International Journal of Computer Applications (0975 - 8887)
Volume 185 - No.29, 2023 August

Intelligent Thread-Specific Rename Register Allocation for Simultaneous Multi-Threading Processors Based on Cache Behavior

An Do
The University of Texas at San Antonio
San Antonio, TX 78249-0669, USA

Wei-Ming Lin
The University of Texas at San Antonio
San Antonio, TX 78249-0669, USA

ABSTRACT

Simultaneous Multi-Threading (SMT) processors allow multiple threads to share resources, such as execution units, caches, and pipelines, in the same processor to improve overall system throughput and utilization. The distribution of the physical register file can have a significant impact on the performance of the system. Hence, the register file is one of the most crucial shared resources. One or a few threads holding too many shared registers can obstruct the execution of other threads, thus hurting overall performance. In this paper, we develop an efficient register-file-sharing algorithm based on the number of L2 cache misses. To determine the relationship between L2 cache misses and rename register utilization, the analysis begins with running programs in a single-threaded environment. This relationship then becomes the foundation to develop an algorithm to optimize the use of shared registers. Simulation on M-sim [8] shows that the proposed algorithm increases the throughput by up to 63% compared to the default case while preserving execution fairness among threads.

General Terms

Simultaneous Multi-threading, Register Rename

Keywords

Simultaneous Multi-threading, Register Rename, Register Capping L2 Cache Miss, Resource Sharing

1. INTRODUCTION

Simultaneous Multi-Threading (SMT) is a method that enables a single processor core to execute multiple threads simultaneously to improve overall system performance. SMT achieves this by introducing thread-level parallelism (TLP). To exploit TLP, SMT allows threads to utilize unused processor resources that would otherwise be idle. For example, if a thread stalls waiting for data, another thread can be executed on the same processor core during the stall, increasing the utilization of the processor.
It is important to note that while SMT can exploit TLP to improve performance by overcoming Instruction-Level Parallelism (ILP) present in a single thread ([3],[12]), it can also introduce performance penalties due to increased contention for resources and reduced cache hit rates. Optimizing SMT to take advantage of TLP involves carefully managing the allocation of resources among threads and minimizing resource contention. Various research on resource-sharing algorithms has been done in the past. Research [11] proposes the ICOUNT policy; in the fetching stage, this policy gives higher priority to a thread that has lower occupancy in the pre-issue stages. DCRA [2] is another resource-sharing algorithm for the fetch stage. It monitors memory performance and allocates more resources for threads that use resources more efficiently. A per-thread capping technique on Issue Queue (IQ) entries [7] improves performance by optimizing the dispatch stage; [15] and[14] also propose techniques to improve the dispatch stage. The distribution of the write buffer is targeted in [4] and [1] to reduce unfair occupation. Previous works targeting the rename stage include paper [16], which proposes a capping technique to limit the number of registers all threads are allowed to use. An autonomous approach, similar to a capping technique, using Neural Network is presented in [13]. The rename stage is an early stage with shared resources. Effectively distributing the registers can have a high impact on the rest of the pipeline. Computer programs behave differently, leading to some programs utilizing registers better than others. Therefore a thread-specific capping technique can maximize the performance gain for every register. Fast threads can achieve better performance improvement than slow threads when given more registers. An algorithm to orchestrate such a distribution scheme without starving the slow threads can lead to substantial overall performance improvement while maintaining execution fairness.
This paper presents an algorithm based on the relationship between shared register utilization and L2 cache misses to optimize the distribution of rename registers for SMT systems. A program performs better when given more rename registers. However, up until a certain amount of rename registers, the performance gain is no longer significant due to bottleneck at another stage. Programs that incur fewer L2 cache misses continue to perform better with more rename registers than programs with a higher number of L2 cache misses. The proposed algorithm detects which programs benefit more from more registers and allocate larger portions of registers to them. The programs that benefit less from more registers are allocated

smaller portions, but not smaller than a predefined limit so that they do not suffer from starvation. This ensures that registers are utilized in a way to gain the most overall performance while maintaining execution fairness. Simulation results show an average performance improvement of up to 63% and 39% in 4-threaded workload and 8-threaded workload respectively. The harmonic IPC sees similar improvement, indicating that the effectiveness of the proposed algorithm is not at the expense of execution fairness.
The remaining of the paper is organized as follows: Section 2 introduces the background of SMT systems and the renaming stage, followed by a walk-through of the simulated environment parameters; motivation is analyzed in detail in Section 4, followed by outlines of the proposed method in Section 5; Section 6 presents the simulation results; Section 7 presents the concluding remarks.

2. BACKGROUND

2.1 Simultaneous Multi-Threading

Simultaneous Multi-Threading (SMT) utilizes resources in a modern processor more effectively by executing multiple independent threads in parallel, resulting in improved performance. By allowing multiple threads to execute concurrently on the same clock cycle, SMT exploits Thread-Level Parallelism (TLP) among threads to take advantage of underutilized resources when ILP in a single thread is not sufficient [3]. Figure 1 illustrates the basic pipeline stages of a 4-threaded system. The system is built from a typical out-of-order super-scalar processor.

