PERFORMANCE ENHANCEMENT WITH SPECULATIVE-TRACE CAPPING AT DIFFERENT PIPELINE STAGES IN SIMULTANEOUS MULTI-THREADING PROCESSORS

Wenjun Wang and Wei-Ming Lin

Department of Electrical and Computer Engineering, The University of Texas at San Antonio, San Antonio, TX 78249-0669, USA

ABSTRACT

Simultaneous Multi-Threading (SMT) processors improve system performance by allowing concurrent execution of multiple independent threads with sharing key datapath components and better utilization of resources. Speculative execution allows modern processors to fetch continuously and reduce the delays of control instructions. However, a significant amount of resources is usually wasted due to miss-speculation, which could have been used by other valid instructions, and such a waste is even more pronounced in an SMT system. In order to minimize the waste of resources, a speculative trace capping technique [1] was proposed to limit the number of speculative instructions in the system. In this paper, a thorough analysis is given to investigate the trade-offs among applying this capping mechanism at different pipeline stages so as to maximize its benefits. Our simulations show that the best choice can improve overall system throughput by a very significant margin (up to 46%) without sacrificing execution fairness among the threads.

KEYWORDS

Simultaneous Multi-Threading, Superscalar, Speculative Execution

1. INTRODUCTION

Simultaneous Multi-Threading (SMT) improves overall system performance by allowing concurrent execution of multiple independent threads for better utilization of the resources provided by modern processor architecture. Essentially, SMT improves the overall performance by exploiting Thread-Level Parallelism (TLP) among threads to overcome the limited availability of Instruction-Level Parallelism (ILP) present in a single thread which inherently limits the potential of a superscalar system [2][3]. There have been various designs targeted on the improvement of overall system performance in an SMT system. Several techniques focus on the fairness of the resources allocation and modify the fetch policy. Among many others, ICOUNT [4] gives priority to threads with the fewest instructions in the system prior to issuing. DCRA [5] monitors the usage of resources by each thread and gives a higher share of the available resources to memory-intensive threads. Several others address the resource allocation at the rename stage [6][7], dispatch stage [8][7] and commit stage [9][10][11].

All modern processors adopt speculative execution to reduce the delays from control instructions. With basic speculative processing, instructions are allowed to speculatively fetch and execute along the predicted path without waiting for the actual outcome of the control instruction. If a miss-prediction occurs, instructions along the speculative trace are then flushed out of the system

DOI:10.5121/caij.2016.3401

and all the resources thus occupied are considered wasted. Traditional branch prediction techniques [12][13][14] can partially alleviate the flush-out problem. Note that this problem is usually exasperated in an SMT system – instructions from different threads intermix through shared components in the pipeline, which potentially extends the length of speculative traces. The amount of miss-speculated instructions flushed out can easily account for a significant percentage of overall instructions fetched. Coupled with the adverse effect that these resources could have been utilized by valid instructions otherwise, net performance gain from speculative execution can be offset to an undesirable degree. Note that this offset does not exist in a non-SMT system.

A speculative trace-capping control mechanism was first proposed in [1] applied at the dispatch stage to limit the maximum number of instructions in a speculative trace allowed beyond this stage. However, the stage where such a capping mechanism is applied can significantly affect the performance outcome. An obvious trade-off exists in between capping at an earlier stage to benefit more from a shorter speculative trace and capping at a later stage to diminish adverse effect from delay in replenishing the pipeline after a correct speculation is resolved. In this paper, a comprehensive study is performed to investigate this trade-off so as to identify the optimal pipeline stage to apply such a control mechanism to maximize its benefits. Our simulations show that when this mechanism is applied at the right stage, not only the overall system performance is improved by up to 46% but the execution fairness is also enhanced by 29%.

The rest of the paper is organized as follows. Background information of SMT systems and speculative execution is first given in Section 2, followed by the descriptions of the adopted simulation environment (simulator and workload) and performance metrics in Section 3. A detailed analysis to illustrate our motivation and conjectures are provided in Section 4. Section 5 explains the proposed method, followed by the simulation results in Section 6. Concluding remarks are then given in Section 7.

2. BACKGROUND

2.1. SMT Processors

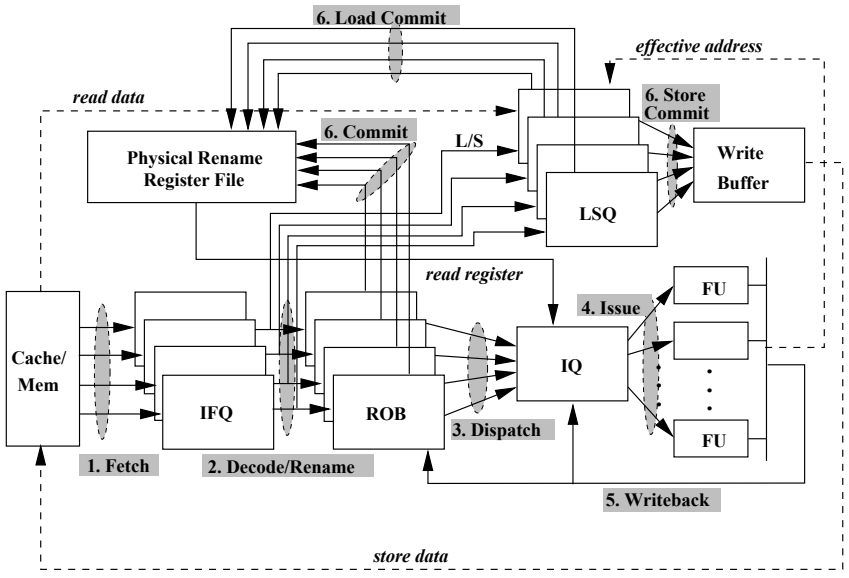

Figure 1. Pipeline Stages in a 4-Threaded SMT System

An SMT processor is built from a typical out-of-order superscalar processor and the basic pipeline stages of a 4-threaded SMT system is shown in Figure 1. Instructions from a thread are *fetched* frommemory (and cache) and put into their respective private Instruction Fetch Queue (IFQ). After the stages of *decode* and register-*rename*, the instructions are allocated to their respective Re-Order Buffer (ROB) and through the *dispatch* stage into the shared Issue Queue (IQ). Load/Store instructions have their operations sent into individual Load Store Queue (LSQ) with their address calculation operationsalso sent into IQ. When instruction-issuing conditions (all operands ready and the required functional unit available) are met, operations are then *issued* to corresponding functional units and have their results *writeback* to their target locations or forwarded to where the results are needed in IQ or LSQ. Load/Store instructions, once their addresses are calculated, will initiate their memory operation. Finally, all instructions are *committed* from ROB (synchronized with Load/Store instructions in LSQ) in order. The most common characteristic of SMT processors is the sharing of key datapath components among multiple independent threads. The shared resources of an SMT system normally include the physical register file (for renaming), various inter-stage bandwidths, buffers (e.g. Issue Queue (IQ) and Write Buffer), and functional units. Due to the sharing of resources, the amount of hardware required in an SMT system can be significantly less than that from employing multiple copies of superscalar processors while achieving a comparable throughput.

