Microprocessors and Microsystems 57 (2018) 42–51

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

A controlled fetching technique for effective management of shared resources in SMT processors

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

ARTICLE INFO

Keywords:
Simultaneous Multi-Threading
Superscalar
Physical register file
Fetch stage

ABSTRACT

Simultaneous Multi-Threading (SMT) is a processor design technique that supports concurrent execution of instructions from multiple threads in every cycle by sharing the key datapath components. In the SMT architecture, the shared resources normally include the physical register file, Issue Queue (IQ), functional units, write buffer and the cache memory. Efficient utilization of the shared resources is critical to achieving high-performance gain. The physical rename register file is one of the most critical shared resources in the SMT architecture due to its being located at forefront of the pipeline stages. The inter-thread sharing of the physical registers reduces the number of registers required in the SMT processors than would have been needed in deploying multiple superscalar processors to achieve a similar throughput. However, due to the nature of sharing, an overwhelming occupancy of the physical register file by any slower threads can lead to a shortage of registers available for the other threads in the system and thus degrade the overall performance. In this paper, we propose an intelligent fetching algorithm for efficient management of the shared physical register file. Even though the primary focus of this paper is to manage the physical register file effectively, it indirectly controls the other shared resources downstream in the pipeline as well. The main goal of this paper is to propose a simple resource management scheme capable of achieving a considerable performance gain that neither incurs a substantial processing or hardware overhead for practical implementation nor requires modifications in the other pipeline stages. We demonstrate that temporarily suspending the slow threads from the system in the fetch stage can improve the overall system performance by a significant margin. An improvement of up to 63% and 68% is achieved when the proposed scheme is applied to the 4-threaded and the 8-threaded system respectively. The throughput of an 8-threaded system with 320 register file entries is significantly higher than the performance of default system with 416 register entries indicating a resource saving of 60%.

1. Introduction

In this paper, we propose a stand-alone resource management mechanism for an SMT (Simultaneous Multi-Threading) system which requires little additional hardware for practical implementation. The proposed method monitors the congestion in the physical rename register file on a window basis and temporarily suspends the thread(s) with excessive register utilization from fetching whenever there is a shortage of rename registers in the system. The proposed mechanism is deployed based on the simple Round Robin fetch policy to impart additional fairness to thread's execution. An improvement of up to 63% and 68% is achieved when our technique is applied to the 4-threaded and the 8-threaded system respectively.

1.1. Background

Traditional multi-Threading architectures eliminate most of the overhead from context switching in typical superscalar architectures by allowing multiple threads coexist in the CPU ready for switching. In addition to the ensuing increase in hardware requirement for the multiple threads, due to the insufficiency of instruction-level parallelism (ILP) existing in typical programs, a very significant amount of issuing slots (so called "pipeline bubbles") are still left unfilled. A fine-grain multi-threading further allows threads to take turn per clock cycle issuing their instructions to fill in more delay slots which are prevalent in a single thread's execution due to data dependencies. Still, limited ILP per thread still prevents this scheme from filling all slots within each clock cycle. Simultaneous Multi-Threading (SMT) takes this one step further by allowing all threads to issue their instructions within the same clock cycle by exploiting the far-more-abundant thread-level

[*] Corresponding author.
E-mail address: zdv592@my.utsa.edu (M.K. Ramanathan).

https://doi.org/10.1016/j.micpro.2018.01.001
Received 25 June 2017; Received in revised form 23 November 2017; Accepted 16 January 2018

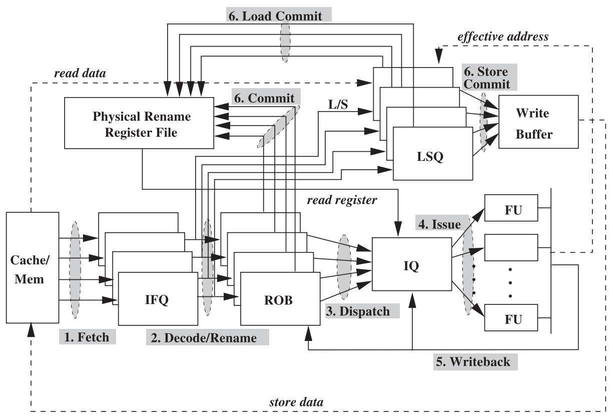

Fig. 1. Pipeline stages in a 4-threaded SMT system.

parallelism (TLP) ([1,2]). Due to the nature of concurrent execution among the threads, some resources (such as buffers and bandwidths) have to be shared among the threads. Thus, the amount of hardware required can be significantly less than that from deploying multiple copies of superscalar processors while achieving a similar throughput.

1.2. SMT system

An SMT processor is constructed from a typical out-of-order superscalar processor. The basic pipeline stages of a 4-threaded SMT system is shown in Fig. 1, and its basic operation flow is briefly described here for the sake of completeness. Instructions from a thread are *fetched* from memory (and cache) and put into their respective private Instruction Fetch Queue (IFQ). During the stages of *decode* and register-*rename*, a register from the shared Physical Rename Register File (PRRF) is assigned to any instruction that involves a write-after-write hazard, and the instructions are allocated into their respective Re-Order Buffer (ROB). In the subsequent *dispatch* stage they are then sent into the shared Issue Queue (IQ). Load/Store instructions have their operations dispatched into individual Load Store Queues (LSQ) with address calculation operations also sent into IQ. When instruction-issuing conditions (i.e., all operands ready and the required functional unit available) are met, the instructions/operations are then *issued* to their 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, while the store instructions are committed into the shared Write Buffer (WB).

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, as just mentioned, PRRF, various machine bandwidth, IQ, functional units, and WB. The PRRF is one of the most critical shared resources since it is located at the forefront of the pipeline stages and thus its "holding delay" is far longer than other shared resources at downstream stages.

1.2.1. Physical Rename Register File (PRRF)

A PRRF is predominantly adopted among multi-threaded processors [3,4] for register renaming, which contains more register slots than those defined in the ISA, the so-called "architectural" registers. On the rename stage, each instruction that writes to an architectural register will be assigned a physical rename register, or simply "physical register" in short, and all subsequent reads of that architectural register will have the data come from the most newly assigned physical register.

During the life of a program execution, except for the very early stage of it, each architectural register is mapped to at least one physical register at any given time and, most of the time, more than one if additional renaming registers are available. Namely, if an ISA has R_a architectural registers then R_a physical registers are considered dedicated to them for a program automatically throughout the program's execution. That is, at least R_a physical registers are considered occupied by this program at any time. The number of physical registers available for renaming will be limited if the size of the physical register file is not much larger than that of the architectural registers. This limitation effected from such a default dedication becomes even more exacerbated in an SMT system where the resource sharing among multiple threads is supposed to allow it to require less amount of resources than what would have been required in multiple single-threaded superscalar systems. For example, with the assumption of a basic ISA with 32 architectural registers, a one-threaded system with 48 physical registers would have 16 free physical registers for renaming. A 4-threaded SMT system is supposed to employ a physical register file with significantly fewer than 192 (4 × 48) entries. However, under the default dedication rule, a 4-threaded system requires a minimum of 128 (4 × 32) physical registers, which does not leave much room for resource saving. If one chooses to adopt the use of 160 physical registers, a saving of 32 registers, it leaves a total of 32 registers (160 − 128 = 32) available among 4 threads for renaming. Throughout the rest of this paper, R_t, R_a, and R_r are used to denote the number of all physical registers, per-thread architectural registers, and all the additional physical registers for renaming, respectively. Thus,

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

where N is the number of threads in the system. This scheme of allocation is displayed in Fig. 2.