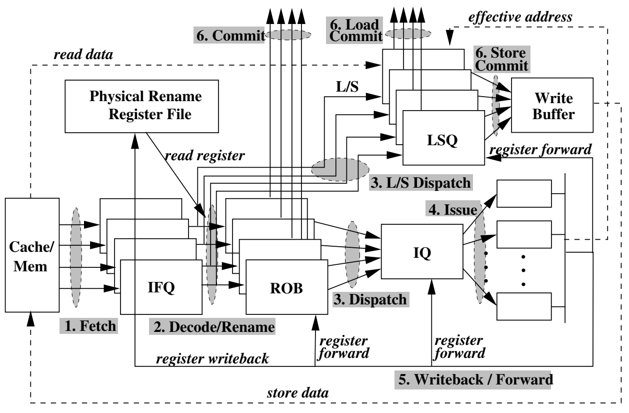


Fig. 1: **Pipeline Stages in a 4-threaded SMT System**

Firstly, instructions from each thread are *fetched* from memory (and cache) and placed into their Instruction Fetch Queue (IFQ). Following the *decode* and register-*rename* stages, they are allocated into their corresponding Re-Order Buffer (ROB) and *dispatched* to the shared Issue Queue (IQ). Load/Store instructions have their address calculation operations dispatched into IQ while also having their operations dispatched into individual Load Store Queues (LSQ). When the instruction-issuing conditions are satisfied (i.e. all operands are ready and the required functional unit is available), the operations are *issued* to the corresponding functional units and their results are *written back* to their target locations or forwarded to where they are needed in IQ or LSQ. Load/Store instructions, upon calculating their addresses, initiate their memory operations. Lastly, all instructions are committed from the ROB in order, synchronized with Load/Store instructions in LSQ. SMT processors typically share key datapath components among multiple independent threads. These shared resources may include the physical register file, machine bandwidths, inter-stage buffers (such as the Issue Queue), functional units, and write buffers. By sharing resources, the hardware required in an SMT system can be significantly reduced, while achieving comparable throughput to multiple copies of superscalar processors. The effective distribution of shared resources among simultaneously executing threads is crucial for achieving desirable performance in an SMT system. Without such effective distribution, shared resources may be occupied by one or very few threads out of proportion, resulting in faster threads impaired by the lack of resources. Consequently, overall system performance may suffer. Among the shared resources, the shared registers in the physical register file are used to eliminate register name dependencies in the rename stage, which is the first stage of shared resources. Disproportionate distribution of the physical register file among threads can easily become a bottleneck along the pipeline stages. Therefore, this paper focuses on the efficient distribution of the physical register file among multiple threads.

2.2 Physical Register File

In this segment, the topic of register renaming and its implementation will be introduced. Register renaming is a useful technique that is employed to avoid name dependencies, such as write-after-read and write-after-write, that arise when registers are reused. This method has gained popularity among modern processors because it is essential to achieve out-of-order execution. By assigning different physical registers to the same architectural register when it is used in subsequent instructions, renaming effectively removes false dependencies. As a result, later instructions can be executed out-of-order without interfering with earlier instructions. For instance, consider the following program segment that contains false dependencies. Out-of-order execution is hindered by write-after-read (e.g., between instructions γ and δ) and write-after-write (e.g., between instructions α and δ).

$$\begin{array}{rcll}
\text{r1} & \longleftarrow & & \text{(instruction } \alpha) \\
 & \vdots & & \\
 & \longleftarrow & \text{r1} & \text{(instruction } \beta) \\
 & \vdots & & \\
 & \longleftarrow & \text{r1} & \text{(instruction } \gamma) \\
 & \vdots & & \\
\text{r1} & \longleftarrow & & \text{(instruction } \delta) \\
 & \vdots & & \\
 & \longleftarrow & \text{r1} & \text{(instruction } \epsilon)
\end{array}$$

If renaming is employed, r1 in instruction δ is assigned to another physical register, allowing δ and instructions after that to execute before instruction γ
Commonly, multi-threaded processors feature a ”physical” register file that contains more physical registers than the number of ”architectural” registers defined in the ISA. Whenever an instruction writes to an architectural register, the physical location is allocated and assigned to that architectural register. Subsequent read instructions of that architecture register will have their data come from the most recently assigned physical register. This mapping between architectural registers and physical registers is recorded in the rename table. With this renaming approach, the availability of physical registers is a major factor in the performance of the system.

A physical register is allocated at the time of renaming and is not freed until the next write instruction on the same architectural register is committed. One way to increase register availability is to expedite the deallocation process. Such modification is proposed in [5] which modifies the deallocation process to expedite the release of a register to reduce register occupation time. However, this approach requires software support from the operating system. Another approach is presented in [6] which holds off register allocation until the complete stage. The tradeoff of this approach is that an instruction might not find a free register to commit to at the complete stage leading to a deadlock. It is challenging to modify the allocation-deallocation process without additional information from the operating system or a great amount of additional hardware. That is not to say there is no room for improvement while maintaining the in-order allocation and deallocation process. A scheme for better utilization of the physical registers can ensure the availability of registers and improve performance. According to previous research, the competition for floating point registers is not as fierce as for integer registers [16]. Therefore, this paper focuses only on the distribution of integer registers.

The register file is shared among threads. However, only the additional registers are truly shared after each thread has populated its defined number of architectural registers. For example, an ISA with 32 architectural registers ensures each thread has all of its architectural registers mapped to at least one physical register at all times. If the register file has extra registers after dedicating 32 registers for each thread, these extra registers will be shared among threads. Figure 2 [17] shows the organization of the register file for this example. R_t, R_a, and R_r are the total number of registers, the number of architectural registers per thread, and the number of extra registers for renaming. Therefore

$$R_r = R_t - N \times R_a$$

where N is the number of threads in the system. In single-threaded systems, if the thread holds on to all the rename registers too long, it cannot execute any more instructions until a previous instruction is committed and releases a register. This can be mitigated by increasing the register file size. However, in multi-threaded systems, the order of committing is independent among threads, leading to the slow threads occupying more registers as the fast threads release them. Without making R_t unreasonably large, it is necessary to have an effective distribution scheme of the rename registers to prevent the slow threads from hindering the system's overall performance.

