Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp1033-1038)

Execution Characteristics of C++ and C Programs on Embedded Processor ARM7TDMI

Ji Weixing Shi Feng Qiao Baojun
Department of Computer Science and Engineering
Beijing Institute of Technology
Beijing 100081
CHINA

Abstract: - This paper presents detailed behavioral measurements of several C++ and C programs on embedded processor ARM7TDMI. By comparing the instruction set usage of C++ and C benchmark programs, we can specify what is needed in an embedded object-oriented processor in order to provide a fully object-oriented system in both hardware and software. Various architectural data related to execution behavior and instruction set usage is collected using ADS 1.2 and profiling software. In addition, the benchmarks explored in this paper have both C++ and C versions. All the programs are computing intensive and used with high frequency in embedded software. Results show that the static size of C++ program is larger than C programs. Although, C++ programs have more call instructions and more memory operations, C++ programs possess smaller function size than C programs. Various suggestions on optimization to be applicable in both hardware and software are appended in our research paper.

Key-Words: - Object-oriented Programming; Program Behavior; Instruction Set Usage; Embedded system; Program size; Function call

1 Introduction

It is widely accepted that object-oriented paradigm can improve code reusability and facilitate code maintenance. Object-oriented programming language, specifically the language C++, has been oftenly used and is replacing procedural languages such as C in many application domains. Although software engineers and software developers embrace C++ for benefits as referred above, earlier studies show that C++ program's behavior is quite different from C programs and tends to be slower when executed on modern processors. Consequently, software systems with real time restrict rarely adopts object-oriented programming due to its performance penalty. The performance overhead is coming from that object-oriented programs are been executed on non-object-oriented processors, since both the compiler and the operating system have to map of object-oriented programming features to the non-object-oriented processor. Therefore, an object-oriented processor is needed to speed up the execution of object-oriented programs. In this paper, we investigate the execution characteristics of several C and C++ benchmarks on processor ARM7TDMI in order to guide the architecture design of our embedded object-oriented processor. Both the C and C++ programs have the same functionality and are compiled by the same compiler. The only difference is the programming paradigm, in which one is procedural, and the other is object-oriented. In this way, we can observe the real performance penalty of C++ programs executed on embedded processors. In the experiment, various architectural data are collected using Arm Development Suit version 1.2. We also measure and present program features such as program size, function invocation and instruction distribution of the benchmarks.

2 Related Work

The empirical behavior investigation of programs falls into two categories: measurements of different aspects of program behavior and instruction set usage on particular architectures. A major work is done by Brad Calder and Dirk Grunwald. They measured the empirical behavior of a group of significant C and C++ programs on the DEC Alpha architecture. The behavior characteristics of C++ programs are identified and optimization that should be applied in those programs is suggested. The results show that C++ programs exhibit behavior that is significantly different from C programs. R. Radhakrishnan and L. John characterized the performance of several C and C++ benchmarks on an UltraSPARC- processor. Various architectural data related to execution behavior of the benchmarks are collected using on-chip performance monitoring counters. They conclude that the programs in the C++ suite incur a higher CPI, higher cache misses, and higher branch mispredictions than the programs in the C suite and C++ application programs have a strong correlation between CPI and branch mispredictions.

Our study is based on the ARM processor and the benchmarks employed are different from those employed in [1] [2]. The benchmark programs contain a suite of tests that measure the relative performance of object-oriented programming (OOP) in C++ versus just writing in plain C-style code. The programs of C++ and C version with exactly the same functionality allow us to characterize the C++ programs more rigorous. Moreover, the benchmarks have considerable computing functions, whose performance are tested for commonly applicable language features and are oftenly adopted in embedded system.

3 Benchmarks and processor

Investigating the execution behavior of C++ programs requires a suite of benchmark written in C and C++. The availability of both C and C++ version should be ideal for comparing and characteristics [1]. And for embedded systems the most frequently used functions, such as fast Fourier transform, encryption and matrix computing programs, should be included. Although the programs in our experiment are really limited, a wide range of applications can be presented due to their computing intensive characters. Our benchmarks for comparison are listed in table 1.

Table 1 The benchmarks

Benchmark	Description
md5	Md5 digest algorithm.
calc	The Reverse Polish Notation Calculator.
complex	Multiplies the elements of two arrays containing complex numbers.
iterator	Dot product programs.
matrix	Matrixes' multiplying programs.
max	Computing the maximum over a vector programs.
radix	Radix sort programs.
fft	Fast Fourier Transform programs.

