Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 7 (21): 3217-3225, 2007
ISSN 1812-5654

A New Technique to Calculate the Exact Process Execution Time with the Help of the Compiler

[1]Ehsan Danesh and [2]Amir Masoud Rahmani
[1]Department of Computer Engineering, Islamic Azad University, Karaj Branch, Hesarak, Tehran, Iran
[2]Department of Computer Engineering, Islamic Azad University, Science and Research Branch, Hesarak, Ashrafi Isfahani, Poonak Sq., P.O. Box 14515/775, Tehran, Iran

Abstract: The scheduling algorithms using the exact process execution time have not been put into practice and execution time is treated as a random variable and some of the methods are statistically estimate it from past observations and some of them collect more data like source process algorithm for estimation but, all of them try to collect more data in deferent ways to estimate a good time for the future execution time of process and make themselves power full than each other but none of them can change their estimation during the time that process is running and this is the weakness. In this study there is a new technique-with the help of the compiler-for calculating the exact process execution time and adding some new data inside process source code to be useful for other algorithms to make them more power full and more. This will pave the way for executing these scheduling algorithms. Furthermore, using this new technique, a new horizon will be opened in CPU scheduling. Finally, a new scheduling algorithm called Shortest Remaining Cycle will be introduced in which the process itself plays a role in its scheduling.

Key word: Process execution time, shortest remaining cycle, compiler, CPU scheduling, cycle counter

INTRODUCTION

There are two sets of algorithms in process scheduling: The first set includes applicable algorithms which are not optimal, like FCFS (First Come First Served). New processes come in at the end of the ready queue, the process at the head of the queue runs until it either terminates or performs I/O (Christopher *et al*., 2002). Another example is RR (Round Robin) which is similar to FCFS, but with preemption. The first process in the queue runs for one time quantum and then moves to back of the queue and the next process in the queue runs. This cycle continues until all processes terminate (Rawat and Kshemkayani, 2003). The second set of the algorithms are optimal but not applicable because they need to know the process execution time. Therefore, they are just ideas or indefinite algorithms such as SPN (Short Process Next). When a process comes in, SPN inserts it in the ready queue based on its length. When the current process is done, SPN picks the one at the head of the queue and runs it (Harchol-Balter *et al*., 2001).

The problem is how do the algorithms understand how long will it take the process to run? In the literature, there are three major classes of solutions to the execution time estimation problem: Code analysis (Reistad and Gifford, 1994), analytic benchmarking/code profiling (Yang and Gerasoulis, 1994) and statistical prediction (Iverson *et al*., 1996). In code analysis, an execution time estimate is found through analysis of the source code of the task. A given code analysis technique is typically limited to a specific code type or a limited class of architectures. Thus, these methods are not very applicable to a broad definition of heterogeneous computing. A class of methods which are more useful in a heterogeneous environment is analytic benchmarking/code profiling. Analytic benchmarking/code profiling was first presented by Freund (Freund 1989) and has been extended by Pease *et al*. (1991) and Khokhar *et al*. (1993). Analytic benchmarking defines a number of primitive code types. On each machine, benchmarks are obtained which determine the performance of the machine for each code type. Code profiling attempts to determine the composition of a task, in terms of the same code types. The analytic benchmarking data and the code profiling data are then combined to produce an execution time estimate. Analytic benchmarking/code profiling has two disadvantages. First, it lacks a proven mechanism for producing an execution time estimate from the benchmarking and profiling data over a wide range of algorithms and architectures. Second, it cannot easily compensate for variations in the input data set.

Corresponding Author: Amir Masoud Rahmani, Department of Computer Engineering, Islamic Azad University, Science and Research Branch, Hesarak, Ashrafi Isfahani, Poonak Sq., P.O. Box 14515/775, Tehran, Iran
Tel: +98(912)1784956

The third class of execution time estimation algorithms, statistical prediction algorithms, make predictions using past observations. A set of past observations is kept for each machine, which are used to make new execution time predictions. The matching and scheduling algorithm uses these predictions (and other information) to choose a machine to execute the task. While the task executes on the chosen machine, the execution time is measured and this measurement is subsequently added to the set of previous observations. Thus, as the number of observations increases, the estimates produced by a statistical algorithm will improve. Statistical prediction algorithms have been presented by Iverson *et al.* (1996). Statistical methods have the advantages that they are able to compensate for parameters of the input data (such as the problem size) and do not need any direct knowledge of the internal design of the algorithm or the machine. However, statistical techniques lack an intrinsic method of sharing observations between machines.