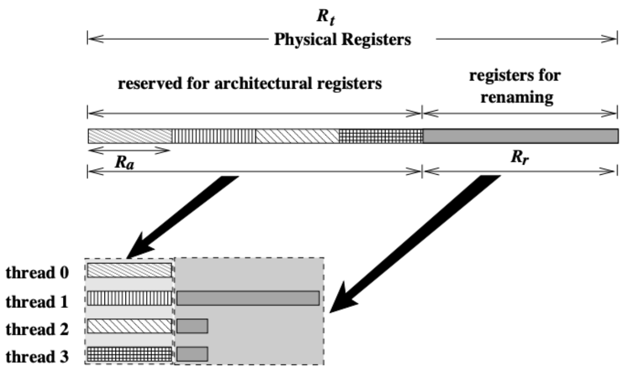


Fig. 2: **Organization of a shared Physical Register File**

3. SIMULATION ENVIRONMENT

3.1 Simulator

M-sim [8], a multi-threaded micro-architectural simulation environment is used to evaluate the performance of the proposed algorithm. M-sim includes all the features necessary for this paper, including models of the key pipeline structures such as the Reorder Buffer (ROB), the Issue Queue (IQ), the Load/Store Queue (LSQ), separate integer and floating-point register files, and register renaming. M-sim can simulate a single-threaded processor or a simultaneous multi-threaded processor. The configuration of the simulated processor is given in Table 1.

Table 1. : **Configuration of the Simulated Processor**

Parameter	Configuration
Machine Width	8 wide fetch/dispatch/issue/commit
L/S Queue Size	48-entry load/store queue
ROB & IQ size	128-entry ROB, 32-entry IQ
Functional Units & Latency(total/issue)	4 Int Add(1/1) 1 Int Mult(3/1)/Div(20/19) 2 Load/Store(1/1), 4 FP Add(2/1) 1 FP Mult(4/1)/Div(12/12) Sqrt(24/24)
Physical registers	integer and floating point as specified in the paper
L1 I-cache	64KB, 2-way set associative 64-byte line
L1 D-cache	64KB, 4-way set associative 64-byte line write back, 1 cycle access latency
L2 Cache unified	512KB, 16-way set associative 64-byte line write back, 10 cycles access latency
BTB	512 entry, 4-way set-associative
Branch Predictor	bimod: 2K entry
Pipeline Structure	5-stage front-end(fetch-dispatch) scheduling (for register file access: 2 stages, execution, write back, commit)
Memory	32-bit wide, 300 cycles access latency

3.2 Workloads

The simulated processor is evaluated by running a mixture of benchmarks from the SPEC CPU2006 benchmark suite [10]. Each program is simulated in a simplescalar environment to be classified by its ILP in accordance with the procedure mentioned in Simpoints tool [9]. There are three types of ILP classification, high ILP, medium ILP (execution bound), and low ILP (memory bound). Table 2 and Table 3 provide the 4-threaded and 8-threaded mixes that feature various ILP combinations to represent diversified workloads.

3.3 Metrics

The sum of individual threads' IPC is a common metric to measure the system performance in SMT processors:

$$\text{Overall\_IPC} = \sum_{i}^{N} \text{IPC}_i \tag{1}$$

Table 2. : **SPEC CPU2006 4-threaded Mixes**

Mix	Benchmarks	Classification(ILP)		
		Low	Med	High
Mix1	libquantum, dealII, gromacs, namd	0	0	4
Mix2	soplex, leslie3d, povray, milc	0	4	0
Mix3	hmmer, sjeng, gobmk, gcc	0	4	0
Mix4	lbm, cactusADM, xalancbmk, bzip2	4	0	0
Mix5	libquantum, dealII, gobmk, gcc	0	2	2
Mix6	gromacs, namd, soplex, leslie3d	0	2	2
Mix7	dealII, gromacs, lbm, cactusADM	2	0	2
Mix8	libquantum, namd, xalancbmk, bzip2	2	0	2
Mix9	povray, milc, cactusADM, xalancbmk	2	2	0
Mix10	hmmer, sjeng, lbm, bzip2	2	2	0

Table 3. : **SPEC CPU2006 8-threaded Mixes**

Mix	Benchmarks	Classification(ILP)		
		Low	Med	High
Mix1	libquantum, dealII, gromacs, namd, soplex, leslie3d, povray, milc	0	4	4
Mix2	libquantum, dealII, gromacs, namd, lbm, cactusADM, xalancbmk, bzip2	4	0	4
Mix3	hmmer, sjeng, gobmk, gcc, lbm, cactusADM, xalancbmk, bzip2	4	4	0
Mix4	libquantum, dealII, gromacs, soplex, leslie3d, povray, lbm, cactusADM	2	3	3
Mix5	dealII, gromacs, namd, xalancbmk, hmmer, cactusADM, milc, bzip2	3	2	3
Mix6	gromacs, namd, sjeng, gobmk, gcc,lbm, cactusADM, xalancbmk	3	3	2