Furthermore, compared to a single-threaded system, an SMT system suffers even more from the potentially long register occupancy time due to the aforementioned program execution behaviors. Long register occupancy time in general leads to a delay of renaming (due to the lack of free physical registers) and thus a blocking of the program's execution. In a single-threaded system, without employing more resources, the blocking is inevitable since the order of committing instructions in the only program can not be altered. On the other hand, in an SMT system, the order of execution/committing instructions among different independent threads does not have to be fixed. Thus, long register occupancy from one slower thread could easily hog most of the physical registers and subsequently stall all other threads' processing, unnecessarily leaving an excessive number of pipeline bubbles unfilled. As a result, it can lead to the degradation of overall system performance. Unfair distribution of physical register file among threads can easily render it a severe bottleneck along the pipeline stages. Thus, the focus of this paper is on the efficient utilization of physical register file among multiple threads.

1.2.2. Related work

There are several fetching policies for an SMT system to improve the resource allocation fairness and utilization. ICOUNT [5] favors the

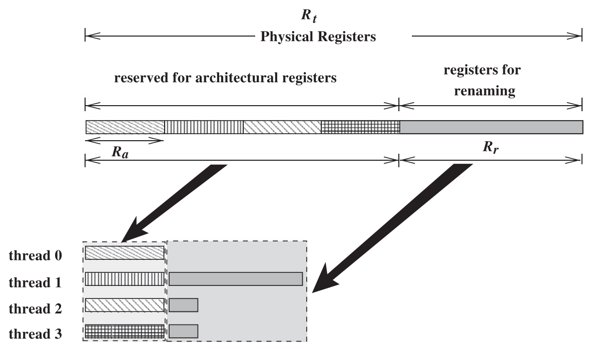

Fig. 2. Sharing of Physical Rename Registers among SMT Threads.

thread with a low resource occupancy which may highly correlate with a high pipeline efficiency. However, ICOUNT is less efficient in handling the threads with a higher cache miss rate since it is unaware of the fact hundreds of clock cycles, which will lead to monopolization of resources for a long time. BRCOUNT [5] 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 [6] fetch policy is built on top of ICOUNT to address the issues arising from the L2 cache miss, it detects and blocks the thread with a pending L2 cache miss from further fetching instructions. However, the detection process must be fast enough to prevent the thread from occupying the most of the available resources. FLUSH [6] is an extension of STALL fetch policy, it deallocates all the resources of the thread with a pending L2 cache miss to make it available for the other threads. But when the resources deallocated from the missing thread remain unutilized it might lead to under-utilization of resources. Moreover, the instructions that are flushed for that thread has to be fetched again which requires extra fetch and power. DCRA [7] dynamically decides the amount of resources required for each thread and prevents the 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 necessary 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 [8] 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. Quinones et al. [9] proposes a premature register deallocation technique to reduce the size of the register file required. In [10] 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 [11] 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 [12] 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. Also, the threshold parameters (overall register file occupancy and the individual register file occupancy) should be handled efficiently to exploit the benefits of this algorithm else it might lead to an undesirable performance outcome. Both the techniques in [11] and [12] are deployed in the rename stage of the pipeline which may cause a bottleneck in the other pipeline stages. For example implementing [12] in an SMT architecture with a shared IFQ might cause a bottleneck in the fetch stage as the instructions of the thread suspended in the rename stage will reside in the IFQ which might lead to under-utilization of IFQ slots. In [13] an intelligent thread-suspension algorithm is proposed for the effective sharing of physical register file among the threads in the system. Even though the proposed technique achieves a significant performance gain it requires its threshold values (individual register occupancy threshold, cache miss threshold, register deallocation threshold) to be managed carefully to exploit the full benefits.

Some of the control techniques [8–10] mentioned above are not a stand-alone process requiring the support of the operating system and the compiler. Also, these techniques are not proposed at the architectural level which requires modifications in the other pipeline stages. The techniques proposed in [11–13] requires extra overhead hardware and high-speed arithmetic units for the practical implementation.

In this paper, we propose a stand-alone resource management scheme at the architectural level which requires a very minimum hardware for its practical implementation. The proposed method monitors the congestion in the physical register file on a window basis and temporarily suspends the thread with the highest register utilization from fetching when there is a shortage of rename registers in the system. We use the overall register utilization rate as the indicator of congestion in the physical register file and adopt a upper threshold and a lower threshold to initiate or terminate the suspension. The technique temporarily suspends the thread from fetching and retains it in the suspension buffer until the congestion in the physical register file falls below a set threshold. The suspension buffer is implemented as a simple First In First Out order (FIFO) queue. Even though the primary focus of the proposed method is to manage the physical register file effectively, it indirectly controls the other shared resources as well. Since in this method the thread which is removed is not allowed to enter the pipeline it relieves the pressure on the other shared resources to a certain level. An improvement of up to 63% and 68% is achieved when the proposed scheme is applied to the 4-threaded and the 8-threaded system respectively. Additionally, the execution fairness is also preserved as demonstrated by the Harmonic IPC improvement of up to 120% and 55% respectively. Besides, the throughput of a 4-threaded system with 160 register file entries is comparable to the performance of default system with 256 entries indicating a resource saving of 60%.

2. Simulation environment

2.1. Simulator

We use the M-sim [14] 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.

2.2. Workloads

Simulations in this paper are performed using the mixed SPEC CPU2006 benchmark suite [15] with mixtures of various levels of ILP for a diverse workloads. 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, i.e. under single-thread simulation using M-sim. Three types of ILPs are identified, low ILP (memory bound), medium ILP and high ILP (execution bound). Instead of estimating the ILP value, we use each benchmark's IPC for classification for the sake of simplification – higher IPC benchmarks are classified as with a higher IPL. Tables 2 and 3 gives the 4-threaded and 8-threaded workloads respectively.

2.3. Metrics

For a multi-threaded workload, throughput or the systems performance is measured in terms of overall IPC which is defined as the sum

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 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 2
4-threaded Workloads for simulation.

