source code yield different energy consumptions based on cohesion measures.**

***Index Terms*—Cohesion, energy consumption, code transformation.**

I. Introduction

Global warming is the biggest and most serious problem faced by us in this century. Climate change is happening and its effects are real. If we do not attempt to stop global warming, it will be too late to save our planet.

One of the culprits that attributes to the above problem is energy consumed by the use of computerization. The underlying technology rests heavily on hardware and software. The proliferation of software development has opened the horizon of computer programs to assist a user in various forms of communications such as chat, e-mail, WWW, and the Internet, etc. However, these programs still exhibit high power consumption owing to a number of short falls, ranging from poor program design, inefficient algorithms, to bad code. One research area to solve the above problems focuses on source code transformation. The inherent difficulty lies in code comprehension and complexity which render the transformation process hard to reduce energy consumption and maintenance.

The objective of this research aims to reduce energy consumption of computer programs. Our approach exploits program related issues such as memory optimization, instruction scheduling and execution, and code rearrangement, etc. We envision that the impacts of source code transformations [1] on software and energy consumption [2] will be a worthwhile undertaking.

The primary principle of our approach to source code transformation deals with design. Cohesion is a measure of how various program components, namely, input/output, variables and their related structure, are strongly-related and focused on the various responsibilities of a program module. The higher the cohesion level, the tighter the components are knitted. In a highly-cohesive system, code readability and the likelihood of reuse is increased, while complexity is kept manageable. Such advantages benefit easy maintenance, code reuse, and most important of all, less energy consumption by the program. When source code is transformed by resorting to higher cohesion, complexity, number of instructions executed, number of pages allocated, and size of memory pages allocated are lower. In this work, we employ SimpleScalar Simulator and Wattch Simulator to measure program energy consumption.

The rest of the paper is organized as follows. Section II briefly describes what cohesion is. Section III elucidates on Power Simulator tools. The proposed method is described in Section IV and the experimental results are in Section V. Section VI concludes the paper with some final thoughts.

II. Cohesion

Cohesion [3] is a measure of how strongly-related is the functionality expressed by the source code of a software module. Methods of measuring cohesion vary from qualitative measures classifying the source text being analyzed to quantitative measures which examine textual characteristics of the source code to arrive at a numerical cohesion score. Cohesion is an ordinal type of measurement and is usually expressed as "high cohesion" or "low cohesion." Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability whereas low cohesion is associated with undesirable traits such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand.

High cohesion [4], [5] emphasizes on how a single module is responsible to the underlying functionality. As applied to C code, if a module that serves the given function tends to be similar in many aspects, the function is said to have high cohesion. In a highly cohesive programming system, high cohesion also attributes to code readability and the likelihood of reuse, while complexity is kept manageable. Nevertheless, some disadvantages of low cohesion persist.

1) Increased difficulty in understanding the program modules.
2) Increased difficulty in maintaining a system, because logical changes in the domain affect multiple modules, and because changes in one module require changes in related modules.
3) Increased difficulty in reusing a module because most

Manuscript received November 27, 2012; revised January 28, 2013.

The authors are with the Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University, Bangkok 10330 Thailand (e-mail: kuad43@gmail.com, peraphon.s@chula.ac.th).

applications won't need the random set of operations provided by a module.

The design level cohesion measures [6]-[8], in order of the worst to the best type, are as follows:

A. Coincidental Cohesion (Worst)

Two outputs of a module have neither dependence relationship with each other, nor dependence on a common input.

B. Conditional Cohesion

Two outputs are flag controlled dependent on a common input.

C. Iterative Cohesion

Two outputs are loop controlled dependent on a common input.

D. Communicational Cohesion

Two outputs are dependent on a common input. An input is used to compute both outputs. But it is used as neither a condition flag to select one of the two outputs nor a loop invariant to compute both outputs.

E. Sequential Cohesion

One output is dependent on the other output.

F. Functional Cohesion

There is only one output in a module.

III. Power Simulators

We will explain the two major components of the simulator, namely, SimpleScalar and Wattch below.

A. Simple Scalar

SimpleScalar [9], [10] is a virtual CPU evaluation tool in cycle level on Linux based platform. 'Virtual' means it does not evaluate the actual processor conducts, but emulates a specific processor by C code. SimpleScalar compiles a given piece of C code with emulated CPU and evaluates the performance by analyzing program execute time. The tool set includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations. All of the simulators distributed with the current release of SimpleScalar can run programs from the above instruction sets. Complex instruction set emulation can be implemented with or without microcode, making the SimpleScalar tools particularly useful for modeling CISC instruction sets.

