Static Profiling of Assembly Code Performance and Optimization Effectiveness using Instructions Performed and Program Latency

Jonathan Paul C. Cempron, Chudrack Shalym Y. Salinas, Roger Luis Uy

Abstract*: Software program optimization for improved execution speed can be achieved through modifying the program. Programs are usually written in high level languages then translated into low level assembly language. More coverage of optimization and performance analysis can be performed on low level than high level language. Optimization improvement is measured in the difference in program execution performance. Several methods are available for measuring program performance are classified into static approaches and dynamic approaches. This paper presents an alternative method of more accurately measuring code performance statically than commonly used code analysis metrics. New metrics proposed are designed to expose effectiveness of optimization performed on code, specifically unroll optimizations. An optimization method, loop unroll is used to demonstrate the effectiveness of the increased accuracy of the proposed metric. The results of the study show that measuring Instructions Performed and Instruction Latency is a more accurate static metric than Instruction Count and subsequently those based on it.

Index Terms*: Assembly Programming, Code Profiling, Performance Metrics. Instruction Set Architecture, Loop Unroll, Vectorization, Compiler, Software Optimization, Time Complexity.

I. INTRODUCTION

Software program optimization for improved execution speed can be achieved through modifying the program. Programs are usually compiled from a high level language into machine low level language. More coverage of optimization and performance analysis can be performed on low level than high level language. Discussed are the process of how a program is transformed from the programmer's code into a language that the processor natively understands. And presents strategic choice on which portion of this process is best for implementing optimizations on.

Optimization improvement is measured in the difference in execution performance. Several methods are available for measuring code performance classified into static and dynamic approaches [1]. Dynamic approaches involves actual program runtime, but less focus on analysis of code. Current

Static approaches involves simplistic analysis of the code, Instruction Count (IC). Discussed are the advantages and disadvantages of IC and alternative methods to more accurately measuring code performance statically.

New metrics presented are named Instructions Performed and Program Latency. These metrics are initially designed to expose effectiveness of unroll optimization performed on code. But can be used to more accurately represent code performance.

An optimization method, loop unroll is used to demonstrate the effectiveness of the increased accuracy of the proposed metric. A method for optimizing assembly code [2] used by popular compilers GCC and ICC [3].

II. CODE TRANSLATION AND OPTIMIZATION

Software is most commonly written on High Level Languages then translated into Low Level Language native to the processor that the software will be executed on [4][5][16][18]. Translation can be done either through Interpretation or Compilation. Interpretation is done on a line per line analysis then execution of the written source code. Examples are Python and R. Compilation is when the whole source code is analyzed and the whole software is translated at once. Example of a compiled language is C.

Software optimization is more commonly performed in compiled language than in interpreted language because of the available knowledge of the complete flow of the program presented at compile time. Also there is more time available to perform optimizations during compile time while in interpreting the language, there is significant overhead caused during optimization.

Optimization can be performed on either high level language or in low level language. However we suppose that more coverage, practicality, and effectiveness of optimization can be achieved on low level. There is more coverage because there are more languages being compiled to a single processor architecture [6]. It is also more practical in that there are fewer architectures than languages to perform optimizations on. It is also more effective because of the closer relationship of the optimization to the actual processor hardware

Revised Manuscript Received on August 19, 2019.

Jonathan Paul C. Cempron, Computer Technology Department, College of Computer Studies, De La Salle University, Manila, Philippines.
Chudrack Shalym Y. Salinas, Computer Technology Department, College of Computer Studies, De La Salle University, Manila, Philippines.
Roger Luis Uy, Computer Technology Department, College of Computer Studies, De La Salle University, Manila, Philippines.

[15][18][23][24][25][30]. Popular processor architecture families that are subject to optimizations are: x86 architectures, ARM architectures, and MIPS architectures.

Consequently, analysis of the optimization is also best done on low level languages. Measuring the performance of a code and also the effectiveness of an optimization is done using performance metrics. Popular metrics are discussed and also a more accurate proposed metric is also presented.

