Code-level Optimization for Program Energy Consumption Tools

Cuijiao Fu, Depei Qian, Tianming Huang, Zhongzhi Luan
fucuijiao@buaa.edu.cn

School of Computer Science and Engineering
Beihang University

Outline

- Introduction
- Methodolgy
- Optimization for dead writes
- Experiment
- Conclusion
- References

Introduction

- **As power and energy consumption are becoming one of the key challenges in the system and software design, several esearchers have focused on the energy efficiency of hardware and embedded systems**
- **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 HPC running complex scientific calculations**
- **Therefore, the optimization of program memory access can bring about significant effects on energy consumption reduction**

Introduction

- **Duo 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**
- **We 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**
- **We found it conveniently to analyze and record the memory accesses during program execution by using Pin**

Introduction

- **We focused on the impact of dead write on program energy consumption**
- **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**

Methodology

- **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**

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

Dead Write

- **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**

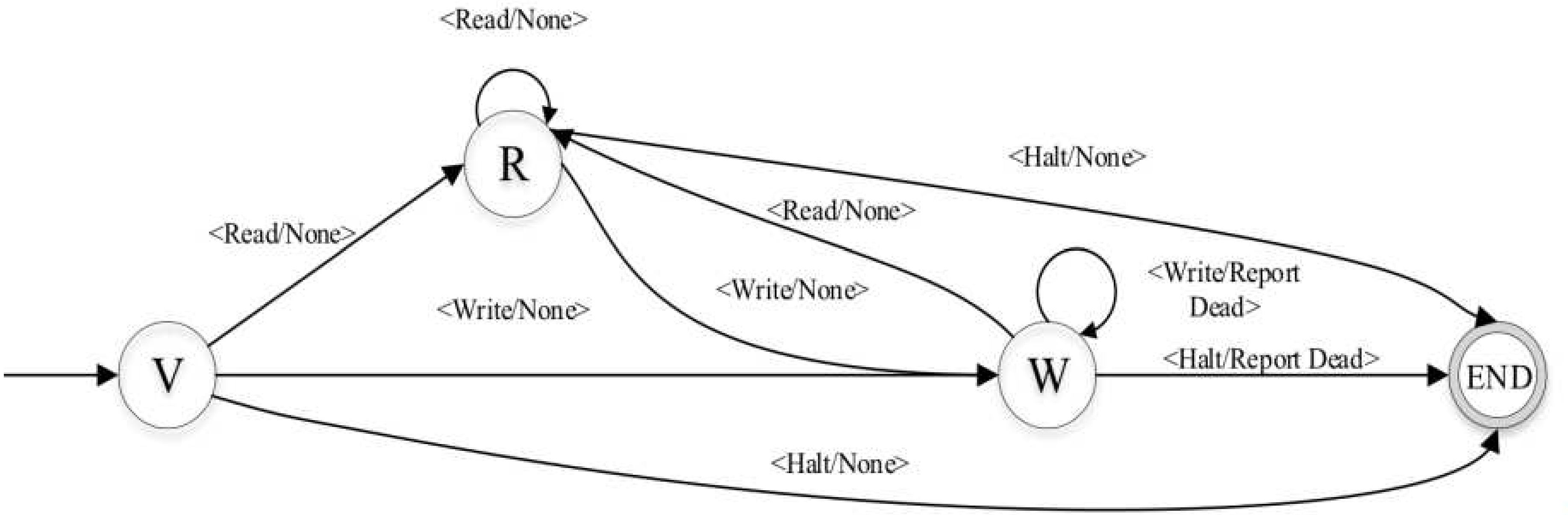

Figure 1 State transition of dead write diagram.

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

Optimization for dead writes

- **There are many causes of dead writing**
  - **For example, Figure 2 is the simplest scenario because of the repeated initialization of an array. in this figure, the function Bar () and function Foo () initializes the array a separately before the function Foo1 () reads it**

```
#define N (0xfffff)
int a[N]
void Foo() {
  int i;
  for ( i=0; i<N; i++ )  a[i] = 0;
}
void Bar() {
   int i;
   for ( i=0; i<N; i++ )  a[i] = 0;
}
void Foo1() {
  int i;
  for ( i=0; i<N; i++ )  a[i] = a[i];
 +1;
}
int main() {
  Foo();
Bar();
  Foo1();
return 0;
}
```

Figure 2.   A simple example for dead write

Optimization for dead writes

- **We analyze two complex situations of the gcc benchmark in SPEC CPU2006 [11]**
  - **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**

```
void  loop_regs_scan (struct  loop * loop,
...)
{...
  last set=(rtx *) xcalloc (-regs>num,
sizeof (rtx));
/*register used in the loop*/
for (each instr in loop) {
...
if(MATCH(ATTERN (insn))==SET || ...)
    count_one_set ...(, last_set, ...);
...
if(block is end)
  memset (last_set, 0, regs->num
  *sizeof(rtx));
}...
}
```

Figure III. Dead writes in gcc due to an inappropriate data structure

Methodolgy

- **It was found through sampling that in the 99.6% case, only 22 different elements per cycle would be written with a new value**
- **The 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.**

Methodolgy

- **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**

```
void cselib_init () {
  ...
  cselib nregs = max reg num();
  /*initializ reg_values to 0 */
  VARRY_ELT_LIST_INIT (reg_values,
   cselib_nregs, ...);
   ...
   clear_table (1);
}
void clear_table (int clear_all) {
  /*reset all elements of reg_values to 0 */
  for (int i = 0; i < cselib_nregs; i++)
    REG_VALUES (i) = 0;
...
}
```

Figure 4. Dead writes in gcc due to excessive reset.

- **This implementation does not initialize the array reg_values, so this dead write could be eliminated by changing the interface**

Experiment

- **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**

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

Calculation method

- **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, Instruction Retired and LLC Misses**

Formula 1

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

- **The energy consumption of the test program can be calculated using Formula 2 since energy is the integral of power over time**
- Formula 2 $$E_{result}=\int_{Tstart}^{Tend}\overline{P_1-P_2}\times(T_{end}-T_{start})$$

Experimental results

- **Result: The average energy consumption is reduced by 13.46%**

TABLE I. 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

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**

Acknowledgment

- This research is supported by the National Key R&D Program (Grant No.2017YFB0202202)
- Thanks to the organizers of the conference and the anonymous reviewers of the paper

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.

References

[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.

The End

Thank You!