Ahead-of-Time (AoT) Instruction Scheduling

M Shifat Hossain
Dept. of Electrical and Computer Engineering
University of Central Florida
Orlando, United States
mshifat.hossain@ucf.edu

Tasbiraha Athaya
Dept. of Computer Science
University of Central Florida
Orlando, United States
tasbiraha.athaya@ucf.edu

Rony Chowdhury Ripan
Dept. of Computer Science
University of Central Florida
Orlando, United States
ronychowdhury.ripan@ucf.edu

M Iffat Hossain
Dept. of Electrical and Computer Engineering
University of Central Florida
Orlando, United States
miffat.hossain@ucf.edu

Maruf Chowdhury
Dept. of Computer Science
University of Central Florida
Orlando, United States
maruf.chowdhury@ucf.edu

Fahimul Aleem
Dept. of Computer Science
University of Central Florida
Orlando, United States
fahimul.aleem@ucf.edu

***Abstract*—Instruction scheduling is an optimization technique that involves arranging the order of instructions in a program to ensure optimal usage of the CPU. Dynamic scheduling occurs during runtime within the processor hardware, allowing instructions to be executed out of order based on resource availability and dependencies. On the other hand, compiler time scheduling involves rearranging the order of instructions at compile time to optimize performance. Dynamic scheduling requires complex hardware support like reorder buffers and register renaming, which can increase the complexity and cost of processor design. Compiler time scheduling's effectiveness is limited by the compiler's ability to analyze and optimize code, and it may not fully adapt to runtime variations in workload or hardware conditions. To address these issues of dynamic and compile-time scheduling, we proposed ahead-of-time (AoT) instruction scheduling. Our AoT scheduler takes an executable binary file as input that was produced by a compiler and reorders instructions within a basic block utilizing a directed acyclic graph (DAG). We evaluated our AoT scheduler's instruction per cycle (IPC) and compared it with Tomasulo's algorithm varying different queue sizes and demonstrated that our AoT scheduler outperforms Tomasulo in any heuristics.**

***Index Terms*—Dynamic Scheduling, Compile-Time Scheduling, AoT Scheduling, Tomasulo's Algorithm, Directed Acyclic Graph**

I. Introduction

Instruction scheduling is an optimization technique that involves arranging the order of instructions in a program to ensure optimal usage of the CPU [1]. This optimization not only minimizes resource conflicts but also alleviates branch delays and reduces stalls in the pipeline, ensuring a smoother flow of instructions through the processor [2]. Ultimately, instruction scheduling plays a crucial role in enhancing overall program performance by orchestrating the execution sequence in a manner that optimally utilizes the available hardware resources [1].

A simple DLX pipeline can be segmented into five stages. These are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB) [3]. In the IF stage, instructions are fetched from memory. The ID stage decodes the fetched instruction, determining the operation to be performed and the operands involved. The EX stage executes arithmetic, logic, or control operations based on the decoded instruction. The MEM stage handles memory access, such as data loads and stores. Finally, the WB stage writes the results of the instruction execution back to the register file. This pipeline design enhances processor throughput by allowing multiple instructions to be processed concurrently, with each stage working on a different instruction simultaneously. Despite its advantages, a simple DLX pipeline architecture also has certain disadvantages. Firstly, data-dependent instructions can introduce stalls in the pipeline, as subsequent instructions must wait for the resolution of data dependencies before they can proceed, impacting overall performance [4]. Secondly, floating-point operations in DLX pipelines typically incur longer latencies compared to other types of operations, potentially leading to pipeline bubbles and reduced throughput. Lastly, the presence of long serial dependence chains within certain instruction sequences can limit the exploitation of parallelism, even with short latencies between pipeline stages.

Dynamic scheduling mitigates the disadvantages of a simple DLX pipeline. Dynamic scheduling occurs during runtime within the processor hardware, allowing instructions to be executed out of order based on resource availability and dependencies [5]. This flexibility can improve performance by exploiting parallelism and reducing stalls caused by dependencies or hazards [5]. However, dynamic scheduling requires complex hardware support like reorder buffers and register renaming, which can increase the complexity and cost of processor design [6].