SimpleScalar is a cycle-accurate architectural level processor simulator. It is distributed free-of-charge to academic non-commercial users, with all source code, making it possible to relatively easily extend the simulator. Ever since SimpleScalar was released, it has become a popular toolset as it included several simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supported non-blocking caches, speculative execution and state-of-the-art branch prediction. SimpleScalar cannot simulate a whole system, i.e., it can only simulate applications, and does not produce power consumption of the whole system as a result of the simulation.

Some inclusion of SimpleScalar power analysis tools such as simpower and wattch are furnished in the form of plugin software. It models physical power comsumption using only numerical expression to measure the gap between actual power consumption and the calculated one. In this research, we use SimpleScalar tool to generate object code.

B. Wattch

Wattch [11], [12] is a simulator that estimates processor power consumption at the architectural level, developed at Princeton University, and is one of the simulators that are based on SimpleScalar. SimpleScalar is used as the cycle level performance simulator that keeps track of which units are accessed per cycle and records the total energy consumed for an application. Wattch uses a modified version of SimpleScalar's sim-outorder, which is extended with an additional number of pipeline stages so that it will be more in line with current microprocessors. Sim-outorder of wattch simulator reports detail on power usage in watts.

IV. Research Methodology

The proposed methodology employs code transformations [13] to reduce the energy consumption at program level. The original C source code is firstly compiled by gcc [14]. The compiled code is then simulated using SimpleScalar and wattch.

The next step applies code transformation to the same C source code and, following the same previous steps, the simulation results, energy, and power consumption are collected. Finally, the processor energy and the system energy are compared to identify the effectiveness of the transformation under analysis. The flow of this code transformation analysis is illustrated in Fig. 1

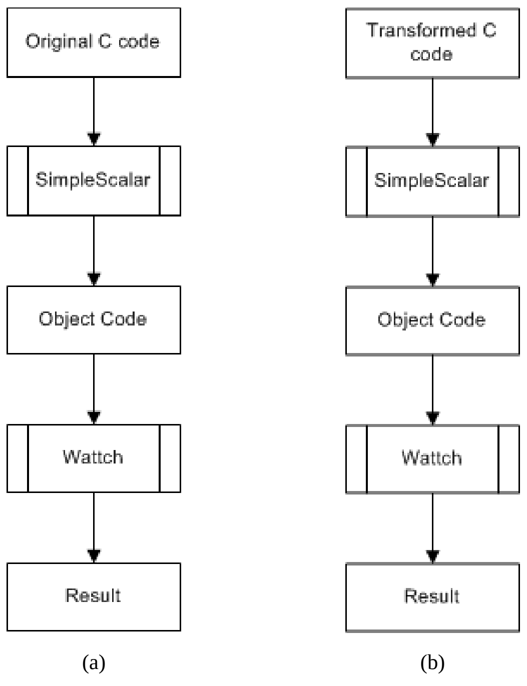

Fig. 1. Analysis flow of original (a) and transformed code (b).

The simulation is performed using a desktop PC having the following specifications: OS Ubuntu 10.4, CPU Intel Core 2

Duo Processor 2 GHz, 4 GB memory.

The transformation methods [15], [16] have been partitioned in four classes, each focusing on a specific code aspect:

1) Loop Transformation
2) Data Structure Transformation
3) Subroutine Transformation
4) Control Structure Transformation

A. Loop Transformation

This class includes transformations modifying either the loop body or the control structure of the loop. The proposed transformation produces positive effects in term of reduction of the number of Instruction Cache (I-cache) and Data Cache (D-cache) misses.

The basic idea is to reduce the size of the loop body in order to decrease the number of the I-cache (Instruction cache) misses. In particular, the transformed codes are distributed in disjoint loops to enable the storing of a complete loop in the cache, preventing to access the upper memory levels.

Loop Transformation is effective with I-cache when a loop body is larger than the cache or than a given number of cache blocks and/or the cache is unified.

D-cache could probably occur when the original loop presents expressions with non-interacting arrays so that different arrays can be distributed on disjoined loop bodies.

B. Data Structure Transformation

This type of transformation either modifies the data structure included in the source code or introduces new data structure or modifies the access mode and the access paths. This transformation aims to maximize the use of register to reduce memory and cache accesses.

Array Declaration Sorting is to modify the local array declaration ordering so that the arrays more frequently accessed are placed on top of the stack.

Array Scope Modification converts local arrays into global arrays to store them within data memory rather than on the stack.

C. Subroutine Transformation

This class of transformations includes the set of source code manipulations operating at subroutine level, typically not considered by compilers, analyzing whether or not it is convenient to modify the subroutine interface.

Compilers usually produce object code by queueing the subroutines depending on the source code structure. Subroutine Queueing Reordering sorts the subroutine declarations according to the subroutine call sequence in order to reduce the I-cache misses.

Substitution of a variable passed as an address with a local variable replaces a routine argument passed as an address with a local copy of a variable. This transformation drives the compiler to use registers, minimizing the energy necessary to access such data.

