2014 International Conference on Computer, Network Security and Communication Engineering (CNSCE 2014)
ISBN: 978-1-60595-167-6

Reducing Energy Consumption in C Programs by Variable Reallocation

Krisada Samrittiyanusorn and Peraphon Sophatsathit
Advanced Virtual and Intelligent Computing (AVIC) Center
Department of Mathematics and Computer Science, Faculty of Science
Chulalongkorn University, Bangkok, 10330, Thailand
krisada.sa@student.chula.ac.th

Keywords: Energy consumption, Variable reallocation, Instruction level, Memory access protection.

Abstract. Energy consumption around the world increases exponentially. One of the causes to blame is electronic devices such as personal computers, embedded devices, and smartphones. To reckon with reducing energy consumption involves efficient hardware and software. This research focuses on the software part, in particular, how to write a program that is energy efficient. The proposed technique is based primarily on local variable reallocation in C programs to exploit the advantages of shared memory and register variable. We analyze the amount of energy consumed at instruction level. Our findings reveal that shared memory is the best choice at the price of memory access protection. The benefits are fewer redundant allocations and memory accesses, thereby less energy will be consumed.

Introduction

Nowadays, energy consumption is one the most concerned issue world-wide. Various electronic devices such as personal computers, smartphones, and embedded devices are the major culprits of high energy sources. Computer programs will play a central role in all relating applications. What follows is the enormity of energy consumed by these devices. There are many ways to reduce the energy consumption on computers attributed by hardware and software. Studies have shown that software is a principal factor on energy consumption in computer systems [1]. Unfortunately, some programmers may only concern about running time or resource utilization of the program and ignore about energy perspective.

A typical computer program in execution stores, retrieves, and processes variables such as local variables, shared memories, and register variables. Heavy use of these variables wastes considerable energy. One remedy is reorganization of the original code to properly allocate variables and parameters, thereby balancing the distribution of energy consumption. This research will address such a compelling issue to demonstrate how the problem can be alleviated.

The organization of this paper is as follows. Section 2 discusses related researches on saving energy consumption of computers. Section 3 describes the proposed approach to appropriate suitable variables for energy saving. In Section 4, a small research tool was built for the experimental purpose to identify the energy consumption at instruction level. Section 5 discusses comparative results obtained from the experiment. Some final thoughts on the trade-off and future work are also given.

Related Work

Saving energy on electronic devices has been the focus of today's green technology. Many research endeavors have been carried out which can be classified into 2 types, namely, hardware and software. Various techniques have been attempted to cope with such problems. A simple, effective, and popular technique is turning off power in wireless card when it is not used [4]. From software standpoint, properly managed of memory allocation and access will help reduce the amount of energy consumption [7]. At a finer grained level, Grochowki and Annavaram [3] analyzed energy per

instruction (EPI) based on Intel processor. Tools for instruction power analysis [5] or energy aware that help developers determine the energy consumed by their programs under development [6] are also available. All these techniques will be further explored in the next section.

Proposed Approach

In C programming language, a local stack is used to store local variables and parameters when a function is called. Access to the variables goes through push/pop operations. Energy consumption occurs during the stack process. This usage is further worsening in repeated calls, whereby power drains are inevitable. The excessive energy consumption of stack use can be reduced by code modification to reallocate these local variables to shared memories and register variables, wherever deemed appropriate. In so doing, repetitive accesses to data will lessen stack process considerably, thereby energy consumption is reduced. By the same token, moving local variables to register variables will also accomplish similar energy savings since data access can be done faster and register variables expend less access effort than stacks.

To explore the operating characteristic of parameter and variable allocation at instruction level, a C program is first compiled into assembly code. Each machine instruction is examined to determine the number of clock cycles used [1], depending on types of instruction. For instance, PUSH takes one clock cycle but consumes 3 latency cycles. Measurement is performed in reciprocal throughput (RT) and latency. RT is the average number of core clock cycles per instruction for a series of independent instructions of the same kind in the same thread [1]. Latency of an instruction is the delay that the instruction generates in a dependency chain. The total clock cycles of every instruction used by the entire program can then be converted to energy consumption [3], which is measured as energy per instruction (EPI). The unit of energy from the average EPI is expressed in nano Joule.

In this study, we will compare the amount of energy consumed by an original local variable allocation with that of shared memory and register variable by arranging the same program code in three respective forms, namely, local, shared memory (global), and register variables. Fig. 1 illustrates such an arrangement.