On the other hand, compiler time scheduling involves rearranging the order of instructions at compile time to optimize performance. This approach leverages static analysis of code to identify instruction-level parallelism and reorder instructions to minimize stalls and improve pipeline efficiency [7]. Compiler time scheduling can produce efficient code without the need for complex hardware, potentially leading to more predictable performance characteristics [7]. However, its

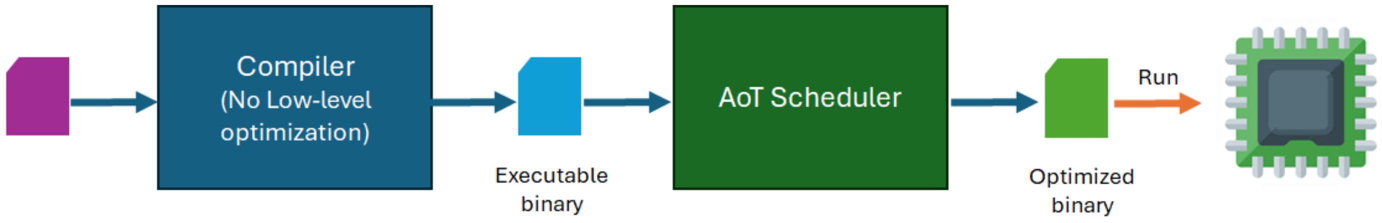


Fig. 1: Project Workflow with AoT system

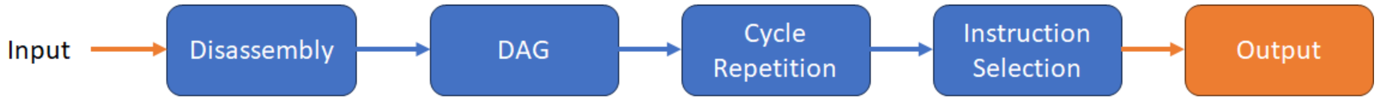


Fig. 2: Workflow diagram of Ahead-of-Time (AoT)

effectiveness is limited by the compiler's ability to analyze and optimize code, and it may not fully adapt to runtime variations in workload or hardware conditions [8].

So there are disadvantages in both dynamic and compile-time instruction scheduling. In this study, we developed a new instruction scheduling technique called Ahead-of-Time (AoT) instruction scheduling that schedules instructions before the actual execution of the program. The objectives of our AoT scheduler are multifaceted. Firstly, it aims to schedule instructions preemptively before program execution to enhance overall efficiency. This involves optimizing scheduling performance, ensuring optimal resource utilization, reducing dependencies between instructions, and improving energy efficiency in instruction execution. Additionally, the system is designed to be adaptive, accommodating various target architectures to effectively leverage hardware capabilities and constraints. These objectives collectively drive the development of a comprehensive instruction scheduling solution tailored to enhance program execution and resource utilization while minimizing energy consumption and dependency issues. To achieve these objectives, our AoT scheduler takes input an executable binary file that was produced by a compiler and reorders instructions within a basic block utilizing a directed acyclic graph (DAG). We evaluated our AoT scheduler's instruction per cycle (IPC) and compared it with Tomasulo's algorithm varying different queue sizes. The contribution of our work can be summarized as follows:

- We propose a new instruction scheduling called ahead-of-time (AoT) instruction scheduling.
- We utilize directed acyclic graph (DAG) to optimize instructions scheduling in a basic block of code.
- We evaluated our AoT scheduler varying different queue sizes and compared it with State of the art Tomasulo Algorithm.

The remainder of the article is structured according to the following. Section II provides the related work of various types of instruction scheduling. In Sections III and IV, we present our design and implementation of our AoT scheduler. In Section V, we discuss our results and finally conclude this article in Section VI.

