International Journal of Computer Systems (ISSN: 2394-1065), Volume 04– Issue 04, April, 2017
Available at http://www.ijcsonline.com/

An Intelligent Fetching algorithm For Efficient Physical Register File Allocation In Simultaneous Multi-Threading CPUs

Madhava Krishnan Ramanathan, 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) is a technique that allows concurrent execution of multiple instructions of independent threads in each cycle by sharing the key data path components. Effectual utilization of these shared resources is vital to improving the systems performance. An instruction miss during the fetch stage usually leads to a large execution latency, thus the thread experiencing frequent cache misses tends to hold the physical registers for a longer period of time resulting in the shortage of registers available for renaming. This develops an intense competition among the threads for the free registers which may severely degrade the overall performance. In this paper, we propose an intelligent fetching algorithm which addresses the congestion in the physical register file by dynamically monitoring the cache misses, the average physical register file occupancy and the register deallocation rate for every thread in the system. This algorithm improves the IPC up to 54% and 59% for a 4-threaded and an 8-threaded SMT system respectively. Additionally the execution fairness is also preserved as demonstrated by the Harmonic IPC improvement of up to 27% and 29% respectively. Besides, the performance of a 4-threaded system with a physical register file size of 160 is comparable to the performance of the default system with the physical register size of 224 indicating a resource saving of 33%.

I. INTRODUCTION

Simultaneous Multi-Threading (SMT) technique enhances the execution efficiency of the traditional superscalar processors by exploiting Thread-Level Parallelism (TLP). In the SMT architecture multiple independent threads can reside in the pipeline during the same clock cycle, which cloaks some short-latency hazards (e.g., data dependencies) that may cause the pipeline to stall [3]. The sharing of key datapath components like the cache memory, physical register file, issue queue, write buffer among the multiple independent threads reduces the amount of hardware required in the SMT architecture than employing multiple copies of the traditional superscalar processors ([1], [2]).

There are several fetching policies for an SMT system to improve the resource allocation fairness and utilization. ICOUNT [4] favors the thread with a low resource occupancy which may highly correlate with a high pipeline efficiency. However, this policy does not directly consider the effect from cache misses, since the thread with a large number of cache misses may mislead this policy. BRCOUNT [4] gives a higher priority to the thread with the fewest unresolved branches in the decode stage, rename stage and the instruction queues, the implementation of which may not be practical latency-wise due to the complexity involved. Both the fetching policies mentioned above achieve resource allocation fairness but fail to account for the execution fairness among the threads in the system. STALL and FLUSH [5] takes care of the issues arising from L2 cache misses, but it is unfair to deallocate all the resources and flush all the instructions from the missing thread because more often it may lead to under-utilization of resources [6]. DCRA [6] dynamically decides the amount of resources required for each thread and averts threads from occupying more than the allocated quota of resources. This policy is efficient but it requires extra hardware and intelligent software support to monitor and to keep track of the resources required for each thread.

There are several research efforts targeted in effective utilization of physical register file which is a critical shared resource in the SMT architecture. In [7] a software-supported register deallocation scheme is presented in which the operating system, with the assistance of the compiler, is allowed to deallocate the physical registers early. [8] proposes a premature register deallocation technique to reduce the size of the register file required. In [9] a technique is proposed to reduce the register file access time by delaying the allocation of registers until a later stage in the pipeline using virtual physical registers. However, all the above-mentioned schemes require auxiliary software and hardware support and most of these techniques pose significant challenges to implementation due to their complexities.

In [1] an efficient register allocation technique is proposed which caps the number of renaming registers for each thread and the threads are not allowed to use more than the capped fraction of registers at any point of time. This technique, though leading to a very respected performance improvement, tends to be inflexible to adapt to various system loads due to its fixed capping scheme. In [2] an allocation algorithm is presented for effective utilization of physical register file by temporarily suspending the thread with the highest register file occupancy when the overall register file utilization exceeds a threshold. A potential drawback to this technique is in that it does not consider the register deallocation rate of the threads; that is, it may not be beneficial to suspend a thread with a better register deallocation rate, even though it holds a large number of registers.

In this paper, we propose an intelligent fetching algorithm

which dynamically monitors the cache misses, the physical register file occupancy and the physical register deallocation rate for each of the threads in the system. A thread is temporarily blocked from fetching if it is found to encounter a large number of cache misses (both instruction and data misses) and holds a large number of registers with a small register deallocation rate. The proposed scheme is at the architectural level and requires no modifications at operating system or compiler. Additionally, our scheme is a stand-alone process in the fetch stage and does not require any modifications in the other pipeline stages. This fetching scheme can therefore be incorporated to any advanced technique designed for other stages in the pipeline for an additive performance gain without corrupting the benefits of each other. Our simulation results indicate that our proposed fetching algorithm improves IPC by 54% and 59% and enhances the Harmonic IPC up to 27% and 29% on a 4-threaded and 8-threaded system respectively. From the resource utilization prospect, with the proposed fetching algorithm, the system is capable of achieving the same performance with a smaller register file size without much overhead in hardware or software. For example, a 4-threaded system with the register file of 160 entries can deliver a throughput in par with that of a default system with 224 entries, indicating a resource saving of 33%.

