# Deep Analysis of Massively Simultaneous Multi-Treading Engine Using Software Simulation

ERAN YERMIYAHU[1] AND ROI KARP[1]

[1] 046853 - Advanced Computers Architectures, The Faculty of Electrical and Computer Engineering, Technion - Israeli Institute of Technology
* eeran@campus.technion.ac.il
* roikarp@campus.technion.ac.il

Compiled March 23, 2023

**Multi-Threaded computing has become an essential tool for researchers and developers to efficiently process vast amounts of data and improve system performance. The ability to exploit hardware in order to improve the overall system performance is expected to continue to grow as the scale and complexity of data continue to increase. In this article, we would like to analyze MSMT using a configurable objected-oriented software simulator.**

http://dx.doi.org/10.1364/ao.XX.XXXXXX

## 1. INTRODUCTION

The soaring ubiquity of big data applications has transformed the landscape of modern computing, precipitating a paradigm shift in the methodologies employed by developers and researchers alike. In this context, multi-threaded computing has emerged as a powerful and indispensable tool for efficient system-level data processing, allowing programs to execute multiple threads concurrently and exploit the parallelism intrinsic to contemporary computing architectures. Multi-threaded computing is of particular relevance to complex applications in scientific simulations, data analytics, and machine learning, which are characterized by a high degree of computational intensity, DRAM approaches, and algorithmic intricacy.

The importance of multi-threaded computing lies in its capacity to significantly enhance performance by switching threads rapidly when an event like cache miss occurs, enabling the resolution of data-intensive problems efficiently. In addition, multi-threaded computing offers an avenue for optimized resource utilization, minimizing costs and reducing the impact of scaling limitations.

Despite its benefits, multi-threaded computing poses several challenges, including complex thread synchronization, load balancing, and resource contention, which can impede performance and scalability. This paper aims to provide a comprehensive overview of multi-threaded computing, highlighting its advantages and challenges across different benchmarks and configurations.

## 2. METHOD AND SETTINGS

### A. MSMT Software Simulator Capabilities

To demonstrate and provide conclusive evidence, we developed a configurable software simulator capable of emulating the execution scope of an MSMT processor given the number of execution units and their latency, along with their respective allocation to different instruction types. We also accounted for the fact that several instructions can access memory by holding the memory address within the CPU registers. To accomplish this, we segregated the execution units into memory access and non-memory access categories for each instruction type.

Moreover, the processing of instructions is reliant on data stored in the processor's registers. As a result, we recognized the necessity of a dependency mechanism and designed a pre-processing phase that determines the actual data dependencies for each command. we assume register renaming mechanism is present and no memory dependencies, therefore we only take into consideration 'read after write' dependencies.

Furthermore, To create a simulation that more realistically and coherently reflects the operation of multi-threaded processing, we incorporated an option to consider the time required for a context switch between threads, which occurs frequently in this type of processing. This option enables the configuration of the number of cycles required for a context switch. Additionally, the simulation provides the option to adjust the miss rate of the cache and the branch predictors as either a constant probability or a monotonous function dependent on the number of threads, using a composition of a sigmoid activation function:

$$MR_n = \frac{1}{1 + \exp(\ln(9) - 0.025 \cdot (n-1)^2)} \quad \textbf{(1)}$$

In addition, the simulator incorporates two distinct schedulers. The first scheduler inserts pending threads

into the execution macro1 , while the second scheduler is responsible for selecting the relevant thread to run based on command dependency and availability of execution units regarding the instruction type. To support a variety of scheduling policies, we have implemented various types of schedulers with different selection policies and complexity.

The Default Scheduler selects the first valid thread, while the Round Robin Scheduler prioritizes valid threads in a round-robin fashion. The LRU Scheduler selects the valid thread that has not been accessed for the longest duration, and the Faint Scheduler selects the least frequently chosen valid thread.

See MSMT Github Repository in order to be review the full code core and use the simulator yourself.