Mix	Benchmarks	Classification (ILP)		
		Low	Med	High
Mix 1	libquantum, dealII, gromacs, namd	0	0	4
Mix 2	soplex, leslie3d, povray, milc	0	4	0
Mix 3	povray, milc, hmmer, sjeng	0	4	0
Mix 4	lbm, cactusADM, xalancbmk, bzip2	4	0	0
Mix 5	hmmer, gobmk, cactusADM, xalancbmk	2	2	0
Mix 6	lbm, bzip2, soplex, leslie3d	2	2	0
Mix 7	gcc, lbm, bzip2, povray	2	2	0
Mix 8	libquantum, lbm, bzip2, namd	2	0	2
Mix 9	dealIII, gromacs, cactusADm, lbm	2	0	2
Mix 10	povray, milc, cactusADM, xalancbmk	2	2	0
Mix 11	dealII, lbm, bzip2, cactusADM	3	0	1
Mix 12	libquantum, leslie3d, povray, namd	0	2	2

Table 3
8-threaded Workloads for simulation.

Mix	Benchmarks	Classification (ILP)		
		Low	Med	High
Mix 1	hmmer, sjeng, lbm, bzip2, povray, cactusADM, xalancbmk, gcc	4	4	0
Mix 2	dealII, gromacs, namd, xalancbmk, hmmer, cactusADM, milc, bzip2	3	2	3
Mix 3	libquantum, dealII, gromacs, namd, soplex, leslie3d, povray, milc	0	4	4
Mix 4	gromacs, namd, sjeng, gobmk, gcc, lbm, cactusADM, xalancbmk	4	3	2
Mix 5	perlbench, gcc, libquantum, soplex, hmmer, sjeng, lbm, bzip2	1	4	3
Mix 6	perlbench, gromacs, namd, gobmk, gcc, libquantum, cactusADM, xalancbmk	3	3	2
Mix 7	gcc, libquantum, cactusADM, xalancbmk, sjeng, leslie3d, gromacs, perlbench	2	4	2
Mix 8	soplex, leslie3d, gcc, perlbench, bzip2, lbm, sjeng, hmmer	0	5	3
Mix 9	gromacs, hmmer, sjeng, gobmk, gcc, lbm, cactusADM, xalancbmk	1	4	3

of each thread's IPC in the system.

$$\text{OverallIPC} = \sum_{i}^{n} \text{IPC}_i \tag{2}$$

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

$$\text{HarmonicIPC} = n / \sum_{i}^{n} \frac{1}{\text{IPC}_i} \tag{3}$$

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

3. Motivation

The algorithm in this paper is designed based on the conjecture that, in an SMT system the physical register file of limited size tends to be the bottleneck for the system performance due to a higher register occupancy rate and disproportionate distribution of registers among the threads. 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 4-threaded and the 8-threaded workloads shown in the Section 2.

3.1. Critical Nature of Physical Rename Register File

We now discuss and demonstrate how the size and distribution of physical register file affects the overall performance of the system. Fig. 3 depicts the average overall IPC obtained under different physical register file sizes (R_t). When R_t increases from 160 to 320 we observe an overall IPC improvement of 36%. As aforementioned, the number of registers available for renaming (R_r) to be shared among the 4 threads varies 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 [11], 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 Fig. 4 for a 4-threaded SMT system. The result is obtained by averaging the register file occupancy for the 12 mixes shown in Section 2 with R_t = 160. Note that in this case only 32 registers are available for renaming (R_r = 32). In about 70% of the time

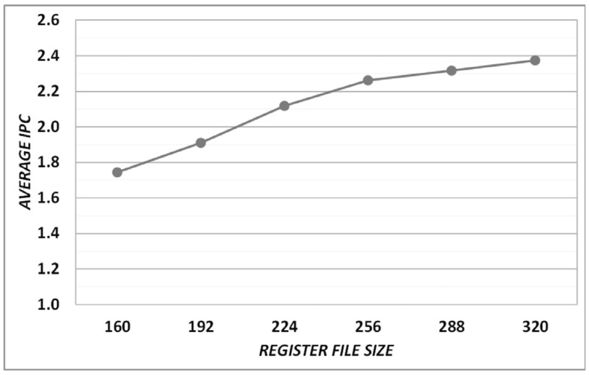

Fig. 3. Physical Register File Size vs IPC on 4-threaded Workloads.

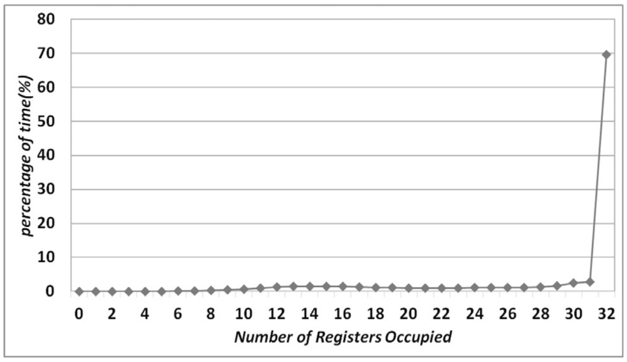

Fig. 4. Average Physical Register File Utilization Rate of 4-threaded Workloads with R_t = 160.

the registers are fully occupied which demonstrates the intense competition prevailing in the system. Such an overwhelming occupancy of physical register file, coupled with a longer holding latency, 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.

3.2. Benefits of thread suspension

To study the impact of initiating the suspension mechanism on the SMT system, we first run simulations by suspending a specified number of randomly selected threads on a window basis for 4-threaded and 8-threaded workloads that are shown in Table 2. That is, during each window of a fixed number of clock cycles, a number of randomly selected threads are suspended from fetching. Fig. 5 shows the IPC improvement attained by randomly suspending threads in every window of 1000 clock cycles for the 4-threaded workloads of $R_t = 192$ and 8-threaded workloads of $R_t = 352$. Note that the effect of suspending one thread from fetching per window in a 4-threaded system is not the same as running 3 threads only, since in the former case the suspended thread still occupies at least R_a rename registers even though it does fetch more new instructions. As aforementioned, these registers will not be released until the next instruction with the same co-renamed register is committed. By randomly suspending one thread and two threads in the 4-threaded SMT system an IPC improvement of 30% and 47% is obtained respectively. Similarly, on the 8-threaded system an improvement of 7%, 14% and 20% is observed by suspending one, two, and three random threads respectively. This clearly ascertains our assumption that thread suspension has a very promising impact on the performance outcome of an SMT system if the shared resources are relatively tight.

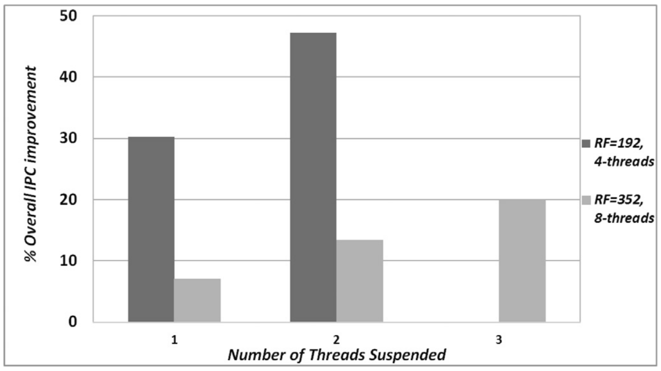

Fig. 5. Overall IPC Improvement % vs Number of Threads Suspended for 4-threaded and 8-threaded Workloads.