By allowing observations to be shared between machines, the execution time estimate on a machine with few observations can be improved by using observations from machines with similar performance characteristics. In these three class collecting more data is equal to estimating better execution time for process obviously collecting more data is equal to more over head for an algorithm.

In this research there is an attempt to overcome the above-mentioned problems in order to be able to calculate the exact process execution time. The first section of the paper tries to explain the new technique (algorithm) which is added to the compiler in order to calculate the program execution time this algorithm can be mentioned in first class (code analysis) of three discussed classes but it does not estimating an execution estimation by analyzing source code algorithm it calculating the exact process execution time by following instruction by instruction of source code during compilation and it can be used where ever that the compiler is used so it could be a very good solution for heterogeneous environment like analytic benchmarking/code profiling but with no data collection and no over head during run time. This algorithm can be used with other three classes combinationally to make them more power full. Another weakness of those three algorithm classes is that they can not change their estimation during process run time and they just estimate a single execution time for each execution of the process but those schedulers that uses algorithm of this paper can dynamically change execution time that estimated for them when process change it's manner during run time. Then by using this capability a new CPU scheduling algorithm will be introduced. In the next some basic concepts and definitions will be reviewed. The added technique to the compiler, i.e., the cycle counter algorithm is introduced. Then, a new applicable scheduling algorithm is presented employing the technique introduced and the last section deals with some conclusions.

MATERIALS AND METHODS

Concepts and definitions

Process and program: A process is a program in execution and a program is a set of instructions put together to reach a determined aim (Conway *et al.*, 1967).

CPU burst: Part of the process which needs a processor in order to be run.

I/O burst: Part of the process which does not need a processor and should come out of the processor to perform some I/O (The processor should have reached to an I/O instruction). So a process will run for a while (the CPU burst), perform some I/O (the I/O burst), then run for a while more (the next CPU burst).

Cycle counter: The name of the program added to the compiler in order to calculate the process execution cycles.

Branch instructions: Conditional jump instructions, unconditional jump instructions and conditional loop instructions that alter the flow control of program and change the number of execution cycles. Examples are JZ, JNZ, JMP, etc. in Assembly language.

Execution cycles: Each program written in every language is at last converted into machine language which is executed in a determined cycle; the execution time of a process could be achieved according to the number of execution cycles of machine language in a program. Of course, beside this big effort, there is still a major difficulty in calculating the process execution time, i.e., unpredictable situations in program flow control or conditional and unconditional jumps that might alter the program flow control and consequently the number of all the cycles or the process execution time and so being preventive in calculating the process execution time.

Main body: The flow control of a program without considering its branch instructions.

Main cycle: The number of the cycles organizing the main body.

Simple instructions: The instructions organizing the main body of the program which do not lead to branch; examples are mathematical or logical instructions such as OR and, SUM, etc. in Assembly language.

Call instructions: This instruction calls a function to do special job in the program.

Loop instructions: A kind of branch instructions which repeat a special part of the program until a specified expression is evaluated to be false.

Loop counter: A register or a place in a memory which stores in itself the number of the times that a loop instruction is supposed to be executed. In x86 Assembly language, the above-mentioned register is CX (Mazidi and Mazidi, 2000).

Branch cycles: The number of the cycles which are supposed to be added to the main cycle (or substituted with the main cycle) as the result of branching caused by branch instructions (the number of these cycles might be negative).

Attaching: The attachment of the execution cycles to the instructions of the machine language which is realized in different stages.

Marking: A procedure done in attaching to the some of instructions in order to declare special situations about them. This concept will be explained in detail while describing the cycle counter algorithm.