2.2. Speculative Execution

Modern processors with speculative mechanism use branch predictors at the *fetch* stage to allow continuous fetching even a control instruction (a branch or a jump with an execution-time dependent target) is encountered. Fetch direction after these control instructions depends on the outcome of the branch predictor. The instructions fetched following these control instructions are called "speculative instructions" which are allowed to proceed through the pipeline even before the branch condition and/or target address is resolved. Such a speculation is not resolved until the control instruction reaches the *writeback* stage. If the speculation is found to be correct, all speculative instructions following it will be processed as normal instructions until another speculation is encountered. In this case, the processor can significantly reduce the delay from an unresolved control instruction compared to a system, which simply suspends the fetching of subsequent instruction until it is resolved. Conversely with an incorrect speculation, all the subsequent speculative instructions, the so-called speculative trace, have to be flushed out of the system, which requires a more complex control logic and leads to resource wastage. Note that such a flush out does not really pose a significant threat to performance for a superscalar system due to the single-thread platform. That is, the resources thus wasted would not have been used by other "valid" instructions anyway. However, in an SMT system, instructions from different threads intermix in time flowing through the shared components in the pipeline, potentially stretching the length of a speculative trace. Threads also potentially block one another, and thus further lengthen the speculative trace and prolong the lifetime of speculative instructions in the system. As aforementioned, in an SMT system, some critical resources (such as register file, IQ, etc.) are shared among concurrent executing threads and instructions from different threads may run into severe competition for available slots due to their limited sizes. These shared resources may be occupied by miss-speculated instructions for a long period of time, which could have been utilized by other valid instructions. Our analysis presented in a later section shows that the percentage of resources thus wasted can be undesirably high which could be especially harmful to the overall system performance.

3. SIMULATION ENVIRONMENT

3.1. Simulator

We use the M-sim [15], a multi-threaded microarchitectural simulation environment to estimate the performance of an SMT system. M-sim explicitly models 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, register renaming,and the associated rename table. M-sim supports both single threaded execution and the simultaneous execution of multiple threads. In SMT mode, the physical register file, IQ, functional units and write buffer are shared among threads. Table 1 gives the detailed configuration of the simulated processor.

Table 1. Configuration of 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 IntMult (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. Workload

Simulations on simultaneous multi-threading use the workload of mixed SPEC CPU2006 benchmark suite [16] with mixtures of various levels of ILP. ILP classification of each mix is obtained by initializing it in accordance with the procedure mentioned in Simpoints tool [17] and simulated individually in a simplescalar environment. Three types of ILP classification are identified, low ILP (memory bound), medium ILP, high ILP (execution bound). Table 2 and

Table 3 give the selected 4-threaded and 8-threaded workload for our simulation with various mixtures of ILP classification types respectively (each number in the table denotes the number of programs with the corresponding classification in the respective mix).

Table 2. Simulated SPEC CPU2006 4-Threaded Workload

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. Simulated SPEC CPU2006 8-Threaded Workload

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

3.3. Metrics

For a multi-threaded workload, total combined IPC is a typical indicator used to measure the overall performance, which is definedas the sum of each thread's IPC:

$$Overall\_IPC = \sum_i^n IPC_i \quad (1)$$

where n denotes the number of threads per mix in the system. However, in order to preclude starvation effect among threads, the so-called Harmonic IPC is also adopted, which reflects the degree of execution fairness among the threads, namely,

$$Harmonic\_IPC = {}^{n}/_{\sum_i^n IPC_i} \quad (2)$$

In this paper, these two indicators are used to compare the proposed algorithm to the baseline (default) system. The following metric indicates the improvement percentage averaged over the selected mixes, which is applied to both $Overall\_IPC$ and $Harmonic\_IPC$, namely,

$$IMP\_\% = (\sum_j^m \frac{IPC_j^{new} - IPC_j^{baseline}}{IPC_j^{baseline}} \times 100\%)/m \quad (3)$$

where m denotes the number of mixes of the workload in our simulation. Note that the improvement percentage represents the average improvement percentage over all mixes, instead of the improvement percentage of average IPC, which may be skewed by large variation among IPC values of different mixes.

4. MOTIVATION

The analysis performed in this paper is based on the conjecture that the benefit brought by speculative execution employed in an SMT system may be out-weighed by the detrimental effect caused by the prolonged wasted resources allocated to the eventual miss-speculated instructions. This conjecture is again based on the two following assumptions:

(1) There is a significant amount of flush-outs in SMT systems
(2) Competition for the shared resources among the threads is high

If the above assumptions are true, flushing out miss-speculated instructions from these already congested shared resources means precious resources being wasted. Properly limiting the chances and amount that these resources are wasted can thus potentially tip the balance. This section is devoted to a comprehensive analysis to support this conjecture and these assumptions. The simulation results presented in this section are based on the system configuration as shown in Table 1 with 10 mixes of a 4-threaded workload as shown in Table 2.

4.1. Flush-out Rate and Throughput Analysis

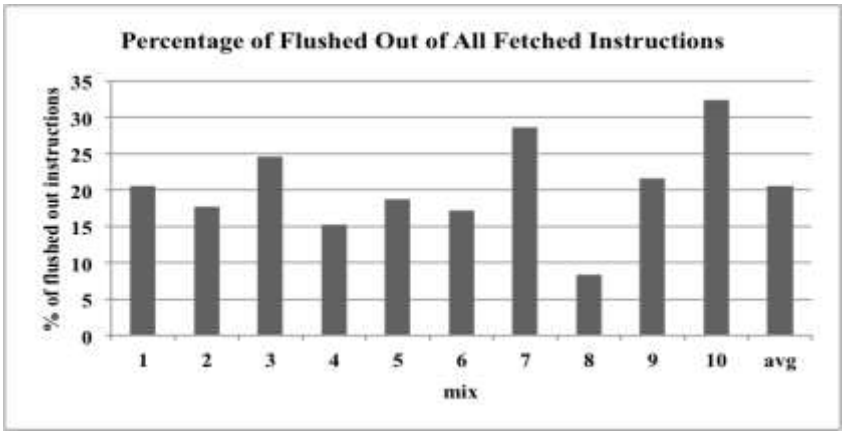

Figure 2. Percentage of Flush-Outs of All Fetched Instructions

Just like in a superscalar system, not all instructions fetched in an SMT system are eventually committed, and the percentage of flushed-out instructions is higher. Figure 2 shows the percentage of flush-outs of all fetched instructions. An average of 21% of fetched instructions are eventually flushed out as shown in Figure 2. For mix10, nearly 32% of fetched instructions are flushed out, which clearly indicates a very significant amount of wasted resources.

We further investigate the flush-out rate at each of the four pipeline stages: fetch, rename, dispatch and commit. Figure 3 shows the throughput of each stage with physical register file size (denoted as R_t) set at 256, that is, the average number of instructions per clock cycle flowing through that stage including the instructions eventually flushed out. As expected, throughput decreases as thepipeline continues to go downstream, due to some additional flushed-out instructions never reaching the next stage in the pipeline. Note that the three main causes for flush-outs are miss-speculations, interrupts and no-ops, with the first cause accounting for over 99% of all flush-outs [1], which is the sole focus of this study, with the other insignificant two ignored.