II. Simulation Environment

A. Simulator

We use the M-sim [10] a multi-threaded microarchitectural simulation model to evaluate the performance of our proposed technique in an SMT system. It is a cycle-accurate model with key pipeline structures including explicit register renaming. The cache memory, physical register file, issue queue, write buffers, functional units are shared among the threads and the ROB, LSQ and branch predictor are exclusive to each thread. Note that the access latency for various levels of cache and memory - 1 clock cycle for L1, 10 for L2 and 300 for memory - are the default values set by the simulator and are considered typical for the modern-day systems. Table 1 gives the detailed configuration of the simulated processor.

B. Workloads

Simulations in this paper are performed using the mixed SPEC CPU2006 benchmark suite [11] with mixtures of various levels of ILP for a diverse workload. ILP classification of each benchmark is obtained by initializing it in accordance with the procedure mentioned in the simpoints tool and simulated individually in a simplescalar environment. Three types of ILPs are identified, low ILP (memory bound), medium ILP and high ILP (execution bound). Table II and Table III gives the 4-threaded and 8-threaded workloads respectively.

C. Metrics

For a multi-threaded workload, throughput or the systems performance is measured in terms of overall IPC which is defined as the sum of each thread's IPC in the system.

$$\text{Overall\_IPC} = \sum_{i}^{n} \text{IPC}_i \qquad (1)$$

TABLE I
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 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

TABLE II
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	povray, milc, hmmer, sjeng	0	4	0
Mix4	lbm, cactusADM, xalancbmk, bzip2	4	0	0
Mix5	hmmer, gobmk, cactusADM,xalancbmk	2	2	0
Mix6	lbm, bzip2, soplex, leslie3d	2	2	0
Mix7	gcc, lbm, bzip2, povray	2	2	0
Mix8	libquantum, lbm, bzip2, namd	2	0	2
Mix9	dealII, gromacs, cactusADM, lbm	2	0	2
Mix10	povray, milc, cactusADM, xalancbmk	0	2	2
Mix11	hmmer, sjeng, lbm, bzip2	0	2	2
Mix12	dealII, lbm, bzip2, cactusADM	0	3	1

TABLE III
Simulated SPEC CPU2006 8-threaded Workload

Mix	Benchmarks	Classification(ILP)		
		Low	Med	High
Mix1	hmmer, sjeng, lbm, bzip2, povray, cactusADM, xalancbmk, gcc	0	4	4
Mix2	dealII, gromacs, namd, xalancbmk, hmmer, cactusADM, milc, bzip2	4	0	4
Mix3	hmmer, sjeng, gobmk, gcc, lbm, cactusADM, xalancbmk, bzip2	4	4	0
Mix4	gromacs, namd, sjeng, gobmk, gcc,lbm, cactusADM, xalancbmk	3	3	2
Mix5	perlbench, gcc, libquantum, soplex, hmmer, sjeng, lbm, bzip2	1	4	3
Mix6	perlbench, gromacs, namd, gobmk, gcc, libquantum, cactusADM, xalancbmk	3	2	3
Mix7	gcc, libquantum, cactusADM, xalancbmk, sjeng, leslie3d, gromacs, perlbench	3	3	2
Mix8	soplex, leslie3d, gcc, perlbench, bzip2,lbm, sjeng, hmmer	3	3	2

where n is the number of threads per mix in the system. However in order to prevent the starvation effect among the threads, the Harmonic IPC is adopted, which demonstrates the level of execution fairness amongst the threads.

$$\text{Harmonic\_IPC} = n / \sum_{i}^{n} \frac{1}{\text{IPC}_i} \quad (2)$$

In this paper, these two parameters are used to compare the proposed algorithm to the default system.

III. Motivation

This section is dedicated to the discussion of the motivation behind developing the proposed fetching algorithm. The algorithm proposed in this paper is based on the conjecture that a thread with a higher number of cache misses tends to occupy the physical register file for a longer period of time, that is, a longer allocation-to-deallocation duration. A load with threads of various miss profiles may lead to congestion and disproportional distribution of registers amongst the threads causing the overall system performance to degrade. This section includes simulation results to corroborate this conjecture and all the simulation results shown in this section are based on the system configuration with the 12 mixes of 4-threaded workload shown in the previous section.

