Code-level Optimization for Program Energy Consumption

Cuijiao Fu, Depei Qian, Tianming Huang, Zhongzhi Luan
School of Computer Science and Engineering
Beihang University
Beijing, China
e-mail: {fucuijiao, depeiq, tianminghuang, luan.zhongzhi}@buaa.edu.cn

***Abstract*—A lot of time is spent on Central Processing Unit (CPU) waiting for memory accesses to complete during the program is being executed, which would be longer because of data structure choice, lack of design for performance, and ineffective compiler optimization. Longer execution time means more energy consumption. To save energy, avoiding unnecessary memory accesses operations is desirable. In this paper, we optimize program energy consumption by detecting and modifying the dead write, which is a common inefficient memory access. Our analysis of the Standard Performance Evaluation Corporation (SPEC) CPU2006 benchmarks shows that the reduction of the program running energy consumption is significant after the dead write in the code was modified. For example, the SPEC CPU2006 gcc benchmark had reduced energy consumption by up to 26.7% in some inputs and 13.5% on average. We think this energy optimization approach has tremendous benefits for the developer to develop more energy-efficient software.**

Keywords-Energy Optimization; Ineffective Memory Access; Energy-efficient Software

I. INTRODUCTION

As power and energy consumption are becoming one of the key challenges in the system and software design, several researchers have focused on the energy efficiency of hardware and embedded systems [1][2], the role of application software in Information Technology (IT) energy consumption still needs investigation. On modern computer architectures, memory accesses are costly. For many programs, exposed memory latency accounts for a significant fraction of execution time. Unnecessary memory accesses, whether cache hits or misses, which lead to poor resource utilization and have a high energy cost as well [3]. In the era where processor to memory gap is widening [4][5], gratuitous accesses to memory are a cause of inefficiency, wasting so much energy, especially in large data centers or High Performance Computer (HPC) running complex scientific calculations. Therefore, the optimization of program memory access can bring about significant effects on energy consumption reduction.

Prior work about on the optimization of energy consumption in computer systems mostly focused on the scheduling of system resources, such as the research and attempt of load balancing in clusters [6]. Due to the complexity of the computer system when the program is running and the uneven level of the developer, it is difficult to modify the program code for energy optimization. Our analysis found that there are a lot of redundant memory accesses in common programs, and the energy waste they cause cannot be eliminated by resource allocation and scheduling. It is very necessary to analyze and optimize the source code of the program.

Fortunately, we found it conveniently to analyze and record the memory accesses during program execution by using Pin [7]. Pin is a dynamic binary instrumentation tool powered by Intel, which provides a rich set of high-level Application Programming Interfaces (APIs) to instrument a program with analysis routines at different granularities including module, function, trace, basic block and instruction. With this tool, we can instrument every read and write instruction, which helps us find out the redundant memory access clips in the program source code.

In this paper, we focused on the impact of dead write on program energy consumption. A 'dead write' occurs when there are two successive writes to a memory location without an intervening read. Our work mainly focuses on the following three aspects. 1) Locating dead writes exactly to the line in the source code of programs. 2) Analyzing and modifying the source code fragments found in 1). 3) Measuring and comparing energy consumption of programs before and after modification of dead writes.

The rest of the paper is organized as follows. Section 2 presents detailed decision process of dead write and sketches the methodology for positioning dead writes in programs' source lines. Section 3 analyses two codes to explore the causes of dead writes and the energy optimization benefits of dead write elimination. Finally, conclusions are drawn in Section 4.

II. METHODOLOGY

Chabbi et al. [8] described a type of redundant memory access and named it *dead write*, which means *two writes to the same memory location without an intervening read operation make the first write to that memory location dead.* This definition gives us a way to reduce energy consumption of programs by optimizing programs' memory access codes.

 ISBN: 978-1-61208-709-2

In the following subsections, we first describe in detail the conditions and scenarios of the formation of dead write. Then, we introduce our methodology to find out the dead writes in programs' source codes.

A. *Dead Write*

For every used memory address, building a state machine based on the access instructions. The state machine state is changed to initial mark **V** (Virgin) for each used memory address, indicating that no access operation is performed, and when an access operation is performed, the state is set to **R** (Read) according to the type of operation. Or **W** (Write). According to access to the same address, the state machine implements state transitions. The following two cases will be judged to be dead write:

1) A state transition from **W** to **W** corresponds to a dead write.
2) At the end of the program, the memory address in the **W** state, meaning that the program did not read it until the end of the operation.