4. Proposed method

The main idea behind the proposed algorithm is to temporarily suspend one or more “slower” threads from fetching if the physical register file is found to be heavily congested. Depending on the degree of congestion relaxation, one or all the threads currently suspended will be revived again for fetching.

The algorithm employs a “window-based” monitoring 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 remove a thread is made by observing the following resource utilization parameters:

- cumulative overall register utilization number – the total number of rename-register-clock-cycles occupied by all the threads during the current window, denoted as RO. This is eventually used to determine the degree of congestion in the physical register file and if the suspension of another thread will be initiated for the next window.
- individual register occupancy rate – the number of registers occupied by a thread in the very last clock cycle at the end of the current window, denoted as U_i for thread i, an indicator of overwhelming resource occupancy. We do not use the ”average register occupancy” of a thread to calculate U_i because of higher degree of complexity involved in its practical implementation.

For the sake of having a more “normalized measurement” for different systems with various register file sizes, an overall register utilization rate will be used, denoted as U_{O}, and

$$U_{\mathrm{O}} = \frac{R_{\mathrm{O}}}{R_r \times W_s}$$

where R_r is the total number of rename registers as defined in Eq. (1) and W_s is the size of the observation window in clock cycles. This ratio is then used to determine if more suspension is called for or some threads currently suspended should be released in the next window. Two threshold values are preset for this purpose:

- θ_{U}: the upper threshold
- θ_{L}: the lower threshold

Fig. 6 shows the sequence diagram of the proposed technique. Our proposed suspension technique is a modified fetching algorithm based on a simple round-robin approach, namely, in each clock cycle, fetching only the threads that are not suspended in a round-robin fashion.

Note that a suspension buffer Ψ is used to retain the IDs of all the threads that are currently suspended, and is processed in a First-In-First-Out (FIFO) fashion. During the current observation window, the cumulative overall register utilization number (R_{O}) is updated/accumulated accordingly. At the end of the window, the overall register utilization rate (U_{O}) is then calculated, along with the individual register occupancy rate (U_i) for each thread i. Three different actions are to take place depending on the relationship between R_{O} and the two preset threshold values:

- $\theta_{\mathrm{U}} \le U_{\mathrm{O}}$:
  when the upper threshold is exceeded, indicating a high potential of blockage, the currently active thread with the highest U_i is appended to the suspension buffer. That is, if the index of this thread is α

  $$\Psi \leftarrow \Psi \cup \{\alpha\}$$

- $\theta_{\mathrm{L}} \le U_{\mathrm{O}} < \theta_{\mathrm{U}}$:
  when the overall utilization falls below the upper threshold but still higher than the lower threshold, a situation we deem viable for releasing one suspended thread back to the system. In this case,

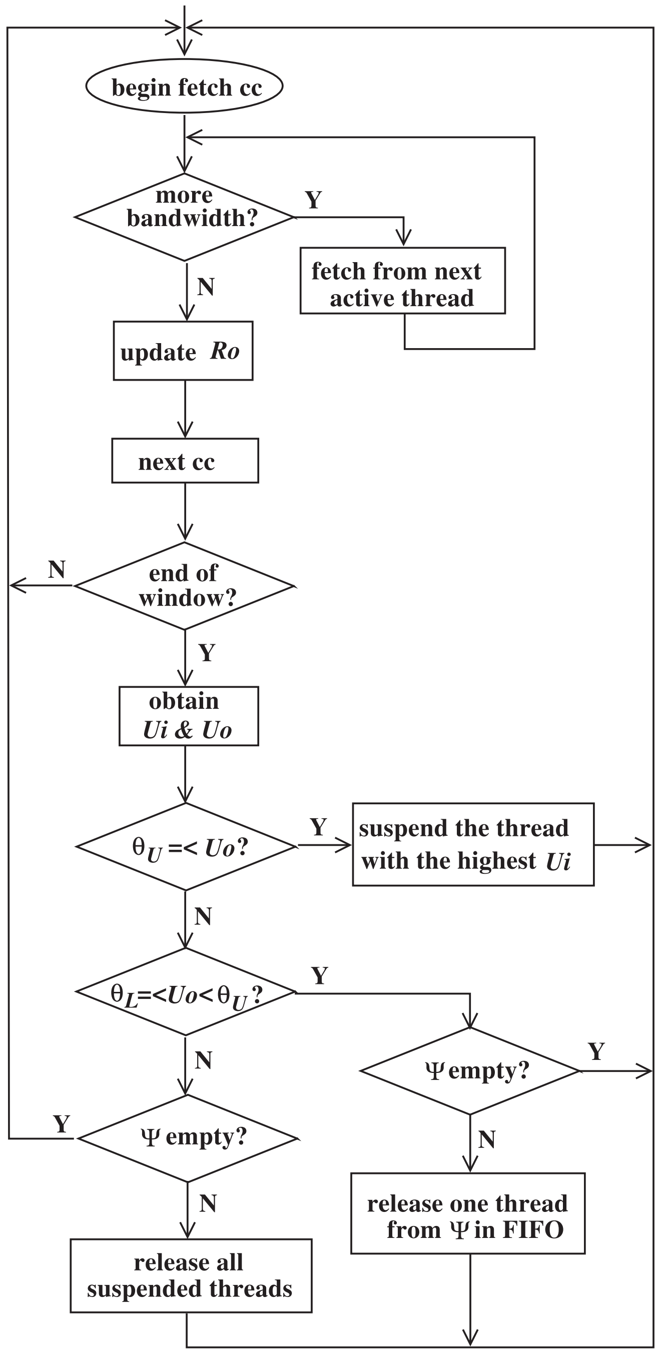
begin fetch cc
more bandwidth?
Y
fetch from next active thread
N
update Ro
next cc
N
end of window?
Y
obtain Ui & Uo
θU =< Uo?
Y
suspend the thread with the highest Ui
N
θL=<Uo<θU?
Y
Ψ empty?
Y
N
N
Y
Ψ empty?
N
release one thread from Ψ in FIFO
release all suspended threads

Fig. 6. Sequence diagram of the proposed technique.

provided that Ψ is not empty, the least recently suspended thread will be released for the sake of fairness and thread starvation prevention:

$\Psi \leftarrow \Psi \setminus \{\Psi_0\}$

where Ψ_0 represents the first entry in the buffer.

- $U_O < \theta_L$:
  when the utilization is less than the lower threshold, a scenario that should allow for full multi-threading capacity, all suspended threads will be released back to the system, that is,

  $\Psi \leftarrow \phi$

Note that removing a thread from fetching will not lead to a blocking in any other pipeline stages. This is because no further fetching is allowed for this thread and its already-fetched instructions in the rename stage can continue to finish and commit safely, thereby releasing the shared resources which can be utilized by the other threads in the system.