Optimization could be of different interests [16][17][20][21][26], of which are: code size, code density [7], speed, memory, data, network, and power consumption. The focus of this paper is speed; shorter execution time.

Optimization for speed is best achieved when the software program takes advantage of the processor architecture's features such as pipelined processors and out of order execution.

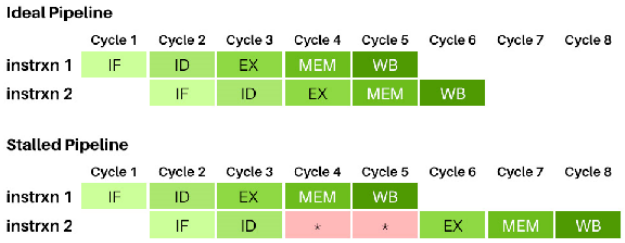

Figure 1. MIPS Pipeline example

Pipeline is a feature wherein several machine instructions can be performed at once. The pipeline is maximized in software by reduction of stalls. A stall processor state is when an execution unit is waiting for a data dependency. Stalls are reduced when dependencies are avoided, this can be done by modifying the assembly code [8][9][10][27]. Figure 1 shows a MIPS pipeline with and without a stall, a stall causes instructions to consume more processor cycles. In an Ideal Pipeline, all the stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB), perform in lockstep; each instruction is completed in 5 cycles. In a Stalled Pipeline however, due to either a dependency on a previous instruction or a lack of resources, an instruction has to be stalled in order for the program to execute correctly.

Software optimization is still closely coupled with knowledge of the computer architecture being used and its actual implementation. One common cause of stalls are jumps in assembly programs. Jumps are typically caused by conditional statements and loops. One method of optimization is by minimizing usage of jumps in loops by performing loop unroll.

Loop unrolling extends source code such that the use of branch instructions is reduced. The method of unrolling a loop will not be discussed but as a summary, loop unrolling increases code length in exchange for reduced latency from branching instructions. As illustrated in Figure 2.

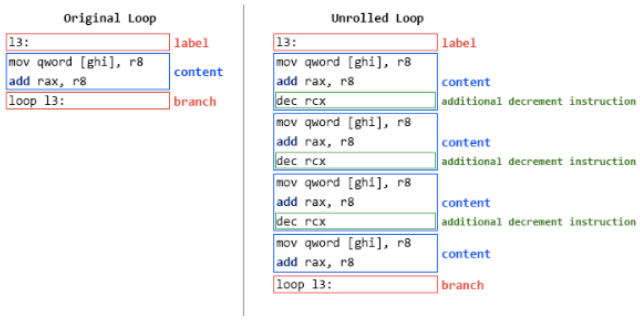

Figure 2. Example of an unrolled loop

III. CURRENT PERFORMANCE METRICS

The effectiveness of any optimization method must be tested and measured before claiming that the method actually optimizes for a specific interest. This measurement is done by using program performance metrics. Code performance metrics have two categories: static and dynamic.

Dynamic metrics are the measurement of the program performance during actual runtime. This can be done through the use of the clock() function in C. Figure 3 illustrates this method. The clock() function is used to obtain the starting and ending times of the program. The execution time of the program is calculated by subtracting the start and end time.

```
1  #include <stdio.h>
2  #include <time.h>
3  int main()
4  {
5     int start = clock();
6     //CALL ASSEMBLY
7     int end = clock();
8     double Etime = (double)(end-start)/clockPerSecond;
9     printf("execution time: %lf", Etime);
10    return 0;
11 }
```

Figure 3. Performance measurement using C function clock()

The simplicity of this measurement approach is its main advantage. A second advantage is the return of measurements in actual time. One of the disadvantages of this method is its unreliability when a program requires user input. Another disadvantageous situation is when the measurement is performed in a multitasking environment; where the results will always include the effects of the other programs running in the same computer. This causes an inaccuracy where the resulting time is more than what the program actually consumes. There are approaches to overcome the challenges presented by multitasking. One approach is to combine the measurement with static profiling of the code.

