International Journal of Computer Applications (0975 - 8887)
Volume 187 - No.33, August 2025

The Incorporation of Register Capping to the Model of the Rename Register File using Markov Chain

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

In modern Simultaneous Multi-Threading (SMT) processors, efficient management of shared resources like the rename register file is crucial for optimizing system throughput and utilization. The register file, being a critical shared resource, significantly influences performance by affecting how multiple threads access and use the available registers. When a few threads monopolize the register file, it can lead to a degradation in overall performance. Register capping is a method of limiting the number of rename registers each thread is allowed to use. The method has been used to improve performance. However, to understand the behavior of multiple threads sharing a register file and the effect that capping has on different combinations of programs, an analytical tool to observe the register file's dynamics is necessary. To address this, this paper develops a theoretical model utilizing Markov Chain to analyze rename register utilization in SMT systems. The model proposed incorporates capping to examine the dynamics of register allocation and usage patterns among concurrent threads across different threads. By varying parameters such as cap value and consumption rates, the model reveals insight into the behavior of register sharing and its impact on overall system performance.

General Terms

Simultaneous Multi-threading, Register File, Queuing Theory

Keywords

Simultaneous Multi-threading, Register Rename, Markove Chain, Queuing Theory

1. INTRODUCTION

Simultaneous Multi-Threading (SMT) is a technique that allows a single processor core to run multiple threads at the same time, enhancing overall system performance. By leveraging thread-level parallelism (TLP), SMT ensures that threads can take advantage of otherwise idle processor resources. For instance, if one thread is stalled while waiting for data, another thread can be executed on the same core during the stall, effectively boosting processor utilization. It is important to recognize that while SMT can leverage Thread-Level Parallelism (TLP) to enhance performance by overcoming the limitations of Instruction-Level Parallelism (ILP) within a single thread [1, 2], it can also introduce performance drawbacks due to increased resource contention and lower cache hit rates. Optimizing SMT to fully exploit TLP requires careful resource management to minimize contention among threads. Numerous studies have been conducted on resource-sharing algorithms. For example, at core level, allocating threads with less inter-thread interference to a single core can improve performance [3]. At thread level, reseach in [4] proposes the ICOUNT policy, which prioritizes threads with lower occupancy in pre-issue stages during the fetch stage. DCRA, proposed in [5], is another resource sharing algorithm for the fetch stage, adjusts resource allocation based on memory performance, giving more resources to threads that utilize them more efficiently. Blocking inefficient threads from fetching to prevent them from taking up too much resources is a method proposed in [6] to improve fair resource utilization.

The rename stage is an early phase in which resources are shared and efficient distribution of the registers at this stage can have a substantial impact on the overall performance of the pipeline. Capping proposed in [7] is a method that limits the number of registers a thread is allowed to use. Capping prevents inefficient threads from taking up too many resources, causing resource contention. When resource contention occurs, inefficient threads hog too much resources, limiting resources for efficient threads, causing poor overall performance.

To demonstrate the existence of resource contention and its effect on performance, a Markov Chain model is used to represent the rename register file as a queue and introduce an inefficient thread to the model. This model takes the instruction input and output rates into the register file as input and calculates the percentage of time the register file remains in each state. These percentages form the basis for determining thread utilization and overall performance. In addition, the model can be modified to incorporate capping, providing insight into how capping affects performance.

2. BACKGROUND

2.1 Simultaneous Multi-Threading

Simultaneous Multi-Threading (SMT) enhances resource utilization in modern processors by enabling the parallel execution of multiple independent threads, thereby improving overall performance. By allowing multiple threads to run concurrently, SMT leverages Thread-Level Parallelism (TLP) to maximize resource usage, particularly when Instruction-Level Parallelism (ILP) within a single thread is insufficient [1]. Figure 1 shows

the pipeline stages of a 4-threaded system, which is based on a conventional out-of-order superscalar processor.

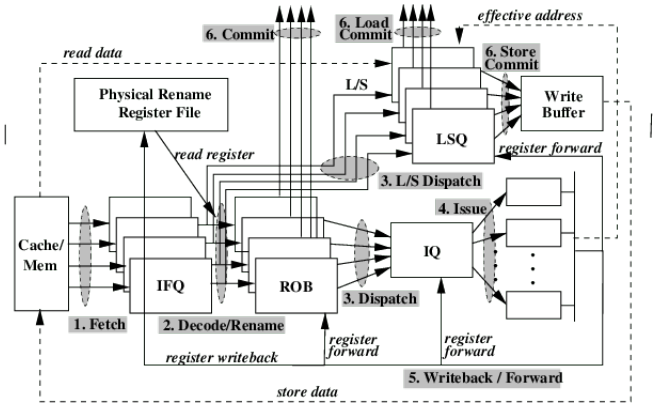


Fig. 1: **Pipeline Stages in a 4-threaded SMT System**

First, instructions from each thread are *fetched* from memory (and cache) and placed into their respective Instruction Fetch Queue (IFQ). After passing through the *decode* and register-*rename* stages, they are allocated to their corresponding Re-Order Buffer (ROB) and *dispatched* to the shared Issue Queue (IQ). Load/Store instructions have their address calculation operations sent to the IQ while their operations are also directed to individual Load Store Queues (LSQ). Once the issuing conditions are met (i.e., all operands are ready and the required functional unit is available), the operations are *issued* to the appropriate functional units, and their results are either *written back* to their destination registers or forwarded as needed to the IQ or LSQ. Upon computing their addresses, Load/Store instructions initiate their memory operations. Finally, all instructions are committed from the ROB in program order, ensuring synchronization with Load/Store instructions in the LSQ.

SMT processors generally share key datapath components among multiple independent threads. These shared resources may include the physical register file, machine bandwidth, inter-stage buffers (such as the Issue Queue), functional units, and write buffers. By sharing resources, an SMT system can significantly reduce hardware requirements while achieving throughput comparable to multiple instances of superscalar processors.

Efficient allocation of shared resources among simultaneously executing threads is critical for optimal performance in an SMT system. Without effective distribution, a small subset of threads can disproportionately occupy shared resources, leading to resource starvation for other threads and affecting overall system performance. Among these shared resources, the physical register file plays a crucial role in eliminating register name dependencies during the rename stage, which is the first stage in which shared resources are utilized. An imbalanced allocation of the physical register file can quickly become a bottleneck across pipeline stages. Therefore, this paper focuses on optimizing the distribution of the physical register file among multiple threads to enhance overall system efficiency.

2.2 Physical Register File

In this section, the concept of register renaming and its implementation is introduced. Register renaming is a crucial technique used to eliminate name dependencies, such as write-after-read (WAR) and write-after-write (WAW), which occur when registers are reused. This method has become widely adopted in modern processors [8, 9] as it is essential for enabling out-of-order execution. By assigning distinct physical registers to the same architectural register across different instructions, register renaming effectively removes false dependencies. Consequently, later instructions can execute out-of-order without being constrained by earlier instructions.

For example, consider the following program segment, which contains false dependencies. Out-of-order execution is impeded by write-after-read dependencies (e.g., between instructions γ and δ) and write-after-write dependencies (e.g., between instructions α and δ).

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

If register renaming is employed, $r1$ in instruction δ is assigned to a different physical register, allowing δ and subsequent instructions to execute before instruction γ. Multi-threaded processors typically implement a ”physical” register file containing more physical registers than the number of ”architectural” registers defined in the ISA. Whenever an instruction writes to an architectural register, a physical register is allocated and mapped to that architectural register. Subsequent read instructions referencing the same architectural register retrieve their data from the most recently assigned physical register. This mapping between architectural and physical registers is maintained in the rename table. Consequently, the availability of physical registers is a critical factor in system performance. A physical register is allocated at the renaming stage and remains occupied until the next write to the same architectural register is committed. One way to increase register availability is to accelerate the deallocation process. A modification proposed in [10] expedites register deallocation process to reduce occupation time; however, it requires software support from the operating system. Another approach, presented in [11], delays register allocation until the instruction reaches the commit stage. The tradeoff of this method is the risk of an instruction failing to find a free register at commit time, potentially causing a deadlock. Modifying the allocation-deallocation process is challenging without additional OS support or substantial hardware overhead. However, improvements can still be made while preserving the in-order allocation and deallocation mechanism. A more efficient utilization scheme for physical registers can enhance availability and system performance. Prior research [7] has indicated that competition for floating-point registers is less intense than for integer registers. Therefore, this paper focuses exclusively 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 been allocated its required number of architectural registers. For instance, in an ISA with 32 architectural registers, each thread is guaranteed to have at least one physical register mapped to every architectural register. If extra registers remain after allocating 32 registers per thread, these additional registers are shared among all threads. Figure 2 illustrates the organization

of the register file [12] in this context. The total number of registers (R_t), the number of architectural registers per thread (R_a), and the number of extra registers available for renaming (R_r) are related as follows:

$$R_r = R_t - N \times R_a \tag{1}$$

where N is the number of threads in the system.
In single-threaded systems, if a thread retains too many rename registers for too long, execution stalls until a previously issued instruction commits and releases a register. This issue can be mitigated by increasing the size of the register file. However, in multi-threaded systems, threads commit instructions independently, leading to slower threads occupying more registers while faster threads release them. Without making R_t excessively large, an effective distribution scheme for rename registers is necessary to prevent slow threads from degrading overall system performance.