Max, Matrix, Iterator and Complex are programs contained in benchmark OOPACK, which contains several programs that both have C version and C++ version [4]. For example, Complex of C-style multiplies the elements of two arrays containing complex numbers, and performs the multiplication by explicitly writing out the real and imaginary parts. In contrast, OOP-style defines complex numbers as a class and complex addition, and multiplication is done using overload operations. The original intention of OOPACK is profiling the optimization issues of certain compiler. Fast Fourier transform is widely used in digital signal processing systems and md5 is used in security systems. Calculator, iterator and radix are also programs that are wildly used in embedded applications.

The ARM7TDMI core is a member of the ARM family of general-purpose 32bit microprocessors and is based on Reduced Instruction Set Computer (RISC) principles [6]. A three-stage pipeline is used in ARM7TDMI, and instructions are executed in three stages: fetch, decode and execution. The ARM7TDMI core has Von Neumann architecture, with a single 32-bit data bus carrying both instructions and data. Only load, store, and swap instructions can access data from memory [7]. This is an un-cached core and processor delivered as a hard macrocell optimized to provide the best combination of performance, power and area characteristics.

4 Experiment Methodology

There has been a great deal of researches on various aspects of the behavior of different kinds of programs in order to improve the performance of their programs with aggressive software optimization. Our focus is on comparing the relative behavior of programs written in C and C++. In particular, we are interested in measuring aspects of behavior that might be exploited with better hardware, which will guide the design of our embedded object-oriented processor. The program and function sizes are also investigated in our research paper.

The Instruction Set Usage Statistics is very important to the design of our object-oriented processor. Both the static instruction sequence and dynamic execution instruction sequence are required to analyze the behavior of programs. Because there is a lack of on-chip performance monitoring counters for ARM processors, we have to make use of software instrument tools to collect the dynamic execution instruction sequence. The systems set up in order to evaluate the benchmark are shown in figure 1.

This experimental process is similar to the one proposed in [2]. Each program code was compiled using the C and C++ compiler of ARM Software

Development Toolkit 1.2, which provide both the code size and the minimum RAM requirement for the data of each kernel. And the optimization option was disabled since we wanted to observe un-optimized instruction sequences to speed up C++ programs using hardware but not software, for example the compiler. Next, we used the ARM Symbolic Debugger to produce a trace file logging instructions and memory accesses. Finally, the profiler parsed the trace file to obtain the numbers of function calls, the number of memory accesses and executed instructions. The profiler is particularly developed for this study.

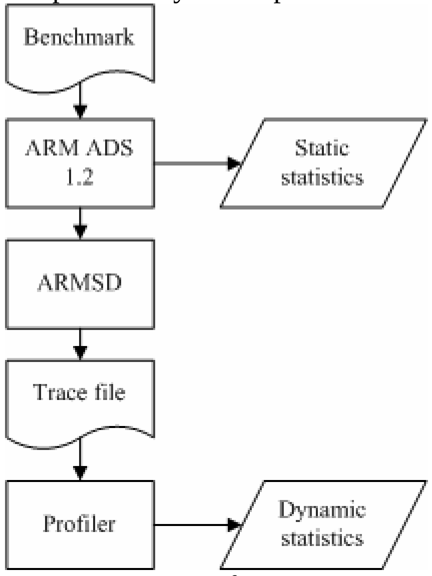

Fig. 1 System set up for statistics

5 Measured execution behavior

Program execution behavior includes cache missing rate, function size, stack depth and code size. Only CPI, function and code size are shown up in our presented paper.

5.1 Static code size and dynamic code size

Table 2 gives information about the static and dynamic size of the code for each program. And the dynamic and static size ratio is illustrated in figure2. Static size shows how many instructions each program contains. Dynamic size represents how many instructions were executed during each program run [1].

Table 2 Static and dynamic code size

Program	Static		Dynamic	
	C	C++	C	C++
md5	3,766	8,815	70,939	128,504
calculator	3,846	16,114	23,642	92,590
complex	3,686	8,952	618,828	763,827
iterator	3,613	8,843	335,895	396,916
matrix	3,637	8,860	13,231,530	14,637,591
max	2,686	7,872	54,131	88,921
radix	2,798	13,641	28,625	93,734
fft	7,342	14,720	268,532	286,654

From the table, we see that the static size of C++ program is larger than the C program, even though they are coded to have the same function. More instructions are executed when running C++ programs. This result is similar to what was observed in the study conducted by R. Radhakrishnan and L. John [2] and shows that C is more effective than C++ in solving problems. Object-oriented paradigm can improve code reusability and facilitate code maintenance, but also results in more instructions. The result of dynamic size and static size ratio in C is higher than in C++. This is also verified in [1].