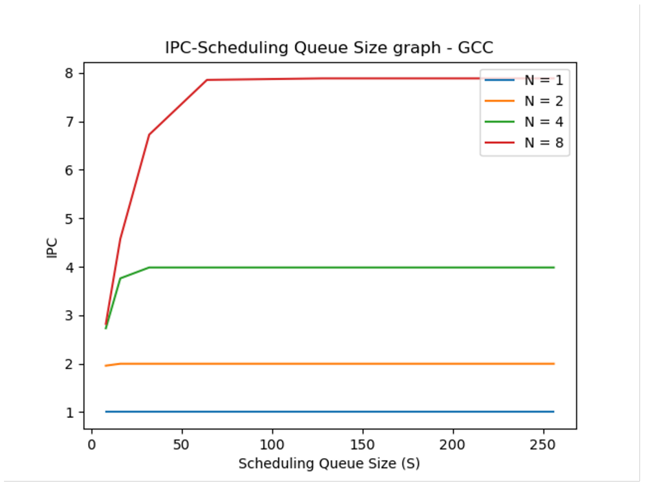
II. Background

Instruction scheduling has been a subject of extensive research and development for several decades, resulting in a vast body of literature encompassing various techniques and optimizations. Tomasulo's Algorithm (1967) developed by Robert Tomasulo, this seminal work introduced a dynamic scheduling algorithm for out-of-order execution, utilizing reservation stations and register renaming to handle dependencies and maximize ILP [9]. It laid the foundation for many subsequent advancements in dynamic scheduling. Gibbons and Muchnick explored the problem of optimal code scheduling for single-issue processors, focusing on minimizing execution time under resource constraints. They introduced the concept of list scheduling and provided theoretical bounds for its performance [10].

Trace Scheduling technique extended instruction scheduling beyond basic block boundaries by scheduling instructions along frequently executed paths, improving performance by considering global program flow (Fisher, 1981)[11]. Reorder Buffer mechanism allowed instructions to be fetched and executed out of order, buffering the results until they could be committed in the original program order, further enhancing ILP and performance (Smith and Pleszkun, 1985)[12]. Several studies investigated the potential of combining compile-time and dynamic scheduling techniques to leverage the strengths of both methods, aiming for a balance between predictability and adaptability. This paper focuses on investigating AoT instruction scheduling techniques, specifically targeting basic blocks within a program. By analyzing data dependencies and utilizing heuristic algorithms, we aim to determine an optimal instruction order within each basic block to enhance execution efficiency.

III. Design

The complete workflow of this process, shown in Figure 1, starts with the source code written in a high-level programming language (i.e., C/C++, Rust, Cython, etc.). This code is fed into a compiler that translates into an executable binary file.



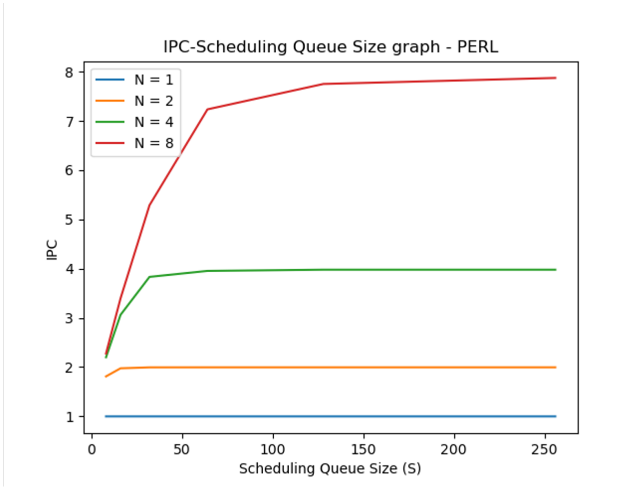


Fig. 3: Results of the simulation using GCC (left) and PERL (right)

The generated executable binary then enters our proposed AoT scheduler. The AoT scheduler then analyzes the binary code and applies its algorithms to rearrange instructions within basic blocks for optimized execution.