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 upper occupancy threshold, θ_U, 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 lower threshold, θ_L, too high will trigger the release of all suspended threads too fast causing the system to clog again immediately, whereas a threshold too low tends to under-utilize the already available resources.

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 renders the extra processing overhead per window harder to justify.

5. Simulation results

The proposed algorithm is tested in the simulation environment [14] with the workloads mentioned in the Section 2. The improvement obtained using the proposed algorithm is calculated using the metrics described in the Section 2.3.

5.1. Parameter settings

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

5.1.1. Window size (W_s)

The first parameter to set for our simulation runs is the window size. Due to the concerns and tradeoffs aforementioned from this parameter's setting, a few tests lead to the selection of window size of 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.

5.1.2. Lower threshold value (θ_L)

First part of our simulation is devoted to the testing of using various values of θ_L when the upper threshold is fixed. Figs. 7 and 8 show the average IPC improvement obtained on the 4-threaded SMT system and 8-threaded SMT system respectively, with θ_L set at 10%, 30% and 50%, when θ_U is fixed at 90% and 95%. All these settings are run with four different register file (R_t) sizes: 160, 192, 224 and 256. From the plots there is a very insignificant variation in performance improvement for various preset values of θ_L. Such an invariance is mostly due to the fact that the overall register utilization rate (U_O) mostly stays in a relatively high range. Only when almost all threads are suspended does it fall below the lower threshold, and it decreases drastically to a very small

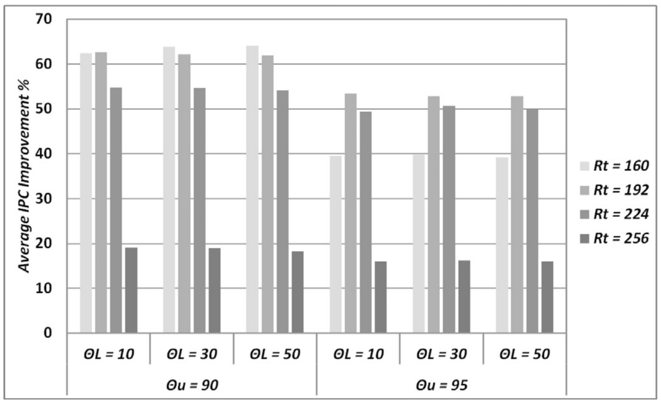

Fig. 7. Average IPC Improvement on 4-threaded Workloads with Various θ_L (*L*th in the plots) Values under Two Different θ_U (*Uth* in the plots) Values.

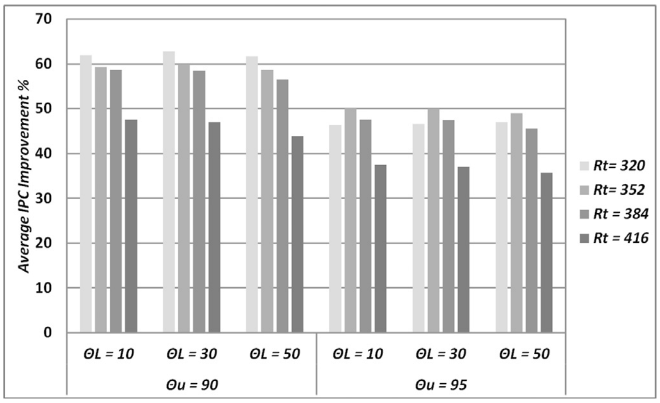

Fig. 8. Average IPC Improvement on 8-threaded Workloads with Various θ_L (*L*th in the plots) Values under Two Different θ_U (*Uth* in the plots) Values.

value which renders the setting of θ_L mostly irrelevant. Such a lower threshold setting allows the system to revive all the threads immediately when the resource utilization has become very inefficient. Due to such an irrelevance, throughout the rest of our simulations, θ_L is present to 30%.

5.1.3. *Upper threshold value (θ_U)*

We then continue to determine the effect from employing different upper threshold values (θ_U). Fig. 9 shows the per-mix IPC improvement obtained for the different values of θ_L. The results are obtained by running simulations on the 12 4-threaded workloads shown in the Table 2 with $R_t = 160$. Across the spectrum, most of the mixes show various degrees of improvement. Mixes 5, 6, 7, 8, 10 and 11 all demonstrate a very significant amount of improvement, with mix 7 tripling its IPC when θ_U is set at 75%. All these mixes are the ones with a huge of amount resource contention, thus suspending threads can easily lead to performance improvement, and most of these mixes have their performance improvement peaked at $\theta_U = 75\%$. Mix 6 even attains an improvement of about 30% when $\theta_U = 100\%$, which indicates a very severe contention in this mix. By observing the performance improvement achieved by mix 4 (combination of 4 low ILP threads, highly memory intensive), mix 5 and mix 6 (combination of 2 low ILP threads and 2 medium ILP threads) we can conclude that our proposed technique works well even on the most memory intensive combinations of mixes.

All the other mixes (1, 2, 3, 4, 9 and 12) instead suffer a very small degree of performance degradation, and the smaller the θ_U is, the worse their performance becomes. Obviously these mixes have a much smaller degree of resource contention and would have been better off without thread suspension. This is clearly demonstrated by the results where once θ_U is set above 90% no more performance degradation is observed, and some of the threads such as thread 4 begins to show some positive improvement. Such a scenario is obviously attributed to the lighter contention which rarely leads to a utilization rate higher than 90%.

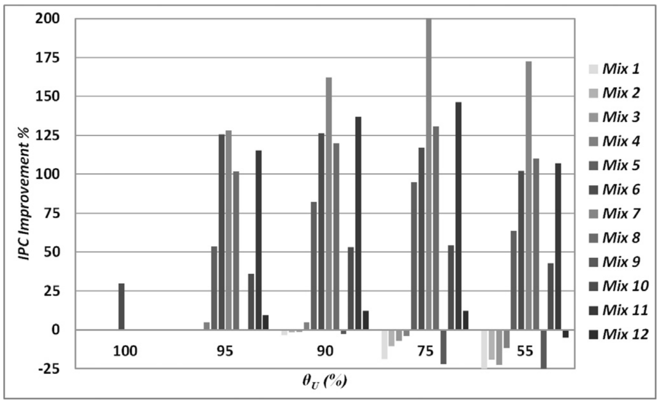

Fig. 9. Per-Mix IPC Improvement on 4-threaded Workloads with Various θ_U Values under $R_t = 160$.

5.2. *Effects on various register file sizes*

More tests are run to determine the effect of applying different θ_U values on different physical register file sizes. Figs. 10 and 11 show the average IPC improvement obtained for different register sizes on the 4-threaded SMT system and the 8-threaded SMT system respectively by varying θ_U. An improvement of up to 63% and 68% is achieved by the 4-threaded and 8-threaded workloads respectively. Peak performance improvement is attained at the range of $75\% \le \theta_U \le 90\%$. Once the threshold is set higher than this range, the benefits of the proposed thread suspension scheme are not as likely to surface since it less likely for the threshold condition to be satisfied. Conversely, when the threshold is set lower than this range (such as at 55%) it leads to unnecessary removal of threads causing the performance to degrade.