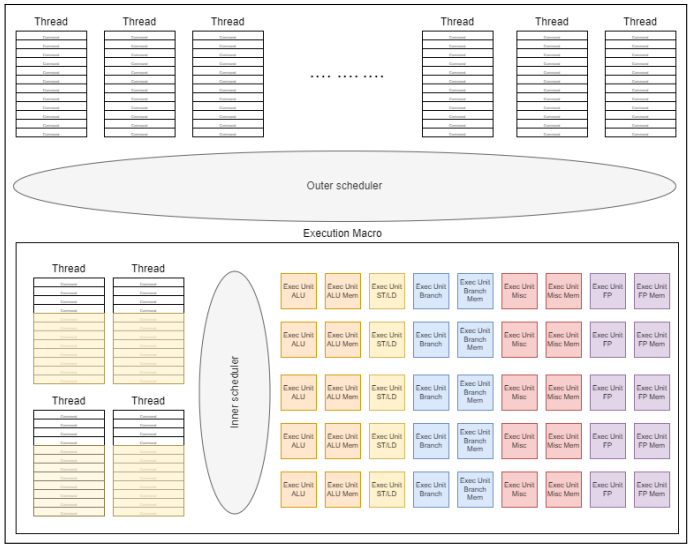


**Fig. 1.** Diagram of MSMT Simulator.

In order to enable simulation, it is imperative for the simulator to receive multiple inputs, including the path to a directory or a file containing various threads configuration files to simulate in parallel, as well as the path to the execution macro configurations. Additionally, a path to an output files directory is required to store the results generated from the simulation process.

In this article, we aim to employ the simulator across various setups and benchmarks to assess the strengths and weaknesses of the MSMT method and analyze the obtained results.

### B. Benchmarks and Command's Core Structure

The use of big data applications and code is widespread in various fields, including scientific simulations, data analytics, image processing, and machine learning. To assess the performance and versatility of MSMT for general purposes, we utilized seven distinct benchmarks from diverse engineering and science domains and categorized their assembly commands into different units. The distribution of execution units for each benchmark is presented below in Table.

**Table 1. Commands Distribution to Execution Unit Types by Benchmark**

| Execution Unit Type | BWAVE | FOTONIK | X264 | GCC | PERL | DEEPSJEN | MCF | Average |
|---|---|---|---|---|---|---|---|---|
| ST/LD | 5.665% | 7.499% | 0.618% | 11.066% | 11.9% | 90.728% | 4.138% | 18.802% |
| ALU | 34.596% | 35.959% | 34.027% | 37.817% | 34.254% | 3.537% | 32.210% | 30.343% |
| ALU + Memory | 26.613% | 22.427% | 25.052% | 18.790% | 20.909% | 0.759% | 25.704% | 19.608% |
| FP | 0.004% | 0.008% | 4.201% | 0% | 0% | 0% | 0% | 1.405% |
| FP + Memory | 0.010% | 0.007% | 9.439% | 0% | 0.026% | 0% | 0% | 2.370% |
| Branch | 14.117% | 14.032% | 5.005% | 13.660% | 13.952% | 3.281% | 16.288% | 11.476% |
| Branch + Memory | 0.006% | 0.001% | 0.002% | 0.050% | 0.065% | 0% | 0.048% | 0.029% |
| Misc | 11.159% | 10.005% | 10.830% | 10.223% | 9.540% | 1.366% | 10.501% | 9.089% |
| Misc + Memory | 10.830% | 10.063% | 10.824% | 8.392% | 9.353% | 0.328% | 11.111% | 8.700% |
| Total with Memory access | 40.123% | 39.996% | 45.936% | 38.299% | 42.253% | 91.815% | 41.001% | 48.489% |
| Total without Memory access | 59.877% | 60.004% | 54.063% | 61.701% | 57.746% | 8.185% | 58.999% | 51.510% |