where N denotes the number of threads that run simultaneously in the system and IPC$_i$ denotes the IPC of each thread.
In addition, to ensure improved overall IPC is not accomplished by the starvation effect, the Harmonic IPC is adopted. Harmonic IPC reflects the execution fairness among the threads:

$$\text{Harmonic\_IPC} = N / \sum_{i}^{N} \frac{1}{\text{IPC}_i} \tag{2}$$

4. MOTIVATION

Previous research [16] and [17] have shown that the imbalance in the distribution of rename registers can be unexpectedly extreme. Figure 3 shows the average percentage of the rename register occupied by each thread, sampled every 50 clock cycles, in a 4-threaded environment for mixes from Table 2 with a register file of size 160. Mixes with small differences among threads do not exhibit extreme competition. Each bar features one standard deviation away from the average. A large standard deviation indicates high fluctuation in the percentage of registers occupied. On average, Mix 3, Mix 8, Mix 9, and Mix 10 experience extreme competition indicated by a single thread occupying 65% of the registers or more in at least 50% of the sampling points. While there is high fluctuation in Mix 3 and Mix 9, one thread still occupies approximately 55% and 65% of the registers 84.1% of the time, respectively. Such dominance can result in heavy performance penalties to the entire system.

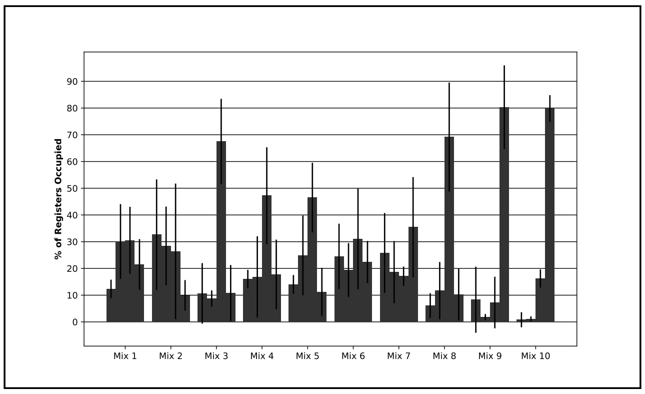


Fig. 3: **Average Register Occupancy of Each Thread**

4.1 Register File Capping Technique

In order to prevent a situation where a single thread dominates, a capping technique was proposed [16]. This technique limits the number of rename registers a thread can use at any given point in time, which is referred to as the ”cap value”. While this fixed-capping approach has shown significant potential, it is not adaptable to varying workloads in real time. In addition, this fixed-capping approach sets a global cap for all threads. This disregards the different needs of each thread. The proposed approach considers the thread’s demand for rename registers, which is determined by the number of L2 cache misses and the utilization of the registers that the thread is permitted to use. The ”cap value” is adjusted accordingly to ensure that threads that benefit from additional registers are given more, while threads that are not utilizing all of their allocated registers have them reallocated for better distribution.

4.2 Reference Systems for Performance Comparison

To present an extensive performance comparison, the default M-sim without any modification is used as a baseline, as well as two other capping algorithms as reference systems. One is a fixed capping approach and the other is a simple intelligent capping approach.
For the fixed capping approach, all threads are capped at one value throughout the entire simulation. Tests are performed on 32 out of all the possible cap values to find the one which provides maximum returns on improvement. This is a fixed and global capping approach. It does not adapt to the workload and all the threads have the same cap value. This approach is not practical because it is not possible to determine the best cap value beforehand. However, it represents the best improvement a fixed and global capping approach can achieve for the sake of comparison.
The simple intelligent control is also a global capping technique, however, it dynamically adjusts the cap value based on the change in the number of instructions committed. In this approach, the cap value starts as $\frac{1}{4}R_r$. After every 5,000 clock cycles, the cap value either increments or decrements based on the direction. At the beginning of the simulation, the direction is not determined. It is up to the designer to choose the initial direction. For this paper, the initial direction is up. Since the initial direction is up, at clock cycle 5,000, the cap value increases by 1. At clock cycle 10,000, the number of instructions committed within this past 5,000 clock cycle

window is compared with the number of instructions committed in the previous window. If the number of committed instructions increases, maintain the same direction when the next window start. On the other hand, if it decreases, the direction is reversed.

5. PROPOSED METHOD

This section introduces the Physical-Register-Allocation algorithm that dynamically adjusts the cap value of each thread based on the number of L2 cache misses, depicted as $L2_m$, and rename register need within a defined time window. The algorithm is designed to ensure optimal utilization of the rename registers and prevent extremely imbalanced distribution.

5.1 Register Demand and L2 cache miss

A high number of $L2_m$ is directly correlated with low IPC. When a program has a high number of $L2_m$, it may hold registers for a longer amount of time. Since the rename is done in Round-Robin, programs that hold registers longer continue to incur more registers. This may lead to programs with high $L2_m$ holding on to more registers than others, which affects other programs' execution flow. Inconveniently, using more registers does not always allow these programs to perform better. To examine the behavior of the programs, each is simulated in a single-threaded environment with $R_t = 128$ to observe its IPC and $L2_m$. Each program has $R_t - R_a = 96$ extra registers to determine how well it can perform when given a large amount of rename registers. For the rest of this paper, the IPC from this single-threaded environment is denoted as Single-Threaded IPC. Each program is simulated with an increased cap value every time to observe how different cap values affect its IPC. Figure 4 shows the percentage of the Single-Threaded IPC each program reaches at increasing cap value starting at 4. Low ILP threads and some medium ILP threads reach up to 90% of their Single-Threaded IPC while using 6 rename registers or less while some medium ILP threads and high ILP threads need an amount close to 32 registers or more to reach their Single-Threaded IPC. To draw a more distinct correlation, Figure 5 shows the cap value to reach 90% of the Single-Threaded IPC and the total $L2_m$. The cap value to reach 90% performance correlate inversely with the total $L2_m$. It is clear that the fewer L2 cache misses there are, the more register the thread demands to get close to its Single-Threaded IPC.