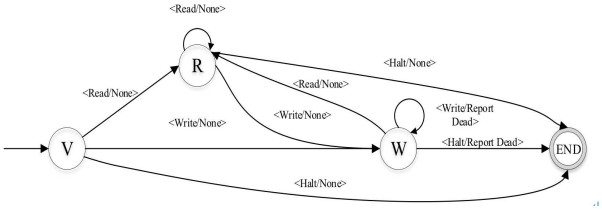

Figure 1 State transition of dead write diagram

A halt instruction transitions the automaton to the terminating state. The *Report Dead* behavior indicates that an invalid write is detected and can be reported.

Because the state machine records every memory access operation from the beginning to the end of the program, false positive or false negative situations can be avoided, and the judgment result is reliable.

B. *Finding dead writes in source lines*

Developing a tool based on CCTLib, a library uses Pin to track each program instruction, and builds dynamic Calling Context Tree (CCT) [9] with the information of memory access instructions. Each interior node in our CCT represents a function invocation; and each leaf node represents a write instruction. After the program is executed, each dead write will be presented to the user as a pair of CCT branches.

Specifically implemented on our tool is the use of shadow memory [10] on the Linux platform to save the state of each memory location. In order to trace dead writes, each memory access instruction to address M is updated according to the state machine of Figure 1 with the state STATE (M), while saving pointers to restore its calling context and reporting dead writes when encountered. When the node in the created call tree reaches the state needs to report dead write according to the transition state of the state machine in Figure 1, our tool will record this context and output all contexts at the end of the entire analysis. By adding the *-g* option to the gcc compiler when compiling the program to be analyzed, the debugging information is obtained so that the contexts is mapped to the source codes.

III. OPTIMIZATION FOR DEAD WRITES

In this section, we discuss the optimal solution for dead write that has been found in programs. There are many causes of dead writing. For example, Figure 2 is the simplest scenario because of the repeated initialization of an array. The Figure 2 shows the function Bar () and function Foo () initializes the array a separately before the function Foo1 () reads it. In the following, we analyze two complex situations of the gcc benchmark in SPEC CPU2006 [11].

```
1   #define N (0xfffff)
2   int a[N]
3   void Foo() {
4     int i;
5     for ( i=0; i<N; i++ )  a[i] = 0;
6   }
7   void Bar() {
8     int i;
9     for ( i=0; i<N; i++ )  a[i] = 0;
10  }
11  void Foo1() {
12    int i;
13    for ( i=0; i<N; i++ )  a[i] = a[i];
14        +1;
15  }
16  int main() {
17    Foo();
18    Bar();
19    Foo1();
20  return 0;
21  }
```

Figure 2 A simple example for dead write

For 403.gcc, after testing each input, it was found that for the input **c-typeck.i**, the dead write is very large, accounting for 73% of the total amount of memory accesses. For gcc with the input **c-typeck.i**, do the following analysis and optimization.

```
1   void  loop_regs_scan (struct  loop * loop, ...)
2   {...
3     last set=(rtx *) xcalloc (-regs>num,
4   sizeof (rtx));
5   /*register used in the loop*/
6   for (each instr in loop) {...
7     if(MATCH(ATTERN (insn))==SET || ...)
8      count_one_set ...(, last_set, ...);
9     ...
10  if(block is end)
11    memset (last_set, 0, regs->num
12    *sizeof(rtx));
13      }...
14  }
```

Figure 3 Dead writes in gcc due to an inappropriate data structure

The code snippet shown in Figure 3 is refined in a frequently-called function named **loop_regs_scan ()** in the file **loop.c**. The function of this part of the code fragment is as follows:

- On line 3, 132KB of space is allocated to the array **last_se**t, with a total of 16937 elements, each element occupying 8KB.
- On Lines 6-14, iterating through each instruction in the incoming parameter loop.
- On line 8-9, if the instruction matches a pattern, the **count_one_set** function is called. The function is to update **last_set** with the last instruction that sets the virtual register.
- On lines 11-12, if the previous module completes, reset the entire **last_set** by calling the **memset ()** in the next loop.