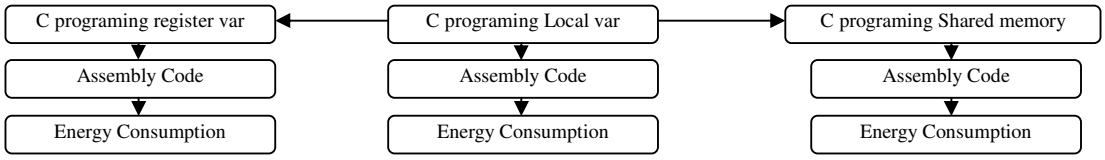

Fig. 1. A comparative energy consumed by local variable, shared memory and register variable.

Two scenarios will be investigated, namely, (1) local variable vs. shared memory or LS, and (2) local variable vs. register variable or LR. A collection of C programs are set up to assist in the analysis. The following case studies will be carried out to exercise both scenarios:

1. Function calls. It is the simplest exercise on parameter allocation, access, and retrieval. Normally, a programmer will use local variables declared in the main function. These variables will subsequently be passed to other functions in the form of parameters. For the first scenario, program modification is done by moving local variables to shared memories, thereby no parameter passing is needed. For the second scenario, the register keyword is simply added to proper local variables.

2. Repeated function calls. The objective is to find code segments that exhibit high energy consumption in a program and behavior of the associated variables/parameters. As such, program improvement can be directed to the right code segments where heavy energy consumptions will be reduced. As a consequence, this case intentionally contrives repeated calls to function for this particular purpose.

3. Function calls to function. This case is intended to investigate the cascading effect of energy consumption consumed by parameter allocation and reference. The complication of such operations, i.e., stack, shared memory, and register variable, at the instruction level are systematically measured and compared.

4. Nested repeated function calls to function. This case culminates all of the above complications to demonstrate as close to actual operation as possible.

Tab.1 illustrates a straightforward comparison of the case studies under both scenarios.

Tab. 1. Example of pseudocode in repeated function calls.

Local variable	Shared memory	Register variable

main() {
var A
for *n* times
function(A)
end for
}
function(para A)
	var A
	main
	for *n* times
	function()
	end for
	}
	function()
		main() {
		register var A
		for *n* times
		function(A)
		end for
		}
		function(para A)

Experimental and Results

We built a tool called KP program to help analyze the test programs. The tool first read an input C program submitted by the user under the above two scenarios. It located local variables and prompted the user to reallocate or alter them to shared memories or register variables. A lookup table was created by the tool to hold all the data selected by the user for shared variable reallocation or register variable alteration. The tool then compiled both original and modified C programs to produce assembly instructions for determining clock cycles and EPI equivalent. The operating environment was hosted by a laptop computer with Intel Core 2 Duo @ 2.00GHz 65 nm, 3 GB RAM running Windows 7. The tool was coded in C# using Microsoft Visual Studio. All test programs were compiled with MinGW which was a ported GNU compiler collection (GCC).

Code analyses are shown in Fig. 2-3. Fig. 2 depicts KP tool running the original program for repeated function calls (case 2). All variables are locally declared. Fig. 3 shows the reallocation from local variables to shared memories. The numbers of instructions to be executed is less than the original version. Alteration of local variables to register variables is straightforward.

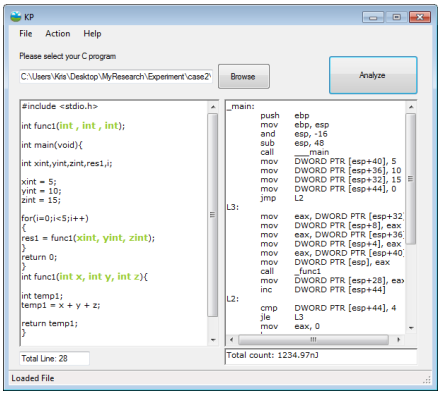

Fig. 2. KP tool running case 2 –local variable .

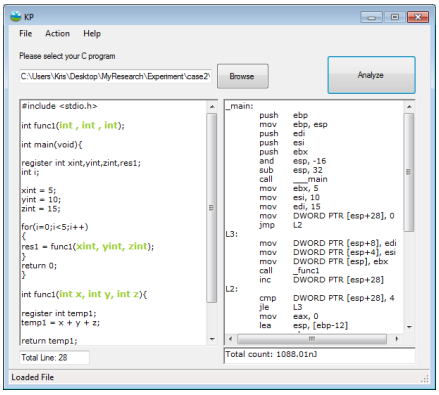

Fig. 3. KP tool running case 2 scenario 1 –shared memory.