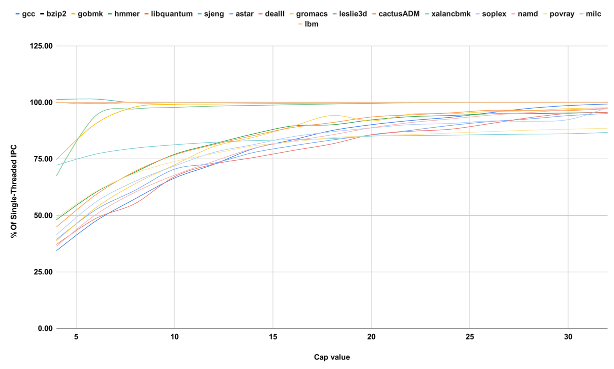


Fig. 4: **Total number of L2 Cache Misses and Cap Value to Reach** 90% **Unrestricted Performance for Each Benchmark**

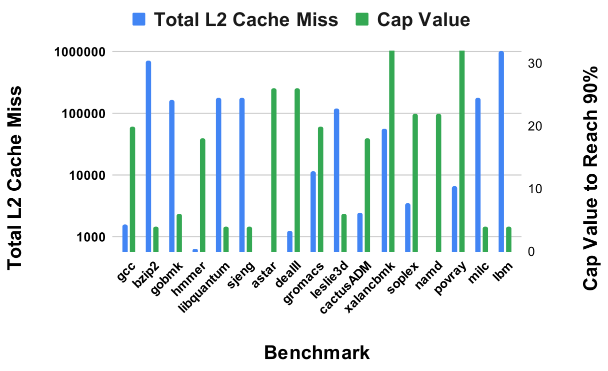


Fig. 5: **Total number of L2 Cache Misses and Cap Value to Reach** 90% **Unrestricted Performance of each Benchmark**

5.2 Proposed Algorithm

The analysis of $L2_m$ and the register demand of the programs is the foundation for the proposed algorithm. The objective of the algorithm is to give programs with higher $L2_m$ a lower cap value and give programs with lower $L2_m$ a higher cap value while keeping a balance among all the cap values. This prevents programs with higher $L2_m$ from unnecessarily occupying more registers while leaving more registers for programs with lower $L2_m$ to maximize their performance. How the algorithm works is as follows. At the end of every window (each window is 2000 clock cycles), the threads are ranked based on their $L2_m$ in that window. To balance out the adjustments, the cap values of half of the threads with lower $L2_m$ are incremented while the cap values of the other half with higher $L2_m$ are decremented. The flowchart of the proposed algorithm is shown in Figure 6.
The algorithm has parameters in place to avoid pulling the cap value too low, pushing it too high, and having a sum of cap values much greater than the number of registers available. C_l, C_h, C_m, and C_s are used to depict the Cap Lower Limit, Cap Upper Limit, Maximum Sum Of Caps, and Sum Of Caps accordingly. While adjusting the cap values, besides looking at the $L2_m$, it is also important to ensure that the sum of caps does not surpass a certain limit. That limit is C_m. Whether a thread can rename also depends on factors other than its cap value, therefore with a $C_m = R_r$, some physical registers are left underutilized. By having

$$C_m = R_r + 2N \tag{3}$$

it leaves a small headroom for threads that can rename to utilize what other threads cannot, despite occupying less than the cap value. Naturally, it is necessary to ensure that $C_s < C_m$ while performing adjustments. $\frac{1}{N}C_m$ is the starting cap value for all threads at the beginning of the simulation. Programs with a high $L2_m$ do not gain much improvement with a high cap value. Many of them reach their Single-Threaded IPC with as little as 6 rename registers. On the other hand, programs with a low number $L2_m$ have a higher demand for rename registers. On average, they need 18 registers to reach 90% of their Single-Threaded IPC, with some needing more than 32. It is important to allocate as much register to such programs as possible without staving programs with high $L2_m$. As a compromise, the hard lower limit is set as