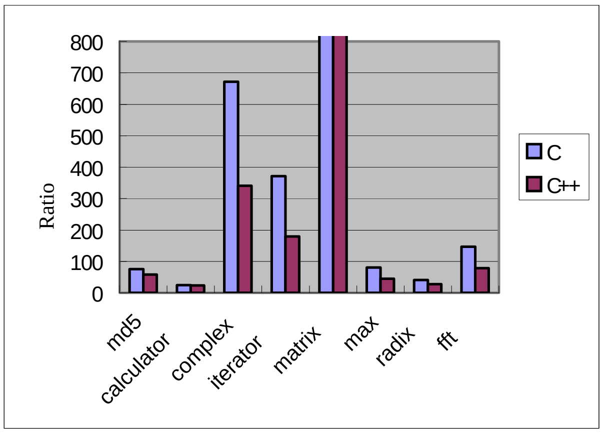

Fig. 2 Program ratio of dynamic and static

5.2 Dynamic function size

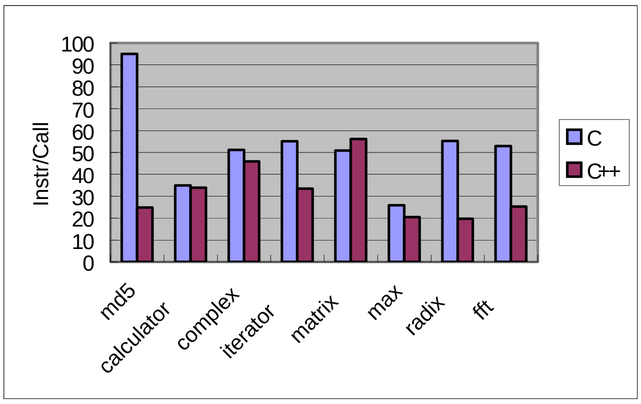

Fig. 3 Instructions/Call for C and C++ programs

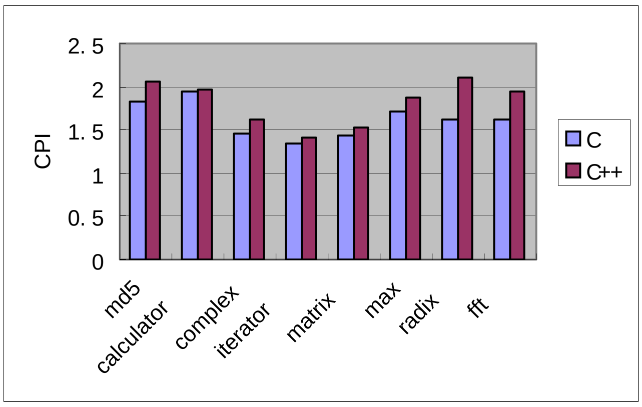

Fig. 4 CPI for C and C++ programs

Dynamic function size is very important to the performance of the programs, since lots of small functions will result in greater overhead than large functions. Figure 3 gives the statistical results of our programs. We can see that the dynamic function size of C++ programs is smaller than C programs. Function size for C++ programs can be attributed to the object-oriented design paradigm. The class encapsulates information by bundling the data items and methods and treating them as a single entity. The class structure hides implementation details and carefully restricts outside access to both the data and operations. This principle deals with information hiding, protection of data integrity, and results in lots of small functions.

5.3 CPI comparison

It is seen that C++ programs execute more instructions than C programs. To quantify the performance, we measured cycles per instruction (CPI) based on total execution cycles. Both dynamic instruction number and total cycles are collected using ARMSD as illustrated in figure 1. From figure 4 we can see that C++ programs show consistently higher CPI than the C programs. The mean CPI for C++ is 1.82, while 1.62 for C programs. This result shows that the instruction composition of C++ programs and C programs are different. Instructions, which take more cycles, are apt to appear more frequently in C++ programs. And this is verified in setion5.1.

6 Instruction Set Usage Statistics

In this section profiling information about instruction set usage is presented. Top 15 frequently used instructions will give an overview of instruction usage in C++ and C programs. And we will discuss function call, control transfer and memory operation instructions in detail.

6.1 Top 15 frequently used instructions

Due to the differences in the execution behavior of C++ and C programs, we made perception that the most commonly and frequently used instructions must be different in them.