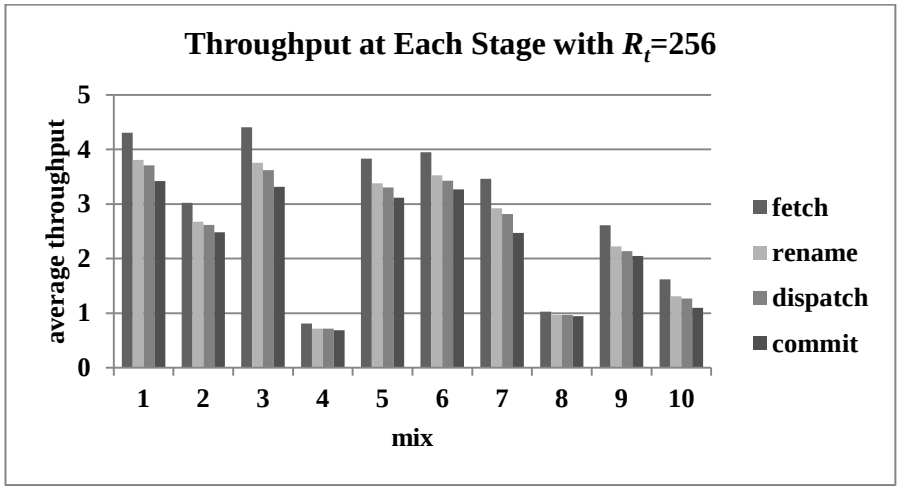

Figure 3. Throughput at Each SMT Pipeline Stage

4.2. Utilization Analysis of Shared Resources

Out of all the wasted resources from miss-speculation in an SMT system, not all are critical to the overall system performance. Some of the resources are private buffers (such as IFQ and ROB), which cannot be utilized by instructions from other threads, and thus congestion of these affects the overall perform to a much lesser degree than that from the shared resources. In this section, utilization of several key shared resources is analyzed, including the shared bandwidths between adjacent pipeline stages, the physical register file (the earliest shared buffer in the pipeline) and IQ (the shared buffer required for a speculation to resolve as early as possible). Study of this is necessary to the affirmation of the two aforementioned assumptions.

(1) *Inter-stage Bandwidth:* In addition to the analysis result shown in Section 4.1. to illustrate the percentages of instructions flushed out, the following analysis shows how a critical shared resource, bandwidth in this case, is wasted due to the same reason. Figure 4 shows the bandwidth wastage between adjacent pipeline stages. Anaverage of 21% of bandwidth is wasted to fetch instructions into IFQ that are eventually flushed out; 9% of bandwidth is

wasted to rename into register files and 7% is wasted to dispatch into IQ. All the wastage could have been utilized by other non-flush-outs for higher overall system performance.

(2) *Physical Rename Register File:* Physical rename register file is the first shared buffer among threads from the start of pipeline. If there are no more physical registers for a thread to rename, none of the available buffers or bandwidth down-stream in the pipeline is going to matter. Thus utilization of this particular buffer is very critical to the overall system performance. Figure 5 displays the occupancy rate of the physical register file with its size (R_t) set at 160. Each point represents an average result fromthe 10 mixes of the 4-threaded workload. As the results show, in nearly 80% of simulation clock cycles, the physical register file is fully occupied, which clearly indicates the need for a larger register file. Instead of increasing the register file size, which contradicts the design philosophy of an SMT processor, a more efficient resource allocation scheme is needed. The next analysis is to investigate the wastage of this resource due to miss speculation. Exploiting such wastage is only meaningful when the register file is full. Figure 6 shows the percentage of miss-speculated instructions in the register file when the register file is full. As the results show, up to 9% of registers are occupied by miss-speculated instructions, which are eventually flushed out, with an average of 3.7% of wasted registers. Note that adverse effect on IPC from the wastage of these resources is not truly reflected by its percentage, but instead much worse in an SMT system, since such a wastage leads to potential blocking of other threads which could have contributed significantly to the overall IPC had the blocking not been there.

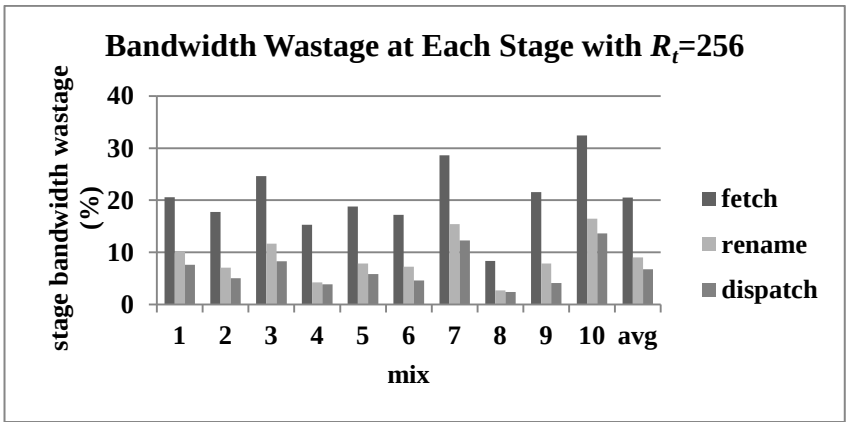

Figure 4. Bandwidth Wastage at Each Stage

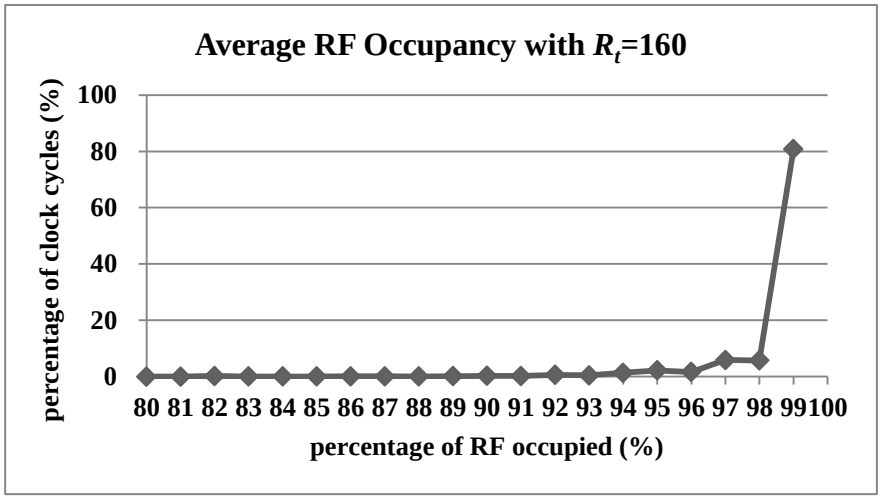

Figure 5. Average Register File Occupancy with R_t=160

(3) *Issue Queue*: A similar analysis is also performed on another critical shared resource, the issue queue. Due to its relatively limited number of entries (compared to the size of ROB at its up-stream stage), the competition among concurrent executing threads for the use ofIQ entries can be severe at times. Figure 7 displays the percentage of simulation clockcyclesin which a given number of IQ entries are occupied. IQ is fully occupied in about 44% of simulation time, mostly due to stagnant thread(s) which have their instructions stuck at IQ for a long period of time waiting for data dependency (such as from cache misses) or resource dependency (on the corresponding functional units). Such a high percentage of full occupancy clearly indicates a need for a better resource management. On the other hand, there is about 15% of time when the IQ is completely empty. Such situation of low IQ occupancy may be a direct outcome of instruction flushes out at IQ, which is partially explained by the next result.