$$C_l = 4 \tag{4}$$

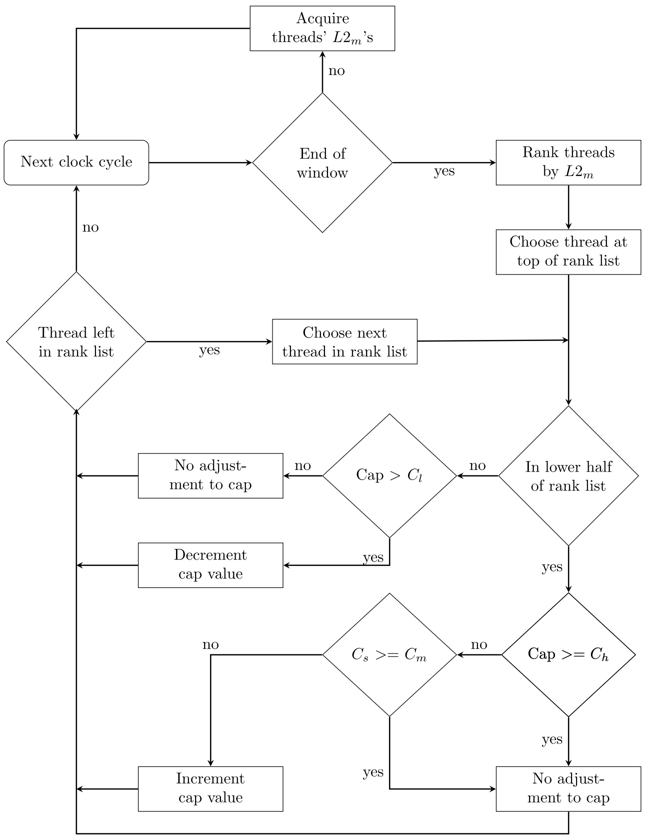


Fig. 6: **Flowchart of proposed algorithm**

While $L2_m$ is a good indication of high ILP programs, it does not always reflect low register occupancy delay. Therefore, there needs to be an upper limit to the cap value to prevent a thread from holding on to many registers for a long period of time. Due to the complementary nature of the adjustment process (half increments and half decrements), and an even portion is $\frac{1}{N}C_m$ registers if the rename registers are split evenly, the upper limit is defined as $C_h = \frac{1}{N}C_m + (\frac{1}{N}C_m - 4)$ or

$$C_h = \frac{2}{N}C_m - 4 \qquad (5)$$

For example, consider a 4-threaded system with a register file size of 160. $R_r = 32$ therefore $C_m = 40$. If two threads consistently have lower $L2_m$ than the other two, the threads with lower $L2_m$ will have cap values of 16 ($C_h = 16$) while the other two will have cal values of 4 ($C_l = 4$).

6. SIMULATION RESULTS

M-sim configured as Table 1 is the tool to simulate the proposed algorithm to compare with the default settings using the mixes in Table 2 and Table 3 for 4-threaded workload and 8-threaded workload.

Figure 7 and Figure 8 show the improvement of the proposed algorithm on different register file sizes, compared to the three reference systems for 4-threaded workload and 8-threaded workload, respectively.

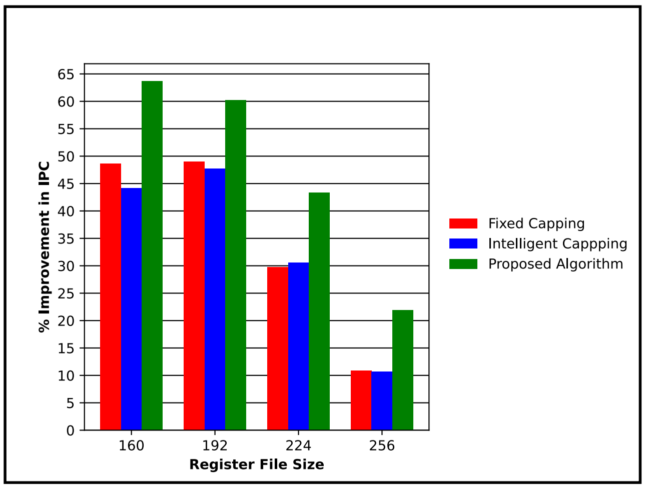


Fig. 7: **IPC Percentage Improvement vs. Register File Size on the 4-threaded Workload**

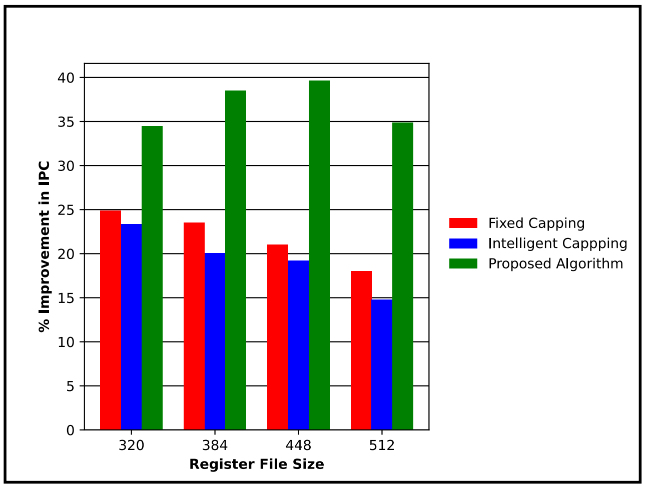


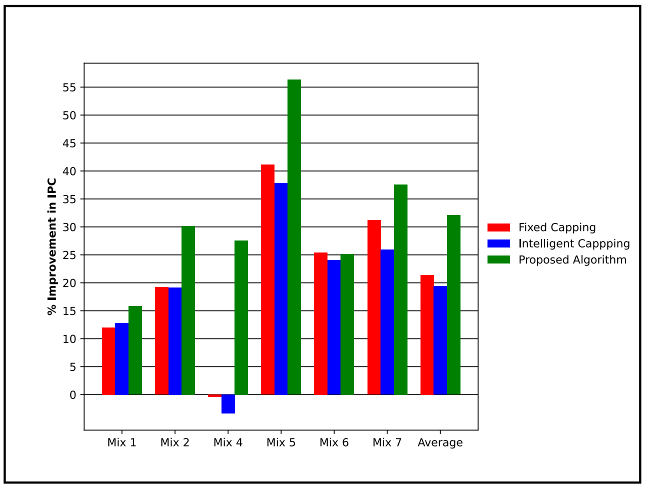
Fig. 8: **IPC Percentage Improvement vs. Register File Size on the 8-threaded Workload**