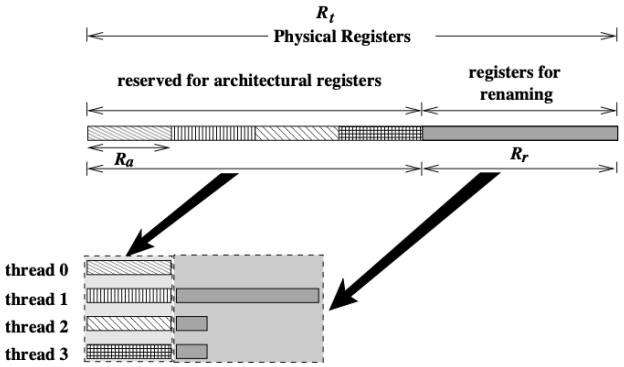


Fig. 2: **Organization of a shared Physical Register File**

There are many studies on improving performance by efficiently allocating the Rename Register File. For example, the capping technique proposed in [7] to limit the number of registers each thread is allowed use. Capping dynamically based on run-time behavior is implemented in [13] for optimum performance. Taking it one step further, Reinforcement Leaning is used in [14] to adjust the cap value of each thread for even better optimization.
To aid in the development of resource sharing algorithms, this paper proposes a mathematical model of the Rename Register File utilization using a Markov Chain model. The model provides insight into the dynamics of the Rename Register File when shared among multiple threads. The model also features capping to mathematically show that recourse allocation improves overall performance when there is contention.

2.3 Queuing Model

In queuing theory, there are many types of queues. The solution to a queue can provide helpful information about the phenomena that queue models [15]. Queuing models have many applications in computer science, such as thread I/O and job scheduling [16]. Currently, there are no available queuing networks to model the Rename Register File that has capping as a feature. To achieve the goal, the model is build on existing work that utilizes queuing theory to model the issue queue in an SMT CPU proposed in paper [17], this approach is extended to the rename stage. The issue queue model provides a solid framework for understanding how instructions are scheduled and executed across multiple threads. The model proposed will incorporate key parameters such as instruction arrival rates, register deallocation rates, and queue lengths to evaluate performance metrics like throughput, and utilization. By comparing these metrics across different parameter combinations, the model provides insights into the optimization of resource allocation in SMT CPUs, ultimately enhancing overall system performance. This work not only validates the applicability of queuing theory to various stages of the CPU pipeline but also offers a comprehensive analytical tool for CPU designers to optimize SMT architectures.
We consider the Rename Register File (RRF) as a finite queue and instructions enter the queue when it is renamed and leave the queue when the register is deallocated. According to the general hypothesis of queuing theory, the number of instructions from a given thread sitting in the RRF can be determined with the following parameters:

—The bandwidth of execution after an instruction has been renamed (i.e. how many instructions can be released in a clock cycle).
—How likely an instruction from this thread is renamed per clock cycle.
—How likely an instruction from this thread is released per clock cycle.

The first parameter can be derived by generalizing all the stages after the rename stage into one execution block. The other two parameters are program dependent. This model does not aim to produce the results of an actual simulation, it rather aims to analyze the general utilization trend of the RRF. Therefore, these parameters should be chosen by the user to observe how different combinations of programs affect the utilization of the rename of the register.
Considering that it is possible to model the RRF as a finite queue, Markov chain is employed to model its transitions. A Markov chain represents a sequence of events where the probability of transitioning to the next state depends solely on the current state. The RRF can model it as a Markov chain with states defined by the combination of instructions currently present in the RRF. For example, when there are two instructions from thread zero and four instructions from thread one, it can be said that it is state $\langle 2, 4\rangle$ in the Markov chain.
A simple complete Markov chain example is shown in Figure 3. In a graphical representation of a Markov chain, states are shown as circles with their labels in the center, and transitions are depicted as arrows with their probabilities indicated nearby. Following standard methods, given the states and transition probabilities of a Markov chain, a transition matrix can be constructed. This is a two-dimensional matrix with a row and column for each state, describing the transition probabilities between states. A transition matrix provides a compact and mathematically rigorous way to represent a Markov chain. To create a transition matrix, list all the states and assign a row and column to each. The matrix, typically denoted as P, contains entries P_i, P_j that correspond to the probability of transitioning from state i to state j in one time step. The transition matrix P for the flow diagram in Figure 3 becomes:

$$P = \begin{bmatrix} 0.6 & 0.4 & 0 \\ 0.2 & 0.5 & 0.3 \\ 0 & 0.9 & 0.1 \end{bmatrix} \tag{2}$$

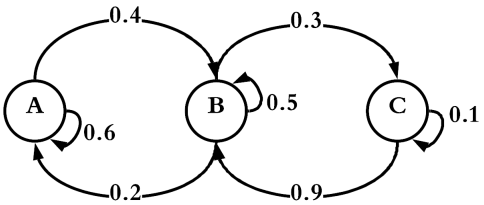


Fig. 3: **Simple Markov Chain**

In addition to being concise, a transition matrix allows us to extract important properties of the underlying Markov chain. The main interest is in the steady-state distribution. For a given transition matrix P, the steady-state distribution, commonly denoted π, is a row vector that exhibits the property

$$\pi P = \pi \tag{3}$$

and importantly, each element π_i represents the steady-state probability of state i, which is the percentage of time the Markov chain will be in state i as it transitions indefinitely. This is obtained by calculating the eigenvector of matrix P with an eigenvalue equal to 1. Finding the steady-state distribution for P in Equation 1 leads to

$$\pi = \begin{bmatrix} 0.27 & 0.55 & 0.18 \end{bmatrix} \tag{4}$$

This means that, in the long run, the Markov chain will spend approximately 27% of the time in state A, 55% of the time in state B, and 18% of the time in state C. Given that a transition matrix P can be derived for an RRF modeled as a queue and then extract the steady-state distribution π, it is possible to determine how often the RRF will be in each state. Recall that a state of the RRF represents a combination of instructions of each thread. Therefore, extracting π informs us how frequently there will be, for example, 1 instructions from thread 1 and 2 instructions from thread 2 in the rename register file. Further more, the consumption rate is an approximation of how likely a thread releases a register, this action happens at the commit stage. The IPC can be approximated if how many registers the thread is holding is known, and what is the probability that those registers are released. With the distribution given by the matrix and the consumption rate, the IPC of each thread can be calculated by multiplying the percentage of time the machine spends in each state by the number of register each according state holds then multiply by the consumption rate. This allows us to observe the change in IPC as the cap value, or consumption rate, or arrival rate changes.

3. SIMULATION ENVIRONMENT

3.1 Simulator

M-sim [18], a multi-threaded microarchitectural simulation environment, is used as a comparison point for the proposed model. M-sim provides all the necessary features for this study, including models of 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. Additionally, M-sim supports the simulation of both single-threaded and simultaneous multi-threaded processors. The configuration of the simulated processor is detailed 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 performance of the simulated processor is evaluated using a mix of benchmarks from the SPEC CPU2006 benchmark suite [19]. Each program is simulated in a SimpleScalar environment and classified based on its Instruction-Level Parallelism (ILP) following the procedure outlined in the SimPoints tool [20]. The benchmarks are categorized into three ILP levels: high, medium, and low. Table 2 presents 4-threaded workload mixes with various ILP combinations, designed to represent diverse computational workloads.

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 2 provide 2-threaded mixes of different ILP combinations to help evaluate the model later on.

Table 3. : **SPEC CPU2006 2-threaded Mixes**

Mix	Benchmarks	Classification(ILP)		
		Low	Med	High
Mix1	gcc, bzip2	1	0	1
Mix2	dealII,xalancbmk	1	0	1
Mix3	libquantumn, sjeng	0	2	0
Mix4	hmmer, povray	0	2	0
Mix5	gobmk, sjeng	0	1	1
Mix6	gromacs, libquantumn	0	1	1

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

$$\begin{aligned} \text{Overall\_IPC} &= \sum_{i=1}^{N} \text{IPC}_i \\ &= \text{(whatever second line)} \end{aligned} \tag{5}$$

where N denotes the number of threads that run simultaneously in the system and IPC_i denotes the IPC of each thread.

4. MOTIVATION

Previous studies [7, 12] have demonstrated that the imbalance in rename register distribution can be unexpectedly severe. Figure 4 illustrates the average percentage of rename registers occupied by each thread, sampled every 50 clock cycles, in a 4-threaded environment using workload mixes from Table 2, with a register file size of 160.

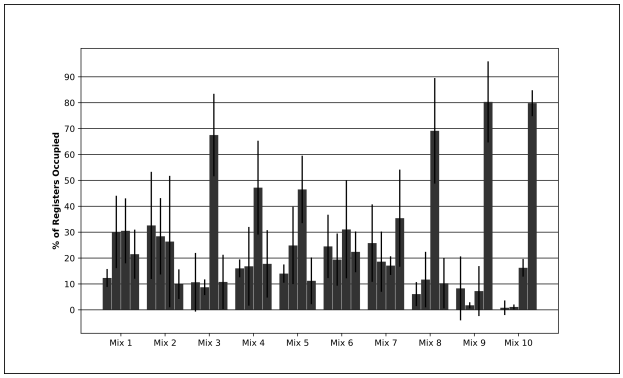


Fig. 4: **Average Register Occupancy of Each Thread**

Mixes with minimal differences among threads do not exhibit extreme competition for registers. Figure 4 also shows one standard deviation from the average, where a larger standard deviation signifies a greater fluctuation in the percentage of occupied registers. On average, Mix 3, Mix 8, Mix 9, and Mix 10 experience intense competition, as indicated by a single thread that occupies at least 65% of the registers in at least 50% of the sampling points. Despite the high fluctuation observed in Mix 3 and Mix 9, a single thread still dominates approximately 55% and 65% of the registers for 84.1% of the time, respectively. This dominance of resources can lead to significant performance degradation to the overall system.