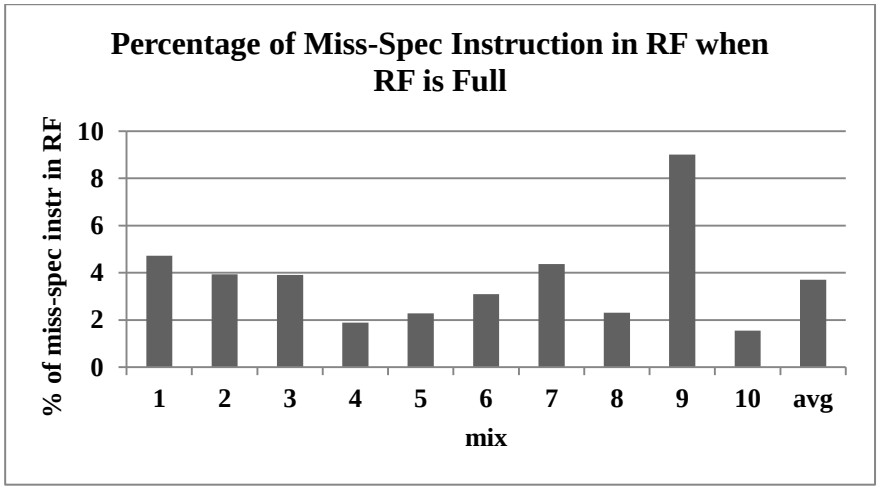

Figure 6. Percentage of Miss-Speculated Instruction in Register File (RF) when It Is Full

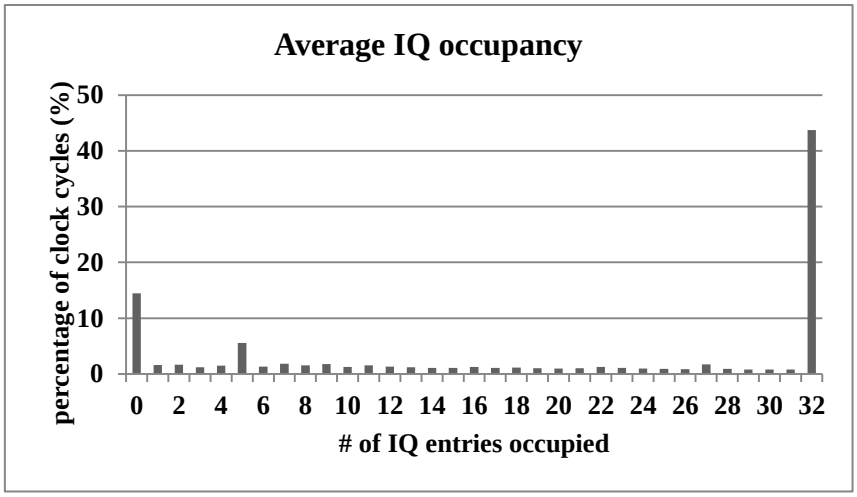

Figure 7. Average IQ Occupancy with IQ Size Equals to 32

Figure 8 shows the percentage of miss-speculated instructions in IQ when IQ is full. Up to 28% of IQ entries are occupied by miss-speculated instructions. An average of 16% of IQ entries are wasted due to miss-speculation. That is, on the average one out of roughly every six entries in IQ could have been used by anotherthread to boost the overall IPC. A combination of high occupancy and high percentage of flush-outs in IQ again ascertains the two assumptions for this study. If at all possible, one should try to prevent miss-speculated instructions from occupying IQ entries as much as possible.

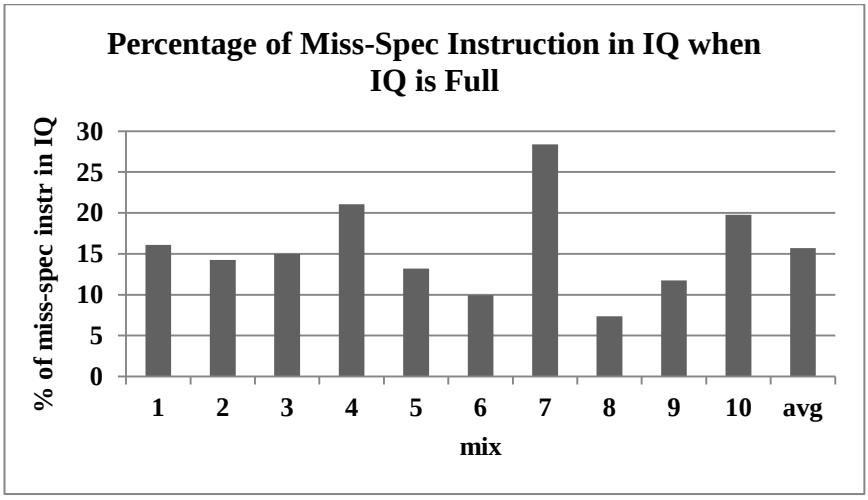

Figure 8. Percentage of Miss-Speculated Instructions in IQ when IQ is Full

5. CAPPING SPECULATIVE TRACES

From the analysis described in the Section 4, we find that a large amount of resources, such as shared bandwidths, the physical register file and IQ, are wasted due to the flush out of miss-speculated instructions. A technique was proposed in [1] to impose a real-time limit on the number of speculative instructions per speculative trace that are allowed to be dispatched into IQ. Although a very significant amount of performance improvement is given from employing this technique at the dispatch stage, one may argue that, from the high percentage of flush-outs and heavy resource competition shown in the previous section, the dispatch stage may not be the best

point of applying the capping mechanism; that is, the speculative trace may already be too long to be desirable in the system taking up too much shared resources and/or using up too much bandwidth in the upper stream of pipeline. In this paper, three different pipeline stages are identified as the targets for the initiation of the capping mechanism:

(1) *Fetch*
(2) *Rename*
(3) *Dispatch*

Essentially, at each of the selected stages, once a trace enters the speculative mode, the number of instructions of this thread that are allowed to proceed passing this stage is limited by a fixed cap value until the speculation is resolved: if correct, the capping is lifted, and if incorrect, all speculative instructions are then flushed out. The reason why only these three stages are targeted is due to that speculation is resolved at the *writeback*stage, and thus there is no reason to cap the trace at this stage or any of the later pipeline stages, the *commit* stage. In addition, since once an instruction enters the *issue* stage its occupancy time on the shared resources, the functional unit, is deterministic, there is very limited resource utilization optimization available at this stage to boost the performance unless more units are employed.

In the proposed capping technique, a counter is used to record the number of speculative instructions fetched/renamed/dispatched so far. Once this number reaches a threshold value (a pre-set cap value), fetching/renaming/dispatching from this thread will be stalled until the respective control instruction is resolved at the*writeback* stage. Figure 9 illustrates the difference among capping at the three respective stages. As shown in the figure, the pre-set cap value is denoted as C and the number of instructions already in the system before the capping point is denoted as X(X_d for capping-at-dispatch, X_r for capping-at-rename and X_f (= 0) for capping-at-fetch). The main objective of this technique is to reduce the wasted resources occupied by these $X + C$ instructions due to miss-speculation. Note that there exists a trade-off stemming from (1) the cap values employed as well as (2) the stage selected. Effectiveness of the capping mechanism is heavily influenced by the selections. Obviously, the larger the $X + C$ is, the more potential resource wastage is which leads to more blocking for other threads to proceed through the pipeline. On the other hand, the smaller the $X + C$ is, the size of the pipeline "bubble" (or gap) due to the capping is bigger. That is, after a correctly speculated trace is resolved and resumed, the subsequent instructions blocked due to the capping will take more time to replenish the pipeline gap again. Especially, in an SMT system, instructions from different threads intermix through shared components in the pipeline and compete for the precious resources,which potentially prolongs the latency to re-supply the subsequent instructions to a correctly speculated trace.