The proposed algorithm results in considerable performance improvement compared to the three reference systems. Compared to the default system, the proposed algorithm performs up to 63% and 39% better for the 4-threaded case and 8-threaded case, respectively. In the 8-threaded case, the algorithm not only provides a substantial improvement over the default but there is also a big gap when compared to the fixed capping and intelligent capping techniques. To our surprise, the simple intelligent algorithm trails behind the fixed capping despite its dynamic nature. This leads us to believe that while the simple intelligent capping is dynamic, it is overly sensitive to the change in the number of instructions committed. A sudden spike in the number of instructions committed can lead to it making adjustments opposite to the behavior of the threads. The fixed capping also represents the best-case scenario for the fixed and global approach while the intelligent control is only a simple implementation of the dynamic and global approach. The combination of dynamic and thread-specific cap value adjustment featured in the proposed algorithm results in superior performance gain compared to both reference algorithms.

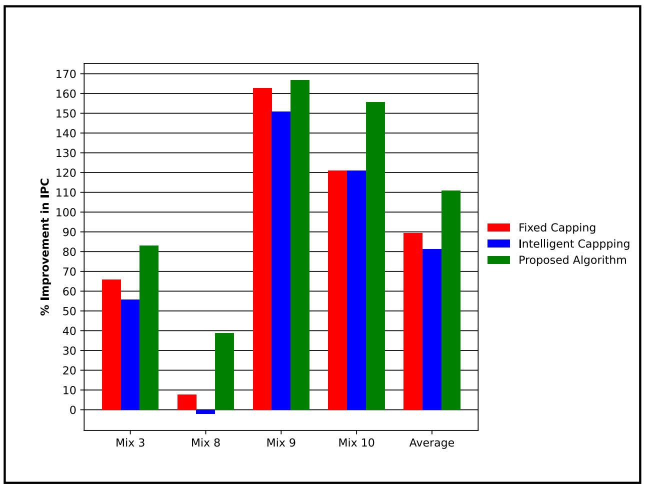
While there is a trend between improvement and register file size in the 4-threaded case, the same cannot be said about the 8-threaded case. The improvement peaks at $R_t = 160$ and decreases as the register file size increases on the 4-threaded workload. As the register file size increases, the competition is not as fierce. The more resources the system has, the effect of the distribution system decreases. Therefore, even when one or a few threads hold on to too many registers, there are enough available registers for the other threads to still perform well. This phenomenon does not manifest in the 8-threaded case for the proposed algorithm. One possible explanation is the diversity of ILP categories in the mixes. All but one mix in the 8-threaded workload contains programs from all three ILP categories. With more mixes containing threads at opposite ends of the ILP spectrum, more threads are more likely to have their cap value hitting the bottom limit of $C_l = 4$, leaving more registers for threads with lower $L2_m$. Figure 5 shows that some programs cannot reach their Single-Threaded IPC even when given 32 registers, this means that the performance improvement can increase as the register file size increases for mixes that have multiple programs at opposite ends of the ILP spectrum in the 8-threaded workload.

For further examination, let us take a closer look at the 4-threaded workload at $R_t = 160$. Figure 9 presents the IPC improvement of each mix.

As mentioned above, Mix 3, Mix 8, Mix 9, and Mix 10 exhibit extreme competition indicated by the dominance of one thread while the other mixes do not exhibit such extreme competition. Figure 9a and Figure 9b show the %IPC improvement of these two groups. The proposed algorithm performs 110% better on average compared to the default in extreme cases. Mix 9 and 10 feature the highest improvement, while the improvement for Mix 8 is not as substantial. In the regular group, there is also a considerable average improvement of 32%.



(a) Regular Cases



(b) Extreme Cases

Fig. 9: **IPC Improvement With $R_t = 160$ On All 4-Threaded Mixes**

6.1 Execution Fairness

It is important for a distribution algorithm to not sacrifice execution fairness to improve overall performance. The improvement would be less meaningful if the algorithm devotes the majority of resources to fast-running threads while critically throttling slower-running threads. To demonstrate the execution fairness of the proposed algorithm, Figures 10 and 11 demonstrate the average IPC improvement on the 4-threaded workload and 8-threaded workload with various register file sizes, respectively.

The proposed method achieves up to 54% and 19% improvement over the default system for the 4-threaded workload and 8-threaded workload respectively. It is fair to say that the algorithm improves the overall performance without sacrificing execution fairness. Interestingly enough, except for the case where $R_t = 320$, both the fixed capping and intelligent capping cause a decrease in harmonic IPC in the 8-threaded workload. The combination of global cap value and diversity of the 8-threaded mixes is a potential culprit. The same cap value for all threads despite them being a part of multiple ILP categories is not ideal. Especially for the simple intelligent capping method where the cap value is driven by the number of instructions committed overall, which closely correlates to the total IPC.