(Short-term) scheduler: A part of the operating system which handles the removal of the running process from the CPU and the selection of the next process, based on a particular strategy.

Cycle counter algorithm: The running process has just CPU burst and I/O burst and the processor runs only the process that is in its own CPU burst. So what the coming algorithms try to do, is to calculate the number of instructions between I/O, because when the process performs its I/O, it comes out of the CPU and does not need to be scheduled any more. Furthermore, while finishing its I/O and returning to the CPU, the process is being treated as a new one. In this algorithm, the compiled program which has no logical or semantic problems and is converted to the lowest level of the machine language is given to the cycle counter algorithm.

The cycle counter algorithm is consisted of five sections with the following titles:

- Simple Instructions,
- Branch Instructions,
- Loop Instructions,
- Call Instructions,
- I/O Instructions,
- All of them are explained later.

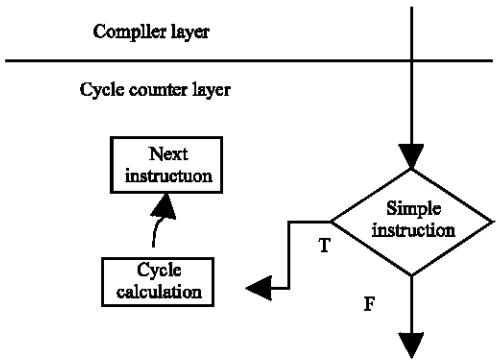

Fig. 1: Behavior of part of the algorithm that reaches simple instructions as mentioned in simple instruction section

Simple instructions: After compiling, the program is given to the cycle counter algorithm in order to calculate its number of the cycles. Then, the cycle counter checks the instructions line by line to recognize the sort of instructions. In Fig. 1, the cycle counter checks whether the instruction is simple or not. This section is one of the simplest parts of the algorithm, because if the condition is true, it just calculates the number of the cycles in a specific instruction, then it goes to the next instruction to recognize it. In this case, if the condition is false, the cycle counter goes to the next part of the algorithm to recognize another kind of instruction without calculating the number of cycles.

Branch instructions: Due to the job they do, branch instructions are so important, because the way of dealing with them, plays a significant role in CPU scheduling. Since a branch instruction jumps over other instructions, it might alter the process flow control. A branch could be one of the four cases below:

- Simple backward conditional jump instruction,
- Simple forward conditional jump instruction,
- Forward conditional jump instruction through I/O,
- Backward conditional jump instruction through I/O.

The first two cases are described here and the last two cases would be discussed. It is worth-mentioning that the number of cycles of branch instructions differ whether the condition is true or false, in this regard, it is necessary to add the number of the branch instruction cycles with main cycle while the condition is false, then it is the time to add the number of the branch instruction cycles-while the condition is true-with the branch cycles supposed to be attached to the branch instruction. In addition, as a branch instruction is

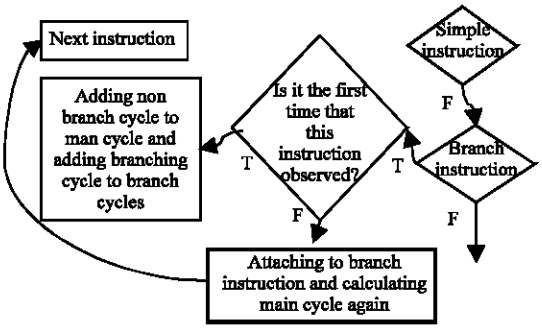

Fig. 2: Behavior of part of the algorithm that reaches branch instructions as mentioned in branch instructions section

being recognized, it is firstly checked whether it is the first or the second time that this instruction is being observed. Recognition for the second time means an end of the branch cycle calculation, in Fig. 2, it is the time to the attaching of the calculated cycle and continuing the main cycle calculation. In addition, the first recognition means that it is the first time that the cycle counter reaches the branch; and as a result, it is a necessity to calculate the branch cycles. From the now on, it is supposed that the cycle counter has reached the branch instruction for the first time.