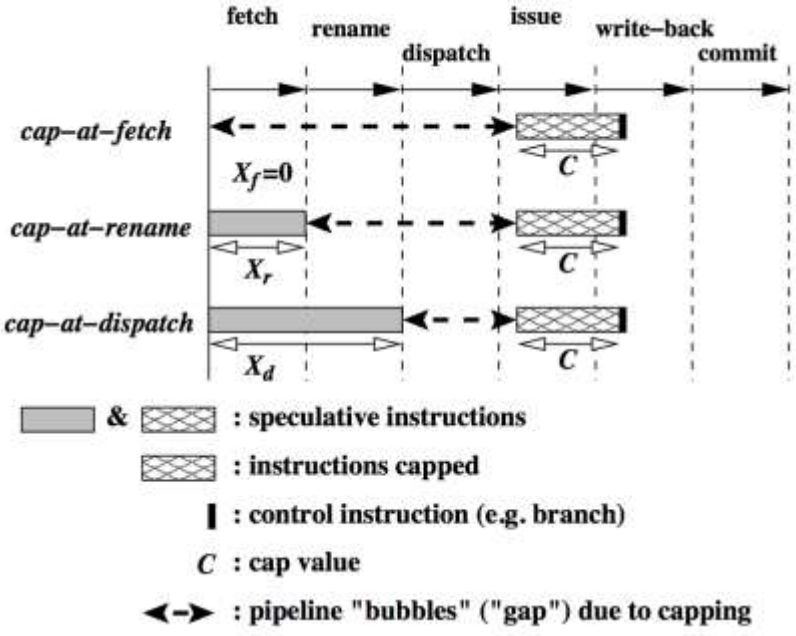

Figure 9. Capping Speculative Traces at the *Fetch*, *Rename*, and *Dispatch* Stages

(1) Cap value selection

A strict (small) cap value *C*, no matter which cap stage is chosen, leads to a smaller $X + C$, entails a performance compromise between losing the benefits from correctly speculated traces and gaining on saving the wasted resources occupied by miss-speculated traces. A larger cap value will instead diminish the benefits acquired from reducing resources wastage due to miss-speculation.

(2) Cap stage selection

With a fixed cap value *C*, capping at an earlier stage, e.g., the fetch stage, can potentially spare more shared resource than capping at a later stage, e.g., the dispatch stage, due to a smaller *X* value. However, such a gain could be easily offset by slowing down a correctly speculated trace as aforementioned due to a larger gap to fill.

6. SIMULATION RESULTS

The proposed technique is simulated with M-sim and compared with the default settings using 10 mixes and 6 mixes for 4-threaded workload and 8-threaded workload as shown in Table 2 and Table 3 respectively. The IPC improvement in this section is evaluated by Equation 3.

6.1. 4-threaded Workload

We first examine the effect on IPC when the cap value is varied with the capping technique applied at the three selected pipeline stages respectively. In order to clearly demonstrate the trade-off effects under various resource availability situations, we choose another parameter, the physical register file (R_t), since it has been shown to be the most dominant shared resources [6]. In the 4-threaded case, four different R_t values are used, ranging from 160 to 256, with 160 giving a very tightly competed resource unit and 256 for a rarely congested one. Cap values are selected to cover the respective relevant range for each R_t value as shown in Figures 10, 11, 12, and 13, respectively.

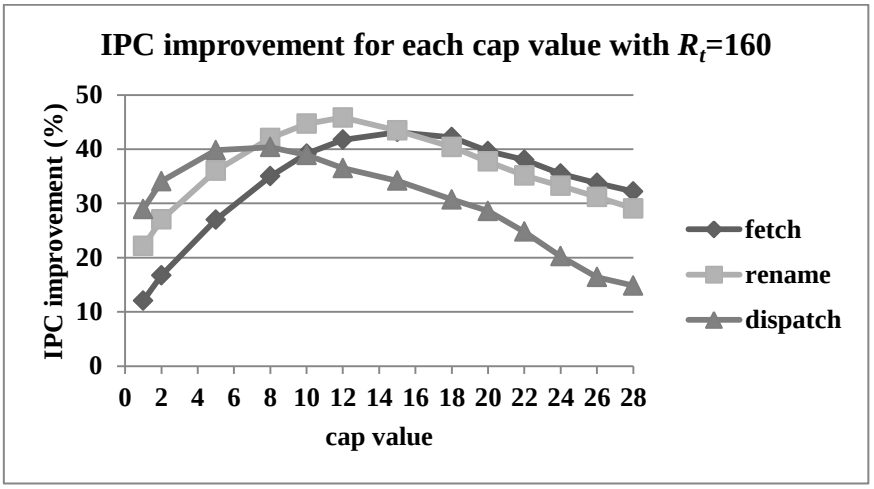

Figure 10. IPC Improvement for Each Cap Value with R_t=160 on 4-Threaded Workload

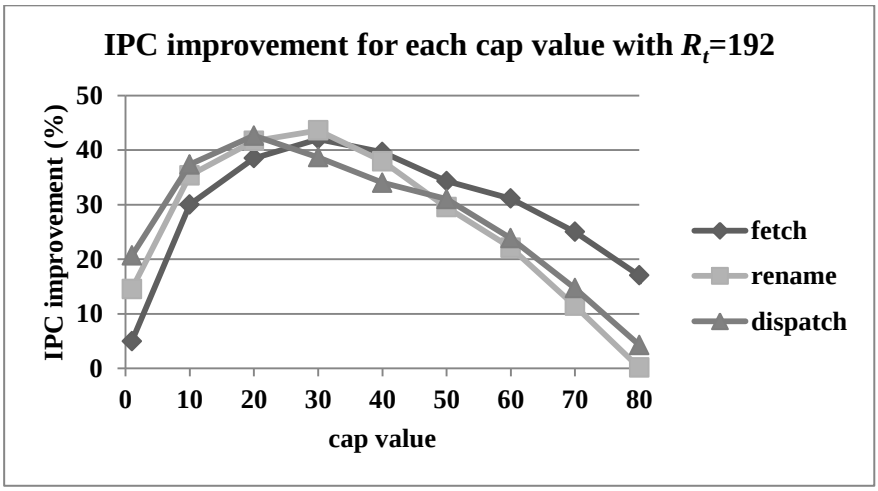

Figure 11. IPC Improvement for Each Cap Value with R_t=192 on 4-Threaded Workload

There are five important aspects of observation and conclusion one can make from these results:

(1) Trade-off due to the cap value (*C*):

The results clearly verify our hypothesis and the first trade-off - for each setting, there is an optimal *C* value which maximizes the overall gain. Setting the *C* value smaller, thus leaving a gap too large, may not be able to compensate the loss of the intended benefit from speculation with the saving of resource wastage. On the other hand, a cap value set larger allows the loss from resource wastage to gradually overtake the benefit from correct speculation. In a nutshell, increasing the cap value will continue to diminish the gain on reducing the wasted resources, while abating the potential loss from interrupting the correctly speculated traces. Further increasing the cap value will eventually completely diminish the benefit from resource saving, defaulting back to the original non-capping case.