A. Effects of Cache Misses

Cache miss occurs when the data or instruction requested by a thread is not found in the cache memory requiring it to be fetched from the main memory. This increases the execution latency thereby impairing the performance of the SMT system. Note that the cache memory is shared among the threads in an SMT system. A thread experiencing frequent cache misses tends to replace the cache blocks more often, which leads to a high probability that this thread may replace some cache blocks that are subsequently necessary for other threads potentially causing thrashing to happen among threads. Another serious impact to the overall system performance is from the usage of the shared physical rename registers – threads with a higher cache miss rate tend to hold their registers longer further decreasing utilization efficiency of registers and thus blocking execution flow of other faster threads. This is illustrated in Figure 1 which shows, for each mix, the relationship between the average register hold time (i.e. the average time from allocating a register to its deallocation) and its cache miss rate. Cache miss rate shown is the overall cache miss rate of the mix, including misses on both D-cache and I-cache. As aforementioned, these results are obtained using the 12 4-threaded workloads shown in the Table II with a physical register file size (denoted as R_f) of 160. In general, a mix with a higher miss rate tends to have a higher average register hold time. This analysis supports our conjecture that threads with more number of cache misses tend to occupy their allocated registers for a relatively longer time, which in turn may easily lead to congestion and disproportionate distribution of registers amongst the threads. This consequence will be more severe in the case of a system with a small physical register file due to heavier congestion.

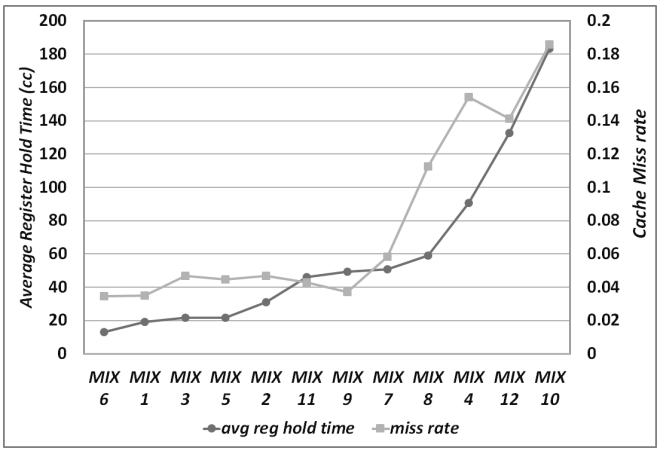

Fig. 1. Average Register Hold Time vs. Overall Cache Miss Rate

B. Critical Nature of Physical Register File

We now discuss and demonstrate how the size and distribution of physical register file affects the overall performance of the system. Figure 2 depicts the average overall IPC obtained under different physical register file sizes (R_f). When R_f increases from 160 to 320 we observe an overall

Fig. 2. Average IPC vs. Physical Register File Size

IPC improvement of 36%. It is important to note that a fixed portion of the physical register file is dedicated to the architectural registers – in a 4-threaded system, a total of $4 * 32 = 128$ registers are reserved for the 4 threads in an ISA with 32 architectural registers. Therefore, the number of registers available for renaming (denoted as R_r) to be shared among the 4 threads vary from 32 to 192. It is evident that the performance of a 4-threaded SMT system continues to improve as the physical register file size increases. Arbitrarily increasing the register file size is not practical due to power and delay constraints. In this paper we instead propose a technique which utilizes the physical register file efficiently and leads to a better performance even with a smaller physical register file. Also, note that the competition for floating point registers in general is not as intense when compared to the integer registers [1], thus the focus of this paper will only be directed to the latter.

In order to illustrate the need for a better register allocation platform for efficient utilization, an additional analysis is performed to show the degree of congestion from competition among the threads for these registers, given in Figure 3 for a 4-threaded SMT system.

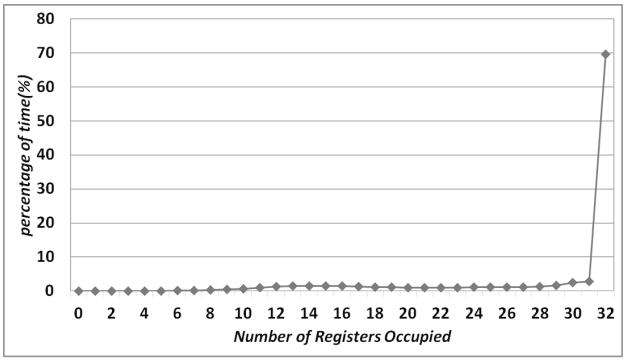

Fig. 3. Average Physical Register File Utilization Rate of 4-threaded System for R_f = 160

The result is obtained by averaging the register file occupancy for the 12 mixes shown in section 2 with R_f = 160. Note that in this case only 32 registers are available for renaming (R_r = 32). In about 70% of the time the registers are fully occupied which demonstrates the intense competition prevailing in the system. Such an overwhelming occupancy of physical register file will lead to shortage of renaming registers for the threads to use. This illustrates that the effective allocation and distribution of registers amongst the threads is a key factor to improve the performance of the SMT system.