Note: while in a forward or backward jump, it is possible to jump over another branch instruction. In such case, the cycle-without satisfying the condition-would be added with the calculated branch cycles.

Simple backward conditional jump instructions: According to the Fig. 3, the cycle counter, after recognizing the jump instruction and calculating branch instruction cycle-while the condition is true or false-determines whether the instruction is simple backward conditional jump instruction or not? If the condition is true, the cycle counter goes to the jump target, leaves the calculation of the main cycle, (because while jumping backward, it goes back to instructions whose main cycle have been calculated) and starts calculating the branch cycles (line by line and instruction by instruction) in order to attach it to the jump instruction, until it reaches itself, then, the cycle counter begins attaching according to Fig. 2.

Simple forward branch instructions: According to Fig. 4, if the condition of the backward jump is false, the cycle counter recognizes the instruction of the type forward jump and in this stage, unlike the previous one, begins to

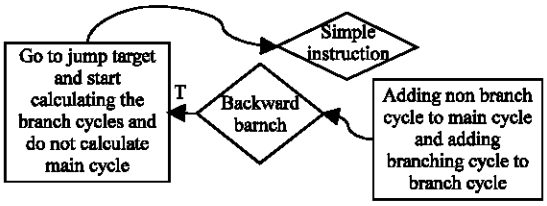

Fig. 3: Behavior of part of the algorithm that reaches simple backward conditional jump instructions as mentioned in simple backward conditional jump instructions section

calculate the branch cycles in addition to the main cycle; it does so from the next instruction to reach the jump target. Since there is a forward jump while the program is being executed in the CPU, so, it can be inferred that the branch instruction jumps even over a situation of itself with no special condition. Therefore, this number of cycles should be added with the branch cycles while calculating the number of branch cycles and then, if I/O instructions would not be recognized inside the branch, the negative number should be attached to the branch instruction.

Loop instructions: The cycle counter might find one of these three cases by encountering loop instructions:

- The Loop Counter may alter inside the loop,
- The Loop Counter may not alter inside the loop,
- There is an I/O instruction inside the loop.

In the first case the loop is being treated like a backward branch instruction (Fig. 2). It means that the number of the cycles of the loop statement is being calculated once and would be attached to it without considering the Loop Counter. Indeed, this kind of loop is considered as a branch instruction.

In the second case, according to Fig. 5, the number of the cycles of the loop statement is calculated once like simple backward branch instructions and then it is multiplied by the Loop Counter:

$$\text{Loop instruction cycles} = \text{Cycles of a loop statement} * \text{Loop Counter} \quad (1)$$

The Loop Counter parameter in the formula 1 can be attached to the loop instruction in the form of both a variable and a constant number. In this case, if the number of Loop Counter is available at the moment of loop cycle calculation, the loop instruction cycles is being calculated and attached to the loop instruction. Otherwise, the Loop

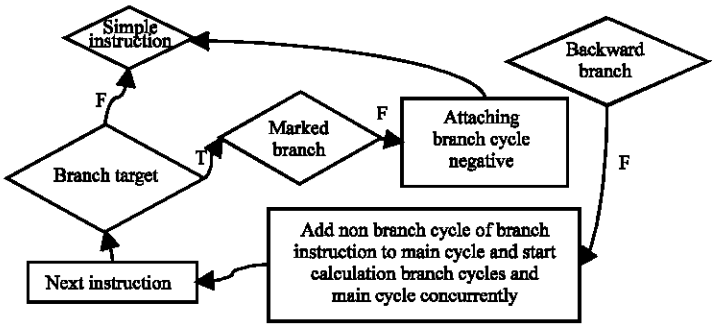

Fig. 4: Behavior of part of the algorithm that reaches simple forward branch instructions as mentioned in simple forward branch instructions section