The output of the AoT scheduler is an optimized binary file where instructions have been strategically reordered. This optimized binary can now be executed on the target CPU.

IV. Implementation

The implementation of the complete AoT process follows Figure 2. Primarily Tomasulo's dynamic scheduling simulator was created, where the fetch rate is set to N (1,2,4 and 8) and the Scheduling queue is to S (8, 16, 32, 64, 128, and 256). The simulation model uses three operations: Type 0, which takes one cycle; Type 1, consisting of two cycles; and Type 2, which takes five cycles. The results of the simulations were acknowledged first without the use of the proposed AoT system as shown in Figure 3 and Table I. Figure 3 shows the IPC vs scheduling queue size (S) for different fetch rates (N) for default Tomasulo's architecture. Table I shows the minimum scheduling queue size (S) that achieves within 5% of the IPC of S=256. It is evident that, close to maximum S performance is possible with very small number of S for different fetch rates.

TABLE I: Optimized scheduling queue size per peak fetch rate

Optimized scheduling queue size per peak fetch rate		
	Benchmark = GCC	**Benchmark = Perl**
N=1	3	4
N=2	7	10
N=4	17	30
N=8	48	81

At the beginning of the implementation of the proposed AoT instruction scheduling approach, the compiled executable is disassembled to analyze the instructions and their dependencies, and basic blocks are identified within the program. Then for each basic block, a Directed Acyclic Graph (DAG) is generated, an example of which is shown in Figure 4.

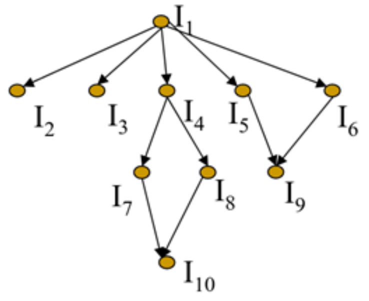
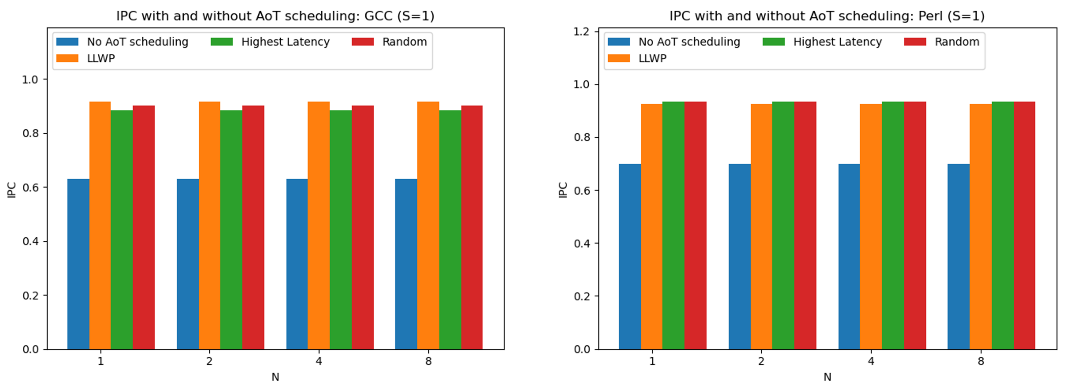