The focus of this paper is on Static methods of determining code performance. One of the most popular metrics for comparison is through Instruction Count (IC). IC is the number of lines that are in the Code Segment of an assembly code program [8][9]. White space lines, and assembler directive are not included in the instruction count.

$$CPU\ Time = \begin{matrix} Instruction\ Count\ x \\ Clock\ Cycles\ Per\ Instructions\ x \\ Clock\ Cycle\ Time \end{matrix} \quad (1)$$

IC is the most simplistic static metric for a program. An advantage of this metric is that it is easy to perform. Another commonly used metric for assembly program code measurement is the CPU Time shown in equation (1). CPU Time includes the actual speeds of the processor thus should yield the actual time [8][9].

$$CPI_{pipelined} = \begin{matrix} Ideal\ CPI + \\ Average\ Stall\ Cycles\ Per \\ Instruction \end{matrix} \quad (2)$$

For pipelined processors, the Clock Cycle per Instruction of equation (1) be replaced with Pipeline CPI. Which considers the processor pipeline feature. This yields a more accurate result in case of computing for a pipelined processor [4].

	Source Code Instructions	Instructions Executed
Program A	1 .code 2 daddiu r1, r0, #0003 3 daddiu r2, r0, #0001 4 label1: 5 dsubu r1, r1, r2 6 bnez r1, label1 7 syscall 0	daddiu r1, r0, #0003 daddiu r2, r0, #0001 dsubu r1, r1, r2 bnez r1, label1 syscall 0 dsubu r1, r1, r2 bnez r1, label1 syscall 0 dsubu r1, r1, r2 bnez r1, label1 syscall 0
Program B	1 .code 2 daddu r1, r0, r0 3 daddu r1, r2, r0 4 daddu r3, r4, r0 5 dsubu r5, r1, r3 6 dmultu r22, r5 7 daddu r22, r3, r5 8 daddu r23, r1, r5 9 syscall 0	daddu r1, r0, r0 daddu r1, r2, r0 daddu r3, r4, r0 dsubu r5, r1, r3 dmultu r22, r5 daddu r22, r3, r5 daddu r23, r1, r5 syscall 0

Figure 4. Instructions Performed of a Looping and Non-Looping Program

Figure 4 above will be used to display the limitation of Instruction Count (IC). Program has an IC of 5 while Program B has an IC of 8. Program B has a higher IC. However when the two programs were ran Program A has more instructions that were performed. This limitation of IC is because it does not have consideration of programming loops. Lepak et al. [4] also agree with the unreliability of instruction count in multiprocessor systems and discuss a simulation methodology for improved performance measurement.

Source Code	Instructions Performed
1 .code 2 daddu r1, r2, r3 3 j label1 4 daddu r3, r4, r5 5 daddu r6, r7, r8 6 daddu r9, r10, r11 7 label1: 8 daddu r12, r13, r14 9 syscall 0	daddu r1, r2, r3 j label1 daddu r3, r4, r5 daddu r12, r13, r14 syscall 0

Figure 5. Instructions Performed of a Jumping Program

Another limitation of Instruction Count is exposed in Figure 5 above. A program has less instructions performed than its IC. This inaccuracy is because IC does not have consideration for branches or jumps.

The inaccuracy of IC is also propagated to the CPU Time equation (1). Since IC is a part of CPU Time. Solving this inaccuracy is important for comparing an optimized and an unoptimized code. That is because an optimized code can have a higher IC but may not necessarily be slower.

IV. OPTIMIZER EFFECTIVENESS

A different metric was developed by this study that would provide a more accurate performance analysis of the program code. The metric takes into consideration programming blocks. The metric uses an analysis of the instructions that will be performed by the code after execution. This solution will be called Instructions Performed for the rest of the discussion.