D. Control Structure Transformation

This class gathers source code transformations optimizing either specific operations or control structures.

Conditional Expression Reordering analyzes a complex conditional expression by rearranging the sub-expressions set in order to save energy by exploiting implicit shortcuts operations. The proposed transformation reassembles the sub-expressions by following two sub-conditions being reordered, placing the sub-condition whose probability to be true is higher.

Function Call Preprocessing associates with a specific function a proper set of macros that will substitute a function call with either an equivalent but low energy function call or a specific result. The transformation skips a function call, or reduces its impact, when its actual parameters allow to directly identifying either the returned value or another equivalent function.

V. Experimental Results

Two samples C source code were taken from [4] for the experiment that served as a standard benchmark. The results of the simulation are collections of the following factors: Clock Cycle, No. of Instructions Executed, Avg. Clock Power, Avg. Total Power, I-Cache Miss, and D-Cache Miss.

Clock Cycle refers to the total number of the processor cycle of the current simulation.

No. of Instructions Executed refers to the number of processor instructions being executed.

Avg. Clock Power refers to the average power in milliwatt (mW.) that is consumed by the processor.

Avg. Total Power refers to the average power in milliwatt (mW.) of overall process.

The Instruction Cache Miss (I-Cache Miss) refers to a cache read miss from an instruction cache.

The Data Cache Miss (D-Cache Miss) refers to a cache read miss from a data cache.

Fig. 2 shows the original C code (having conditional cohesion) in comparison with the transformed C code (having functional cohesion).

Table I summarizes the simulation statistics based on simulation run of code listing 1.

TABLE I: Results of Simulation of Code Listing 1

Parameter	Original	Transformed	%
Clock Cycle	11,034.0	10,878.0	-1.41
No. of Instr Executed	8368.0	8354.0	-0.17
Avg. Clock Power (mW)	28.8	28.4	-1.39
Avg. Total Power (mW)	79.6	78.5	-1.38
I-Cache Miss	335.0	333.0	-0.60
D-Cache Miss	436.0	436.0	-0.00

Table II summarizes the simulation statistics based on simulation run of code listing 2.

TABLE II: Results of Simulation of Code Listing 2

Parameter	Original	Transformed	%
Clock Cycle	11,762.0	11,589.0	-1.47
No. of Instr Executed	6690.0	6782.0	+1.38
Avg. Clock Power (mW)	30.7	30.5	-0.65
Avg. Total Power (mW)	84.9	84.3	-0.71
I-Cache Miss	343.0	339.0	-0.71
D-Cache Miss	426.0	405.0	-4.93

Original C code

```
void main()
{
  int n1 = 100;
  int n2 = 200;
  int flag = 1;
  int arr1[100];
  int arr2[200];
  int sum1;
  int sum2;

  Sum1_or_Sum2(n1, n2, flag, arr1,
arr2, &sum1, &sum2);
}
void Sum1_or_Sum2(
  int n1, n2;
  int flag,
  int arr1[], int arr2[],
  int *sum1,  int *sum2)
{
  int i;

  *sum1 = 0;
  *sum2 = 0;
  if (flag == 1)
    for (i = 0; i < n1; i++)
      *sum1 =  *sum1 + arr1[i];
    else
      for (i = 0; i < n2; i++)
        *sum2 =  *sum2 + arr2[i];
}
```

Transformed C code

```
void main()
{
  int n1 = 100;
  int n2 = 200;
  int flag = 1;
  int arr1[100];
  int arr2[200];
  int sum1;
  int sum2;

  if (flag == 1)
    Sum(n1, arr1, &sum1);
  else
    Sum(n2, arr2, &sum2);
}

void Sum(
  int n,
  int arr[],
  int *sum)
{
  int i;

  *sum = 0;
  for (i = 0; i < n; i++)
    *sum =  *sum + arr[i];
}
```

Fig. 2. Original VS transformed code listing 1.

Fig. 3 shows the original C code (having communicational cohesion) in comparison with the transformed C code (having functional cohesion).

Original C code

```
void main()
{
  int n = 100;
  int arr[100];
  int sum;
  int prod;
  float avg;

  Sum_and_Prod(n, arr, &sum,
&prod, &avg);
}

void Sum_and_Prod(
  int n,
  int arr[],
  int *sum,
  int *prod,
  float *avg)
{
  int i;

  *sum = 0;
  *prod = 1;
  for (i = 0; i < n; i++)
  {
    *sum =  *sum + arr[i];
    *prod =  *prod * arr[i];
  }
  *avg =  *sum / n;
}
```

Transformed C code

```
void main()
{
  int n = 100;
  int arr[100];
  int sum;
  int prod;
  float avg;

  sum = Sum(n, arr);
  prod = Prod(n, arr);
  avg = sum / n;
}

void Sum(int n, int arr[ ])
{
  int i;
  int sum = 0;

  sum = 0;
  for (i = 0; i < n; i++)
  {
    sum = sum + arr[i];
  }
  return sum;
}

void Prod(int n, int arr[])
{
  int i;
  int prod;

  prod = 1;
  for (i = 0; i < n; i++)
  {
    prod = prod * arr[i];
  }
  return prod;
}
```

Fig. 3. Original VS transformed code listing 2.