Table 3 lists the top 15 frequently used instructions. The percentage in the brackets is the arithmetic mean of all instruction percentages in the 8-benchmark programs. Moreover, it is obvious from the table that the order of specific instruction is different. Table 3 suggests that ADD (addition instruction) is the most frequently used instructions in C programs. ADDC (addition with carry), ADDS (addition with flag update), ADCS (addition with carry and flag update), and CMP (comparison) also have higher position in C programs. The second most frequently used instruction is MOV, which is used to move data from register to register in RISC processors. Logic instruction ORR is the third rank and memory load instruction LDR follows next.

Table 3 Top 15 frequently used instructions

Order	C	C++
1	ADD (9.93%)	LDR (21.87%)
2	MOV (7.86%)	ADD (12.56%)
3	ORR (5.64%)	MOV (8.78%)
4	LDR (5.61%)	EOR (4.21%)
5	ADC (5.04%)	LDRB (4.17%)
6	ADDS (4.83%)	STR (3.78%)
7	ADCS (4.03%)	ORR (3.08%)

8	BIC (3.92%)	ADC (2.72%)
9	UMULL (3.42%)	ADDS (2.60%)
10	TST (3.29%)	CMP (2.41%)
11	B (3.15%)	BL (2.36%)
12	CMP (3.10%)	B (2.21%)
13	MOVS (2.76%)	ADCS (2.19%)
14	LDMIA (2.69%)	BIC (2.16%)
15	BICNES (2.63%)	TST (1.97%)

In contrast, in the C++ column we can see the most frequently used instruction is memory load instruction LDR and its percentage of usage is amazingly high and up to 21.87%. After LDR, instruction ADD and MOV are used more frequently than other instructions in the table. We can conclude that though ADD and MOV instructions are not mostly used in C++, they still have higher percentage of usage than to be used in C programs. Logic instruction EOR (xor) takes place of ORR has the fourth rank, and ORR falls to the seventh one. Both memory load byte LDRB and memory store instruction STR are appeared in the top 15 lists. Another instruction, which does not appear in C column, is the function call instruction BL. Bit clear (BIC) and bit test instruction (TST) now locate at the bottom of C++ column. Another difference, which is worth paying attention to, is the percentage of top 4 instructions in both columns. The percentage of ADD, MOV, ORR and LDR are nearly consecutive in C column, while the percentage of LDR, ADD, MOV and EOR are nearly twice to the subsequent instruction.

Memory operations in C++ are also found more in number as compare to C program. This fact is also extracted from the table that number of function call is also prominent in number in C++ program. However, the difference between JUMP instructions is not considerable in the two. Logic instruction EOR has a significant percentage in C++ programs, while it is not in the list of C programs. And BIC and TST have lower percentage in C++ programs. These conclusions are explained and verified in the later sections.

6.2 Function Calls

The number of function call is very important to analyze precisely as the instruction level parallelism available from a program is heavily influenced by the frequency of function calls and average function size. Processor has to save registers and pass parameters to the callee functions. Figure 5 depicts the function calls made in our benchmark. We see that C++ programs made more function calls than C programs except matrix (Bench Mark). The reason is that most methods of class in program matrix are defined inline. This gives us a clue that the function call overhead can be partially eliminated by compiler optimization. Much work has already been done to reduce function call overhead in C++ programs [3] [8].

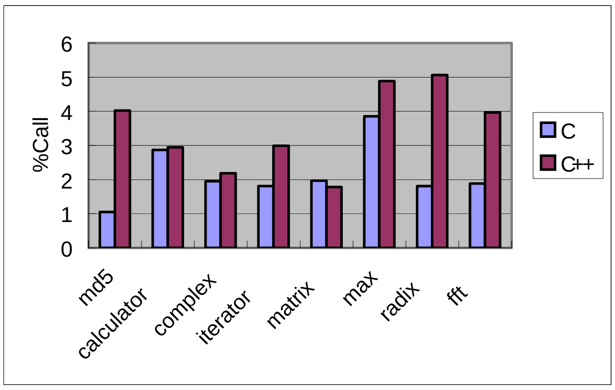

Fig. 5 the percentage of function call in benchmark

6.3 Control Transfer Instructions

Modern architectures using deep instruction pipelines and speculative execution rely heavily on predictable control flow changes, and control transfer instructions cause unpredictable changes in program control flow [9]. The number of control transfer instructions determines the block size of code. The frequency of control transfer instructions like branches, jumps and calls which cause a break in the program and this is really a mess to the super pipelined processor engineers.

Figure 6 gives the percentage of control transfer in benchmark programs. From Figure 6, we can summarize C++ programs have more control transfer instructions than C programs. This indicates that it's more difficult for modern super pipelined and super scalar processor to find the required instruction level parallel.