To prevent a situation where a single thread dominates register usage, a capping technique was proposed [7]. This technique imposes a limit, referred to as the "cap value," on the number of rename registers a thread can use at any given time. When a thread reaches the cap value, it is no longer allowed to rename, even if free registers are available during its turn. These free registers are reserved for threads that are using fewer registers. Figure 5 shows, for all mixes in 2, the average difference in IPC between no capping versus when capping is implemented 2. Notice that there is an improvement in performance starting from cap value of two; This is because, for some mixes, there is substantial improvement even with low cap values.

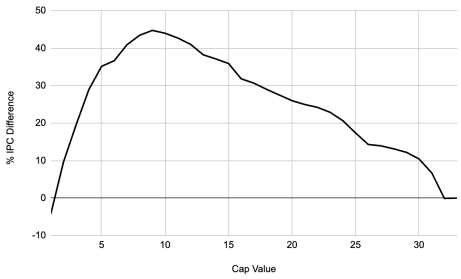


Fig. 5: **% IPC Difference vs. Cap Value**

Considering existing evidence, an explanation for why capping is effective is proposed. Threads in lower ILP classification hold on to a register longer than threads in higher ILP classification. However, threads are renamed in round robin. Lower ILP threads will continue to rename regardless of how many of their instructions are still holding the register, resulting in higher utilization of the RRF. Without a scheme to control, low ILP threads will slowly hog up the majority of the registers, leaving few registers for high ILP thread, causing them to perform poorly. Capping works because it stops low ILP threads from reaming if they are already holding a certain amount of registers. The next chapter will confirm this explanation with a theoretical model.

5. SINGLE-THREADED RRF MODEL

We begin by considering a system that has instructions only from one thread. To model this system as a Markov chain, the initial step is to describe the state space. The state space of an RRF Markov chain model comprises all possible combinations of RRF occupancy. Therefore, the state space for a system with instructions from only one thread can be described as $S = \{s_i | i \in \mathbb{Z} \cap [0, N_Q]\}$ Where N_Q is the size of the RRF and s_i represents the state when the RRF has exactly i instructions currently residing in it. This means that the state space consists of one state for each possible number of instructions up to N_q, the size of the RRF. Figure 6 shows a graphical representation of the state space.

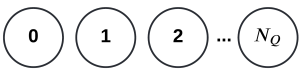


Fig. 6: **State Space of Single Thread Model**

where N_Q is the size of the RRF and s_i represents the state when the RRF has exactly i instructions currently residing in it. This

means that the state space consists of one state for each possible number of instructions up to N_q, the size of the RRF. Figure 5 shows a graphical representation of the state space.
To complete the Markov chain model, augment the state space with transition probabilities between each pair of states. The transitions of interest are the changes in the number of instructions in the RRF after each clock cycle. In each clock cycle, two events affect the number of instructions in the RRF:

—Rename to the RRF

—Instruction released after subsequent instruction is committed

For each of these two events, derive a transition matrix. First, derive a transition matrix C which represents the consumption of instructions out of the RRF in any clock cycle. Then, derive a transition matrix A which represents the flow of instructions into the RRF in any clock cycle. By multiplying these matrices together, the complete transition matrix of the RRF can be derived; that is, $P = C \times A$.

5.1 Single-Threaded Consumption Model

The section demonstrates the construction of the consumption matrix C to represent the consumption of instructions from the RRF. The number of instructions released is limited by two factors:

—The number of instructions ready to be released.

—Execution bandwidth

Instructions are renamed and hold on to that instruction until the subsequent instruction that uses the same architectural register commits. Therefore, at any given time, the number of instructions that are candidates for released is probabilistically distributed based on the expectation of how many are ready to be released.
Consider the transition from state s_i to state s_j. If $j > i$, the transition probability should be zero during the release stage since it is not possible to release a negative number of instructions. Thus:

$$p_{s_i,s_j} = 0, \text{ if } i - j < 0 \tag{6}$$

where p_{s_i,s_j} denotes the transition probability.
Next, due to the fact that the RRF cannot release more instructions than the execution bandwidth. With F represent execution bandwidth:

$$p_{s_i,s_j} = 0, \text{ if } i - j > F \tag{7}$$

We now consider the case where $i - j = F$, meaning the entire execution bandwidth is fully utilized. Let ρ represent the probability that an instruction is ready to be released, and assume that each instruction in the RRF is ready to be released. Given i instructions in the RRF, the transition probability for this situation (at least F instructions are ready) can be calculated as:

$$p_{s_i,s_j} = \sum_{k=F}^{i} \binom{i}{k} \rho^k (1-\rho)^{i-k}, \text{ if } i - j = F \tag{8}$$

Lastly, to take in consideration of the case where $0 \leq i - j < F$, meaning fewer instructions are ready to be released than execution bandwidth. This scenario reduces to the probability that exactly $i - j$ instructions in the RRF are ready to be released given i instructions in the RRF. Therefore:

$$p_{s_i,s_j} = \binom{i}{i-j} \rho^{i-j} (1-\rho)^j, \text{ if } 0 \leq i - j < F \tag{9}$$

For the special case when $j = 0$, meaning every instruction in the RRF was released, this equation simplifies to:

$$p_{s_i,s_j} = \binom{i}{i} \rho^i (1-\rho)^0 = \rho^i \tag{10}$$

This represents the probability of all instructions in the RRF are ready to be released.
To summarize the transition probabilities from state s_i to state s_j during the release stage:

$$p_{s_i,s_j} = \begin{cases} 0, & \text{if } i - j < 0 \\ \binom{i}{i-j} \rho^{i-j} (1-\rho)^j, & \text{if } 0 \leq i - j < F \\ \sum_{k=F}^{i} \binom{i}{k} \rho^k (1-\rho)^{i-k}, & \text{if } i - j = F \\ 0, & \text{if } i - j > F \end{cases} \tag{11}$$

We can use these cases to populate a release transition matrix (consumption matrix) for the Markov model of an RRF.
To modify this model to implement capping, any transition where the initial state, s_i, is greater than the cap value is 0 because it is not possible to have more registers than the cap value. Therefore to generate transition probabilities during release stage with a cap value of N_{cap} is :

$$p_{s_i,s_j} = \begin{cases} 0, & \text{if } i - j < 0 \\ \binom{i}{i-j} \rho^{i-j} (1-\rho)^j, & \text{if } 0 \leq i - j < F \text{ and } i \leq N_{cap} \\ \sum_{k=F}^{i} \binom{i}{k} \rho^k (1-\rho)^{i-k}, & \text{if } i - j = F \text{ and } i \leq N_{cap} \\ 0, & \text{if } i - j > F \\ 0, & \text{if } i > N_{cap} \end{cases} \tag{12}$$

5.2 Single-Threaded Arrival Model

This section introduce the derivation the transition matrix A to represent the arrival of new instructions into the RRF. During the rename stage of the pipeline, instructions are decoded and renamed. The number of instructions renamed depends on:

—The number of instructions decoded.

—The number of empty entries in the RRF

Assume that the number of instructions decoded in a random clock cycle (i.e., the arrival rate of instructions) follows a probability mass function (pmf) $a(x)$. By neglecting the effect of limited bandwidth and given $a(x)$, the instruction arrival model can be constructed for any state of the RRF.
Consider a system in state s_i. For any state s_j, a Markov chain to be built to model the probability of transitioning from state s_i to state s_j during the rename stage of the pipeline by partitioning the range of values that i and j may take on.
Firstly, observe that the RRF cannot lose instructions during the rename stage (at least 0 must arrive). Therefore:

$$p_{s_i,s_j} = 0, \text{ if } j < i \tag{13}$$

Secondly, consider the case $i \leq j < N_Q$. This transition implies that $j - i$ instructions have been renamed, but the RRF is not full. In this scenario, the probability of transitioning to state s_j is the probability that exactly $j - i$ instructions are renamed. Since the arrival of instructions is modeled by $a(x)$, we have:

$$p_{s_i,s_j} = a\,(j-i)\,,\ \text{if } i \le j < N_Q \tag{14}$$

Thirdly, consider the case $j = N_Q$. Transitioning from state s_i to state s_j implies that enough instructions have arrived to fill the RRF. Since this is the final case in a partition of the probability space, the likelihood of this case is the complement of the total likelihood of the previous cases:

$$p_{s_i,s_j} = p_{s_i,s_{N_Q}} = 1 - \sum_{\substack{s_k \in S \\ k \neq N_Q}} \alpha\,(k-i)\,,\ \text{if } j = N_Q \tag{15}$$

To summarize the transition probabilities from state s_i to state s_j during rename:

$$p_{s_i,s_j} = \begin{cases} 0, & \text{if } j < i \\ a\,(j-i)\,, & \text{if } i \le j < N_Q \\ 1 - \sum\limits_{\substack{s_k \in S \\ k \neq N_Q}} \alpha\,(k-i)\,, & \text{if } j = N_Q \end{cases} \tag{16}$$

Suppose the instruction arrival rate into the RRF follows a Poisson distribution with a mean λ equal to 1. That is, the number of instructions ready for rename follows the distribution:

$$a\,(k) = \frac{\lambda^k e^{-\lambda}}{k!} = \frac{e^{-1}}{k!} \tag{17}$$

where $a\,(k)$ is the probability of k instructions being ready rename to the IQ in any clock cycle. By using the principles outlined above, it is possible to systematically populate the arrival-stage transition matrix A for the RRF's Markov model.