IV. Proposed Method

The main idea behind the proposed algorithm is to temporarily suspend the thread in the fetching stage if it is found to potentially cause congestion in the physical register file. The algorithm employs a "window-based" suspension scheme; that is, in each current window of duration several critical statistics are gathered for the system so as to determine the action to take for the next time window. The decision to suspend a thread is made by observing its three resource utilization parameters:

- register occupancy rate – average number of registers occupied by it, an indicator of disproportional resource occupancy;
- register deallocation rate – the number of registers occupied are deallocated within the current window, an indicator of processing pace of the respective thread;
- cache miss rate – number of cache misses in the current window, an indicator of both processing speed and resource-overwhelming potential.

A threshold value for each of the above-mentioned parameters are set in advance and at the end of the current window if any thread in the system is found to "exceed" these preset threshold values then that thread will be suspended in the succeeding window. Figure 4 shows the sequence diagram of the proposed algorithm.

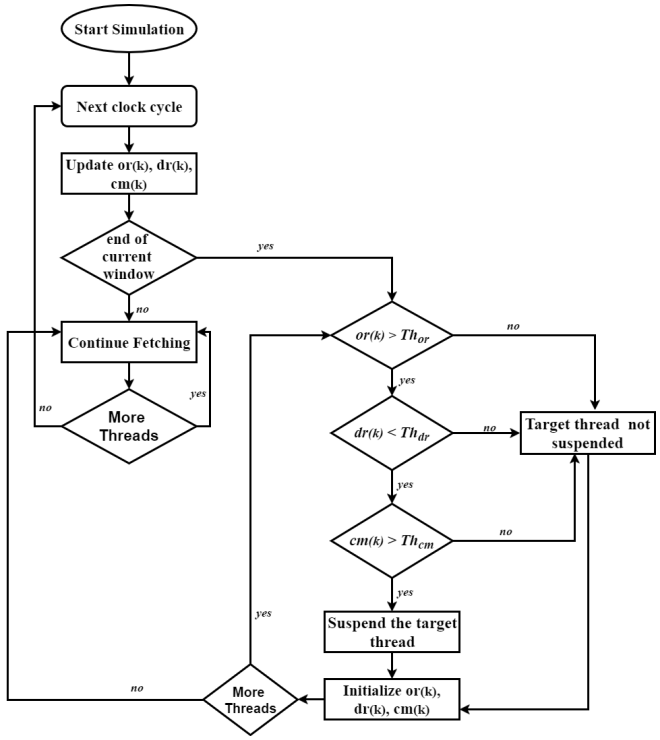

Fig. 4. Flowchart of the Proposed Algorithm

As indicated earlier, the physical register file is shared among the threads with R_r registers being truly shared. Thus, throughout this paper all the register occupancy quantification and threshold computations are made with respect to R_r. The algorithm monitors each thread in the system in a window basis and at the end of the current window it calculates the individual register occupancy of thread k, denoted as $or(k)$. If the average number of registers used by a thread is $o(k)$ then

$$or(k) = \frac{o(k)}{R_r}$$

At the end of every window, the algorithm then examines if the individual register occupancy exceeds the preset individual register occupancy threshold value (denoted as Th_{or}) for every thread to decide if the thread has to be suspended in the succeeding window; that is, when

$$or(k) > Th_{\text{or}} \tag{3}$$

However, a thread with a large $or(k)$ should not be suspended if it continues to deallocate these registers in an acceptable rate. Thus, a second parameter, register deallocation rate (denoted as $dr(k)$) is employed, to be calculated as

$$dr(k) = \frac{d(k)}{W_s}$$

where $d(k)$ denotes the number of registers deallocated by the thread in the current window, and W_s is the size of the window. Thus, the second criterium to satisfy for a thread to be suspended becomes:

$$dr_(k) < Th_{\text{dr}} \tag{4}$$

where Th_{dr} is the respective threshold.

The third criterium centers around the cache miss rate. Let $cm(k)$ denotes the number of cache misses encountered by thread k during the current window. The final suspension criterium is then

$$cm(k) > Th_{\text{cm}} \tag{5}$$

where Th_{cm} is the respective threshold.

As indicated in the flowchart, a thread is suspended only each of three conditions in Eq. 3, 4 and 5 is satisfied.

Note that any algorithm or technique designed with an intention to eradicate the monopolization of resources by a single thread may experience some pitfalls, and only when the benefits achieved can outweigh ensuing damages then the technique can be considered a successful one. Such a tradeoff exists in many aspects of this algorithm in terms of the selection of the threshold values, the size of the window, etc. For example, if the register occupancy threshold (Th_{or}) is preset to a value too high then very likely the intention of the proposed algorithm to eliminate the monopolization of the physical register file by a single thread will not be served well since the threshold condition is less likely to be satisfied, rendering the suspension mechanism less effective. On the contrary if this threshold value is set too low the algorithm will in turn suspend some threads unnecessarily causing under-utilization of resources and degradation of system performance. Likewise a register deallocation rate threshold (Th_{dr}) too high will unnecessarily suspend threads that are flowing through the system nicely even they may occupy the shared resources more than others. A value set too low in this instead again renders the the proposed suspension mechanism ineffective since the threshold condition is less likely to be met. A similar tradeoff happens in the setting of the cache miss rate threshold value, Th_{cm}.

