Dynamic Capping of Physical Register Files in Simultaneous Multi-threading Processors for Performance

Hasancan Güngörer(✉) and Gürhan Küçük

Department of Computer Engineering, Yeditepe University, Istanbul, Turkey
hasancangungorer@gmail.com, gkucuk@cse.yeditepe.edu.tr

Abstract. Today, Simultaneous Multi-Threading (SMT) processors allow sharing of many datapath elements among applications. This type of resource sharing helps keeping the area requirement of a SMT processor at a very modest size. However, a major performance problem arises due to resource conflicts when multiple threads race for the same shared resource. In an earlier study, the authors propose capping of a shared resource, Physical Register File (PRF), for improving processor performance by giving less PRF entries, and, hence, spending less power, as well. For the sake of simplicity, the authors propose a fix PRF-capping amount, which they claim to be sufficient for all workload combinations. However, we show that a fix PRF-capping strategy may not always give the optimum performance, since any thread's behavior may change at any time during execution. In this study, we extend that earlier work with an adaptive PRF-capping mechanism, which tracks down the behavior of all running threads and move the cap value to a near-optimal position by the help of a hill-climbing algorithm. As a result, we show that we can achieve up to 21% performance improvement over the fix capping method, giving 7.2% better performance, on the average, in a 4-threaded system.

Keywords: Processor performance · Shared resource management
Simultaneous Multi-Threading

1 Introduction

Simultaneous Multi-Threading (SMT) processors are merely superscalar processors, which allow simultaneous execution of instructions coming from multiple threads. Compared to single-thread superscalar processors, they promise better throughput by allowing more efficient utilization of available datapath resources. To keep the design cost at its lowest level, many of the resources are shared among running threads. For instance, Physical Register Files (PRFs) are not replicated but are enlarged to compensate renaming requests coming from multiple threads. Here, one faulty assumption is that threads show similar behavior and do not harm each other. However, shared resources give SMT processors a major disadvantage over their superscalar counterparts: resource conflicts. These resources can be claimed by any of the running threads at any given time, and, unfortunately, the assumption about the behavior of different threads being similar is almost always wrong. Some of the threads are resource hungry

T. Czachórski et al. (Eds.): ISCIS 2018, CCIS 935, pp. 41–48, 2018.
https://doi.org/10.1007/978-3-030-00840-6_5

but do not commit too many instructions. We can classify such threads as harmful, since they can easily clog one or many shared datapath resources up, and, as a result, they can reduce the overall system throughput, almost instantly. In contrast, some of the threads may request just enough resources to keep their throughput at a reasonable pace. Such threads can be classified as genuine threads, since they become the reason for achieving high overall system throughput. These threads do not ask for any unnecessary amount of resources. Finally, there is also a third class of threads, which hardly asks for any type of resources. We can classify them as harmless threads, since they have low instruction level parallelism resulting in no claim for any resources.

There is considerable amount of work involved in this research area. On reducing the pressure on PRF, Lo et al. apply compiler and operating system extensions to release PRF entries as soon as they are detected to be useless [1]. For instance, all PRF entries allocated for an idle thread might be immediately released. The authors also propose techniques to release PRF entries immediately after their last use. In another study, Monreal et al. propose virtual physical registers that do not require any storage bindings [2]. With the help of virtual physical registers PRF allocation time can be delayed, again reducing the pressure on the PRF. Here, in our proposed method, we also reduce pressure on the PRF by dynamically capping it according to runtime behavior of threads.