The rationale behind each case study was to determine the amount of energy consumed by program instructions under different functions. We began with simple function calls (case 1). The tool analyzed and compared clock cycles used, and the energy consumed by each scenario. For example, the original C program contained 30 instructions that utilized 27.31 clock cycles and 300.41 nJ of energy. Under the first scenario (LS), the number of instructions, clock cycles, and energy consumed were 21, 18.98, and 208.78, respectively. Similarly, statistics of the second scenario (LR) came out to be 35, 28.63, and 314.93, respectively. Obviously, shared memories exhibited a sizable savings (-30.50%), while register variables showed a slightly higher consumption (+4.60%) than that of the local variables. The story was different for repeated function calls (case 2), where shared memories continued to saving energy consumption (-35.60%), and register variables gained on the original local variables (-11.90%). As programs became more complicated, savings on energy consumed were even

more noticeable. The function calls to function (case 3) exhibited such benefits. Shared memory savings went from -30.50% to -31.00%, while the numbers on register variable were down from +4.60% to +2.90%. For nested repeated function calls to function (case 4), the numbers were even more interesting. Shared memories showed -34.10%, while register variables were -7.00%. Tab. 2 and 3 summarize all the statistics taking only RT factor into account, while Tab. 4 incorporates additional latency factor. Fig. 4 shows the total clock cycles used in all 4 cases. A similar pattern is obtained by total number of instructions as shown in Fig. 5.

Tab.2. (RT) Instruction clock cycles and number of instructions.

Case	Instruction Clock Cycle			Number of Instruction		
	Local	Register	Shared Memory	Local	Register	Shared Memory
1	27.31	28.63	18.98	30	35	21
2	112.27	98.91	72.27	124	116	84
3	42.97	44.28	29.64	46	52	32
4	190.57	177.16	125.57	204	206	139

Tab. 3. (RT) Energy consumption by allocation scheme (nano Joule).

Case	Local	Register	Shared memory	Shared Memory VS Local	Register VS Local	Shared Memory VS Register
1	300.41	314.93	208.78	-30.50	+4.60	-33.70
2	1234.97	1088.01	794.97	-35.60	-11.90	-26.93
3	472.67	487.08	326.04	-31.00	+2.90	-33.06
4	2096.27	487.08	1381.27	-34.10	-7.00	-29.12

Tab. 4. (RT+Latency) Energy consumption by allocation scheme (nano Joule).

Local	Register	Shared memory	Shared Memory VS Local	Register VS Local	Shared Memory VS Register
839.41	809.93	538.78	-35.81	-3.51	-33.47
3654.97	2375.01	1564.97	-57.18	-35.02	-34.12
1352.67	1279.08	854.04	-36.86	-5.44	-33.23
6199.27	5446.76	3801.27	-38.68	-12.14	-30.21

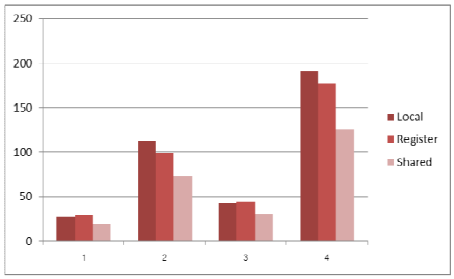

Fig. 4. Total number of clock cycles (Y-axis) for all cases (X-axis).

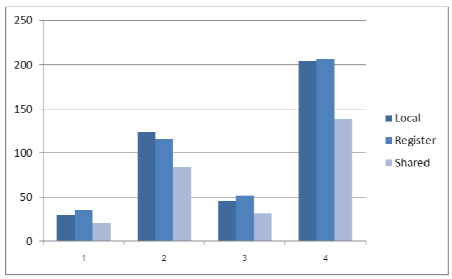

Fig.5. Total number of instructions (Y-axis) for all cases (X-axis).