Instructions Performed is the number of instructions executed by the program during runtime. This takes into consideration blocks of code that are repeated while the program is running. Blocks of code that are ignored on runtime are also accounted. For a more accurate CPU time in equation (1), the Instruction Count can be replaced by Instructions Performed.

Instruction Latency is similar to instructions performed with added consideration to the latency of each instruction. As different instructions take different time in some architectures such as the x86_64 architecture. For the x86_64, latency per instruction information is defined by their document for recommendations for compiler developers in [11] and other processor related documents [19][22][29].

V. METHODOLOGY

The method for computation of proposed metrics Instructions Performed and Program Latency is explained by demonstration. Below are Fibonacci programs written in different processor assembly languages. The computation for the proposed metrics are shown alongside the code.

Source Code	Block Type	Instrxn Count	Block Repetition	Total
1 .data 2 fib: .word64 0,0,0,0,0,0,0,0,1,1 3 .code 4 daddiu r8, r0, 8 ;last address 5 daddiu r9, r0, 8 ;find 8 fib values 6 daddiu r10, r9, 1 7 dmult r9, r10 ;first address 8 mflo r9	Fall Block	5	x1	5
9 fibloop: 10 daddu r10, r9, r0 11 lw r11, fib(r10) ;load first val 12 daddi r10, r10 0xFFF8 13 lw r12, fib(r10) 14 daddu r13, r12, r11 ;add to second val 15 daddi r10, r10, 0xFFF8 16 sw r13, fib(r10) ;store to mem 17 daddi r9, r9, 0xFFF8 18 bne r9, r8, fibloop	2Way Block	9	x8	72
			Instructions Performed: 77	

Figure 6. Instructions Performed of MIPS64 Fibonacci code

Figure 6 shows a MIPS64 code for computing 8 Fibonacci numbers and stores those numbers into memory. The steps for computing Instructions Performed is displayed in the table to the right of the code. The steps in computing the Instructions Performed are presented as columns, read from left to right.

Source Code	Block Type	Instrxn Count	Block Repetition	Total
1 segment .data 2 fib dq 0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0, 0x1,0x1 3 segment .text 4 global _start 5 start:				
6 mov rcx, 8 ;find 8 fib values	Fall Block	1	x1	1
7 fibloop: 8 mov rax, qword[(fib+rcx*8)] ;load first val 9 add rax, qword[(fib+rcx*8)-8] ;add to second val 10 mov qword[(fib+rcx*8)-16], rax ;store to memory 11 loop fibloop	2Way Block	4	x8	32
			Instructions Performed: 33	

Figure 7. Instructions Performed of x86_64 Fibonacci code

Figure 7 shows the same Fibonacci code but implemented for the x86_64 architecture. The following steps are done in order to compute for the Instructions Performed. First step is to separate the code into basic programming blocks. Second is to determine the type of blocks. Third is to count the number of instructions in each block. Fourth is to determine the number of repetitions the block will be performed throughout the program. This can be determined by analyzing the value assigned to the loop counter before entering the block. Typically, only a 2way block has more than 1 repetition. If the number of repetitions cannot be determined, just assume that the block is executed once. A single execution is chosen as default because a block of code would usually be at least used once. Fifth step is to multiply the Instruction Count and block repetition. Sixth and final step is to sum the products and the result is the Instructions Performed.