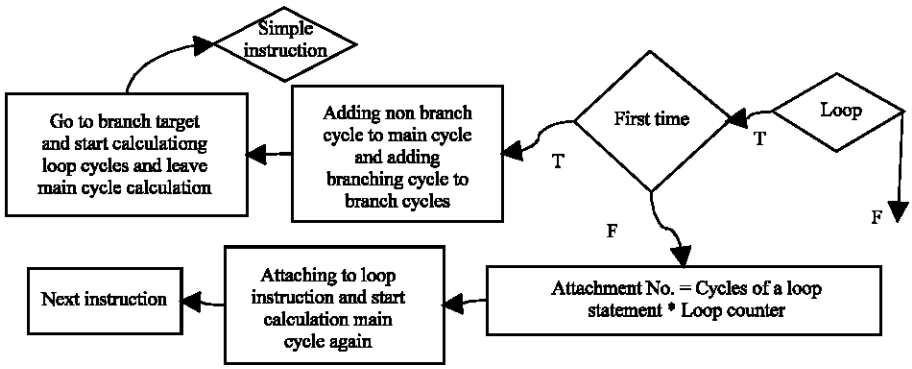

Fig. 5: Behavior of part of the algorithm that reaches loop instructions as mentioned in loop instructions section

Counter is attached to the instruction as a variable multiplied by a constant number in order to be calculated while executing, then the loop instruction cycles would also be determined based on formula 1.

Call instructions: The way the number of the cycles of a function is calculated, entirely depends on the existence of I/O instructions within it, so the calculations are two types:

- The function without I/O.
- The function including I/O.

When the function does not contain I/O instructions, it is treated like a program without I/O having a main cycle and might include some branch cycles. Therefore, it is treated like a separate program and its main cycle is added with the main cycle of the program and the branch cycles of branch instructions inside it, is attached to themselves (Fig. 6).

I/O instructions: All the I/O instructions would be explained in this section. These instructions are accompanied with I/O, causing the calculated execution cycles change until the next I/O, so in order to calculate its cycle, previous methods can not be used. Therefore, these instructions-in the calculation of which I/O plays a significant role-should be made distinct from other instructions by marking, in order to be recognized while executing. In this case, the alter cycle could be calculated and consequently be used.

Simple I/O: The simple I/O has two properties:

- No branch or loop instructions effect,
- Not being the last I/O instruction within a function.

When the cycle counter reaches a simple I/O, it means the cycle between two I/Os is calculated and should be attached to the previous I/O instruction according to the Fig. 7, now it is time for the cycle counter to begin a new counting between the new I/O instruction and the next one.

I/Os within a branch instruction: If the cycle counter reaches an I/O within a branch instruction two cases happen:

The branch instruction has forward jump, so in this case the cycle counter goes to the next instruction and adds the instruction's cycles with both the main cycle and branch cycles (Fig. 4). In Fig. 8, by reaching the first I/O inside the branch instruction, the cycle counter marks that branch instruction and stops counting its cycle until reaching jump target, but during this stage, the cycle counter does not stop main cycle calculation between I/Os and it just stops branch cycle calculation and when the cycle counter reaches jump target, it starts calculating a new branch cycles for branch instruction until it reaches first I/O after jump target, then it attaches the number of the cycles from jump target to the first I/O after jump instruction to the branch instruction. In this case, as the program executes the branch instructions, the scheduler-noticing the marked instruction-deletes the number of calculated main cycle and substitutes it with the new attached branch cycles, with no addition or subtraction.

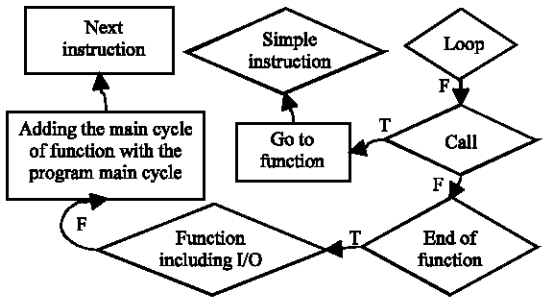

Fig. 6: Behavior of part of the algorithm that reaches the function without I/O as mentioned in function without I/O section