To modify this model to feature capping, consider the RRF to have the same size as the cap value (N_{cap}). Since it is not possible to have more register than the cap value

$$p_{s_i,s_j} = 0,\ \text{if } j > N_{cap} \tag{18}$$

Therefore to generate transition probabilities from state s_i to state s_j during rename, with a cap value of N_{cap}:

$$p_{s_i,s_j} = \begin{cases} 0, & \text{if } j < i \\ a\,(j-i)\,, & \text{if } i \le j < N_{cap} \\ 1 - \sum\limits_{\substack{s_k \in S \\ k \neq N_{cap}}} \alpha\,(k-i)\,, & \text{if } j = N_{cap} \\ 0, & \text{if } j > N_{cap} \end{cases} \tag{19}$$

5.3 Single-Threaded Complete Model

We have partitioned the RRF's behavior during each clock cycle into two components: the consumption model C to represent the instructions leaving the RRF, and the arrival model A to represent the arrival of new instructions into the RRF. Now, to model the change in the RRF between clock cycles by combining these two models. Specifically, the usage of the arrival and consumption models to comprehensively describe the behavior of the RRF within a single Markov model.

Consider an RRF in state s_i at the beginning of a clock cycle, meaning there are currently i instructions in the RRF. During the next clock cycle, the RRF undergoes two changes: the release stage, followed by the rename stage. To determine the probability of transitioning from state s_i to state s_j during one clock cycle, it is necessary to add the events needed to happen such that after the rename stage, RRF is in stage s_j. We can rewrite the transition process as:

$$s_i \xrightarrow{C} \text{post-release stage} \xrightarrow{A} s_j \tag{20}$$

Denote the post-release stage as some arbitrary state denoted as s_m:

$$s_i \xrightarrow{C} s_m \xrightarrow{A} s_j \tag{21}$$

The transition $s_i \xrightarrow{C} s_m$ denotes the transition from the initial state to any arbitrary states, meaning the i^{th} row of the consumption matrix. From all of these arbitrary states it is necessary to transition to the targeted state $s_m \xrightarrow{A} s_j$. Each arbitrary state s_m can be transitioned to target state s_j by multiplying the i^{th} row by the j^{th} column in the arrival matrix. Summing up all of these transitions gives us the probability $s_i \to s_j$. This operation is the same as the dot product of the i^{th} row of the consumption matrix and the j^{th} column of the arrival matrix. Therefore, to compute the complete model

$$P = C \times A \tag{22}$$

6. MULTI-THREADED RRF MODEL

In this chapter, a system with instructions from arbitrarily-sized set of unique threads $\{I_1, I_2, \cdots, I_T\}$ is considered, which means a system with instructions from T different threads for some $T \in N$, where each I_t denotes a unique thread. A system like this one resembles a realistic SMT processor where instructions come in from different threads. We take the simple probability models derived in Chapter Four and show that by using the joint probabilities of instructions from each thread, it is possible to develop a model of RRF with multiple number of threads.

6.1 State Space And Matrix Representation

We define the state space and the assignment of a state to an RRF that can contain instructions from multiple threads. In a system with T threads, the RRF at any given clock cycle can be characterized by the number of instructions from each thread. Let n_t be the number of instructions of thread I_t currently residing in the RRF. A state can be denoted as a T-tuples $\langle n_1, n_2, \ldots, n_T \rangle$. The collection of all possible T-tuples, the state space of the system, which can be denote as:

$$S = \left\{ \langle n_1, n_2, \ldots, n_T \rangle \,|n_t \in [0, N_Q] \cap Z, \sum_t n_t \le N_Q \right\} \tag{23}$$

where T is the number of unique threads and N_Q is the size of the RRF. Each state is denoted as a list of integers n_t with $0 \le n_t \le N_Q$. In addition, there are only N_Q registers total; therefore the sum of these integers cannot be larger than N_Q. A graphical representation of the state space of a two-thread model with $N_Q = 3$ is presented in Figure 7.

Using the theory of permutation and combination, the number of states for a system with T threads and an RRF size of N_Q is:

$$|S| = \frac{(T + N_Q)!}{N_Q! T!} \tag{24}$$

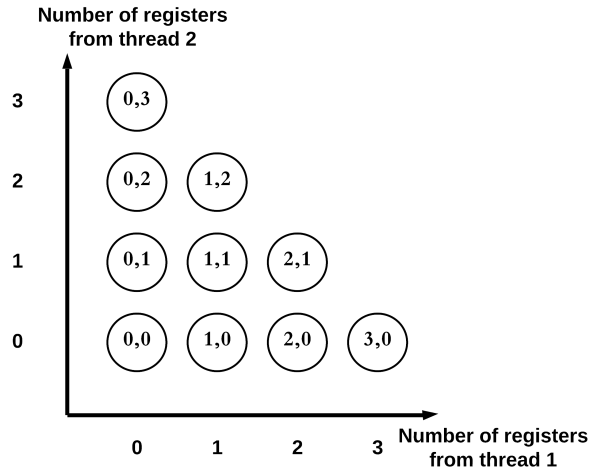


Fig. 7: **State Space Of Two-Threaded Model**

this reveals the size of the state space to be of exponential of T and N_Q. When the number of threads and number of registers in the RRF increase, state space of the model increases exponentially of T and N_Q. Therefore, it is computationally expensive to study four-threaded environments where the RRF has to be bigger to accommodate all threads. The subsequent subchapters present a general model for any arbitrary number of threads and registers. However, the configuration of the model that is used for analysis is a two threads and RRF size of 32 registers.

To model a multi-thread system, utilize transition matrices to represent the usage of the RRF and its transitions during both the arrival and consumption stages of the pipeline. Previously, when dealing with a single thread, matrices of size $|S| \times |S|$ are used , where S is the state space and $S = N_Q+1$. Each state s had its own row and column, with the entry $P_{i,j}$ representing the probability of the transition $s_i \rightarrow s_j$. In the multi thread scenario, following a similar approach, but states are now lists of integers rather than integers. For each model, use a transition matrix of size $|S| \times |S|$. Each state $\langle n_1, n_2, \ldots, n_T \rangle$ is mapped to an index and assigned one row and one column in the matrix. Each element of the matrix $P_{\langle i_1,i_2,\ldots,i_T\rangle,\langle j_1,j_2,\ldots,j_T\rangle}$ represents the probability of the transition from state $\langle i_1, i_2, \ldots, i_T \rangle$ to state $\langle j_1, j_2, \ldots, j_T \rangle$.

6.2 Multi-Threaded Consumption Model

First part of the model is to derive a model for the release stage and build a transition matrix C to describe the probabilities of RRF transition when registers are released. Chapter Five present the equation to compute the probability of transition for an RRF with instructions from one thread when instructions are released in one clock cycle based on the current state of the RRF and the consumption rate. This equation can be used as a marginal probability and show that the transition probability in a system with instructions from an arbitrary number of unique threads can be derived from the joint probability of instructions from all threads described in the single-threaded model.

Let us consider a transition from state ⟨2,1⟩ to state ⟨1,0⟩. Given that the execution bandwidth F is 2 and $(i_1–j_1)+(i_2–j_2) = 2$, this transition utilize the entire bandwidth and will have the probability equal to the sum of probabilities of having at least (1,1) instructions have been in the RRF long enough to be released but only (1,1) are executed and ready to be released. (n_1, n_2) depicts a combination of n_1 instructions from thread I_1 and n_2 instructions from thread I_2. Given that there is an execution bandwidth, there are three cases to consider.

6.2.1 Case $\sum_t (i_t - j_t) > F$.
In this case, the number of registers being released is greater than the execution bandwidth. It is not possible to release more than the execution bandwidth, therefore:

$$p\left(s_i \xrightarrow{C} s_i\right) = 0 \tag{25}$$

6.2.2 Case $\sum_t (i_t - j_t) < F$.
Let us look at these states in more details. Suppose the RRF is in some state $s_i = \langle i_1, i_2, \ldots, i_T \rangle$ transitioning to destination state $s_j = \langle j_1, j_2, \ldots, j_T \rangle$. For this transition to occur, for each thread t the RRF releases $i_t - j_t$ instructions. Therefore, the transition probability of this case is the joint probability of the independent events that is each thread t releases $i_t - j_t$ instructions. This results in:

$$p\left(s_i \xrightarrow{C} s_i\right) = \prod_{t=1}^{T} p^{[t]}_{i_t,j_t} \tag{26}$$

where $p^{[t]}_{i_t,j_t}$ is the probability in the consumption matrix in the single-threaded RRF model for thread t. For example, in a three-threaded system with threads $\{I_1, I_2, I_3\}$ with execution bandwidth of 10, the probability of transitioning from state ⟨5, 6, 7⟩ to state ⟨2, 3, 4⟩ is given by:

$$C_{\langle 5,6,7\rangle,\langle 2,3,4\rangle} = C^{[1]}_{5,2} \cdot C^{[2]}_{6,7} \cdot C^{[3]}_{7,4} \tag{27}$$

Thus this case can be calculated with:

$$C_{s_i,s_j} = \prod_{t=1}^{T} C^{[t]}_{i_t,j_t} \tag{28}$$

6.2.3 Case $\sum_t (i_t - j_t) = F$.
This case exhausts the execution bandwidth. In the single-threaded model, this is the transition where $i - j = F$ which is the probability that at least $i - j$ instructions are ready to be released. Only on thread is using the entire execution bandwidth. However, in the multi-threaded model, multiple threads share the execution bandwidth, resulting in various instances where the execution bandwidth becomes fully utilized. For example, an RRF size of and an execution bandwidth of 2, if the initial state is ⟨2,3⟩, states ⟨0,1⟩, ⟨1,1⟩, ⟨2,1⟩ take up the entire execution bandwidth. These states are called Boundary States. Figure 8 shows the available states if the initial state is ⟨2,3⟩ with red states being non-reachable states because they hold more registers in either thread than the starting state and the gray states being non-reachable states due to the execution bandwidth.

Looking at the example provided earlier where the RRF transitions from state ⟨2,1⟩ to state ⟨1,0⟩, this probability can be derived from the cases where at least two instructions are ready to be released and the combination of the first two instructions to be released is one from thread I_1 and one from thread I_2.