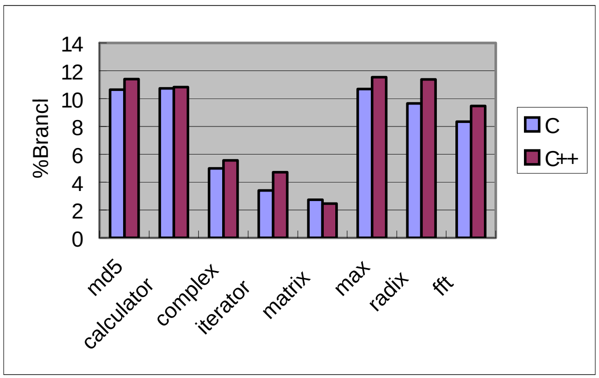

Fig. 6 Control transfer instructions

6.4 Memory Operations

Modern computer architectures are sensitive to the number of memory operations, because memory has become the bottleneck of the overall system and cannot shovel data in and out fast enough to keep up with the work being done. The measurement of load and store memory instructions is given in table 4. Although hardware mechanisms such as caches seek can be used to mask some of these problems, memory operations are still choke point for C++ programs.

Table 4 describes that memory operation percentage of C++ programs ranges from 11% to 35%. That is, load and store are application related and can range over a wide field.

Table 4 Load and store instructions analysis.

Program	%Load		%Store	
	C	C++	C	C++
md5	19.6	24.6	10.0	11.2
calc	21.7	22.0	12.2	12.2
complex	9.8	11.0	5.5	7.8
iterator	6.1	7.1	4.3	4.9
matrix	9.7	14.8	4.1	7.1
max	10.71	12.3	8.0	8.4
radix	11.7	21.8	8.0	12.1
fft	12.5	18.2	7.3	10.8

7 Conclusion

In this paper, we analyze dynamic behavior of programs in C++ and C using ARM7TDMI processor. As compared to existing techniques, our work is on the embedded processor and the benchmark programs have both C and C++ versions. We observed that the static size of C++ programs is larger than C programs, even though the functionality and input/output of both versions are the same. We also observed that the function size of C++ programs is smaller than C programs. What is more, C++ programs have more control transfer and memory operation instructions, higher CPI than C programs. The top 15 most frequently used instructions of C++ programs are different from C programs.

All these observations narrate that semantic gap of C++ programs and modern embedded processors can not been ignored in the design of embedded object-oriented processors. The performance of processor will be heavily affected due to more control transfer instructions appear in instruction pipeline. Therefore, the processor should provide a fast method manipulating procedure, which means function invocation and RETURN should get more support from hardware to speed up program execution. And this problem can be partially eliminated by compiler as referred above. The memory load and store problem will be more serious when executing C++ programs. Some specific technique should be explored to shovel data in and out quickly enough to catch up with the processor.

Although the profiling result in this paper confirms some of prior work in comparing C and C++ programs, the measurement of cache missing and stack depth cannot be done for lacking of software and hardware support. Moreover, further study should be done on other embedded processors, such as DSP processor; this is planned to be done in future.

References:

[1] Brad Calder, Dirk Grunwald, Benjamin Zorn. Quantifying Behavioral Difference Between C and C++ programs, *Journal of Programming Languages*, Vol.2, No.4, 1994, pp. 313-351.

[2] R.Radhakrishnan, L. John, Execution Characteristics of Object Oriented Programs on the UltraSPARC-II. *HIPC'98 5th International Conference*, 1998, pp. 202-211.

[3] Brad Calder, Dirk Grunwald. Reducing Indirect function call overhead in C++ programs. *In Annul Symposium on Principles of Programming Languages*, 1994. pp.397-408.

[4] Alexander Chatzigeorgiou. Perfromance and power evaluation of C++ object-oriented programming in embedded processors. *Information and software Technology*, 2003, 45:195-201.

[5] Dennis C. Lee, Patrick J. Crowley etc. Execution Characteristics of Desktop Applications on Windows NT. *ACM SIGARCH Computer Architecture News*, 1998, Vol.26, No.3, pp. 27-38.

[6] ARM Limited. *ARM7TDMI Technical Reference Manual*. 2001.

[7] ARM Limited. ARM Instruction Set Quick Reference Card. 2003.

[8] Donzellini G., Nervi S., Object oriented ARM7 coprocessor. *In Proceedings of the Thirty-Frist Hawaii International Conference*, 1998, pp:243-252.

[9] Da-Chil David Tang, Ann Marie Grizzaffi Maynard. Contrasting Branch Characteristics and Branch Predictor Performance of C++ and C Programs. *In Performance, Computing and Communications Conference*, 1999, pp:275-283.