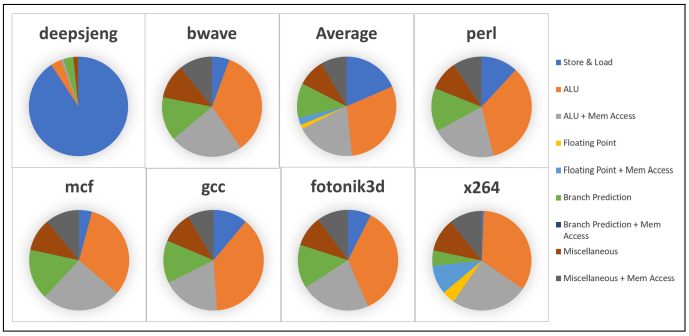


**Fig. 2.** Distribution of command's type over tested benchmarks.

The tabulated data yields salient trends worthy of investigation. Firstly, the marked divergence in the types of commands across benchmarks accentuates the indispensable necessity of testing a diverse array of benchmarks to efficaciously gauge the multifarious advantages of multithreading. Secondly, the ALU emerges as the most sought-after execution unit, closely pursued by Memory accesses and Branches.

Albeit the relative proportion of store/load commands is inconsiderable, the ratio of memory access commands is tantamount to that of non-memory access commands. This finding underscores the paramountcy of minimizing cache and branch prediction misses in single-thread performance, as any decrement in performance in these areas can impede progress in the field considerably.

Another piece of information that was gathered regarding the characteristics of the benchmarks is the depth of their dependencies. when looking at Figures 3 and 4 it is clear that the absolute majority of command dependencies is zero(no dependency) or 1(dependant of previous

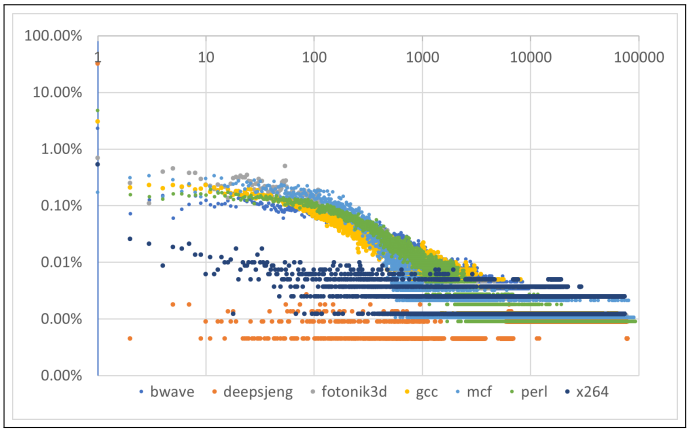


**Fig. 3.** A visual view of all the dependencies in the different benchmarks, both axis are with logarithmic scale.

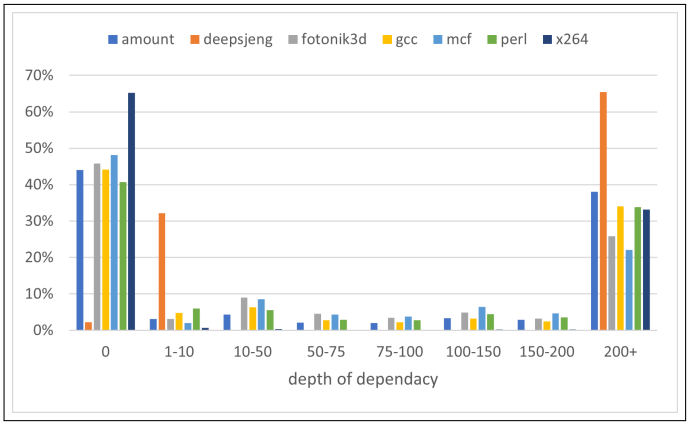


**Fig. 4.** A visual view of the same data as in 3 when binned

command). this information is crucial when deciding on any processor architecture, especially a MSMT processor.

Additionally, in order to gather data from the entirety of each benchmark and not just a specific portion, we implemented a uniform sampling approach where each benchmark was sampled in constant steps. The step size was determined as a function of the length of the benchmark to ensure a uniform number of commands across all benchmarks and minimize variability between different runs, thereby facilitating an unbiased test. In addition, to obtain accurate performance measurements regardless of the local machine that runs the code, a software clock was emulated with a progressively increasing number of cycles, and each test was executed in parallel using multiprocessing coding, enabling the acquisition of highly precise runtime data for subsequent simulations.