This piece of code will produce a large number of dead writes, because the program spends a lot of time to reset the **last_set** to zero. In the module, only a very small number of elements of the array would be used in one cycle. However, at the beginning of the allocation, the largest array size possible for **last_set** is used. It means there are a large number of elements that were repeatedly reseted and cleared when they have not been accessed. It was found through sampling that in the 99.6% case, only 22 different elements per cycle would be written with a new value. Thus, a simple optimization scheme is: we maintain an array of 22 elements to record the index of the modified element of the **last_set.** Reseting only the elements of the subscript stored in the array when the reset is cleared. Reseting the entire 132KB array if the encounter array is overflow, then call **memset ()** at the end of the period to reset the entire array.

Another dead write context was found in **cselib_init ()**. As shown in Figure 4, the macro **VARRY_ELT_LIST_ INIT ()** allocates an array and initializes to 0. Then, the function **clear_table ()** initializes the array to 0 again, apparently resulting in a dead write. By reading the source code, there is a more lightweight implementation for **clear_table ()**. This implementation does not initialize the array **reg_values**, so this dead write could be eliminated by changing the interface.

```
1   void cselib_init () {
2     ...
3     cselib nregs = max reg num();
4     /*initializ reg_values to 0 */
5     VARRY_ELT_LIST_INIT (reg_values,
6      cselib_nregs, ...);
7      ...
8     clear_table (1);
9   }
10  void clear_table (int clear_all) {
11    /*reset all elements of reg_values to 0 */
12    for (int i = 0; i < cselib_nregs; i++)
13      REG_VALUES (i) = 0;
14  ...
15  }
```

Figure 4 Dead writes in gcc due to excessive reset

IV. EXPERIMENT

In this section, we actually take the readings of the hardware performance counters by sampling them while the program is running. Those readings are the input of the Power Model [12] we had published in 2016. The output of the model is the power of the whole system. Obviously, time-based integration of power is energy consumption.

A. Experiment environment

We used PAPI [13] to get the readings of the hardware performance counters and gcc to compile the programs with option -g before they are analyzed by dead write analysis tool. Detailed hardware configuration of the experiment platform is shown in Table I.

TABLE I. HARDWARE CONFIGURATION

Component	Description($)
CPU	2.93GHz Intel Core i3
Memory	4GB DDR3 1333HZ
Hard Disk	Seagate Barracuda 7200.12
Net	1000Mb/s Ethernet

B. Calculation method

In our prior work [12], we have presented a full system energy consumption model based on performance events, and its accuracy had been verified. We use it in our work this time.

In the model, we calculated full system power as the linear regression of three kinds of readings of the hardware performance counters according to performance Events. As shown in Formula 1. The three kinds of performance Events are **Active Cycles** ({Cycles *in which processor are active*.), **Instruction Retired** (*The instruction* (*micro-operation*) *leaves the "Retirement Unit".)* and **Last-Level Cache (LLC) Misses** (Count *each cache miss condition for references to the last level cache.*).

$$P_{system} = 23.834 + ActiveCycles \times 2.093 + InstructionRetired \times 72.113 + LLC_{Misses} \times 47.675 \quad (1)$$

When the host computer does not run the test program, it also has background programs running, and the components are also consuming power. Therefore, the energy consumption, when the host computer is not running the test program, should be removed to see more obvious contrast. Firstly, reading the host hardware performance counters' value when the test program is not running. Then, using Formula 1 to calculate the long-term power average value $\overline{P}_2$ which is taken as the background power of the host. The energy consumption of this part can be calculated as $\overline{P}_2$ multiple the running time (which is T_{end} - T_{start}). The final energy consumption will be energy caused by P_1 subtract that from $\overline{P}_2$. Therefore, the energy consumption of

the test program can be calculated using Formula 2 since energy is the integral of power over time.

$$E_{result} = \int_{Tstart}^{Tend} P_1 - \overline{P_2} \times (T_{end} - T_{start}) \quad (2)$$

When the test program is running, the three hardware performance counters in Formula 1 are sampled every 5 seconds. The calculated system power is connected to each sampling point using a Bezier curve. Then, the energy consumption is calculated by integrating the time with Formula 2.