Note that, for the 4-threaded workloads, all the register file sizes except $R_t = 160$ achieve their peak performance when θ_U is set to 75%, while for $R_t = 160$ peak performance is attained at a higher threshold value at $\theta_U = 90\%$. Such a distinction originates from the fact that, for a system with a smaller register file size such as $R_t = 160$ (R_r is only 32), the register occupancy rate is constantly high, thus setting a threshold too low (such as at 75%) will lead to an excessive amount of unnecessary thread suspensions which causes the system performance to decrease significantly. A similar situation happens at $R_t = 320$ for the 8-threaded workloads. Another observation interesting to us is at $\theta_U = 100\%$ where the improvement for the 8-threaded workloads is virtually zero since, with R_r being at least 64, it is very unlikely for the

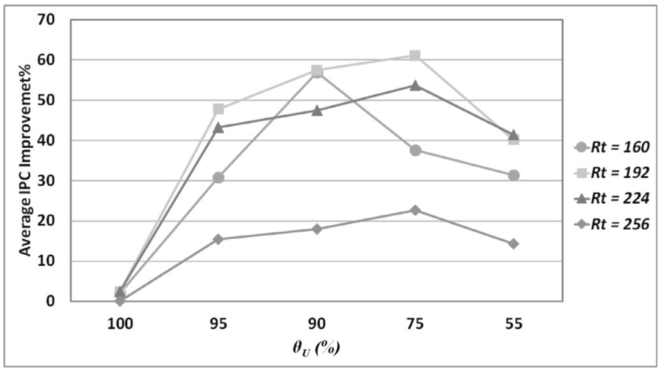

Fig. 10. Average IPC Improvement vs. θ_U on 4-threaded Workloads with Different Register File Sizes.

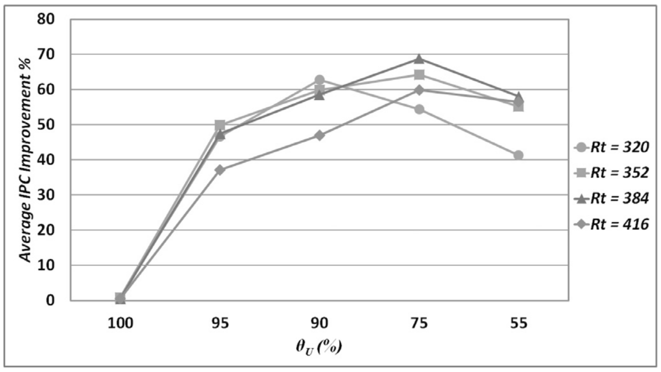

Fig. 11. Average IPC Improvement vs. θ_U on 8-threaded Workloads with Different Register File Sizes.

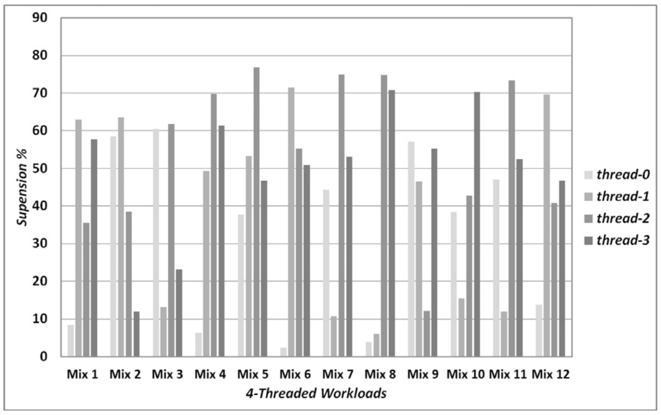

Fig. 12. Per-Thread Suspension Rate on 4-threaded Workloads under R_t = 160.

average register utilization to be equal to 100% for the suspension mechanism to be triggered. For the 4-threaded case, other than the R_t = 256 case (R_r = 128), all achieve some marginal performance improvement, up to 3 percent.

To illustrate that the imposed suspension mechanism does not lead to constant suspension of any single thread which in turn presents a disaster in execution fairness, further analysis is made on the per-thread suspension rate on each mix of the 4-threaded workloads. Fig. 12 shows the percentages of windows that a thread is under suspension in all the mixes of the 4-threaded workloads with R_t = 160. The θ_U is set to 75% since this setting leads to the best improvement for the 4-threaded workloads with R_t = 160. Our analysis reveals that no single thread is being constantly suspended. All threads in the system get suspended in some windows but slower threads tend to be suspended more frequently than fast threads. The reason that no one particular thread gets suspended all the time with our approach is that a suspended thread will eventually be revived (usually within a couple of windows after it is first placed in the suspension queue) once its instructions still in the pipeline get processed and committed and thus dropping its register occupancy below the set threshold. The significant improvement in overall IPC with our proposed technique comes from a combination of a small sacrifice in the slower threads and a larger gain in the faster threads, which also explains the increase in harmonic IPC, to be shown later.

5.3. Performance comparison with other optimization techniques

In addition to the performance comparison with the default system presented in the previous section, we also compare the performance of the proposed method with other well-known fetching techniques described in the introduction section of this paper. Fig. 13 gives a complete performance comparison in overall IPC among the Round-Robin

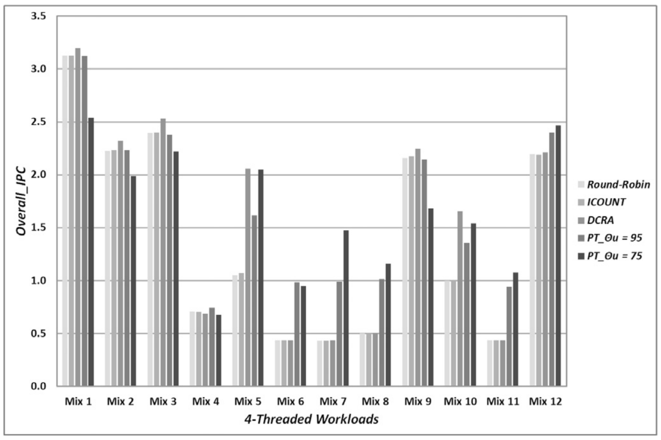

Fig. 13. Per-Mix Overall IPC on 4-threaded Workloads for Different Fetch Policies under R_t = 192.

policy, ICOUNT [5], DCRA [7] and the proposed technique (PT) with θ_U = 95% and θ_U = 75% Our technique with θ_U = 95% is significantly better than the Round-Robin policy and ICOUNT policy in 8 of the 12 mixes tested and relatively identical to the latter two in the other 4 mixes. When compared to the DCRA policy, our technique outperforms DCRA by a very significant margin (some over 100%) for most memory intensive workloads (such as mixes 5, 6, 7, 8, 10 and 11), while achieving very similar results in other mixes. In a nutshell, our technique is highly effective on mixes with slow and memory-intensive threads due to its suspension mechanism. This benefit from our technique is not as obvious in the cases where memory traffic is not as prevalent.