## 3. RESULTS AND CONCLUSIONS

### A. Single Thread versus Multi Thread Performance

In this section, we present an evaluation of the performance of each thread on the MSMT simulator isolated and additionally a joint performance of all benchmarks on the MSMT simulator, with the aim of assessing the self-thread performance relative to the overall system performance in each use case. Moreover, we conducted an analysis of the utilization of each category of execution unit to gauge the degree to which the given hardware was leveraged during each simulation. A standardized execution macro configuration was fixed across all benchmarks, and the percentage increase in miss rates over memory accesses and branch predictions in the thread-shared simulation was calculated using the function described by formula 1. The selection of threads outside and inside the execution macro was determined using the Round Robin scheduling policy.

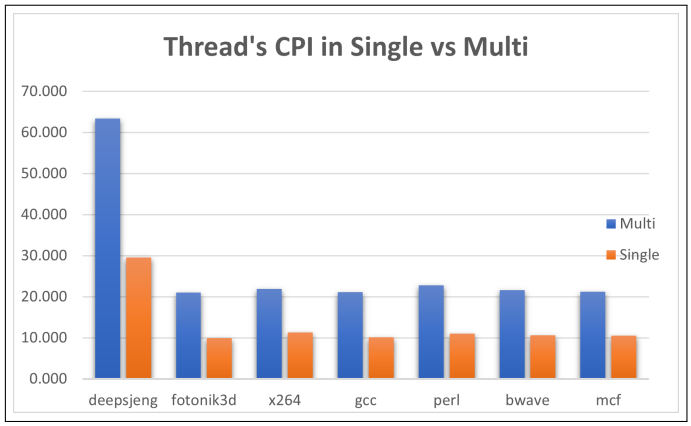


**Fig. 5.** Comparison of Single Thread Isolated versus Joint Benchmarks Simulation.

The graph displayed above demonstrates a decrease in the performance of individual threads in the isolated scenario for each benchmark compared to the CPI of each thread in the joint benchmark simulation. This suggests that executing benchmarks with multiple threads necessitates a greater number of cycles for their completion, a phenomenon that is anticipated given the scheduling policies and endeavors to preserve fairness using the Round Robin policy. Despite these considerations, the overall performance of the system remains notably subpar, especially with respect to the utilization of available execution units.

The presented data provides a genuine comparison between the performance of running a single thread and executing multiple threads simultaneously. In the single-thread scenario, the utilization of each processing unit is low and the average utilization is extremely low. However, when executing the benchmarks concurrently, the average utilization increases, and the system CPI decreases. Therefore, it can be inferred that loading more threads improves overall performance and utilization, despite the fact that the results are still suboptimal and the hardware is capable of delivering better performance.

Therefore, loading additional threads is expected to yield improved performance. Although this approach may slightly affect the performance of individual threads,

**Table 2. Utilization and CPI of Average threads Isolated Run versus Joint Run**

| Execution Unit Type | Average Isolation Scenario | Joint Threads Scenario |
|---|---|---|
| ST/LD | 6.343% | 13.959% |
| ALU | 2.757% | 3.402% |
| ALU + Memory | 5.577% | 6.930% |
| FP | 1.244% | 0.664% |
| FP + Memory | 2.148% | 1.951% |
| Branch | 0.975% | 1.296% |
| Branch + Memory | 0.008% | 0.008% |
| Misc | 6.701% | 9.036% |
| Misc + Memory | 6.607% | 9.104% |
| Average Utilization | 3.996% | 5.150% |
| System CPI | 13.290% | 8.624% |
| Jain's Fairness Index | 1.0% | 0.9378% |

the system can execute more instructions, and the machine is better utilized. In terms of fairness, it can be concluded that the Round Robin scheduling policy successfully maintains fairness among threads, albeit with a slight deviation from the optimal fairness due to command dependencies and the dynamic allocation of execution units.