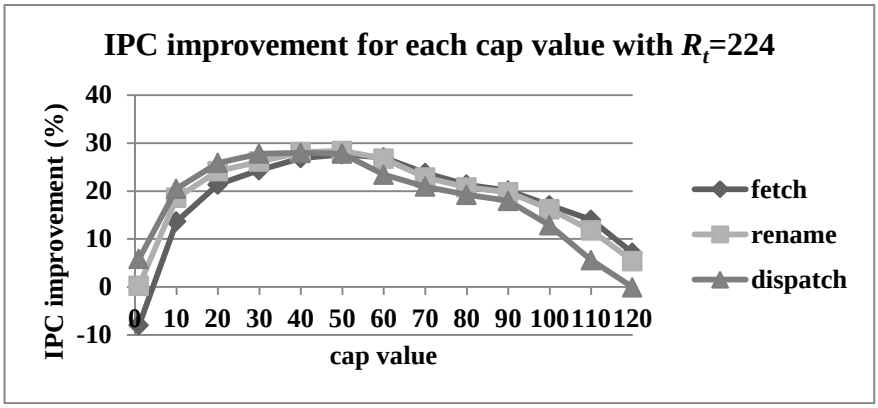

Figure 12. IPC Improvement for Each Cap Value with R_t=224 on 4-Threaded Workload

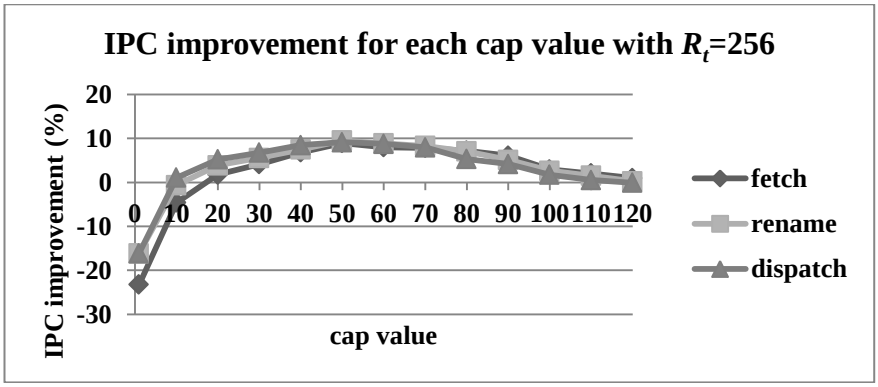

Figure 13. IPC Improvement for Each Cap Value with R_t=256 on 4-Threaded Workload

(2) Effectiveness of capping with respect to the setting of R_t:

The loss of intended speculation benefit from setting the cap value too small is more prominent when the shared resources ($R_t$in this case) are less stringent which can afford the resource wastage. For example, with the case when the capping is imposed at the *fetch* stage, setting the cap value at 1 still gives an average IPC improvement of 12% for R_t= 160, while leading to only 5% for R_t= 192, and begins to show a net negative impact at −8% for R_t= 224 and −24% for R_t= 256. This observation clearly shows the impact of $R_t$on the selection of the cap value C. The larger $R_t$is, the smaller cap value the system can afford to have. Also, the overall effectiveness of capping continues to dwindle as the shared resource size increases. This clearly reflects the degree of competition among the threads for the registers.

(3) Optimal cap value point with respect to the setting of R_t:

Optimal IPC gain occurs at a different cap value for each $R_t$setting. For the example of capping-at-*dispatch* case, when R_t= 160, optimal IPC happens at when cap value is set at 15, and the location continues to rise: 30 for R_t= 192, 45 for R_t= 224 and 50 for R_t= 256. That is, the larger the shared resources are given, the larger the cap value needs to be set to prevent the loss from pipeline bubbles to overcome the benefit from resource saving.

(4) Trade-off from where the capping is imposed:

Among the three stages, capping-at-*rename* almost consistently delivers the best result, outgaining the other two by another 2 − 3%. This comes from an obvious tradeoff in between capping at an earlier stage to benefit more from a shorter speculative trace and capping at a later stage to diminish adverse effect from delay in replenishing the pipeline after a correct speculation is resolved. That is, as shown in Figure 9, with the same cap value imposed, capping-at-*fetch* allows for a shorted overall speculative trace (X_f= 0) before it is resolved and thus leading to the smallest amount of resource wastage. However, by having a shorter trace in the system, the ensuing larger pipeline gap (bubble) takes a longer time to replenish if the speculation is resolved to be correct. On the other hand, capping-at-*dispatch* allows for the longest trace (largest X) to exist thus leading to the smallest benefit from curtailing resource wastage, but does not suffer as much in replenishing the pipeline.

(5) Optimal cap value point with respect to the stage of capping:

Further investigating the results leads to another interesting observation that the optimal cap value that leads to the highest IPC gain differs among the three stages of capping. This optimal cap value continues to shift downward as the stage of capping further moves down-stream. For example, for R_t= 160 (Figure 10), capping-at-*fetch* peaks at cap value equal to 15, capping-at-*rename* shifts to 12 and capping-at-*dispatch* further down to 8. That is, capping at an earlier stage has to adopt a larger cap value to prevent the aforementioned instruction- replenishing delay from severely offsetting the benefit from resource saving.

6.2. 8-threaded Workload

A similar simulation run is performed on the 8-threaded case with all the results displayed in Figures 14, 15, 16, and 17 with the $R_t$value being set at 320, 384, 448 and 512, respectively. These sizes are chosen to allow for the same resource allocation quota per thread compared to the 4-threaded case[1]. Compared to the results from the 4-threaded workload, the 8-threaded results are very similar except that the distinction among results from capping at the three different stages is much more pronounced. Again, the respective five observations are briefly summarized in the following:

(1) Trade-off due to the cap setting is in general similar to the 4-threaded results.

(2) Effectiveness of capping is even better than the 4-threaded one, consistently up to 35%, even for a very large register file. In addition, even a very small cap value would lead to improvement in most of the cases. All these clearly indicate a heavier resource competition among the higher number of threads even the per-thread quota remains the same. That is, a system with more threads has a higher chance of imbalanced resource occupancy, thus it tends to benefit more from the capping technique.

[1]Note that, for the rename physical register file, n × 32 entries are automatically pre-reserved for the n architectural register files, where n is the number of concurrent threads. Thus, R_t= 160 for 4-threaded case actually leaves a total of 32 (160 − 4 × 32) for renaming register allocation among the 4 threads – a quota of 8 per thread; similarly, R_t= 320 for 8-threaded case allows for 64 (320 − 8 × 32) rename registers, also a quota of 8 per thread.