We first derive the probability that at least two instructions are ready to be released by subtracting the probability that fewer than

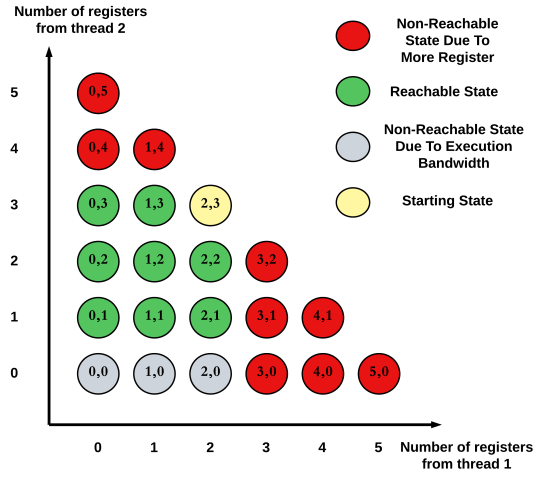


Fig. 8: **Reachable and Non-Reachable States Due To Execution Bandwidth**

two instructions are ready to be released from 1. To achieve this, iterate over the destination states that satisfies $\sum_t (i_t - j_t) < F$ (which is 2 in this case), where $\langle i_1, i_2 \rangle$ is $\langle$2,1$\rangle$ and sum all their probabilities. If the probability of at least two instructions are ready to be released is denoted as $p(2^+)$, and let S* represent the set of all reachable states that are not boundary states:

$$p\left(2^+\right) = 1 - \sum_{s_k \in S^*} p\left(\langle 2,1 \rangle \xrightarrow{C} s_k\right) \tag{29}$$

where each term $p\left(\langle 2,1 \rangle \xrightarrow{C} s_k\right)$ are available from the previous case since those events do not lead to a boundary state.
After obtaining $p\left(2^+\right)$, another probability needed to be considered is the probability of reaching only reaching state $\langle$1,0$\rangle$. This means that one of the first two instructions is from thread I_1 and the other is from thread I_2. The total number of combinations can be calculated as $\frac{(1+1)!}{1!1!} = 2$, each with a probability of

$$P^1_{I_1} \cdot P^1_{i_2} \tag{30}$$

where P_{I_1} and P_{I_2} represents the probability that an instruction ready to be released is from thread I_1 and I_2 respectively. This probability can be computed from the consumption rate of that thread as follows:

$$P_{I_1} = \frac{\rho_1}{\rho_1 + \rho_2}, P_{I_2} = \frac{\rho_2}{\rho_1 + \rho_2} \tag{31}$$

Putting the two parts together, it is possible to calculate the probability of the transition to a boundary state in the example of $\langle 2,1 \rangle \rightarrow \langle 1,0 \rangle$ with

$$\left[1 - \sum_{s_k \in S^*} p\left(\langle 2,1 \rangle \xrightarrow{C} s_k\right)\right] \frac{(1+1)!}{1!1!} P^1_{I_1} \cdot P^1_{i_2} \tag{32}$$

This equation can be generalized for transition from arbitrary state $s_i = \langle i_1, i_2, \ldots, i_T \rangle$ to arbitrary state $s_j = \langle j_1, j_2, \ldots, j_T \rangle$ where $\sum_t (i_t - j_t) = F$ as:

$$P\left(s_i \xrightarrow{C} s_j \middle| \sum_t n_t = F\right) = \left[1 - \sum_{s_k \in S^*} \left(s_i \xrightarrow{C} s_k\right)\right] \frac{(\sum n_t)!}{\prod n_t!} \left[\prod_{t=1}^{T} p_{I_t}^{n_t}\right] \tag{33}$$

where $n_t = i_t - j_t$ depicts the number of instructions from thread I_t that are ready to be released and S^* is the set of all reachable states:

$$S^* = S \backslash \left\{ \langle n_1, n_2, \ldots, n_T \rangle \,|\, \sum_i n_i = F \right\} \tag{34}$$

6.3 Multi-Threaded Arrival Model

Similar to the state space definition in the consumption model, for the transition $s_i \xrightarrow{A} s_j$ to occur during the rename stage, there are two conditions to consider.

6.3.1 Case $\sum_t j_t < N_Q$.
This case is when the RRF is not full at the end of the rename stage and is simple to derive. Similar to the consumption matrix, the transition probability is the joint probability of the independent events that $j_t - i_t$ instructions from each thread t are renamed to the RRF:

$$p\left(s_i \xrightarrow{A} s_j\right) = \prod_{t=1}^{T} p^{[t]}_{i_t, j_t} \tag{35}$$

where $p^{[t]}_{i_t, j_t}$ is the probability in the arrival matrix in the single-threaded RRF model for thread t. Thus the entires in the arrival matrix for the muli-thread model where $\sum_t j_t < N_Q$ follow

$$A_{s_i, s_j} = \prod_{t=1}^{T} A^{[t]}_{i_t, j_t} \tag{36}$$

6.3.2 Case $\sum_t j_t = N_Q$.
This case is similar to the case $\sum_t (i_t - j_t) = F$ in the consumption matrix. The RRF is filled up during the rename stage, creating boundary states and is more complex to derive. Unlike the consumption matrix where the boundary states may be different for each starting state, all starting states have the same boundary states due to the hard limit on how many instructions can be in the RRF regardless of how many instructions are currently in the starting state. For example, with RRF of size 3 and there are 2 threads like presented in Figure 6, states $\langle 0,3 \rangle, \langle 1,2 \rangle, \langle 2,1 \rangle$, and $\langle 3,0 \rangle$ are boundary states for any arbitrary starting state. Let us look at the transition from state $\langle 0,0 \rangle$ to $\langle 2,1 \rangle$. The probability can be derived from the events where at least three instructions are ready to be renamed and the combination of the first two instructions to be renamed is two from thread I_1 and one from thread I_2.
Similar to how the consumption matrix is derived, first calculate the probability that at least three instructions are ready to be renamed using the same formula

$$p\left(3^+\right) = 1 - \sum_{s_k \in S^*} p\left(\langle 0,0 \rangle \xrightarrow{A} s_k\right) \tag{37}$$

where S* represents the set of all states that $\sum_t j_t < N_Q$ which are non-boundary states and $p\left(\langle 0,0\rangle \xrightarrow{A} s_k\right)$ is the pobability of reaching those states.
Then, calculate the probability of reaching only reaching state $\langle 2,1\rangle$. This means that two of the first three instructions are from thread I_1 and the other is from thread I_2. The total number of combinations can be calculated as $\frac{(2+1)!}{2!1!} = 3$, each with a probability of

$$P_{I_1}^2 \cdot P_{i_2}^1 \tag{38}$$

where P_{I_1} and P_{I_2} represents the probability that an instruction ready to be renamed is from thread I_1 and I_2 respectively. This probability can be computed from the arrival rate of that thread as follows:

$$P_{I_1} = \frac{\lambda_1}{\lambda_1+\lambda_2}, P_{I_2} = \frac{\lambda_2}{\lambda_1+\lambda_2} \tag{39}$$

Putting the two parts together, it is possible to calculate the probability of the transition to a boundary state in the example of $\langle 0,0\rangle \rightarrow \langle 2,1\rangle$ with

$$\left[1 - \sum_{s_k \in S^*} p\left(\langle 0,0\rangle \xrightarrow{A} s_k\right)\right] \frac{(2+1)!}{2!1!} P_{I_1}^2 \cdot P_{i_2}^1 \tag{40}$$

This equation can be generalized for transition from arbitrary state $s_i = \langle i_1, i_2, \ldots, i_T\rangle$ to arbitrary state $s_j = \langle j_1, j_2, \ldots, j_T\rangle$ where $\sum_t j_t = N_Q$ as:

$$\begin{aligned} &P\left(s_i \xrightarrow{A} s_i \middle| \sum_t j_t = N_Q\right) = \\ &\left[1 - \sum_{s_k \in S^*} \left(s_i \xrightarrow{A} s_k\right)\right] \frac{(\sum n_t)!}{\prod n_t!} \left[\prod_{t=1}^{T} p_{I_t}^{n_t}\right] \end{aligned} \tag{41}$$

where $n_t = i_t - j_t$ depicts the number of instructions from thread I_t that are ready to be renamed and S^* is the set of all reachable states:

$$S^* = S \backslash \left\{ \langle n_1, n_2, \ldots, n_T\rangle \middle| \sum_i n_i = N_Q \right\} \tag{42}$$

6.4 Register Capping Modification

This section discuss the modifications needed to introduce capping into the multi-threaded model. For the consumption matrix, boundary states occur when the RRF release as many as the execution bandwidth allows. It is unrelated to the size of the RRF therefore the transition probabilities to boundary states are not affected when the number of registers each thread is allowed to use is capped. Capping can be implemented into the consumption matrix by setting the transition probabilities to and from states that have the sum of instructions from all the threads equal to zero like so

$$p(s_i \xrightarrow{C} s_j) = 0, \text{ if } \sum_t i_t > N_{cap} \text{ or } \sum_t j_t > N_{cap} \tag{43}$$