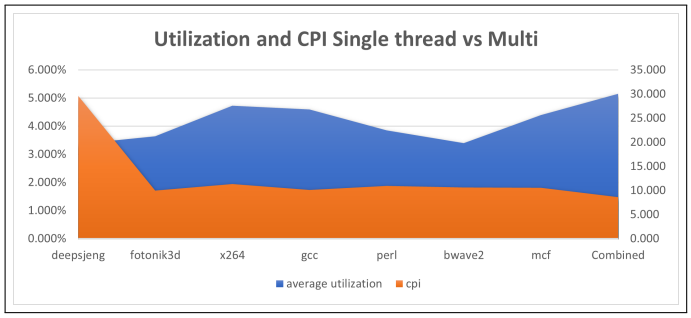


**Fig. 6.** CPI and Utilization of Isolated Benchmark and Joint Threads Run.

As depicted in 6, the joint execution of threads results in an increase in utilization and a decrease in the overall CPI.

### B. Benchmarks IPC as Function of Thread's Congestion

This section endeavors to examine the impact of overloading threads on the overall system performance IPC, with a view to determining the extent to which the hardware can be optimally utilized. It has been observed that despite the hardware's considerable potential, the execution units are not being utilized effectively. In order to explore the limitations of the hardware, each benchmark was subjected to a simulation involving an escalating number of threads to obtain the CPI for each simulation, thereby enabling a visual representation of the system's capacity to handle congestion and execute the benchmark. It is important to underscore that the simulations of each benchmark were homogeneous, involving the simultaneous testing of the same benchmark multiple times, leading to competition over the same resources.

The presented graph depicts the relationship between the IPC of each benchmark and the number of threads employed. The ensuing discussion aims to elucidate the key trends inferred from the results and identify noteworthy scenarios within specific benchmarks.

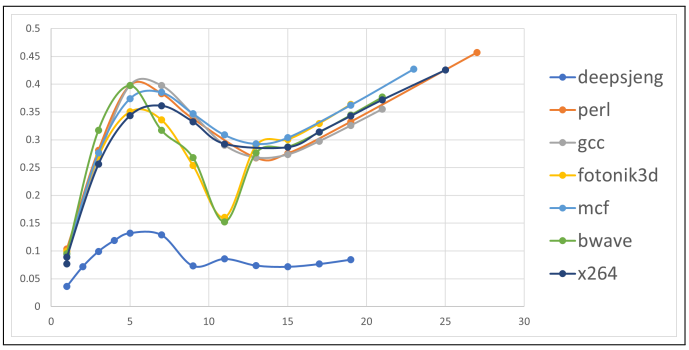


**Fig. 7.** IPC of Benchmarks as Function of Number of Threads.

An overarching trend that is discernible is the IPC's augmentation with an increase in the number of threads, as the hardware is more efficiently exploited, and the cache and branch predictors are not yet saturated. However, a tipping point is reached when the IPC declines to owe to the deterioration in the cache and branch predictor's miss rates. This suggests that the cache and branch predictors are no longer effective, resulting in an elevated miss rate, and necessitating an approach to memory for every memory access command. Consequently, we can anticipate the steepest slope from benchmarks that frequently approach memory, such as deepsjeng.

When examining each benchmark individually, it is possible to compare their performance and comprehend the significance of the miss rate and the location of the valley as a function of the number of threads. For instance, deepsjeng benchmarks necessitate significantly more memory access commands than any other benchmark, as indicated in Table 1, and thus, the IPC values are poorer because of the high dependency on the miss rate and the extremum is lower than the other benchmarks. In addition to that, when the number of threads is high, performance improves gradually and gently, considering the time required for context switching and accessing DRAM, which consumes a considerable amount of cycles when fewer threads are running. The remaining benchmarks exhibit similar behavior, with variations in the slope of the multi-threading area and the depth of the valley.

Despite the ineffectiveness of the cache and branch predictors, there is discernible improvement in performance across all benchmarks with an increase in the number of threads. This can be attributed to a higher likelihood of obtaining an available execution unit and an independent, valid command as the number of threads increases. This, in turn, leads to better hardware utilization and higher

IPC. Under the simulator assumptions, there is no bandwidth limitation, therefore, better performance is expected while increasing the number of threads. Nevertheless, a real system is highly dependent on the available memory bandwidth, which may prevent the thread's access to memory and delay the execution progress.