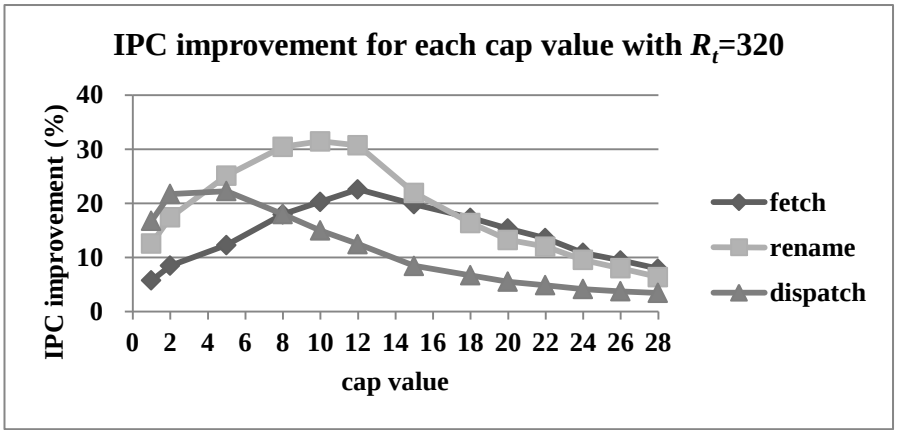

Figure 14. IPC Improvement for Each Cap Value with R_t=320 on 8-Threaded Workload

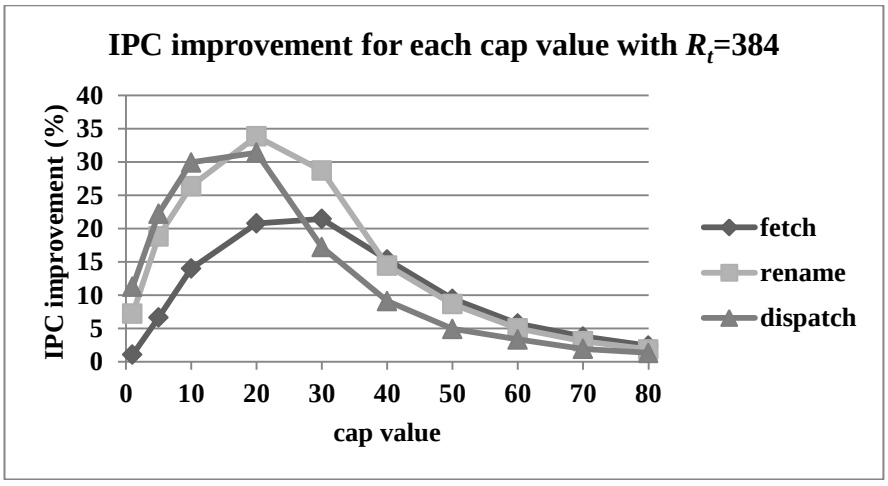

Figure 15. IPC Improvement for Each Cap Value with R_t=384 on 8-Threaded Workload

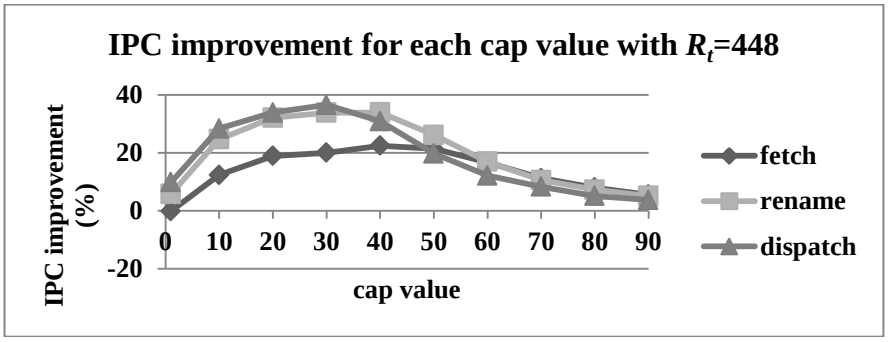

Figure 16. IPC Improvement for Each Cap Value with R_t=448 on 8-Threaded Workload

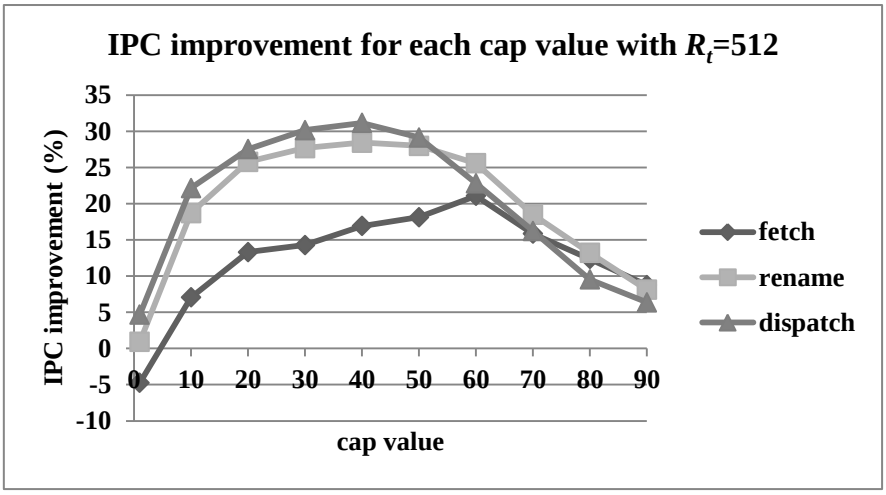

Figure 17. IPC Improvement for Each Cap Value with R_t=512 on 8-Threaded Workload

(3) Optimal cap value point shifts higher as $R_t$increases, similar to the 4-threaded case; however, compared to the 4-threaded case, this optimal point position is comparably lower, another indication of heavier resource competition.

(4) Capping-at-*rename* is the best among the three when $R_t$is small and capping-at-*dispatch* takes over the top when $R_t$becomes larger. This implies that the pipeline bubbles created by capping are harder to fill as $R_t$increases.

(5) Optimal cap value point shifts downward as the stage of capping further moves down-stream, a scenario even more prominent than the 4-threaded case.

6.3. Execution Fairness

To further demonstrate that overall IPC improvement by the proposed technique does not jeopardize the execution fairness due to the capping process, Figure 18 and Figure 19 display the Harmonic IPC (defined in Equation 2) for each mix with R_t = 160 for 4 threads and R_t= 448 for 8 threads respectively. During these simulation runs, stage-wise respective overall-IPC-optimal cap values are used – 15, 12, and 8 for the fetch, rename and dispatch stages, respectively on 4-threaded workload, and 40, 30, and 30 for the 8-threaded workload. The reason why these optimal cap values are adopted here in this part of simulations is that usually the highest degree of unfair execution happens when the overall system throughput (overall IPC) is optimized by favoring faster threads over slower threads. As shown in Figure 18, even under the situation where overall system throughput is maximized, irrespective of which stage the capping is imposed, the technique still manages to enhance the Harmonic IPC by up to 29%, an astonishing result in assuring execution fairness. An improvement of up to 6% is shown for the 8-threaded workload.

Combining all the results presented in this section, including analysis of the two trade-offs on overall IPC and execution fairness, we can see that rename stage in most cases is the best stage to apply the capping mechanism, with its result outperforming those from applying at the other two stages by up to 10%. This choice is especially more preferable when the amount of resources available is tight. All selections come without any suffering on execution fairness. That is, properly capping speculative threads not only prevents undesirable amount of resource wastage

from flush-outs but also balances the proceeding of threads by minimizing the resource-overwhelming and blocking from any thread.