Discussion

The results obtained from our KP tool did not come as a surprise, as they were well-established programming facts. Our findings merely reinstated local, register, and shared memory variable scoping principles. From our findings, shared memory reduced energy consumption by approximately 30% in simple function calls, and 35% in nested function calls. Register variable, on the other hand, was effective only when repetitive accesses were called for. It was however not as efficient as shared memory. Energy consumption comparison between shard memory and register variable reveals that the former saves approximately 33% and 25% in simple function calls and nested function calls, respectively.

An interesting finding that is not stated in many literatures is the total energy consumption of different variable scoping. We found that local variables consumed the most energy. From the

instruction level, it was apparent that stack process took considerable clock cycles which resulted in high latency and energy consumption. Fig. 6 illustrates the results obtained in Tab. 3, while Fig. 7 shows the results from Tab. 4. The shortfall of shared memory is due to memory access protection as it violates information hiding principle in Software Engineering. The side effect of shared access is what programmers should heed and practice with care.

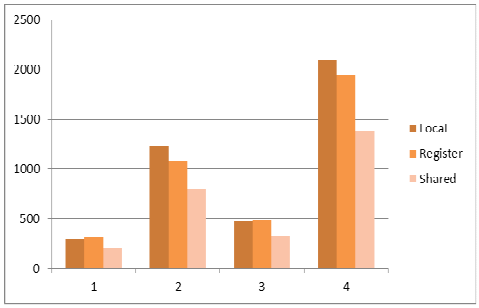

Fig. 6. Comparison of energy consumption (Y) for all cases (X).

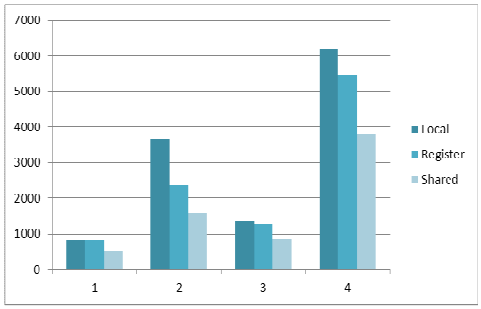

Fig. 7. Comparison of energy consumption (Y) for all cases (X).

Conclusion and Future Work

We investigated energy consumption used by local variable, register variable, and shared memory in C programs. Experiments were conducted in 2 scenarios by means of a KP tool. We found that shared memory significantly saved the most energy consumption over local and register variables. Under repetitive accesses, register variable provided considerable reduction of energy consumption. However, the outcomes were not any surprised. The gains from shared memory could possibly be offset by violation penalty of "good" software engineering practice, e.g., side effects, information hiding, portability, etc. The issue at hand is whether software engineering or energy consumption is crucial to producing theoretically sound or environmentally conscious products.

The KP tool greatly helps identify code fragment to reduce the energy consumption from programming point of view. However, it is currently limited to basic analysis. We plan to improve our tool to be able to analyze complicated C programs in wider research aspects.

References

[1] Kostas Zotos, Andreas Litke, Er Chatzigeorgiou, Spyros Nikolaidis, George Stephanides, "Energy complexity of software in embedded systems". ACIT - Automation, Control, and Applications 2005.

[2] Agner Fog, Instruction Tables Lists of Instruction Latencies, Throughputs and Micro Operation Breakdowns for Intel, AMD and VIA CPUs, Copenhagen University College of Engineering.

[3] E. Grochowski and M. Annavaram, Energy per instruction trends in Intel microprocessors, Technical rezport, Microarchitecture Research Lab, Intel Corporation, Santa Clara, CA, Mar 2006.

[4] S. Eugene, B. Paramvir and Michael J. Sinclair, "Wake on wireless: an event driven energy saving strategy for battery operated devices", in MobiCom '02 Proceedings of the 8th annual international conference on Mobile computing and networking, 2002, pages 160-171.

[5] Vivek Tiwari, Sharad Malik, Andrew Wolfe and Mike Tien-Chien Lee, "Instruction Level Power Analysis and Optimization of Software", Journal of VLSI Signal Processing Systems, 1996, pages 223-238.

[6] H. Timo, E. Christopher, K. Rüdiger and Wolfgang Schröder-Preikschat, “SEEP: exploiting symbolic execution for energy-aware programming”, in HotPower '11 Proceedings of the 4th Workshop on Power-Aware Computing and Systems, No. 4, 2011.

[7] Mike Tien-Chien Lee and Lee Vivek TiwariA, “Memory allocation technique for low-energy embedded DSP software”, IEEE Symposium on Low Power Electronics, San Diego, CA, Oct 1995.