However, because the multi-thread consumption matrix is derived from the sing-thread consumption matrix and all of the states that utilize more than the cap has already been taken care of in the single-threaded matrix, it is not necessary to do this step. For the arrival matrix, boundary states occur when the RRF becomes full during the rename stage or when any thread reaches its cap value. The set of boundary states are directly affected if there is a limit to how many instructions each thread is allowed to use, therefore it is necessary need to make modifications to the arrival matrix. Let N_{cap} represents the cap value, T represents number of threads, and N_Q represents the size of the RRF. Boundary state is now defined as a state where either the RRF is full or one or multiple threads reach the cap value. Let us first calculate the transition probability to non-boundary states. Similar to the non-capping model, the transition probability of non-boundary state is the joint probability of the independent events that $j_t - i_t$ instructions from each thread t are renamed to the RRF. For all states where $\sum_t j_t < N_Q$ and $\forall t, t \in [0,T] : j_t < N_{cap}$ transition probabilities can be calculated as

$$p\left(s_i \xrightarrow{A} s_j\right) = \prod_{t=1}^{T} p_{i_t,j_t}^{[t]} \tag{44}$$

where $p_{i_t,j_t}^{[t]}$ is the probability in the arrival matrix in the single-threaded RRF model for thread t. To get the probability of all the remaining states, there are two cases to consider. The first case is when the RRF is not full even if all the threads occupy as much as the cap value $N_{cap} \cdot T < N_Q$. To derive the arrival matrix for this case is quite simple. The transition probability is the joint probability of the independent events that $j_t - i_t$ instructions from each thread t are renamed to the RRF. Transition probabilities can be calculated as

$$p\left(s_i \xrightarrow{A} s_j\right) = \prod_{t=1}^{T} p_{i_t,j_t}^{[t]} \tag{45}$$

where $p_{i_t,j_t}^{[t]}$ is the probability in the arrival matrix in the single-threaded RRF model for thread t. Due to the fact that the RRF is not full, the entire matrix can be calculated using the above formula.
The second case features states where the RRF is full $N_{cap} \cdot T \geq N_Q$. This case is more complex to derive and requires a process called Absorption. Absorption is when the transition to a state that is possible when there is no capping but not possible when capping is applied due to over occupation gets absorbed by transitions to states lower occupation. For example, consider a model with RRF of size 5 ($N_Q = 5$), 2 threads total ($T = 2$), cap value of 3 ($N_{cap} = 3$), and arrival rate λ_1 and λ_2 when capping is not applied and when capping is applied. Figure 9 shows all the states available to state $\langle 0,0\rangle$ when capping is implemented with green states as reachable states and red states as non-reachable states. When capping is not applied, state $\langle 1,4\rangle$ is possible because there are 5 instructions in the RRF total. However, when capping is applied, this state is no longer possible because the second thread occupies more than 3 registers. The transition probability to this state needs to be absorbed by the two immediate transitions to states with lower occupancy, namely $\langle 0,4\rangle$ and $\langle 1,3\rangle$. The absorption starts with the transition to impossible states with the highest occupancy down to first possible state. In case of this example, the absorption stops at state $\langle 1,3\rangle$, however absorption needs to be applied to transitions to state $\langle 0,4\rangle$ again until it reaches a possible state. $\langle 1,4\rangle$ is absorbed by multiplying its transition probability by the probability that the

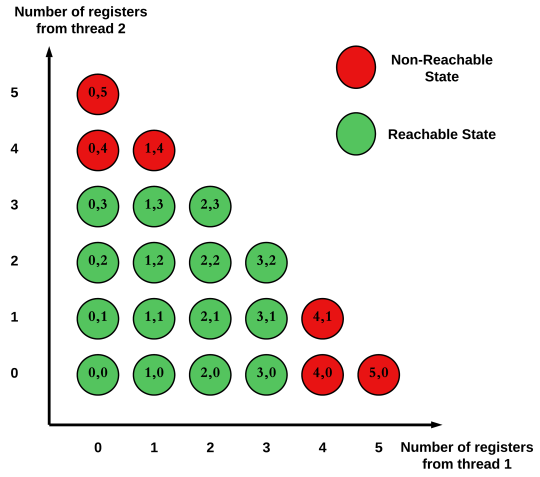


Fig. 9: **Reachable and Non-Reachable States when Cap is 5**

next instruction is coming from thread I_t (p_{I_t}) add add this product to the state where there is one less instruction from thread I_t. The probability that the next instruction is coming from thread I_t is presented above

$$p_{I_t} = \frac{\lambda_t}{\sum_{k=1}^{T} \lambda_k} \tag{46}$$

Assuming an initial state $\langle 0,0 \rangle$, the absorption of transitioning to $\langle 1,4 \rangle$ is as follows:

$$\begin{aligned} p\left(\langle 0,0\rangle \xrightarrow{A} \langle 0,4\rangle\right) =& p^*\left(\langle 0,0\rangle \xrightarrow{A} \langle 0,4\rangle\right) + \\ & p\left(\langle 0,0\rangle \xrightarrow{A} \langle 1,4\rangle\right) \cdot \frac{\lambda_1}{\lambda_1+\lambda_2} \end{aligned} \tag{47}$$

$$\begin{aligned} p\left(\langle 0,0\rangle \xrightarrow{A} \langle 1,3\rangle\right) =& p^*\left(\langle 0,0\rangle \xrightarrow{A} \langle 1,3\rangle\right) + \\ & p\left(\langle 0,0\rangle \xrightarrow{A} \langle 1,4\rangle\right) \cdot \frac{\lambda_2}{\lambda_1+\lambda_2} \end{aligned} \tag{48}$$

where $p^*(s_i \xrightarrow{A} s_i)$ is the transition probability before absorption during that specific absorption process and $p(s_i \xrightarrow{A} s_i)$ is the new transition probability after absorption. The absorption of the transition to state $\langle 0,4 \rangle$ is different because there is only one immediate state with lower occupancy, $\langle 0,3 \rangle$. For such states, the absorption is simply

$$\begin{aligned} p\left(\langle 0,0\rangle \xrightarrow{A} \langle 0,3\rangle\right) =& p^*\left(\langle 0,0\rangle \xrightarrow{A} \langle 0,3\rangle\right) + \\ & p\left(\langle 0,0\rangle \xrightarrow{A} \langle 0,4\rangle\right) \cdot \frac{\lambda_2}{\lambda_2} \end{aligned} \tag{49}$$

Figure 10 provides a visualization of the absorptions mentioned above.

The absorption equation can be generalized for absorption of an arbitrary transition $p\left(\langle i_1, i_2, \ldots, i_T\rangle \xrightarrow{A} \langle j_1, j_2, \ldots, j_T\rangle\right)$ where $j_x - 1 > 0,\ \forall x \in [0, T]$ as:

$$\begin{aligned} & p\left(\langle i_1, i_2, \ldots, i_T\rangle \xrightarrow{A} \langle j_1 - 1, j_2, \ldots, j_T\rangle\right) = \\ & p^*\left(\langle i_1, i_2, \ldots, i_T\rangle \xrightarrow{A} \langle j_1 - 1, j_2, \ldots, j_T\rangle\right) \\ & + p\left(\langle i_1, i_2, \ldots, i_T\rangle \xrightarrow{A} \langle j_1, j_2, \ldots, j_T\rangle\right) \cdot \frac{\lambda_1}{\sum_{k=1}^{T} \lambda_k} \end{aligned}$$

$$\begin{aligned} & p\left(\langle i_1, i_2, \ldots, i_T\rangle \xrightarrow{A} \langle j_1, j_2 - 1, \ldots, j_T\rangle\right) = \\ & p^*\left(\langle i_1, i_2, \ldots, i_T\rangle \xrightarrow{A} \langle j_1, j_2 - 1, \ldots, j_T\rangle\right) \\ & + p\left(\langle i_1, i_2, \ldots, i_T\rangle \xrightarrow{A} \langle j_1, j_2, \ldots, j_T\rangle\right) \cdot \frac{\lambda_2}{\sum_{k=1}^{T} \lambda_k} \end{aligned}$$

$$\ldots$$

$$\begin{aligned} & p\left(\langle i_1, i_2, \ldots, i_T\rangle \xrightarrow{A} \langle j_1, j_2, \ldots, j_T - 1\rangle\right) = \\ & p^*\left(\langle i_1, i_2, \ldots, i_T\rangle \xrightarrow{A} \langle j_1, j_2, \ldots, j_T - 1\rangle\right) \\ & + p\left(\langle i_1, i_2, \ldots, i_T\rangle \xrightarrow{A} \langle j_1, j_2, \ldots, j_T\rangle\right) \cdot \frac{\lambda_T}{\sum_{k=1}^{T} \lambda_k} \end{aligned}$$

Using this absorption process it is possible to derive the arrival matrix for the multi-threaded case with capping by following these steps:

(1) Generate arrival matrix for multi thread without capping.
(2) Gather the non-reachable boundary states (states that occupy more than cap value).
(3) Apply the absorption process on boundary states that are non-reachable due to capping recursively until possible states are reached

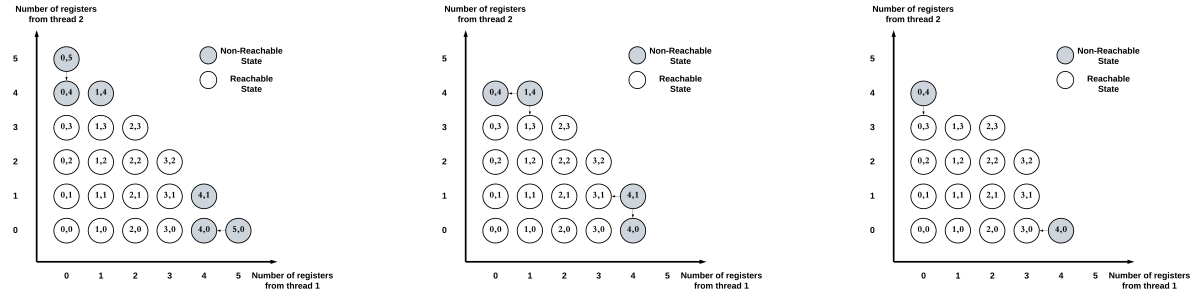
When all the non-reachable states have been absorbed by reachable states, the states that the algorithm stops at are the new boundary states.

6.5 Complete Multi-Threaded Model

After modifications to apply capping, it is possible to combine the matrices to complete the model using $P = C \times A$. P represents the clock-to-clock transition of the RRF.