Selection of a proper window size (W_s) also can matter significantly in the effectiveness of the proposed technique. A window size too large does not allow the technique to adapt well to changes of the threads' real-time behaviors; on the other hand, a window size too small not only has the system susceptible to some threads' short-term temporal fluctuations in behaviors but also makes the extra processing overhead per window harder to justify.

Also note that in the algorithm there is another parameter instilled to control, for a thread under suspension during a window, the "ratio" of suspension; that is, a suspended thread can be actually only "partially" suspended still allowing it to fetch during some portion of the clock cycles during the window. The suspension ratio (denoted as S) indicates the number of clock cycles per 100 in which the suspended thread is actually barred from fetching. This partial suspension mechanism, instead of a complete suspension (when $S = 100\%$), is employed to give the threads which encounter cache misses a slower fetching pace instead of completely shutting it down during a long window. Selection of the value S can very well affect the outcome of the imposed mechanism.

V. Simulation Results

The proposed algorithm is tested in the simulation environment [10] with the workloads mentioned in the section II. The improvement obtained using the proposed algorithm is calculated using the metrics described in the section II.

A. Parameter Settings

Each of the following subsections is devoted to the study of various parameter settings.

1) Window Size (W_s) and Suspension Ratio (S): The first parameter to set for our simulation runs is the window size W_s. Due to the concerns and tradeoffs aforementioned from this parameter's setting, a few tests lead to the selection of W_s = 1000 for all the simulation runs in this research, which is small enough to adapt well to temporal changes while large enough to absorb the extra processing overhead. On the setting of the suspension ratio S, Figure 5 depicts the per-mix IPC improvement obtained on a 4-threaded system with R_f = 160 (R_r = 32) when the proposed algorithm is applied. It is evident

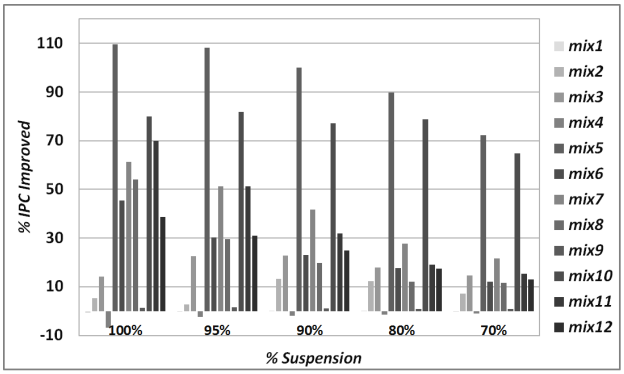

Fig. 5. Per-Mix IPC Improvement vs. Suspension Ratio ($S\%$) for a 4-threaded System with R_f = 160

from this result that most of the mixes (except mix 1 and mix 4) attain peak performance improvement when suspended for 100% of the time ($S = 100\%$) with the improvement ranging from 3% to 108%.

More tests are run to determine the effect of applying different S values on different physical register file sizes. Figure 6 and Figure 7 portray the average IPC improvement for different physical register file sizes where each point in the graph represents the average IPC obtained by applying the proposed algorithm over 12 mixes of 4-threaded workloads and 8 mixes of 8-threaded workloads respectively. The maximum average IPC improvement of up to 41% and up to 17% is achieved using the proposed technique for the 4-threaded and 8-threaded system respectively and for all the register file sizes used the maximum improvement is attained for 100% suspension. Allowing a thread responsible for congestion to fetch for even 5% of time in a window still causes congestion to prevail in the physical register file and it remains a bottleneck for the improvement of system performance. That is, in general, a complete suspension of the selected threads throughout the window is better than any partial suspension. Also from this result, the improvement continues to decrease when the register file size increases, a clear indicator that the proposed suspension mechanism is not as useful when there is a larger amount of shared resources

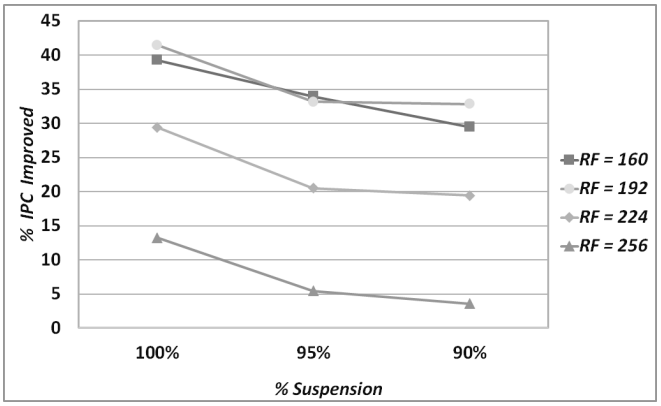

Fig. 6. IPC Improvement vs. Suspension Ratio ($S\%$) for a 4-threaded System with Different R_f