However, while performance improves with an escalation in the number of threads, there is a fixed peak in performance that is dependent on the hardware, which cannot be surpassed, as known as IPC max. Thus, a scenario of an optimal number of threads arises, in which performance remains optimal, but fairness and single-thread performance decline.

$$IPC_{Max} = \left( \sum_{i=1}^{unit-types} \frac{1}{\text{latency}_i \cdot \text{Units}_i} \right)^{-1} \quad (2)$$

### C. IPC and Fairness within Different Schedulers Policy

In the ensuing section, we will undertake an evaluation of performance and fairness measurements pertaining to a simulation featuring an increasing number of benchmark batches encompassing all seven benchmarks represented previously. We have opted for this particular use case to achieve a highly diversified scenario that is both generic and closely resembles real-life circumstances, thus affording a generic and close-to-reality emulation. Notably, the performance measurements have been fine-tuned, given the significant variances among the benchmarks. To ensure that the simulation does not encounter a scenario in which some benchmarks finish executing while others are still active, we have implemented measures to validate that the system has achieved a steady state across all active threads following the completion of the initial thread. It is important to note that, unlike a real-world system, new threads are not streamed to the processor during the simulation.

Specifically, we calculate the IPC and Jain's fairness index for each scheduling policy applied to both the outer and inner schedulers for an escalating number of benchmark batches. The outer scheduler is responsible for scheduling threads to enter the execution macro, while the inner scheduler identifies the thread that executes a valid command for each unit. To facilitate our analysis, we have implemented four distinct policies, each targeting a unique aspect of the scheduling process, with some policies geared toward promoting fairness. For the sake of convenience, the Default Scheduler selects the first valid thread, while the Round Robin Scheduler prioritizes valid threads in a circular fashion. The LRU Scheduler selects the valid thread that has remained inactive for the longest duration, and the Faint Scheduler chooses the least frequently selected valid thread. By employing these policies, we aim to achieve a comprehensive evaluation of the scheduling process and determine the most effective policy for optimizing both performance and fairness.

In 8, the presented graphs indicate minor variations among the benchmarks results under different scheduling

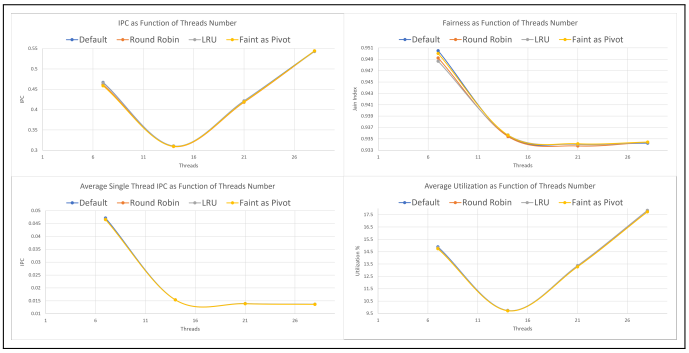


**Fig. 8.** Performance Measurements of Benchmarks Batches Under Different Scheduling Policies (With Realistic Miss Rate).

policies. However, the main trends remain consistent in all scenarios. Upon meticulous scrutiny of the data, it becomes evident that the IPC aligns with the findings of the preceding section, wherein a decline in performance arises due to ineffective cache and branch predictors, thereby leading to a mounting miss rate with an increase in thread count. Conversely, an upswing in performance manifests with thread overload, which is reflected in the average utilization of execution units that exhibit a robust correlation with system performance. The employment rate of these units also serves as a reliable proxy of system performance. Nevertheless, single-thread performance demonstrates a monotonous decrease as the number of threads escalates due to system congestion and resource contention. A particularly intriguing graph depicts fairness, revealing a mystifying phenomenon of almost identical outcomes across different scheduling algorithms. Although the Round Robin Policy shows slightly better fairness results with an increase in thread count, the difference is negligible and the plots ultimately converge.