7. CONCLUSION

This paper proposes an optimized register-file-distribution method, based on the number of L2 cache misses and the demand for renaming registers. The algorithm is developed by observing how well each program performs, given various cap values. Correlating this with the number of L2 cache misses, the algorithm adjusts the cap values of each thread to meet its demand while not compromising other threads. This method achieves promising results compared to the default case and two other reference systems. The findings in this paper open a path for potential future improvements. The per-thread cap values adjustment mechanism presented in this paper is only one of the possible ways to balance the cap values among threads. There are other mechanisms to be explored. While the number of L2 cache misses is a good indicator of register demand, there are still gray areas. In addition, the correlation may only hold up for the set of programs presented in the paper. The search for another parameter or a parameter to be used in conjunction with L2 cache misses has the potential

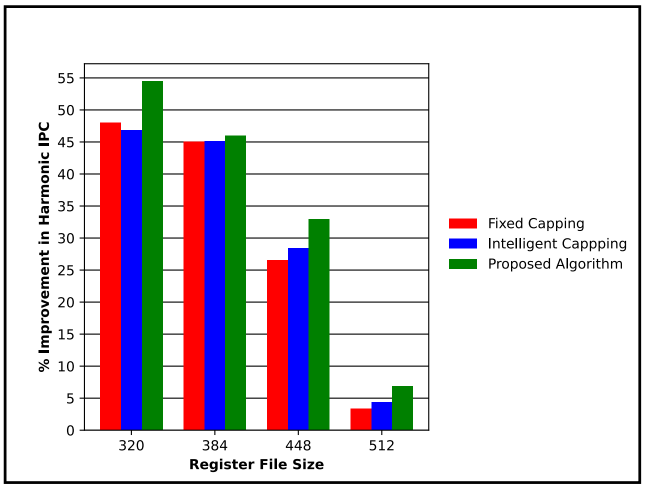


Fig. 10: **Harmonic IPC Percentage Improvement vs. Register File Size on the 4-threaded Workload**

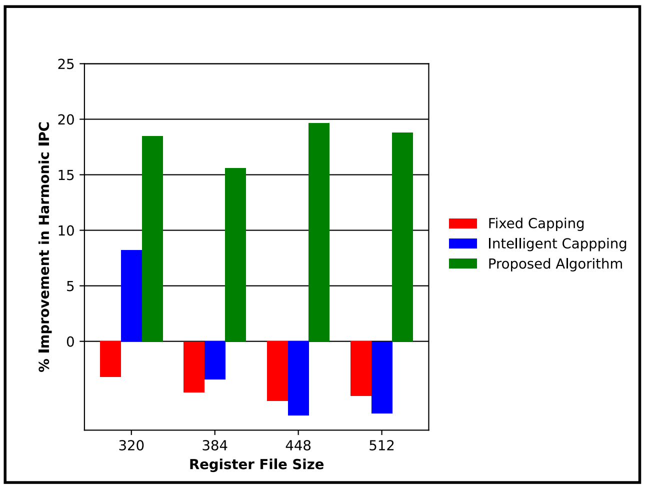


Fig. 11: **Harmonic IPC Percentage Improvement vs. Register File Size on the 8-threaded Workload**

to improve the performance, as well as make the algorithm more adaptable to a broader variety of workloads.

8. REFERENCES

[1] Shane Carroll and Wei-Ming Lin. Latency-aware write buffer resource control in multi-threaded cores. *Int. J. Distrib. Parallel Syst.(IJDPS)*, 7, 2016.

[2] Francisco J Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernández. Dynamically controlled resource allocation in smt processors. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 171–182. IEEE, 2004.

[3] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 136–145, 1992.

[4] Sherifdeen Lawal, Yilin Zhang, and WM Lin. Prioritizing write buffer occupancy in simultaneous multi-threading processors. *Journal of Emerging Trends in Computing and Information Sciences*, 6(10):515–522, 2015.

[5] Jack L Lo, Sujay S Parekh, Susan J Eggers, Henry M Levy, and Dean M Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):922–933, 1999.

[6] Teresa Monreal, Antonio González, Mateo Valero, José González, and Víctor Viñals. Dynamic register renaming through virtual-physical registers. *Journal of Instruction Level Parallelism*, 2:4–16, 2000.

[7] Tilak Kumar Develapura Nagaraju, Caleb Douglas, Wei-Ming Lin, and Eugene John. *Effective Dispatching in Simultaneous Multithreading (SMT) Processors by Capping Per-thread Resource Utilization*. PhD thesis, University of Texas at San Antonio, 2011.

[8] Joseph Sharkey, Dmitry Ponomarev, and Kanad Ghose. M-sim: a flexible, multithreaded architectural simulation environment. *Techenical report, Department of Computer Science, State University of New York at Binghamton*, 2005.

[9] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices*, 37(10):45–57, 2002.

[10] SPEC. Standard performance evaluation corporation. https://www.spec.org.

[11] Dean M Tullsen, Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, and Rebecca L Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 191–202, 1996.

[12] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, 1995.

[13] Wenjun Wang and Wei-Ming Lin. Real-time physical register file allocation with neural networks for simultaneous multi-threading processors. *International Journal of High Performance Systems Architecture*, 8(3):146–158, 2018.

[14] Yilin Zhang, Marcus Hays, Wei-Ming Lin, and Eugene John. Autonomous control of issue queue utilization for simultaneous multi-threading processors. In *Proceedings of the High Performance Computing Symposium*, pages 1–8, 2014.

[15] Yilin Zhang and Wei-Ming Lin. Capping speculative traces to improve performance in simultaneous multi-threading cpus. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1555–1564. IEEE, 2013.

[16] Yilin Zhang and Wei-Ming Lin. Efficient physical register file allocation in simultaneous multi-threading cpus. In *33rd IEEE International Performance Computing and Communications Conference (IPCCC 2014), Austin, Texas*, 2014.

[17] Yilin Zhang and Wei-Ming Lin. Intelligent usage management of shared resources in simultaneous multi-threading processors. In *Proceedings of the*

International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), page 27. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2015.