Source Code	Block Type	Instrxn Latency	Block Latency	Block Repetition	Total
1 segment .data 2 fib dq 0x0,0x0,0x0,0x0,0x0,0x0,0x0, 0x0,0x0,0x1,0x1 3 segment .text 4 global _start 5 start:					
6 mov rcx, 8 ;find 8 fib values	Fall Block	1	1	x1	1
7 fibloop: 8 mov rax, qword[(fib+rcx*8)] ;load first val 9 add rax, qword[(fib+rcx*8)-8] ;add to second val 10 mov qword[(fib+rcx*8)-16], rax ;store to memory 11 loop fibloop	2Way Block	1 1 1 1 8	11	x8	88
			Program Latency: 89		

Figure 8. Program Latency of x86_64 Fibonacci code

Figure 8 shows the computation for Program Latency for the x86_64 Fibonacci program. To compute for Program Latency are the following steps. First is to separate the code into blocks. Second is to determine the type of block. Third is to determine the latency per each instruction (refer to processor documentation for this step). Fourth is to sum the latencies per each block. Fifth is to determine the block repetition. Sixth is to multiply the block repetition and block latency. Seventh and final step is to sum the products and the result is Program Latency.

VI. TEST AND RESULT

Testing the effectiveness of the proposed metrics Instructions Performed and Program Latency is demonstrated via comparison of computed values on an optimized and an unoptimized version of the Fibonacci codes presented. The optimization method used is loops unrolling. The proof of loop unroll effectiveness can only be exposed in increased accuracy offered by the proposed metrics.

Source Code	Block Type	Instrxn Count	Block Repetition	Total
1 segment .data 2 fib dq 0x0,0x0,0x0,0x0,0x0,0x0, 0x0,0x0,0x1,0x1 3 segment .text 4 global _start 5 start:				
6 mov rcx, 8 ;find 8 fib values	Fall Block	1	x1	1
7 fibloop: 8 ;###UNROLL### base block 9 mov rax, qword[(fib+rcx*8)] 10 add rax, qword[(fib+rcx*8)-8] 11 mov qword[(fib+rcx*8)-16], rax 12 dec rcx ;###UNROLL### counter update 13 mov rax, qword[(fib+rcx*8)] 14 add rax, qword[(fib+rcx*8)-8] 15 mov qword[(fib+rcx*8)-16], rax 16 dec rcx ;###UNROLL### counter update 17 mov rax, qword[(fib+rcx*8)] 18 add rax, qword[(fib+rcx*8)-8] 19 mov qword[(fib+rcx*8)-16], rax 20 dec rcx ;###UNROLL### counter update 21 mov rax, qword[(fib+rcx*8)] 22 add rax, qword[(fib+rcx*8)-8] 23 mov qword[(fib+rcx*8)-16], rax 24 loop fibloop ;###UNROLL### branch	2Way Block	16	x2	32
			Instructions Performed: 33	

Figure 9. Instructions Performed of x86_64 unrolled Fibonacci

Figure 9 shows an unrolled loop from the program shown in Figures 7 and 8 unrolled with a factor of 4 wherein the loop instruction is placed at the bottom and a decrement of the loop counter is placed in between each unroll block. The unrolled block resulted in an instruction count of 16, while the original program has an instruction count of 4 for the same block. The Instruction Count is increased for the unrolled program, but the instructions performed are the same, 33 Instructions Performed.

Source Code	Block Type	Instrxn Latency	Block Latency	Block Repetition	Total
1 segment .data					
2 fib dq 0x0,0x0,0x0,0x0,0x0,0x0, 0x0,0x0,0x1,0x1					
3 segment .text					
4 global _start					
5 start:					
6 mov rcx, 8 ;find 8 fib values	Fall Block	1	1	x1	1
7 fibloop:	2Way Block		23	x2	46
8 ;###UNROLL### base block		1			
9 mov rax, qword[(fib+rcx*8)]		1			
10 add rax, qword[(fib+rcx*8)-8]		1			
11 mov qword[(fib+rcx*8)-16], rax		1			
12 dec rcx ;###UNROLL### counter update		1			
13 mov rax, qword[(fib+rcx*8)]		1			
14 add rax, qword[(fib+rcx*8)-8]		1			
15 mov qword[(fib+rcx*8)-16], rax		1			
16 dec rcx ;###UNROLL### counter update		1			
17 mov rax, qword[(fib+rcx*8)]		1			
18 add rax, qword[(fib+rcx*8)-8]		1			
19 mov qword[(fib+rcx*8)-16], rax		1			
20 dec rcx ;###UNROLL### counter update		1			
21 mov rax, qword[(fib+rcx*8)]		1			
22 add rax, qword[(fib+rcx*8)-8]		1			
23 mov qword[(fib+rcx*8)-16], rax		1			
24 loop fibloop ;###UNROLL### branch		8			
					Program Latency: 47