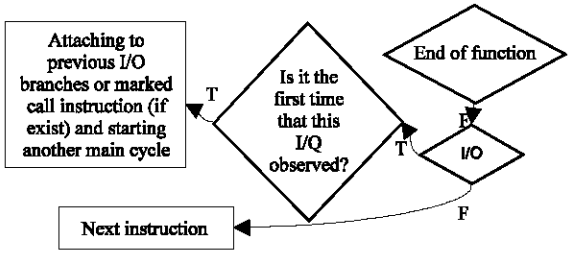

Fig. 7: Behavior of part of the algorithm that reaches simple I/O as mentioned in simple I/O section

The branch instruction is a loop or has a backward jump, so in this case, (Fig. 9), after returning to the jumping place, the cycle counter starts counting the branch cycles, without dealing with the main cycle and then reaching at the first I/O. It marks the branch instruction and calculates the cycles to that I/O and attaches it to the branch instruction. The remaining calculating procedure is like simple forward conditional jump instruction.

The last I/O instruction inside a function: While calling a function, the cycle counter goes to the function and begins to calculate the main cycle of function as well as to add the program's instruction cycle. If the cycle counter reaches an I/O inside a function, (Fig. 8), it stops the calculation and attaches the calculated cycle to the previous I/O outside the function. It also marks the call instruction, (Fig. 10) and calculates the cycles between the function's I/O and attaches them to I/O instruction. The major problem appears when the cycle counter reaches the last I/O inside the function, because due to the calling of a function in the different sections of a program, the cycle counter does not understand which number to attach to the last I/O inside the function. It is because of the fact that by calling in the different sections, the number of instructions between this I/O and other I/Os changes continuously, although attaching more than one number to an instruction would be impossible. This problem also exists for the branch instruction inside the function that jumps forward over

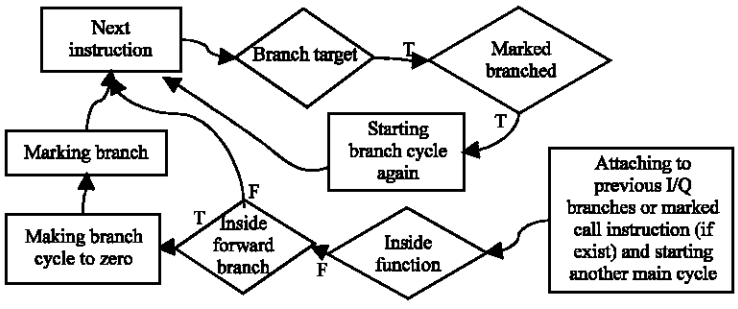

Fig. 8: Behavior of part of the algorithm that reaches I/O within the branch instruction that has forward jump as mentioned in I/O within a branch instruction section

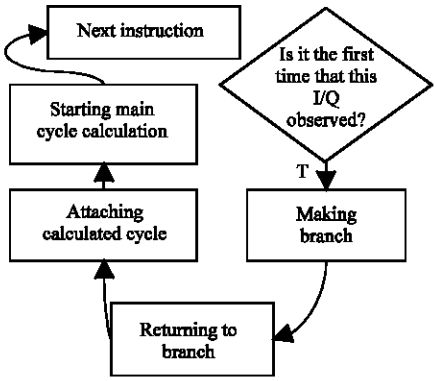

Fig. 9: Behavior of part of the algorithm that reaches I/O within the branch instruction that has backward jump as mentioned in I/O within a branch instruction section

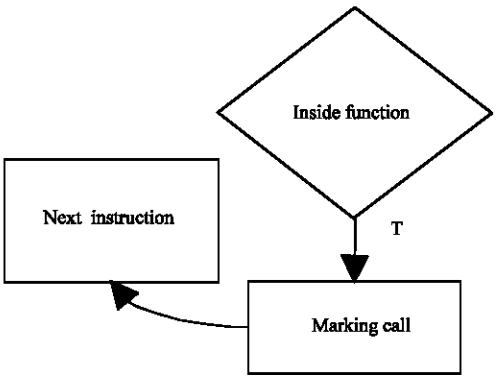

Fig. 10: Behavior of part of the algorithm that reaches first I/O within a function as mentioned in last I/O instruction inside a function section