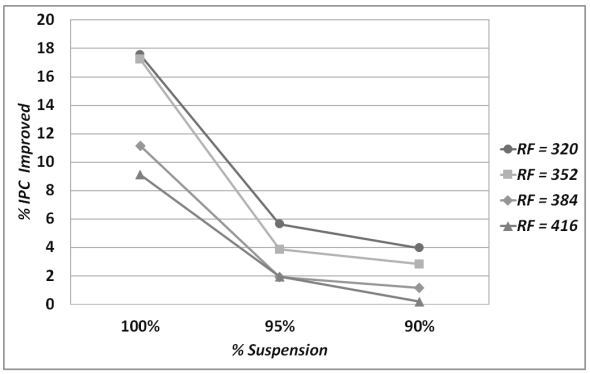

Fig. 7. IPC Improvement vs Suspension % for a 8-threaded system with different R_f

available, a situation that does not call for as many threads to be suspended.

Note that all the results shown throughout the rest of simulation runs are based on $W_s = 1000$ and $S = 100\%$.

2) Individual Register File Occupancy Threshold (Th_{or}): Figure 8 and Figure 9 depict the average IPC improvement obtained when varying the Th_{or} for different R_f with 4-threaded and 8-threaded workloads. It can be inferred from

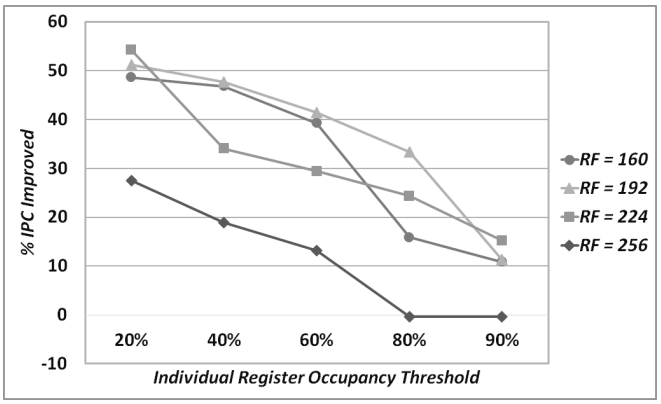

Fig. 8. Average IPC Improvement vs. Individual Register File Occupancy Threshold for Different R_f on A 4-threaded System

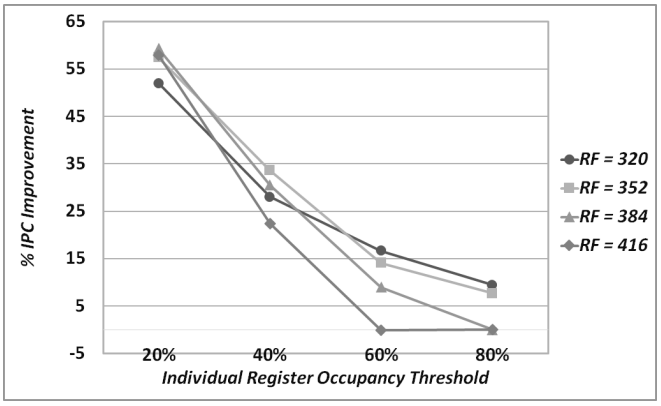

Fig. 9. Average IPC Improvement vs. Individual Register File Occupancy Threshold for Different R_f on An 8-threaded System

these plots that the proposed technique achieves the best improvement when the threshold is set a relatively low value. A very obvious trend from these results shows that, if the threshold is set a high value, the suspension condition becomes harder to satisfy and thus the benefit from the proposed suspension technique is less likely to surface. This is clearly demonstrated in the cases when the threshold is set higher than 70% for the 4-threaded case and 60% for the 8-threaded case the improvement is almost next to nil.

3) Individual Register Deallocation Rate Threshold (Th_{dr}): Figure 10 illustrates the average IPC improvement obtained when varying Th_{dr} for different register file sizes on a 4-threaded SMT system. Each point in the graph represents the IPC improvement obtained averaging over 12 4-threaded workloads for different values of Th_{dr}. The best performance of

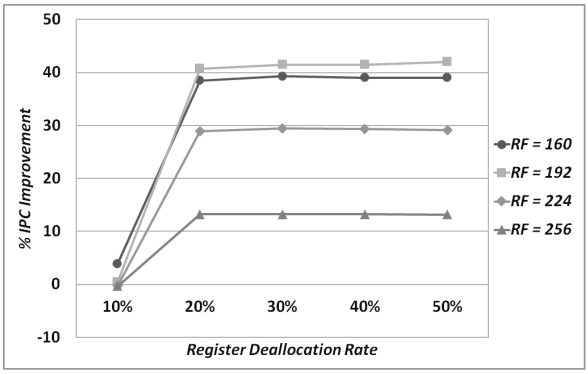

Fig. 10. Average IPC Improvement vs Register Deallocation Threshold for Different R_f on 4-threaded System

the proposed algorithm is reflected when register deallocation threshold is preset to any number above 30% which means, with $W_s = 1000$, the thread will be suspended if the number of register deallocated in the current window is less than 300. Any threshold value higher than 30% does not incur much difference in performance, which implies that in most of the mixes the deallocation rate is upper bounded by about 30%.