Figure 10. Program Latency of x86_64 unrolled Fibonacci

Figure 10 is the same program as shown in Figure 9 but displays computation for Program Latency. The computed program latency for the unrolled program is 47 which is less than 89 from the original program. While having a higher Instruction Count than the original program, the unrolled program has less latency. This is because the use of high latency instructions has been reduced by unrolling.

	Original Fibonacci	Unrolled Fibonacci
Instruction Count	5	17
Instructions Performed	33	33
Program Latency	89	47

Figure 11. Different metrics on x86 Fibonacci code

Figure 11 shows the comparison of Instruction Count, Instructions Performed, and Program Latency from the original x86 Fibonacci code to the unrolled Fibonacci code. It is shown that the original code has less Instruction Count. Both have the same Instructions Performed. But the unrolled program has less Program Latency.

	MIPS64 Fibonacci
Instruction Count	14
Instructions Performed	77
Program Latency	77

Figure 12. Different metrics on MIPS Fibonacci code

Figure 12 displays the Instruction Count, Instructions Performed, and Program Latency for the MIPS64 Fibonacci code shown in Figure 6. It is important to note that for the case of MIPS64 programs, the Instructions Performed is the same as the Program Latency because the MIPS64 architecture has uniform latency for all of its instructions.

VII. ANALYZING TIME COMPLEXITIES

Big O notation used for measuring code performance in terms of growth function of an algorithm's frequency count of its basic operation [12][13][14]. The proposed method for computation can be extended to also express time complexities in terms of big O notation. Not all types of big O notation can be detected and expressed and are limited to the following: O(1), O(n), O(nx) where x is a non-zero positive integer. The extension procedures is placed on the loop detection portion:

1. if no loop is detected, then O(1),
2. if a loop is detected and there is no nested-loop, then O(n),
3. if a loop is detected and nested loop, then O(nx) where x is the layer of the deepest nest.

It should also be noted that for O(nx) the iterations of the loops are assumed to be the same for all layers thus nx does not cover all cases. Example 1 assume a 3 layer nested loop where the 1st, 2nd, and 3rd layers have the same n iterations, then our output O(n3) is correct. Example 2, assume a 3 layer nested loop where the layers 1st has n iterations, 2nd has o, and 3rd has p, then our output O(n3) is not correct because n is not a single number, we could average (n+o+p)/3 for a better approximate but it is still not accurate. Example 3 assume a 3 layer nested loop where 1st has n, 2nd has o, 3rd has a constant 7 iterations, then our output O(n3) is not correct because the deepest layer is 3 but the third layer is actually a constant and the correct answer is O(n2).

VIII. CONCLUSION

The presented metrics: Instruction Performed and Program Latency provide a more accurate representation of code performance than Instruction Count based metrics because of its increased accuracy. Approximation of Time Complexities are also presented. The presented methods also provide a tool to prove that a higher Instruction Count does not necessitate a faster running program as shown in the case of an unrolled loop. From the results in Figure 11 it can be seen that the effectiveness of loop unroll can best be statically determined through computing the Program Latency. An unrolled program has longer Instruction Count but has significantly better Program Latency, outperforming the original code.

It is important to note that the relationship between Instructions Performed and Program Latency is dissimilar from that of the Instruction Count and Program Latency. The worst comparative result that Instructions Performed can provide is failing to expose performance gain, but it will not be the reverse from the Program Latency as was in Instruction Count. Thus, Instructions Performed is still more accurate than Instruction Count.