the last I/O. Before solving such a problem, it is necessary to explain the kind of marking, known as second mode marking. This sort of marking is just applicable to those branch instructions inside a function, that have jumped forward over the last I/O. As the cycle counter reaches the end of a function, it firstly marks the last I/O and then attaches to it the number of the calculated cycles from that I/O to the last instruction. If there exists a branch instruction which have jumped forward over an I/O, the number of cycles from the jumping place to the last instruction of function is calculated-like the previous algorithm-and in addition to attaching this number to that branch instruction, it is marked by the second mode marking. While executing, this kind of marking suggests that the instruction is within the function and its attached number should be added with the attached number of the call instruction in a function.

The number of the attached cycle to call instructions is equal to the main cycle that is calculated by the cycle counter immediately after recognizing a function with an I/O and returning from the function call to the main program until reaching the first next I/O, (Fig. 11).

In this regard, as the program reaches the last I/O inside a function or the last forward branch over I/O and enters it, the scheduler adds the number of the attached cycles with the number of the cycles attached to the call instruction of that function and calculates the number of the cycles to the next I/O.

It is worth-mentioning that attaching the cycles to the instructions, as well as calculating the number of the cycles while the program is running, should be done atomically in order to prevent from scheduling again.

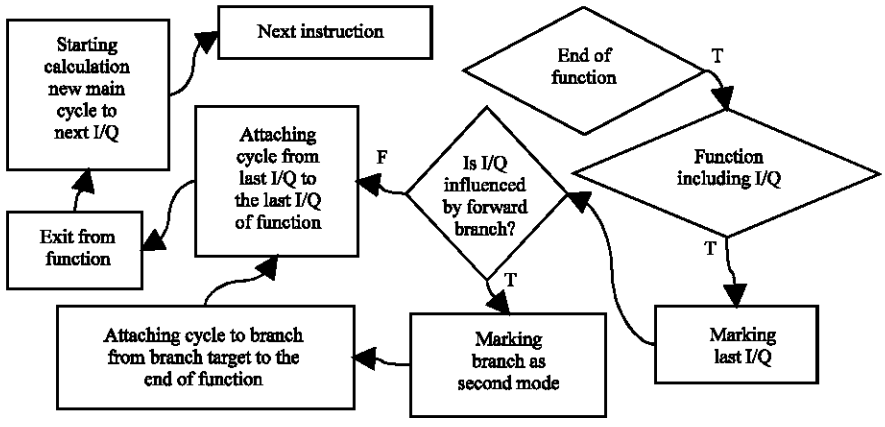

Fig. 11: Behavior of part of the algorithm that reaches last I/O inside function as mentioned in last I/O instruction inside a function section

RESULTS AND DISCUSSION

Shortest remaining cycle: A new scheduling algorithm: In this section, a new scheduling algorithm called SRC (Shortest Remaining Cycle) will be explained which is based on the cycle counter algorithm; it means that this scheduling algorithm just works with those processes whose cycles numbers are calculated by the cycle counter algorithm and attached to their instructions. SRC schedules the processes based on their process execution time, giving priority to them.

Like the previous parts, the cycle counter algorithm starts calculating the number of the cycles between two I/Os (CPU burst) in a program. SRC scheduling considers the fragmented codes existing between two I/Os as a new process and puts it in a group of processes to be scheduled. SRC allocates CPU to the process which has the least number of attached cycles. It is worth-mentioning that the cycle which is given to the scheduler is the main cycle of the program itself. After allocating CPU to the process, SRC immediately preempts the CPU for the new process according to one of the following factors:

- Entering of a process to the ready queue with the shorter number of cycles compared with the remaining cycles of the current process,
- Execution of an I/O instruction,
- Referring to a branch which increases the number of the cycles of a process and consequently decreases the priority of that process to other processes in the queue.