Using P, it is possble to use the steady state to examine the IPC as well as the occupancy of each thread. The consumption rate (ρ) can be multiplied with how likely a state is to release a register to approximate the IPC. To calculate the rate of release of each thread, by multiplying each element in each row of the matrix where $j_t > i_t$ for that thread by the number of registers that state releases ($j_t - i_t$) for that thread and add them all up. Then multiply by how many percentage of time the steady state spends in that state to get an approximation of the IPC. Observe this IPC as the cap value is varied to study the effect capping has on performance.

For the sake of simplicity and computation cost, this paper studies a 2-threaded model with $N_Q = 32$ and arrival rates of $\lambda_1 = \lambda_2 = 4$. N_Q is chosen to be 32 because it is enough registers for 2 threads to run properly but not too many where they are redundant. $\lambda_1 = \lambda_2 = 4$ is a high arrival rate compared to the consumption rate ($0.01 \leq \rho \leq 0.7$) so that the RRF is more likely to be fully utilized. The sections analyzes three different combinations of



(a) First Round (b) Second Round (c) Third Round

Fig. 10: Absorption Steps for Model $N_Q = 2$, $T = 2$, and $N_{cap} = 3$

consumption rates, $(\rho_1 = 0.3, \rho_2 = 0.5), (\rho_1 = 0.2, \rho_2 = 0.7)$, and $(\rho_1 = 0.01, \rho_2 = 0.5)$ and vary the cap value from 0 to 32 to analyze the changes in occupation and effect of capping. Figure 11 shows the occupation of the thread with lower consumption rate across different cap values. As the difference in the consumption rates increases, the occupation of the slow thread dominates. In the case of $(\rho_1 = 0.01, \rho_2 = 0.5)$ the slow thread occupies as much as 28 out of 32 registers.

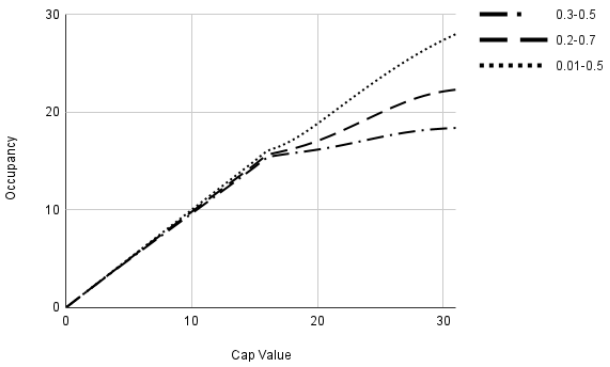


Fig. 11: **Cap Value vs. Occupancy of Slow Thread in Model when** $N_Q = 32$

This trend demonstrates the starvation effect when a slow thread holds on to too many resources. As the cap value increases, the slow thread continues to rename while still holding on to registers, causing a high occupancy. When a slow thread consumes most of the resources, the fast thread does not have enough space to execute, resulting in a negative effect on the IPC.

Compared with the simulation results of M-Sim, it can be observe that there is a similar behavior in the occupancy of slow threads, except for the case $(\rho_1 = 0.2, \rho_2 = 0.7)$. Figure 12 shows the occupation of the thread with lower consumption rate across different cap values of Mix 3, Mix 5, and Mix 1. Mix 3 consists of two threads with the same ILP classification, representing the case where consumption rates are similar, comparable to case $(\rho_1 = 0.3, \rho_2 = 0.5)$. Similarly, Mix 5 is comparable to the case $(\rho_1 = 0.2, \rho_2 = 0.7)$ where one thread is in the medium ILP classification and the other is in the high ILP classification. Finally, Mix 1 is comparable to case $(\rho_1 = 0.01, \rho_2 = 0.5)$ where the difference in consumption rates is the greatest.

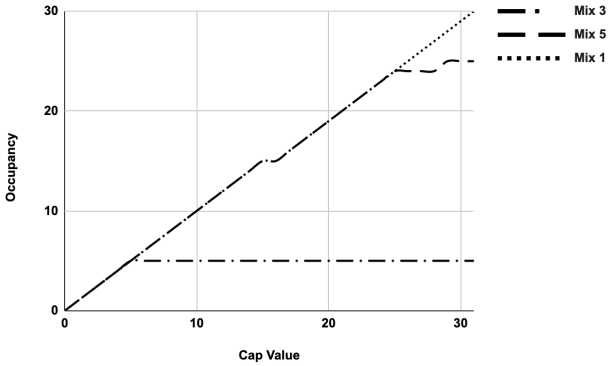


Fig. 12: **Cap Value vs. Occupancy of Slow Thread in Simulation when** $N_Q = 32$

The theoretical model provides a similar trend to the actual simulation in two of the cases that are examined, $(\rho_1 = 0.2, \rho_2 = 0.7)$, and $(\rho_1 = 0.01, \rho_2 = 0.5)$, with a few differences. When the cap value is less than half of the rename register file, the model and simulation show that the occupancy increases linearly with the cap value. However, the model predicts that when the cap value is greater than half of the register file, the occupancy does not increase at the same rate. In actual simulation, the occupancy continues to increase at the same rate until the very last few cap values. This shows that while the model shows the starvation effect of the slower thread, the model fails to predict the magnitude of starvation. For example, in the case $(\rho_1 = 0.01, \rho_2 = 0.5)$ the slower thread only holds 28 registers on average at the highest cap value, while Mix 1 holds 31 registers on average.

In the case $(\rho_1 = 0.2, \rho_2 = 0.7)$, the model provides a different trend from Mix 3. There are other factors in the pipeline besides the rename stage that lead to both threads in Mix 3 to require only a maximum of 6 registers on average despite the fact that there are more registers to use. The model fails to reflect cases like this because of its nature of modeling only the rename stage. With

an arrival rate higher than the consumption rate, the threads will always try to use all the registers available.

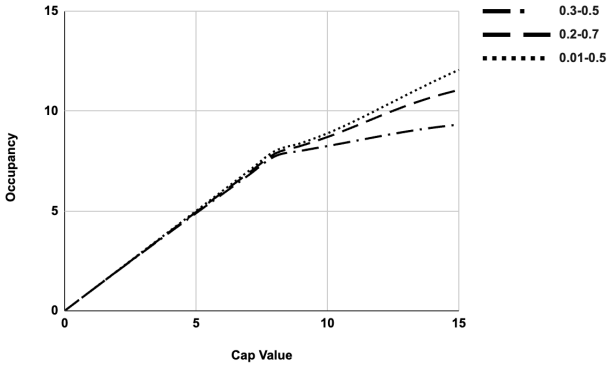


Fig. 13: **Cap Value vs. Occupancy of Slow Thread in Model when** $N_Q = 16$

When keeping all parameters the same and decrease $N_Q = 16$ to show more competition, the trend of the model and the simulation has some differences to the configuration where $N_Q = 32$. Looking at figures 13 and 14, the model predicts that the trend for case $(\rho_1 = 0.2, \rho_2 = 0.7)$ is closer to the trend of Mix 5. The model still fails to predict the magnitude of starvation in the case $(\rho_1 = 0.01, \rho_2 = 0.5)$. It appears that there is a limit of how many registers a thread can take despite being allowed to use more. In the simulation, a thread can take up to almost if not all registers on average if there is not a cap to limit it. This is a limitation of the model that can be addressed in future revisions. Despite lowering N_Q to foster more competition, Mix 3 still only uses as many as six registers per thread, further confirming that occupancy is affected by other parts of the pipeline. When each thread only uses up to 6 registers and there are 16 registers available, there is no competition to be had between each thread.

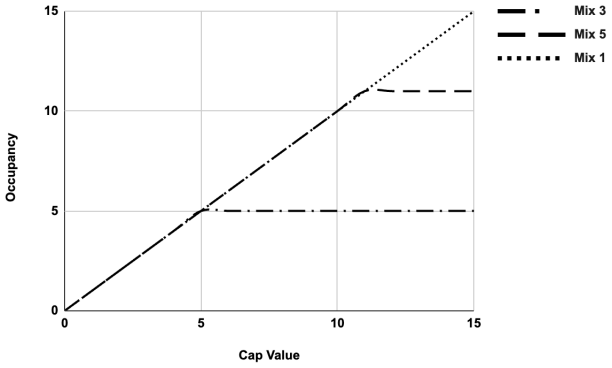


Fig. 14: **Cap Value vs. Occupancy of Slow Thread in Simulation when** $N_Q = 16$

We again reduce $N_Q = 10$ to foster competition even for Mix 3, where the maximum occupancy of both threads is 13 in other configurations, to see if the model can mimic the simulation's behavior. However, figures 16 and 15 show that the simulation still does not show competition between threads in Mix 3. Both threads in Mix 3 take up less registers than they do when allowed more registers like in other comparison configurations.