C. Result

Since the same benchmark is running on different inputs, the functions in it invoked are different, so optimization tests are performed for different inputs. For some benchmarks, such as **bzip2**, because the program execution time is too short to sample an accurate reading, which are not suitable for energy consumption measurement. According to 403.gcc, it has a long execution time so that we can observe the changes in energy consumption before and after dead write optimization under different inputs. The results are shown in Table II. All the energy consumptions were calculated by using the methods described in previous parts.

TABLE II. CHANGES IN ENERGY CONSUMPTION FOR GCC

Input	Energy consumption (J)		%Reduction
	before	after	
166.i	141.65	128.48	9.3
200.i	207.34	203.2	2
c-typeck.i	182.37	137.69	24.5
cp-decl.i	133.36	115.76	13.2
expr.i	153.13	127.4	16.8
expr2.i	197.48	169.64	14.1
scilab.i	98.46	97.8	0.8
g23.i	254.07	219.26	13.7
s04.i	227.0	166.39	26.7
% Average		13.46	

The average energy consumption is reduced by 13.46%, which has a significant effect. The result shows that finding and the dead writes in the program code can significantly reduce the energy consumption of the programs.

V. CONCLUSIONS

This paper proposes an optimization method for program energy consumption. The method is based on the optimization of dead write, a widely-existing redundant memory access in the source code. Finding out and eliminating the dead writes in programs, which could increase system efficiency and reduce energy consumption. From the experimental results, the effect is significant. Subsequent work should be focused on developing the tools based this paper, which allow more developers to use simple operations to optimize energy consumption of written program code.

ACKNOWLEDGMENT

This research is supported by the National Key R&D Program (Grant No.2017YFB0202202).

REFERENCES

[1] E. Capra, C. Francalanci, and S.A. Slaughter, “Is software green? Application development environments and energy efficiency in open source applications”, Information & Software Technology, vol. 54, no. 1, pp. 60–71, 2012.

[2] I. Manotas, L. Pollock, and J.Clause, “Seeds: a software engineer’s energy-optimization decision support framework”, Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 503–514.

[3] P. Hicks, M. Walnock, and R. M.Owens, “Analysis of power consumption in memory hierarchies”, International Symposium on Low Power Electronics and Design, 1997, pp. 239–242.

[4] B. Jacob, “The memory system: you can’t avoid it, you can’t ignore it, you can’t fake it”, Synthesis Lectures on Computer Architecture, vol.4, no. 1, 2009, pp.1-15.

[5] S. A. Mckee, “Reflections on the memory wall”, in Conference on Computing Frontiers, 2004, p. 162.

[6] R. Azimi, M.Badiei, X. Zhan, N. Li, and S. Reda, “Fast decentralized power capping for server clusters”, in IEEE International Symposium on High Performance Computer Architecture, 2017, pp. 181–192.

[7] C.K.Luk et.al, “Pin: building customized program analysis tools with dynamic instrumentation”, 2005, pp.190–200.

[8] M.Chabbi, and J. Mellor-Crummey, “Deadspy: a tool to pinpoint program inefficiencies”, Proceedings of the Tenth International Symposium on Code Generation and Optimization(CGO’12), pp. 124-134.

[9] M. Chabbi, X. Liu, and J. Mellor-Crummey, “Call paths for pin tools”, IEEE/ACM International Symposium on Code Generation and Optimization, 2014, pp. 76–86.

[10] N. Nethercote and J. Seward, “How to shadow every byte of memory used by a program”, International Conference on Virtual Execution Environments, 2007, pp. 65–74.

[11] J. L. Henning, “Spec cpu2006 benchmark descriptions”, ACM SIGARCH Computer Architecture News, vol. 34, no. 4, pp. 1–17, 2006.

[12] S. Yang, Z. Luan, B. Li, G. Zhang, T. Huang, and D. Qian, “Performance events based full system estimation on application power consumption”, IEEE International Conference on High Performance Computing and Communications, 2017, pp.749–756.

[13] P. J. Mucci, S. Browne, C. Deane, and G. Ho, “Papi: A portable interface to hardware performance counters”, DoD Hpcmp Users Group Conference, 1999, pp.7–10.

 ISBN: 978-1-61208-709-2