By considering the above-mentioned factors, starvation problem occurs for the processes with a long execution cycles, for which an aging solution should be clarified. One solution is to reduce n cycles of the remaining cycles for each process in every m number of CPU cycles (m >> n) by increasing the priority of processes; so the fourth factor could be defined as the following:

- Reaching of a process in the ready queue to a priority higher than the priority of the current process.

It is important to mention that the only parameter in defining priority in this suggested algorithm is the number of the cycles in a program. For instance, it is impossible to discover the remaining time of a process, only on the basis of the total number of the cycles of the processes existing in the ready queue. The reason is that in a long time execution, due to the execution of the aging solution, it would be possible for some process to have the same number of cycles or zero cycles (note: the number of the execution time in a program for scheduling would never be negative). On the other hand, if multiple processes with the same priority are runnable, one of the previous methods of scheduling like FCFS could be used.

An advantage of using this new scheduling algorithm is that, running on CPU is fairly given to the processes themselves, until they could do their scheduling through their environmental conditions. In fact, who is more suitable to decide on a CPU scheduling than the process itself?

CONCLUSION

The aim of this research is the optimal utilization of the existing facilities, like compiler, with the least changes-like adding the cycle counter algorithm-in order to define a new technique for using in different applications. Some facilities (calculated cycle of each part of the process) are added to the processes by compiler, which can be used in CPU scheduling directly or with other data for solving deadlock problems in OS's that could be the future work of the algorithm, this algorithm can play role in heterogeneous environments with a very low overhead during compiling and no over head during runtime (specially when hardware support is being used) because it is an architecture free algorithm and is not relate to any special architecture, another ability of the algorithm is that it can change it's estimation with the help of the process itself during run time if process manner changes, the ability that the other algorithms do not have it, so a new scheduling algorithm will be introduced in which the process itself plays a role in its scheduling. On the other hand, this new technique could be used for opening new horizons to better solutions and more modern possibilities in the future operating systems.

REFERENCES

Christopher, D., Gill and Ron K. Cytron Douglas and C. Schmidt, 2002. Multi-paradigm scheduling for distributed real-time embedded computing. Proc. IEEE,. 91: 183-197.

Conway, R.W., W.L. Maxwell and L.W. Miller, 1967. Theory of Scheduling, Addison Wesley.

Freund, R., 1989. Optimal Selection Theory for Superconcurrency. In: Proceedings of Supercomputing Conference. IEEE Comput. Soc. Press, pp: 13-17.

Harchol-Balter, M., N. Bansal, B. Schroeder and M. Agrawal, 2001. SRPT Scheduling for Web Servers. Lecture Notes in Comput. Sci., 2221: 11-21.

Iverson, M.A., F.O¨ Zgu¨ner and G. Follen, 1996. Run-time Statistical Estimation of Task Execution Times for Heterogeneous Distributed Computing. In: Proceeding of High Performance Distributed Computing Conference, pp: 263-270.

Khokhar, A.A., V.K. Prasanna, M.E. Shaaban and C.L. Wang, 1993. Heterogeneous computing: Challenges and opportunities. IEEE Comput., 26: 18-27.

Mazidi, M.A. and J.G. Mazidi, 2000. The 80×86 IBM PC and Compatible Computers. 3rd Edn. Prentice Hall.

Pease, D., A. Ghafoor, I. Ahmad, D.L. Andrews, K. Foudil-Bey, T.E. Karpinski, M.A. Mikki and M. Zerrouki, 1991. PAWS: A performance evaluation tool for parallel computing systems. IEEE Comput., 24: 18-29.

Rawat, M. and A. Kshemkayani, 2003. SWIFT: Scheduling in web servers for fast response time. In 2nd IEEE International Symposium on Network Computing and Applications. IEEE Comput. Soc., pp: 51.

Reistad, B. and D.K. Gifford, 1994. Static dependent costs for estimating execution time. In: Proceeding of ACM Conference on LISP and Functional Programming, pp: 65-78.

Yang, T. and A. Gerasoulis, 1994. DSC: Scheduling tasks on an unbounded number of processors. IEEE Trans. Parallel and Distributed Syst., 5: 951-967.