4) Cache Miss Threshold (Th_{cm}): Figure 11 shows the performance of the 4-threaded SMT system with different

values of cache miss threshold (Th_{cm}) for R_f = 160. The maximum performance improvement is achieved when this threshold value is set at 5. This results clearly reveals that

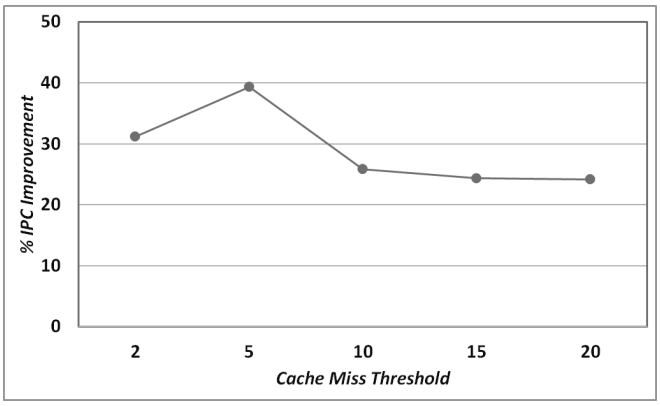

Fig. 11. Average IPC Improvement vs Cache Miss Threshold on A 4-threaded System with R_f = 160

even a small number of cache misses from a thread during a window can lead to a substantial blocking on the shared resources for other threads. This result also corroborates our claim in the previous section that it is important to initiate the suspension mechanism on a thread in the starting stages of congestion to get the maximum benefit out of the proposed technique. If a thread with the higher number of cache misses is allowed to fetch in the succeeding window it inflates the congestion in the system by holding the registers for a longer period without. This is critical in the system with a small register file.

B. Performance Comparison with Optimal Set-up

Equipped with all the best parameter settings obtained from the analysis aforementioned, the proposed suspension technique is then compared to the default system. Results are shown in Figure 12 and Figure 13 for the 4-threaded and 8-threaded system respectively by varying physical register file size (R_f). The results not only show a consistent improvement in IPC from employing the proposed technique, but also indicate that a 4-threaded system with a smaller

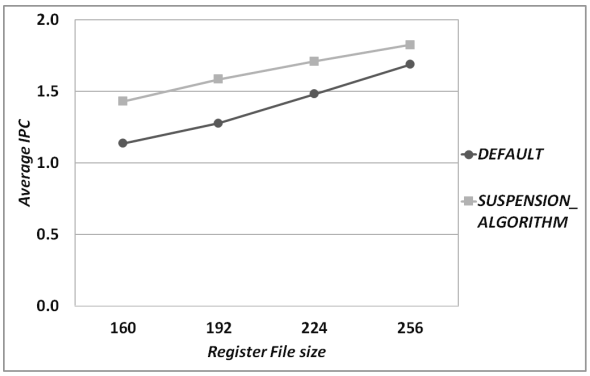

Fig. 12. IPC Comparison for A 4-threaded SMT System of Various R_f

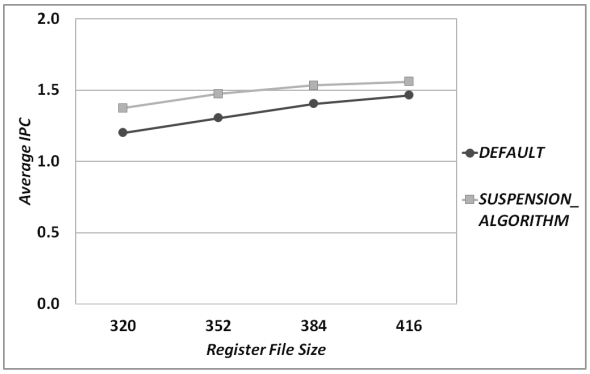

Fig. 13. IPC comparison for An 8-threaded SMT System of Various R_f

register file size $R_f = 160$ using the proposed technique can deliver a performance comparable to the default system with $R_f = 224$, demonstrating a resource saving of 33% in the number of physical registers (R_f) and 67% in the number of rename registers R_r (32 vs. 96). Similarly the performance of an 8-threaded system with 320 register file entries using the proposed suspension technique is comparable with the performance delivered by the default system with 384 register file entries exhibiting a resource saving of 17% in the number of register entries and 50% in the number of rename registers (64 vs. 128).