Fig. 4: An example of the creation of DAG

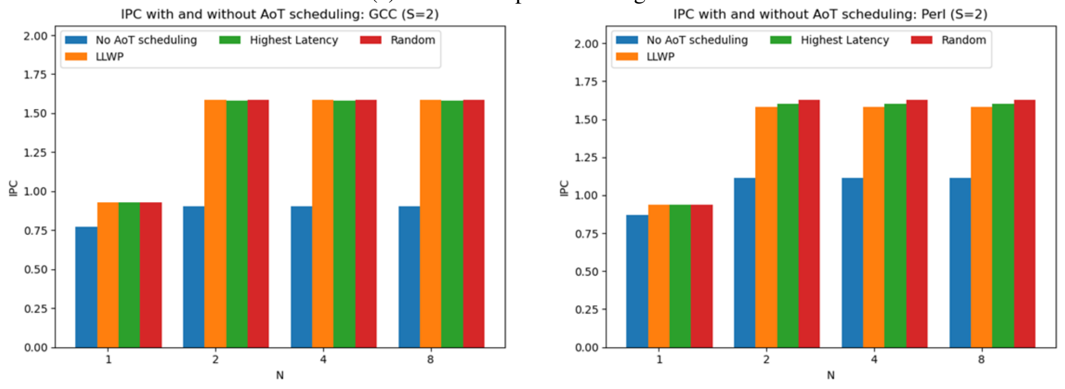
To be noted, Nodes in the DAG represent instructions, and edges represent dependencies (WAR, RAW, and WAW hazards) between them. Then it iterates through cycles, scheduling one or more instructions in each cycle based on the chosen heuristic. Three heuristics were applied in implementing the proposed method: LLWP (Longest Latency, Widest Path): Prioritizes instructions with the longest latency and the most successors to minimize potential stalls in the pipeline. Highest Latency: Schedules instructions with the highest latency first. Random: Selects instructions randomly (used as a baseline for comparison). After that instructions whose dependencies are fulfilled are added to a ready list and become candidates for scheduling in the next cycle. In each cycle, the chosen heuristic is applied to select an instruction from the ready list. The selected instruction is scheduled for execution in the current cycle, and its successors are added to the ready list if their dependencies are met. After continuous repetition, if all instructions are scheduled, then the output binary is generated. The Pseudocode of the algorithm is shown in Figure 6.

V. Evaluation

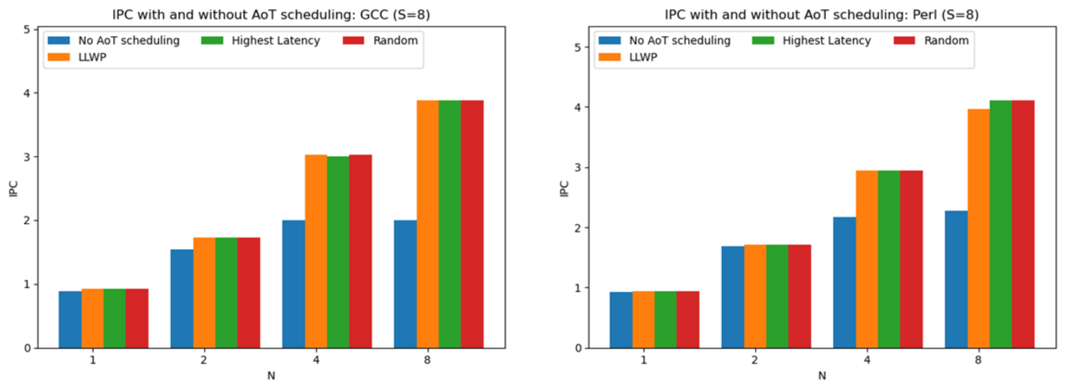
The results of the AoT process are evaluated through different scheduling queue sizes (S=1,2,8 and 256). The process is completed using both GCC and Perl. Figure 5a shows