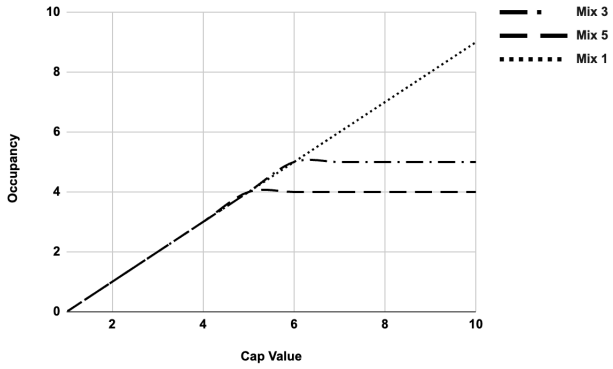


Fig. 15: **Cap Value vs. Occupancy of Slow Thread in Simulation when** $N_Q = 10$

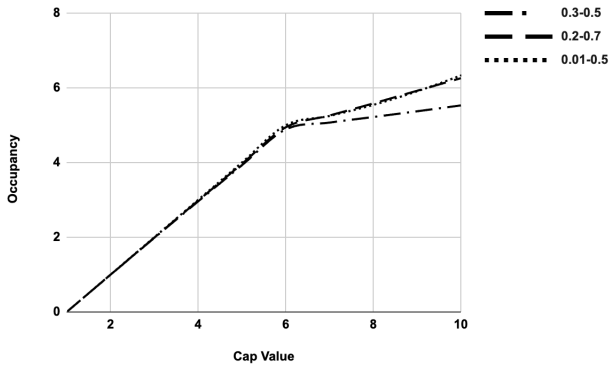


Fig. 16: **Cap Value vs. Occupancy of Slow Thread in Model when** $N_Q = 16$

When there is a limit on the number of registers that each thread can use with capping, the IPC improves as shown in Figure 17.The total IPC starts low because with a low cap value, neither thread can perform well. As the cap value increases, the IPC increases as there are more registers to use. However, when the cap value is too high, the slow thread dominates in utilization and causes the total IPC to decrease. This trend matches the trend in the simulation shown in Figure 18. The case $(\rho_1 = 0.01, \rho_2 = 0.5)$ and Mix 5 show the highest competition between threads; therefore, the total IPC is higher when the cap value is around half the size of N_Q and dramatically decreases as the cap value approaches N_Q. The case $(\rho_1 = 0.3, \rho_2 = 0.5)$ and Mix 5 have threads that are similar in performance, therefore, the total IPC increases with the cap value until it reaches a peak and then decreases slightly at higher cap values. The model predicts that case $(\rho_1 = 0.2, \rho_2 = 0.7)$ has

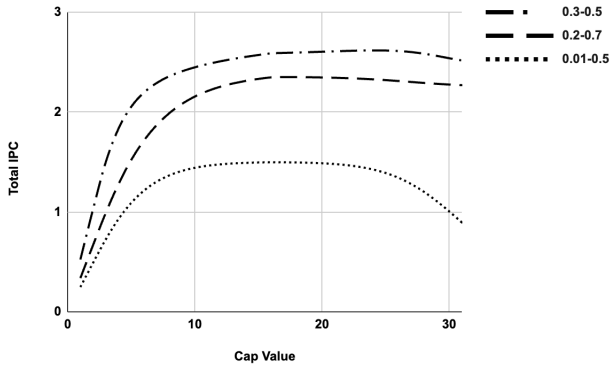


Fig. 17: **Cap Value vs. Total IPC in Model when** $N_Q = 32$

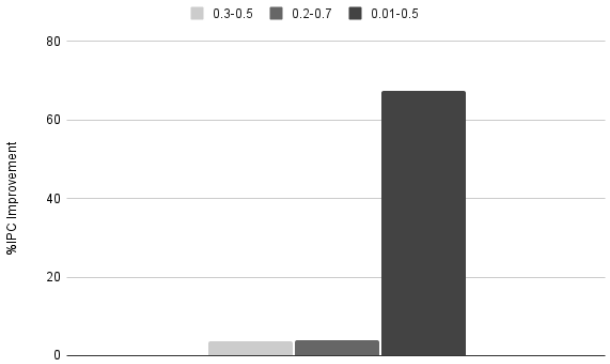


Fig. 19: **%IPC Improvement**

an IPC trend similar to case $(\rho_1 = 0.3, \rho_2 = 0.5)$ due to the small difference in performance between each thread, however, in simulation, because the total register usage at maximum is lower than the registers available, the IPC hits a plateau once the cap value reaches the number of registers each thread uses at its most.

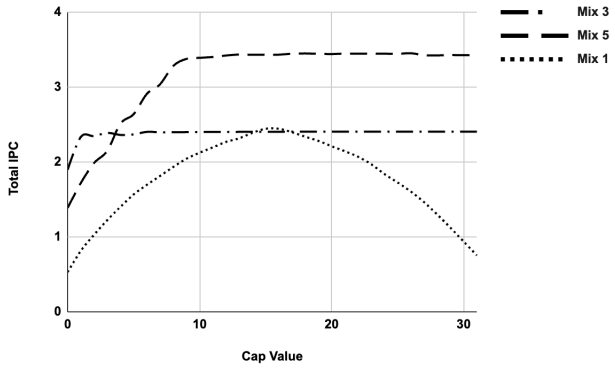


Fig. 18: **Cap Value vs. Total IPC in Simulation when** $N_Q = 32$

Figure 19 shows that at the cap value where the total IPC peaks, the first two cases $(\rho_1 = 0.3, \rho_2 = 0.5)$ and $(\rho_1 = 0.2, \rho_2 = 0.7)$ do not show much improvement over the absence of capping. They are not very far in terms of consumption rate relative to each other. However, in cases where the difference in consumption rate is considerable, for example, $(\rho_1 = 0.01, \rho_2 = 0.5)$, the improvement is substantial.
Observations of the model confirm that slow threads hurt overall performance by taking up too many registers, starving the fast threads. The higher the difference between the consumption rates, the more performance can be gained by employing capping. The following chapters derive a capping algorithm that adjusts the cap value based on how the threads perform compared to each other.

7. CONCLUSION

This research presents a theoretical model of the Rename Register File using queuing theory. The model features the utilization of the RRF in a multi-thread environment when capping is applied. Analysis of the model provides a better understanding of the phenomena that occur in the RRF. Although not complex enough to simulate actual program behavior, it mimics the behavior of an RRF model under different consumption rate combinations and capping values. As there are other stages that the model cannot take into account.
To better model the behavior of an actual RRF, the model needs to take into consideration of other inputs such as cache miss, instruction type, ALU type, and other parameters of a CPU. The stock Markov Chain is limited by one type of incoming rate and one type of out-going rate. In order to simulate the RRF better, there needs to be further modifications to the Markov Chain.
Further more, the formulae in this paper requires intensive calculations when there are many registers and threads. Optimization is needed to model larger CPU's with more threads and registers.

8. REFERENCES

[1] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, T. Nishizawa, An elementary processor architecture with simultaneous instruction issuing from multiple threads, in: Proceedings of the 19th annual international symposium on Computer architecture, 1992, pp. 136–145.

[2] D. M. Tullsen, S. J. Eggers, H. M. Levy, Simultaneous multithreading: Maximizing on-chip parallelism, in: Proceedings of the 22nd annual international symposium on Computer architecture, 1995, pp. 392–403.

[3] M. Navarro, J. Feliu, S. Petit, M. E. Gómez, J. Sahuquillo, Synpa: Smt performance analysis and allocation of threads to cores in arm processors (2023). `arXiv:2310.12786`.
URL `https://arxiv.org/abs/2310.12786`

[4] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. 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, 1996, pp. 191–202.

[5] F. J. Cazorla, A. Ramirez, M. Valero, E. Fernández, Dynamically controlled resource allocation in smt processors, in: 37th International Symposium on Microarchitecture (MICRO-37'04), IEEE, 2004, pp. 171–182.

[6] S. Carroll, W.-M. Lin, Applied on-chip machine learning for dynamic resource control in multithreaded processors, Parallel Processing Letters 29 (03) (2019) 1950013. doi:10.1142/S0129626419500130.

[7] Y. Zhang, W.-M. 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, pp. 5–7.

[8] D. Kanter, Intel's sandy bridge microarchitecture, https://www.realworldtech.com/sandy-bridge/ (Sep 2010).

[9] D. Kanter, Amd's bulldozer microarchitecture, https://www.realworldtech.com/bulldozer (Aug 2010).

[10] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, D. M. Tullsen, Software-directed register deallocation for simultaneous multithreaded processors, IEEE Transactions on Parallel and Distributed Systems 10 (9) (1999) 922–933.

[11] T. Monreal, A. González, M. Valero, J. González, V. Viñals, Dynamic register renaming through virtual-physical registers, Journal of Instruction Level Parallelism 2 (2000) 4–16.

[12] Y. Zhang, W.-M. 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), The Steering Committee of The World Congress in Computer Science and Computer . . . , 2015, p. 27.

[13] H. Güngörer, G. Küçük, Dynamic capping of physical register files in simultaneous multi-threading processors for performance, in: Computer and Information Sciences, Springer International Publishing, 2018, pp. 41–48.

[14] H. Zhan, V. S. Sheng, W.-M. Lin, Reinforcement learning-based register renaming policy for simultaneous multithreading cpus, Expert Systems with Applications 186 (2021) 115717. doi:https://doi.org/10.1016/j.eswa.2021.115717.
URL https://www.sciencedirect.com/science/article/pii/S095741742101099X

[15] A. Papoulis, S. Pillai, Probability, Random Variables and Stochastic Processes, McGraw-Hill series in electrical engineering: Communications and signal processing, Tata McGraw-Hill, 2002.
URL https://books.google.com/books?id=g6eUoWOlcQMC

[16] K. Trivedi, Probability and Statistics with Reliability, Queuing and Computer Science Applications, Wiley, 2016.

[17] S. Carroll, W.-M. Lin, A queuing model for cpu functional unit and issue queue configuration, ArXiv abs/1807.08586 (2018).
URL https://api.semanticscholar.org/CorpusID:49904986

[18] J. Sharkey, D. Ponomarev, K. Ghose, M-sim: a flexible, multithreaded architectural simulation environment, Techenical report and Department of Computer Science and State University of New York at Binghamton (2005).

[19] Standard Performance Evaluation Corporation, Standard Performance Evaluation Corporation (SPEC), https://www.spec.org, accessed: May 29, 2025 (2025).

[20] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, ACM SIGPLAN Notices 37 (10) (2002) 45–57.