Another performance comparison is carried out against other current state-of-the-art rename register optimization techniques, including the ones mentioned in the introduction section. As shown in the Fig. 14, these include the fixed register capping method in [11] and two of the thread suspension techniques described in [12] and [13]. The fixed register capping method in [11] reports an improvement in IPC of up to 45% and 33% for the 4-threaded system and the 8-threaded system respectively. This fixed capping method is unable to adapt to the real-time fluctuation in the system loads, and it tends to suffer when there is a significant variation in register requirement among the threads which is more prevalent in a system with more competing threads and a small number of rename registers. The thread suspension technique in [12] claims an improvement in IPC of up to 25% and 24% for the 4-thread system and the 8-thread system and the suspension scheme in [13] outlines an improvement of up to 54% and 59%. As described in the Section 1.2.2 all these techniques involve a higher degree of hardware

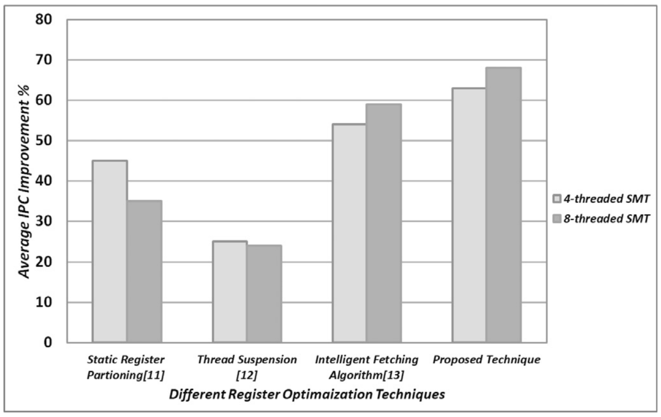

Fig. 14. Average IPC Improvement on 4-threaded and 8-threaded Workloads for the Different Register Optimization Techniques.

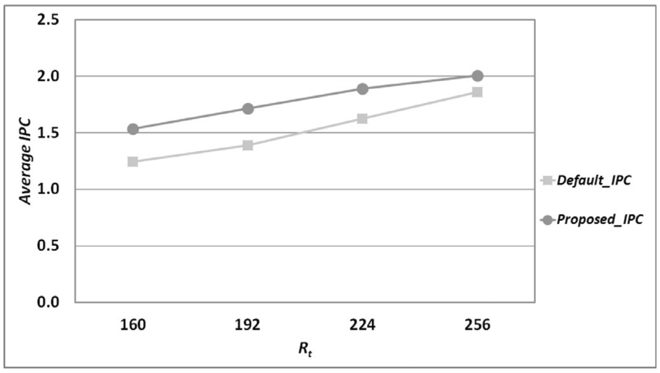

Fig. 15. Average IPC Comparison between The Proposed Technique and The Default System on 4-threaded Workloads with Various Register File Sizes.

complexity when it comes to practical implementation. The thread suspension techniques in [12] and [13] have numerous threshold values to maintain, thus requiring more high speed arithmetic units to be practical. Instead, our proposed technique not only requires a small overhead in hardware due to its per-window (vs. per-cycle) data sampling, but also outperforms all these techniques by at least 20% in the average IPC improvement.

5.4. Performance comparison with optimal set-up

Equipped with all the best parameter settings obtained from the analysis aforementioned, the proposed controlled fetching technique is then compared to the default system. Results are shown in Figs. 15 and 16 for the 4-threaded and 8-threaded workloads respectively by varying the physical register file size (R_t). The results not only show a consistent improvement in IPC from employing the proposed technique, but also indicate that the 4-threaded workloads with a smaller register file size $R_f = 160$ using the proposed technique can deliver a performance comparable to the default system with $R_f = 224$. This reflects a resource saving of 33% in the number of physical registers (R_t) and 67% in the number of rename registers R_r (32 vs. 96). Similarly the performance of the 8-threaded workloads with 320 register file entries using the proposed controlled fetching technique outperforms the default system with 416 entries, exhibiting a resource saving of 30% in the number of register file entries and 60% in the number of rename registers (64 vs. 192).

5.5. Execution fairness

Overall effectiveness 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 a significant sacrifice in execution fairness or even thread starvation.

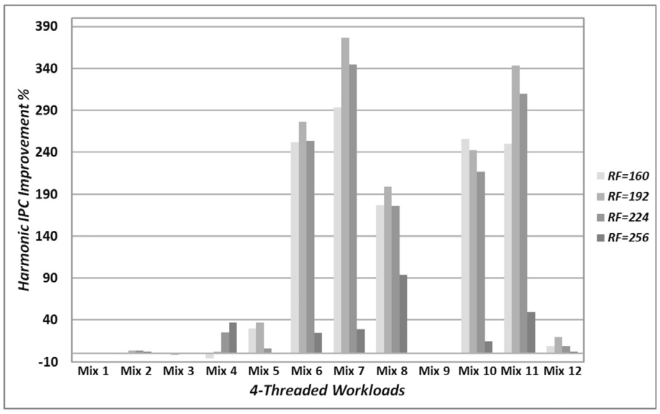

Fig. 17. Per-Mix Harmonic IPC Improvement on the 4-threaded Workloads with Various Register File Sizes.

To further investigate how execution fairness can still be maintained while achieving a significant gain in overall IPC, another analysis in performed to show how each of the mixes performs in terms of harmonic IPC. Fig. 17 shows the per-mix Harmonic IPC improvement from the proposed technique on the 4-threaded workloads under various register file sizes. The values of θ_U and θ_L are set to their respective values (90% and 30%) for optimal overall IPC improvement so as to inspect any potential loss of execution fairness. A majority of the mixes deliver a very notable improvement in harmonic IPC, with the highest being 360% by mix 7. Out of the all the mixes, there is only one mix with a small register file size that suffers a notable loss in fairness, although still to a very small degree by about 7%. As mentioned earlier, the significant improvement in overall IPC with our proposed technique comes from a combination of a small sacrifice in the slower threads and a larger gain in the faster threads, which also explains the increase in harmonic IPC. For example, under register file size of 160 for mix#5, the four individual IPC values are 0.22, 0.29, 0.21 and 0.16 from the default system, resulting an overall IPC of 0.89, while our approach obtains the individual IPC values of 0.32, 0.47, 0.18 and 0.26 with an overall IPC of 1.24, an improvement of 39%. Note that the third thread is slowed down by about 15% but all other three threads sustain a gain from 45% to 97%. With the combination of the significant gains and a small sacrifice, the harmonic IPC also increases from 0.21 to 0.27.

Figs. 18 and 19 illustrate the average improvement on Harmonic IPC (defined in Section 2.3) attained by deploying this controlled algorithm on the 4-threaded and an 8-threaded workloads respectively for different physical register file sizes.