7. CONCLUSIONS

A very thorough analysis was given in this paper in identifying the ideal pipeline stage to place the speculative trace capping mechanism in order to maximize system throughput and to understand the causes. The analysis results from this paper set a foundation for future extensions in further customizing the capping technique to best exploit the trade-offs discussed in this paper. One obvious direction is to dynamically adjust the cap value in real time based on the run-time speculative prediction result; that is, a speculative trace is allowed to grow longer if it has a higher chance of correct speculation. Such a dynamic approach not only can minimize the pipeline gap caused by the capping when the speculation has a higher chance of being correct, while under the same scheme can also curtail the wastage of shared resources when it is less expected to be correct. Another potential direction is to have a hybrid capping mechanism, which imposes multiple capping points at different pipeline stagesfor each individual trace. One intended benefit from this technique is to spread and thus dilute the congestion around all stages, instead of allowing all speculative instructions to flow into the *writeback* stage. Bottleneck problem should be lessened with this approach.

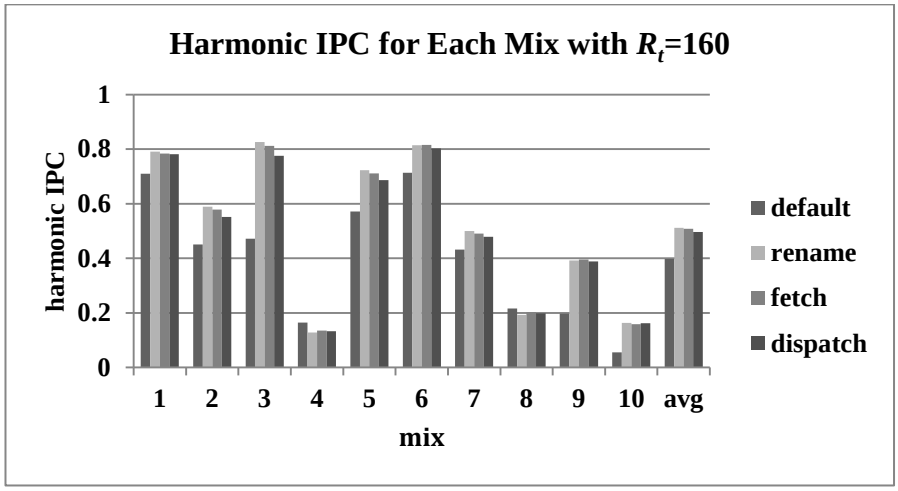

Figure 18. Harmonic IPC for Each Mix with R_t=160 on a 4-Threaded Workload

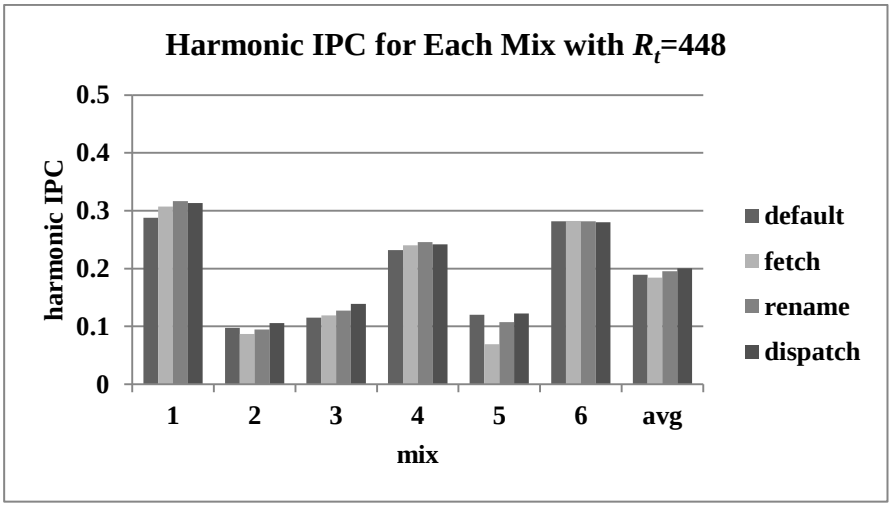

Figure 19. Harmonic IPC for Each Mix with R_t=448 on an 8-Threaded Workload

REFERENCES

[1] Y.Zhang,andW.-M.Lin,"CappingSpeculative Traces to Improve Performance in Simultaneous Multi-Threading CPUs", Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International IEEE, 2013.

[2] H.Hirate, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y.Nakase and T. Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads", Proc. of 19th Annual International Symposium on Computer Architecture, 1992, May, pp. 136-145.

[3] D.Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", Proc. of 22nd Annual International Symposium on Computer Architecture, 1995, May, pp. 392-493.

[4] D. M. Tullsen, S. J. Emer, H. M. Levy, J. L. Lo and R. L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multi-Threading Processor", Proc. of 23rd Annual International Symposium on Computer Architecture, 1996, May, pp. 191-202.

[5] F. J. Cazorla, A. Ramirez, M. Valero and E. Fernandez, "Dynamically Controlled Resource Allocation in SMT processors", Proc. of 37th International Symposium on Microarchitecture, 2004, Dec, pp. 171-192.

[6] Y. Zhang and W.-M. Lin, "Efficient Physical Register File Allocation in Simultaneous Multi-Threading CPUs", 33rd IEEE International Performance Computing and Communications Conference (IPCCC 2014), Austin, Texas, Dec, 2014.

[7] Y. Zhang and W.-M. Lin, "Intelligent Usage Management of Shared Resources in Simultaneous Multi-Threading Processors", The 21st International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'15), Las Vegas, NV, July, 2015.

[8] Y. Zhang, C. Douglas and W.-M. Lin,"Recalling Instructions from Idling Threads to Maximize Resource Utilization for Simultaneous Multi-Threading Processor", Journal of Computers and Electrical Engineering, 40 (2013).

[9] Y. Zhang and W.-M. Lin, "Write Buffer Sharing Control in SMT Processors", PDPTA'13: The 19th International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, July 22-25, 2013.

[10] S.Lawal,Y.ZhangandW.-M.Lin,"PrioritizingWriteBufferOccupancy in Simultaneous Multithreading Processors", Journal of Emerging Trends in Computing and Information Sciences, Vol. 6, No. 10, pp. 515-522, Nov 2015.

[11] S. Carroll and W.-M. Lin, "Latency-Aware Write Buffer Resource Control in Multithreaded Cores", International Journal of Distributed and Parallel Systems (IJDPS), Vol. 7, No. 1, Jan 2016

[12] S.McFarling, "Combining Branch Predictor", DEC Western Research Laboratory Technical Note TN-36, June 1993.

[13] S. Pan, K. So and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", 5th International Conference on Architectural Support for Programming Languages and Operating Systems, 1992, Oct, pp. 76-84.

[14] J. E. Smith, "A Study of Branch Prediction Strategies", In the Proceedings of the 8th Annual International Symposium on Computer Architecture, 1981, May, pp. 135-148.

[15] J. Sharkey, "M-Sim: A Flexible, Multi-threaded Simulation Environment", Tech. Report CS-TR-05-DP1, Department of Computer Science, SUNY Binghamton, 2005.

[16] Standard Performance Evaluation Corporation (SPEC) website, http://www.spec.org/.

[17] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, "Automatically Characterizing Large Scale Program Behavior", Proc. of 10th International Conference on Architectural Support for Programming Languages and Operating Systems, 2002, Oct, pp. 45-57.