The most accurate metric presented, Program Latency, requires deeper knowledge of the processor as it requires information regarding the latency of each instruction. This information can sometimes be inherent to the processor architecture, as in MIPS64 having uniform latency for each

instruction shown in Figure 12. Latency per instruction can also be provided by the processor architecture manufacturer, as is the case in x86_64 architecture with [11]. But in some cases, this kind of information is not easily available, thus Program Latency cannot be computed for some architectures. Presented in this paper was a more accurate method of statically determining code performance and an approximate of the time complexity of a program designed to display the effectiveness of an optimization by more accurately exposing performance differences. The accuracy is with the cost of tediousness and should be implemented into software that automatically computes for the Instructions Performed and Program Latency. Other future work includes further testing of the static profiling in terms of time complexity.

REFERENCES

1. Y. Han "Application performance evaluation on different compiler optimizations" (ACSA) 410. Vol. 2 No. 3 ISSN 2166–2924. 2013.
2. V. P. Bharadwaj and M. Rao, "Compiler optimization for superscalar and pipelined processors," 2016 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER), Mangalore, 2016, pp. 232-236. doi: 10.1109/DISCOVER.2016.7806224
3. R. S. Machado, R. B. Almeida, A. D. Jardim, A. M. Pernas, A. C. Yamin and G. G. H. Cavalheiro, "Comparing Performance of C Compilers Optimizations on Different Multicore Architectures," 2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Campinas, 2017, pp. 25-30. doi: 10.1109/SBAC-PADW.2017.13
4. K. Lepak, H. Cain, M. Lipasti. 2003. Redeeming IPC as a Performance Metric for Multithreaded Programs. 12th International Conference on Parallel Architectures and Compilation Techniques; 2003; New Orleans, LA, USA, USA. p. 232-243.
5. T. L. Casavant, "Low-level programming of parallel supercomputers," Proceedings COMPSAC 88: The Twelfth Annual International Computer Software & Applications Conference, Chicago, IL, 1988, pp. 274-275. doi: 10.1109/CMPSAC.1988.17185
6. S. Aletan and W. Lively, "Architectural design methodology for supporting high level programming languages," Proceedings. 1988 International Conference on Computer Languages, Miami Beach, FL, USA, 1988, pp. 356-363. doi: 10.1109/ICCL.1988.13084
7. MCKEE S, WEAVER V. 2009. Code Density Concerns for New Architectures. 2009 IEEE International Conference on Computer Design; 2009; Lake Tahoe, CA, USA. p. 459-464.
8. Patterson, D. and Hennessy, J. Computer Architecture A Quantitative Approach, 5th ed. Massachusetts, USA: Waltham, 2012, pp. 148-334.
9. Patterson, D. and Hennessy, J. Computer Organization and Design The Hardware / Software Interface. Massachusetts, USA: Burlington, 2009.
10. D. Patti, A. Spadaccini, M. Palesi, F. Fazzino and V. Catania, "Supporting Undergraduate Computer Architecture Students Using a Visual MIPS64 CPU Simulator," in IEEE Transactions on Education, vol. 55, no. 3, pp. 406-411, Aug. 2012. doi: 10.1109/TE.2011.2180530
11. Intel Corporation, (2010). A Guide to Vectorization with Intel C++ Compilers [Online]. Available: https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf
12. H. El-Aawar, "An application of complexity measures in addressing modes for CISC- and RISC-architectures," 2008 IEEE International Conference on Industrial Technology, Chengdu, 2008, pp. 1-7. doi: 10.1109/ICIT.2008.4608682