Deriving from the aforementioned analysis, it is deduced that the scheduling policy has a negligible impact on overall system performance, thereby rendering efforts invested in implementing fairness potentially redundant. A rudimentary scheduling algorithm would suffice to handle scheduling without causing any performance degradation. Our assumption is premised on the correlation between the miss rate and scheduling routines in the MSMT simulator. Each time the scheduler selects a valid command, there exists a probability of a miss, which causes the command and the corresponding thread to be popped out from the execution macro and replaced with a new thread. From the perspective of the scheduler, the vacated thread has been selected but has not made any progress. Moreover, in cases where the miss rate is high, a valid thread is chosen to execute memory-accessing commands in all policies, despite the high likelihood that the thread will be required to wait for a penalty before accessing the DRAM and being evicted from the execution macro. The scheduler then selects a new thread that follows the same sequence of events. We infer that the scheduling pro-

cess has little impact on performance results, whereas the benchmark comprises a significant proportion of memory-accessing instructions. In such scenarios, most commands would have to wait for the penalty and context switch delay, thereby potentially compromising the scheduling policy.

In order to explore these observations, we have chosen to eliminate the increment of the miss rate as the number of threads increases and settled a fixed miss rate of 1%

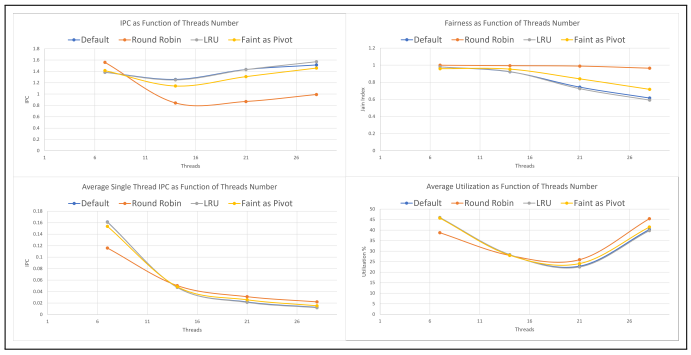


**Fig. 9.** Performance Measurements of Benchmarks Batches Under Different Scheduling Policies (With 1% Constant Miss Rate).

In the present evaluation, discernible disparities among the schedulers are evident, wherein the Round Robin policy outperforms the other policies in terms of fairness. However, this policy yields suboptimal performance and results in lower overall system IPC than the other policies. Furthermore, the default scheduler is found to decrease Jain's fairness index, that's because it simple and considers nothing while scheduling threads. On the other hand, The Faint as Pivot scheduler is a good compromise because it balances the performance and the fairness to the middle ground.

### D. Hardware Tuning Evaluation

When discussing the advantages of MSMT hardware, it is imperative to consider and examine the provided hardware and the features it enables. The simulation's flexibility provides a convenient way to emulate different hardware configurations in order to evaluate the performance and utilization of a given configuration. In this section, we have chosen to evaluate the performance results of an increasing number of threads that co-exist inside the execution macro. As we increase the number of threads we aim to reduce the context switch penalty times and perhaps achieve better performance and utilization. Additionally, we analyze the performance outcomes of an escalating instruction window size. This is equivalent to the scope of the execution macro for each thread, permitting a fixed number of valid commands to choose from. A larger instruction window enhances the number of available commands and diminishes the dependency barrier of each thread.

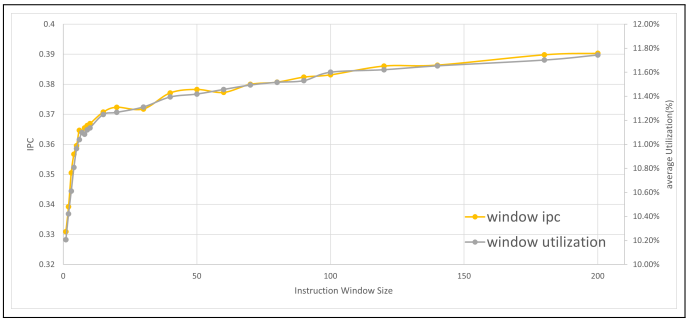


**Fig. 10.** IPC and Utilization as a Function of Instruction Window Size).

Upon examining Figure 10, it is apparent that there is a strong correlation between increasing the instruction window size up to 10 instructions and both the IPC and utilization. However, as the instruction window size continues to increase beyond this point, the correlation between the IPC and utilization becomes less pronounced, due the dependencies of the command within each thread inside the execution macro. This trend is not surprising, given the insights gleaned from Figure 4, which illustrates that the majority of command dependencies occur within the last 10 commands window.