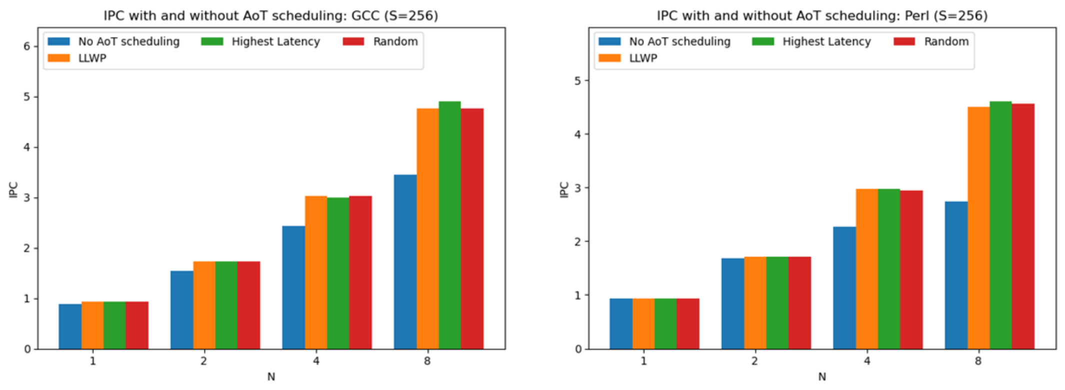
(a) Result comparison using S=1



(b) Result comparison using S=2



(c) Result comparison using S=8



(d) Result comparison using S=256

Fig. 5: AoT instruction scheduling result comparison using different heuristics and scheduling queue sizes.

the result for both GCC and Perl while using S=1, which means Tomasulo's algorithm is not applied in this part. It clearly demonstrates that AoT scheduling performs better in any Heuristics than the standard procedure. By increasing the number of scheduling queue sizes (S=2,8,256), it can be observed that the average Instructions Per Cycle (IPC) gets better with the increase of fetch rate (N) than applying no AoT scheduling. A remarkable difference can be observed where 8 instructions are being fetched at a time. Figure 5b, 5c and 5d shows the results where S=2,8 and 256 respectively.

```
I. Initialization:

  1. for each instruction $i$ in block
       $a$. initialize Avail(i,e) appropriately
       $b$. if $i$ is ready, add it to ReadyL
  2. $\forall\ 0 \leq c <$ MaxExecTime, $W[c] \leftarrow \emptyset$
  3. cycle $\leftarrow$ 1


II. Repeat until all instructions are scheduled:

  1. $i \leftarrow$ ChooseInstr(cycle, ReadyL, DAG)
  2. Start($i$) = cycle
  3. for each outgoing edge $e : i \rightarrow s$
       $a$. Avail(s,e) $\leftarrow cycle + ExecTime(i)$
       $b$. if Avail(s,e) has been initialized for all edges coming in to $s$:
            i. c $\leftarrow$ MAX$_e$ Avail(s,e)
            ii. c $\leftarrow$ c MOD MaxExecTime
            iii. $W[c] \leftarrow W[c] \cup s$
  4. cycle $\leftarrow$ cycle + 1
  5. ReadyL $\leftarrow$ ReadyL $\cup$ $W$[cycle mod MaxExecTime]
  6. $W$[cycle mod MaxExecTime] $\leftarrow \emptyset$
```

Fig. 6: Pseudo-code Algorithm of AoT implementation

VI. Conclusion

Ahead-of-time (AoT) instruction scheduling presents a promising approach to optimize instruction execution without the need for specialized hardware or complex compilation processes. One of the key advantages of AoT scheduling is its ability to eliminate the need for complex dynamic scheduling hardware. This makes it particularly well-suited for resource-constrained environments such as embedded systems, where minimizing hardware complexity and power consumption are crucial considerations. Additionally, by shifting the scheduling process to an earlier stage, AoT reduces the burden on compilers for low-level optimizations, potentially leading to faster compilation times. While the initial results of the AoT scheduling algorithm are promising, further research is necessary to explore its full potential. This includes investigating more sophisticated scheduling heuristics that consider factors like register pressure, branch prediction, and cache behavior. Additionally, evaluating the algorithm on a wider range of architectures and workloads, including those with complex instruction sets and irregular memory access patterns, will provide a more comprehensive understanding of its effectiveness and limitations.

Acknowledgment

We would like to express our sincere gratitude to Professor Jongouk Choi for his invaluable support and guidance throughout the course of this project. His expertise and insights have been instrumental in shaping our research and achieving our objectives. We would also like to extend our appreciation to our teaching assistants, Noureldin and Sevitha. Their feedback and contributions have been highly valuable in enhancing the quality of our work.