Other results of coincidental, iterative, and sequential cohesion, in comparison with functional cohesion yield similar outcomes.

VI. Conclusion

In this paper, we propose a simple technique to reduce energy consumption of a computer program using design cohesion measure. This method is to transform a given piece of C code to increase tighter cohesion level. Both samples source code were simulated on the designated simulation tool environment. The results showed that the original source code consumed more power than the transformed code. This was because we could reduce both I-cache and D-cache misses, clock cycle, and power consumed, with an exception of increase in number of instructions executed in code listing 2. Although we were able to reduce the energy consumption upon improvement with cohesion, the amount of reduced energy was not significantly noticeable. However, the proposed approach exhibited promising opportunities in larger programs. The higher the level of cohesion is attained, the more power is conserved by a program. As such, good design level code translates into less energy consumption by subsequent programming applications.

The windfall benefits from the proposed code transformation technique are program readability, better design, lower complexity, and more code reuse.

References

[1] C. Brandolese, W. Fornaciari, and F. Salice, "Code-level transformations for software power optimization," *CEFRIEL*, Tech. Rep. N. RT-02-004, 2002.

[2] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "The impact of source code transformations on software power and energy consumption," *J. Circuits Systems and Computers*, vol. 11, no. 5, pp. 477-502, May 2002.

[3] E. B. Allen and T. M. Khoshgoftaar, "Measuring coupling and cohesion: An information-theory approach," in *Proc. Sixth International Software Metrics Symposium*, pp. 119-127, November 1999.

[4] J. M. Bieman and B.-K. Kang, "Measuring design-level cohesion," in *Proc. IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 111-124, Feb. 1998.

[5] G. Gui and P. D. Scott, "Coupling and cohesion measures for evaluation of component reusability," in *Proc. the 2006 International Workshop on Mining Software Repositories*, May 2006.

[6] B. D. Bois, S. Demeyer, and J. Verelst, "Refactoring-Improving Coupling and Cohesion of Existing Code," in *Proc. 11th Working Conf. Reverse Eng.*, pp. 144-151, November 2004.

[7] J. M. Bieman and L. M. Ott, "Measuring functional cohesion," in *Proc. IEEE Transl. Software Engineering*, vol. 20, no. 8, pp. 644-657, August 1994.

[8] T. M. Meyers and D. Binkley, "An empirical study of slice-based cohesion and coupling metrics," *ACM Transl.*, vol. 17, no. 1, December 2007.

[9] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," in *Proc. IEEE*, Feb. 2002, pp. 59-67.

[10] D. Burger and T. Austin, "The simplescalar tool set, version 2.0," Technical report, Computer Sciences Department, University of Wisconsin, Jun 1997.

[11] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level poweranalysis and optimizations," in *Proc. 27th Annual International Symposium on Computer Architecture*, Jun. 2000.

[12] J. Chen, M. Dubois, and P. Stenstrom, "Integrating Complete-System and User-level Performance/ Power Simulators: The Sim Wattch Approach," in *Proc. IEEE*, IEEE Press, Jul. 2007, vol. 27, no. 4, pp. 34-48.

[13] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 1st ed., Addison Wesley, 1999.

[14] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: an overview," in *Proc. IEEE Int. Symp. Low Power Electronics,* Digest of Technical Papers, 1994, pp. 38-39.

[15] H. Falk and P. Marwedel, "Control flow driven splitting of loop nests at the source code level," in *Proc. Europe Conference and Exhibition*, Mar. 2007, pp. 410-415

[16] G. Cai and C. H. Lim, "Architectural Level Power/Performance Optimization and Dynamic Power Estimation," in *Proc. Cool Chips Tutorial, in Conjunction with MICRO 32*, Nov. 1999, pp. 90-113.

Nattachar Ia-Manee is a Lieutenant of The Royal Thai Naval pursuing his Masters in Computer Science. His research interest is in Software Engineering.

Peraphon Sophatsathit received his Bachelor Degree in Industrial Engineering from Chulalongkorn University, Masters in Industrial Engineering and Computer Science from University of Texas at Arlington USA, and Ph.D. in Computer Science from Arizona State University USA.He is currently an associate professor in Computer Science at the Department of Mathematics and Computer Science, Faculty of Science, Chulalongkonrn University, Thailand. His research interests are Software Engineering and Operating Systems.