C. Execution Fairness

The significance of a fetching algorithm also depends on the degree of execution fairness it achieves amongst the threads in the system. The throughput improvement imparted to the system becomes insignificant if it is achieved through unfair execution amongst the threads or by causing thread starvation. Harmonic IPC defined in section II demonstrates the execution fairness attained by deploying this fetching algorithm. Figure 14 and Figure 15 illustrate the Harmonic IPC improvement attained by deploying this algorithm on a 4-threaded and an 8-threaded system respectively for different physical register file sizes. Various suspension ratio (S) values again are tested here to see if there is any effect on the issue of fairness. The plots indicate that suspending a thread for 100%

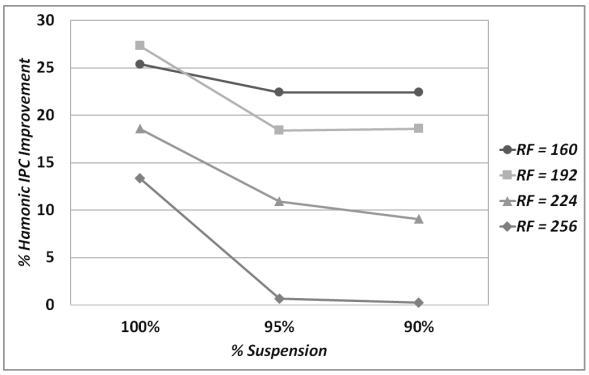

Fig. 14. Harmonic IPC Improvement vs. Suspension % for A 4-threaded SMT System of Various R_f

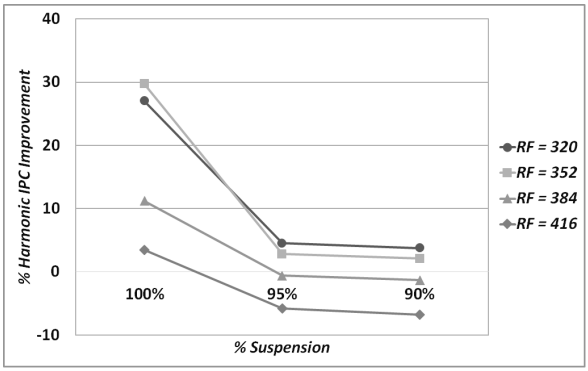

Fig. 15. Harmonic IPC Improvement vs. Suspension % for An 8-threaded SMT System of Various R_f

of time achieves maximum execution fairness amongst the threads making it an optimal suspension ratio to be deployed in the proposed fetching algorithm. For a 4-threaded system the proposed technique achieves an throughput improvement of 41% exhibiting a harmonic IPC improvement of 27% and for a 8-threaded system it achieves a throughput improvement of 18% exhibiting a harmonic IPC improvement of 29% indicating that the proposed algorithm does not cause unfair execution or thread starvation among the threads.

VI. Conclusion

SMT system needs its shared resource to be effectively distributed amongst the threads to achieve a high-performance gain. Overwhelming occupancy of the physical register file by a thread will degrade the performance of the SMT system. As illustrated in this paper the thread with a large number of cache misses tends to hold the registers for a longer period of time which will may cause congestion in the shared resource especially in the physical register file. This paper introduced an effective fetching algorithm which addresses the congestion in the partially shared physical register file. This fetching algorithm achieves a substantial performance gain by effectively distributing the physical register file among threads without sacrificing the execution fairness. This algorithm neither needs much extra hardware support nor requires any modifications in the pipeline stages. we believe that this intelligent fetching algorithm can be implemented alongside with other advanced techniques in different pipeline stages to extract a greater gain in the performance.

References

[1] 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, December 5-7, 2014.

[2] W. Wang and W.-M. Lin, "Efficient Physical Register File Allocation with thread suspension for Simultaneous Multi-Threading processors," 25th International Conference on Software Engineering and Data Engineering, SEDE 2016 - Denver, United States

[3] S. Carroll and W.-M. Lin, 'Dynamic Issue Queue Allocation Based on Reorder Buffer Instruction Status," International Journal of Computer Systems (ISSN: 2394-1065), Volume 02, Issue 9, September, 2015 Available at http://www.ijcsonline.com/

[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", *In the Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 191-202, May 1996

[5] D. M. Tullsen, J. A. Brown, "Handling Long-latency Loads in a Simultaneous Multithreading Processor", *In Proceedings of the 34th International Symposium on Microarchitecture,pp. 318-327, December 2001*

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

[7] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy and D. M. Tullsen, "Software-directed Register Deallocation for Simultaneous Multi-Threading Processors", *IEEE Transaction on Parallel and Distributed Systems*, Vol 10, Issue 9, pp 922-933, Sept 1999

[8] E. Quinones, J. Parcerisa, A. Gonzalez "Leveraging Register Windows to Reduce Physical Registers to the Bare Minimum", *IEEE Transaction on Computers*, Vol. 59, No. 12, Dec 2010

[9] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, V. Vinals, "Dynamic Register Renaming through Virtual-Physical Registers", *Journal of Instruction Parallelism*, Vol. 2, 2000

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

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