References

[1] P. Faraboschi, J. A. Fisher and C. Young, ”Instruction scheduling for instruction level parallel processors,” in Proceedings of the IEEE, vol. 89, no. 11, pp. 1638-1659, Nov. 2001, doi: 10.1109/5.964443.

[2] M. Alipour, S. Kaxiras, D. Black-Schaffer and R. Kumar, ”Delay and Bypass: Ready and Criticality Aware Instruction Scheduling in Out-of-Order Processors,” 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 2020, pp. 424-434, doi: 10.1109/HPCA47549.2020.00042.

[3] Shen, J. P., & Lipasti, M. H. (2005). Modern Processor Design: Fundamentals of Superscalar processors. McGraw-Hill Higher Education.

[4] Sprangle, E., & Carmean, D. (n.d.). Increasing processor performance by implementing deeper pipelines. ACM Digital Library. https://doi.org/10.1145/545214.545219

[5] A. Bonasu, S. R. Karmunchi and N. Wang, ”Design of Efficient Dynamic Scheduling of RISC Processor Instructions,” 2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), Vancouver, BC, Canada, 2020, pp. 0236-0240, doi: 10.1109/IEMCON51383.2020.9284902.

[6] Alshehri, Y.A.; Mordhah, N. Use Dynamic Scheduling Algorithm to Assure the Quality of Educational Programs and Secure the Integrity of Reports in a Quality Management System. Information 2021, 12, 315. https://doi.org/10.3390/info12080315

[7] G. Lj. Djordjević, M. B. Tošić, A Compile-Time Scheduling Heuristic for Multiprocessor Architectures, The Computer Journal, Volume 39, Issue 8, 1996, Pages 663–674, https://doi.org/10.1093/comjnl/39.8.663

[8] T. Hagras and J. Janecek, ”A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems,” 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., Santa Fe, NM, USA, 2004, pp. 107-, doi: 10.1109/IPDPS.2004.1303056.

[9] Tomasulo, R. M. (1967). An Efficient Algorithm for Exploiting Multiple Arithmetic Units. IBM Journal of Research and Development, 11(1), 25-33.

[10] Gibbons, P. B., & Muchnick, S. S. (1986). Efficient instruction scheduling for a pipelined architecture. In Proceedings of the SIGPLAN ’86 Symposium on Compiler Construction (pp. 11-16).

[11] Fisher, J. A. (1981). Trace scheduling: A technique for global microcode compaction. IEEE Transactions on Computers, C-30(7), 478-490.

[12] mith, J. E., & Pleszkun, A. R. (1985). Implementation of precise interrupts in pipelined processors. ACM SIGARCH Computer Architecture News, 13(3), 36-44.

Appendix

These are answers to the questions asked in the project instruction manual:

Q3.A) The goal of a superscalar processor is to achieve an IPC that is close to peak fetch rate (which is the peak theoretical IPC of the processor). Given this goal, please explain the relationship between S and N.

Answer: In Tomasulo's architecture, as we increase the fetch rate (N), the scheduling queue size (S) needs to increase as well to achieve close to ideal performance. This adjustment is necessary because a higher fetch rate implies that more instructions enter the pipeline per cycle. Consequently, the instruction queue must expand to accommodate this increased volume adequately. This expansion ensures optimal storage of all instructions needed, preventing pipeline stalls caused by insufficient space in the instruction queue.

Q3.B) With different benchmark (trace) files, one IPC is higher or lower than the other with the same microarchitecture configuration. What could be reason?

Answer: Benchmark 2 (PERL) exhibits a reduced IPC across all microarchitecture configurations compared to benchmark 1 (GCC). This variance might stem from benchmark 2 containing a higher number of genuine dependencies. Genuine dependencies can lead to pipeline stalls because certain instructions may need to wait for others to finish, causing hardware components to reach full capacity. Additionally, benchmark 2 may have more instructions that require a longer time to complete. Given that instructions can take 1, 2, or 5 cycles to finish, the completion time for any benchmark can vary significantly based on the opcodes used.