Very much to our delighted surprise, a maximum improvement of up to 120% and 55% are obtained for the 4-threaded SMT system and

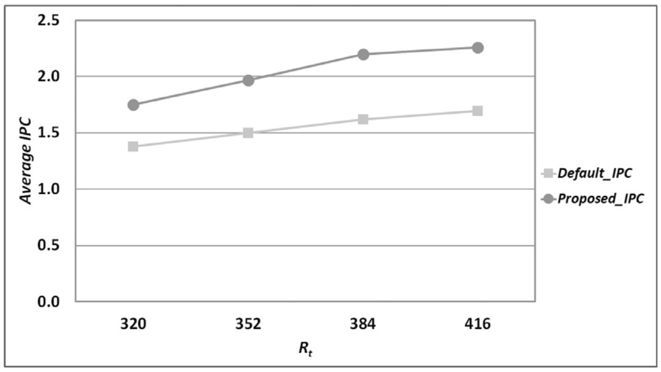

Fig. 16. Average IPC Comparison between The Proposed Technique and The Default System on 8-threaded Workloads with Various Register File Sizes.

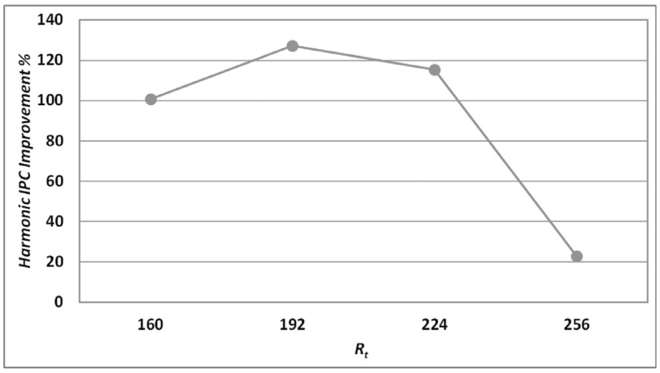

Fig. 18. Average Harmonic IPC Improvement on the 4-threaded Workloads with Various Register File Sizes.

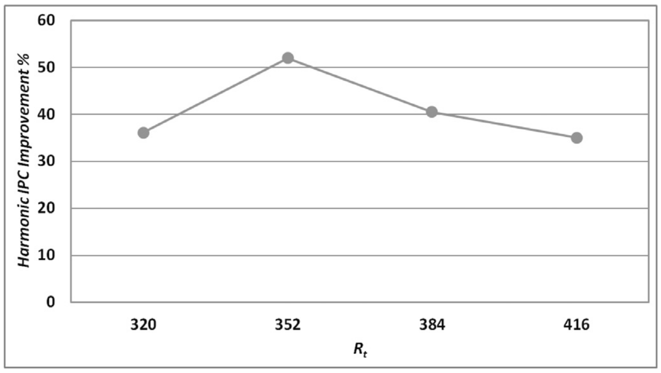

Fig. 19. Average Harmonic IPC Improvement on the 8-threaded Workloads with Various Register File Sizes.

the 8-threaded SMT system respectively. This shows that the proposed controlled fetching algorithm achieves performance improvement without any unfair execution or thread starvation.

6. Conclusion

This paper proposed a simple stand-alone resource management technique which works at the architectural level and is also easily feasible in terms of practical implementation. Moreover this controlled fetching method can be modified to manage the other shared resources directly – it can be similarly employed to manage issue queue or write buffer or any other resources by monitoring their respective utilization numbers. This algorithm neither needs much extra hardware support nor requires any modifications in the pipeline stages. We believe that this algorithm can be implemented alongside with other advanced techniques in different pipeline stages to extract a greater gain in the performance. An obvious future direction along this line of research is to develop an adaptive threshold system to better dynamically adjust to various combinations of threads in real time.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.micpro.2018.01.001.

References

[1] H. Hirate, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, T. Nishizawa, An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads, Proceedings of the 19th Annual International Symposium on Computer Architecture, (1992), pp. 136–145.
[2] D. Tullsen, S.J. Eggers, H.M. Levy, Simultaneous multithreading: maximizing on-chip parallelism, Proceedings of the 22nd Annual International Symposium on Computer Architecture, (1995), pp. 392–403.
[3] D. Kanter, Intels Sandy Bridge Microarchitecture, Real World Technologies Website, 2010. http://www.realworldtech.com/sandy-bridge/ .
[4] C. Angelini, AMDs bulldozer and bobcat architecture pave the way, 2010. http://www.intel.com.
[5] 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 multi-threading processor, Proceedings of the 23rd Annual International Symposium on Computer Architecture, (1996), pp. 191–202.
[6] D.M. Tullsen, J.A. Brown, Handling long-latency loads in a simultaneous multi-threading processor, Proceedings of the 34th International Symposium on Microarchitecture, (2001), pp. 318–327.
[7] F.J. Cazorla, A. Ramirez, M. Valero, E. Fernandez, Dynamically controlled resource allocation in SMT processors, Proceedings of the 37th International Symposium on Microarchitecture, (2004), pp. 171–182.
[8] J.L. Lo, S.S. Parekh, S.J. Eggers, H.M. Levy, D.M. Tullsen, Software-directed register deallocation for simultaneous multi-threading processors, IEEE Trans. Parallel Distributed Syst. 10 (9) (1999) 922–933.
[9] E. Quinones, J. Parcerisa, A. Gonzalez, Leveraging register windows to reduce physical registers to the bare minimum, IEEE Trans. Comput. 59 (12) (2010).
[10] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, V. Vinals, Dynamic register renaming through virtual-physical registers, J. Instruct. Parallelism 2 (2000).
[11] Y. Zhang, 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).
[12] W. Wang, 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.
[13] M.K. Ramanathan, W.-M. Lin, An intelligent fetching algorithm for efficient physical register file allocation in simultaneous multi-threading CPUs, Int. J. Comput. Syst. (IJCS) 4 (4) (2017) 78–85.
[14] 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.
[15] Standard performance evaluation corporation (SPEC) website, http://www.spec.org/.

Madhava Krishnan Ramanathan pursued his masters in computer engineering at The University of Texas at San Antonio. His research interests include Computer Architecture, High-Performance Computing and Mobile Computing. He recently graduated his college and is pursuing Ph.D. in Computer Engineering. Madhava has received a lot of honors and scholarships for his excellence in academics and research from the Department of Electrical and Computer Engineering at UTSA.

Dr. Wei-Ming Lin is currently a professor of Electrical and Computer Engineering at the University of Texas at San Antonio (UTSA). He received his Ph.D. degree in Electrical Engineering from the University of Southern California, Los Angeles, in 1991. Dr. Lin's research interests are in the areas of distributed and parallel computing, computer architecture, computer networks, autonomous control and internet security. He has published over 100 technical papers in various conference proceedings and journals. Dr. Lin's past and on-going research has been supported by NSF, DOD, ONR, AFOSR, etc.