13. B. Holland, G. R. Santhanam, P. Awadhutkar and S. Kothari, "Statically-Informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities," 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), Raleigh, NC, 2016, pp. 79-84. doi: 10.1109/SCAM.2016.23
14. H. El-Aawar, "CISC vs. RISC Hardware and Programming Complexity Measures of Addressing Modes," Proceedings of the 2nd International Conference on Perspective Technologies and Methods in MEMS Design, Lviv, 2006, pp. 43-48. doi: 10.1109/MEMSTECH.2006.288660
15. Seyfarth, R. "Introduction to 64 bit Assembly Language Programming for Linux", School of Computing University of Southern Mississippi Hattiesburg, 2014.
16. Aho, A., et al., Compilers Principles, Techniques, & Tools, 2nd ed. Boston, Pearson Addison Wesley, 2007.
17. Ming, L. and Qixian, C., "A Research for the Optimization of MIPS Instruction Set Simulation," in Computer Science & Education, 2009. ICCSE '09. 4th International Conference, Nanning, 2009, pp. 1886-1888.
18. J. M. Sibigtroth, "Programming fuzzy logic in assembly language," IEEE Technical Applications Conference. Northcon/96. Conference Record, Seattle, WA, USA, 1996, pp. 456-458. doi: 10.1109/NORTHC.1996.564981
19. Lomont, C., "Introduction to Intel Advanced Vector Extensions," 2011.
20. Mckee, S.A. and Weaver, V.M., "Code Density Concerns for New Architectures," in Computer Design, 2009. ICCD 2009. IEEE International Conf., Lake Tahoe, CA, 2009, pp. 459-464.
21. Kleir, R.L. and Ramamoorthy, C.V., "Optimization Strategies for Microprograms," in Computers, IEEE Transactions, 1971 Volume: C-20, Issue: 7, pp. 783-794.
22. O. Lempel, A. Peleg and U. Weiser, "Intel's MMX/sup TM/ technology-a new instruction set extension," Proceedings IEEE COMPCON 97. Digest of Papers, San Jose, CA, USA, 1997, pp. 255-259. doi: 10.1109/CMPCON.1997.584723
23. M. Lingling, Q. Xiaojie, Z. Zhihong, Z. Gang and X. Ying, "An Assessment Tool for Assembly Language Programming," 2008 International Conference on Computer Science and Software Engineering, Hubei, 2008, pp. 882-884. doi: 10.1109/CSSE.2008.111
24. Steven Holzner and Peter Norton Computing, Inc., "Advanced Assembly Language". New York, NY, 1991.
25. NASM Development Team (2015) "The Netwide Assembler: NASM" [Online]. Available: http://www.nasm.us/doc/nasmdoc1.html
26. B. Blaha and D. Wunsch, "Evolutionary programming to optimize an assembly program," Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600), Honolulu, HI, USA, 2002, pp. 1901-1903 vol.2. doi: 10.1109/CEC.2002.1004533
27. Imagination Technologies, Inc. "MIPS Architecture for Programmers Volume I-A: Introduction to MIPS64 Architecture rev.6.01". Sunnyvale: Imagination Technologies Inc., 2014
28. D. Fujiwara, N. Ishiura, R. Sakai, R. Aoki and T. Ogawara, "Reverse Engineering from Mainframe Assembly to C Codes in Legacy Migration," 2016 5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI), Kumamoto, 2016, pp. 1058-1063. doi: 10.1109/IIAI-AAI.2016.37
29. Intel Corporation. "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2". Intel Corporation, 2015"
30. Kusswurm, D. "Modern x86 Assembly Language Programming", CreateSpace Independent Publishing Platform, 13 Jul 2017

AUTHORS PROFILE

Jonathan Paul C. Cempron. A Graduate of BS Computer Science major in Computer Systems Engineering in De La Salle Unversity. Research focus on Computer Architecture and Computer Vision**.**

Chudrack Shalym Y. Salinas. A Graduate of BS Computer Science major in Computer Systems Engineering in De La Salle Unversity. Research focus on Computer Architecture.

Roger Luis Uy. Assistant Professor, Computer Technology Department, College of Computer Studies, De La Salle University, Manila Philippines. Research interest includes applying concepts of Computer Architecture to other interdisciplinary fields such as Bioinformatics**.**