It is worth noting that this correlation between the instruction window size, IPC, and utilization has important implications for optimizing processor performance. Specifically, by adjusting the instruction window size, it is possible to achieve higher IPC and utilization rates, thereby enhancing overall processor efficiency. However, as with any optimization strategy, it is important to strike a balance between maximizing performance and minimizing resource consumption, as excessively large instruction windows can lead to increased hardware complexity and energy consumption. Therefore, further research is needed to determine the optimal instruction window size for a given set of workload and hardware constraints.

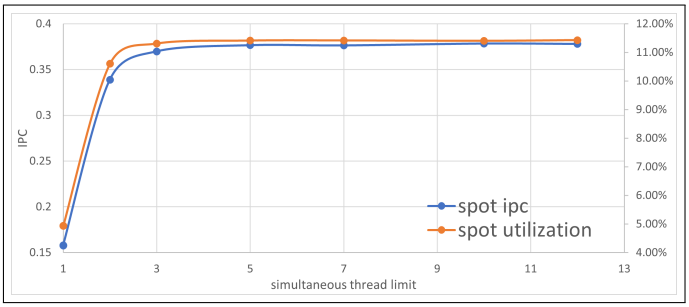


**Fig. 11.** IPC and Utilization as a Function of Thread's Number In Execution Macro). the number of execution units in these runs is 1 for each unit type Except the alu units which has 2 and the non memory branch unit which also has 2

Upon examining the results presented in Figure 11, a clear trend emerges regarding the relationship between

the number of threads that can run simultaneously and both IPC and utilization rates. Specifically, we observe a significant improvement in both metrics as the number of threads increases from 1 to 2. Subsequently, there is a slight improvement in performance as the number of threads increases from 2 to 3, but beyond that point, there is no further improvement. We postulate that this trend is due to the bottleneck effect that arises when there is only one or two execution units of each type. In such a scenario, the number of running threads becomes less important, as the execution units themselves become the limiting factor.

These findings have important implications for optimizing processor performance in multi-threaded environments. In particular, they suggest that increasing the number of threads beyond a certain threshold may not lead to commensurate performance gains, and may even be counterproductive in some cases. Instead, it may be more effective to focus on improving the efficiency of existing execution units, or on optimizing the scheduling of tasks within the processor to minimize idle time and maximize throughput. Further research is needed to determine the optimal number of threads for a given set of workload and hardware constraints, and to develop new strategies for maximizing processor performance in multi-threaded environments.

## 4. CONCLUSIONS

The utilization of the MSMT processor allows for the most efficient use of command congestion. Therefore, it is crucial to identify the optimal hardware configuration that best suits the system's needs. A simple solution would be to envision a processor with no hardware limitations, regardless of die size, yield, power consumption, and so on. Upon running such a simulation, we achieved an IPC of 1.4826, which exceeded all of the realistic results observed in other tests.

Upon reviewing the results of the tests, we identified clear trends between the configuration parameters and system performance. We implemented a setup that we believe strikes a good balance between performance and cost. This test yielded an IPC of 0.454, which is an improvement over all of the realistic results we observed in other tests, albeit not by as large a margin as the unrestricted simulation.

The quest for the ideal configuration necessitates extensive investigation. Initially, we must formulate a hardware cost function that accounts for the quantity of each unit, the execution macro's abilities, and the scheduler algorithm's intricacies. This function estimates the manufacturing and operational costs. Next, we must develop a target function that harmonizes system performance, such as IPC and fairness, with cost. Once we have constructed this function, we can employ iterative optimization techniques, like gradient descent, to discover local minima and attain an optimal MSMT processor configuration for a specific benchmark.

It is important to note that all of the tests conducted in this part of the study used 14 threads, including two of each benchmark.