There is also a variety of resource partitioning techniques that target better resource utilization [3]. Choi et al. propose a hill-climbing based method, which runs in periods of time called epochs. At each epoch, one of the threads is given more resources than its actual share, and the overall performance is recorded. After testing each thread's performance with extra resources, a greedy-type decision algorithm selects the thread with the best performance and increases its allocated resource size. In another study, Wang et al. focus on a metric, which they call Committed Instructions Per Resource Entry (CIPRE) [4]. Again, the execution time is divided into epochs, and the CIPRE value of all running threads are calculated at the end of each epoch. The thread with the highest CIPRE value is selected to receive a resource increase, since it proves that it can give the best throughput per allocated resource entry. Finally, Cazorla et al. propose the DCRA mechanism, which is an acronym for Dynamically Controlled Resource Allocation [5]. Here, threads are classified as slow/fast and active/inactive, and slow threads receive more resources than fast threads can have. Active and slow threads receive the largest share of the resources. All these techniques require a complex hardware for collecting hardware statistics and partitioning multiple datapath resources.

In an earlier study, the authors propose a PRF-capping mechanism that allocates a small amount of PRF entries for each thread [6]. They claim that, with this simple mechanism, they can regulate the Issue Queue (IQ), the Re-Order Buffer (ROB) and the Load/Store Queue (LSQ). They also show that a fixed cap size is enough to satisfy any type of workloads, and they report up to more than 44% IPC improvements for a 4-threaded SMT system.

In this study, we show that a fixed cap size does not really gives optimal performance in all workload combinations as stated in the earlier work. As shown in Fig. 1, there are a variety of cap sizes that give near-optimal IPC values for different workloads. Here, benchmarks from the SPEC CPU 2006 suite are combined to create a set of 4-threaded workloads mixtures (see Table 2). When we examine the figure, we see

that PRF cap size of 8 is the best for Mix 1, whereas the same cap size gives the worst performance for Mix 2 and Mix 5. We also see that Mix 10 is almost insensitive to cap size. In this study, our main motivation is to dynamically pinpoint a near-optimal cap size and use it. Our results show that our proposed method tracks down the optimal cap size for each of these workloads and achieves up to 21% better performance compared to a fixed cap size approach as proposed in [6].

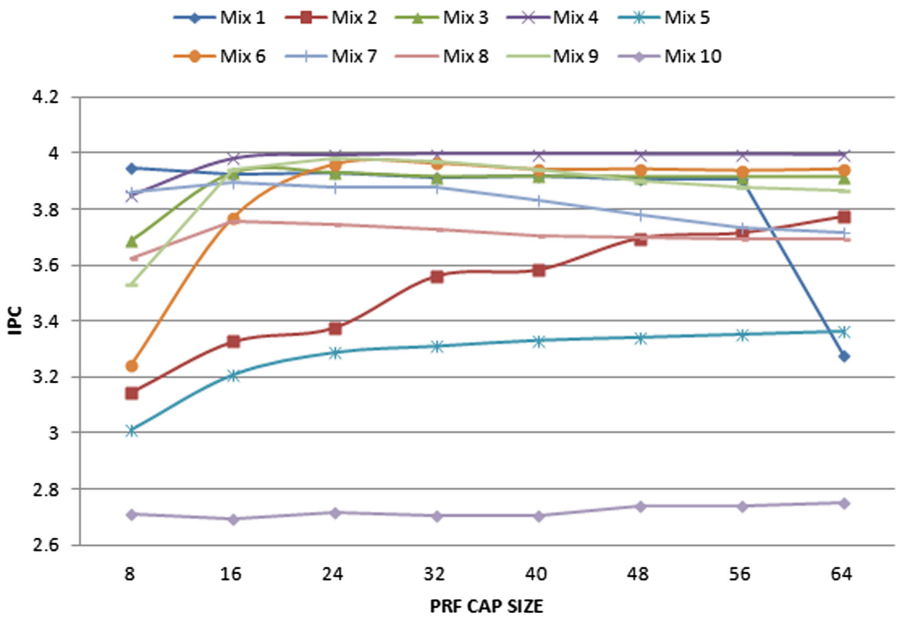

Fig. 1. Effect of various cap sizes on SMT performance

2 The Proposed Method

Physical Register Files (PRFs) are one of the most critical resources to achieve high throughput in SMT processors. At the instruction dispatch stage, each decoded instruction with a destination register is given a PRF entry to eliminate false data dependencies among instructions. This is known as the register renaming process. When the PRF becomes full, the frontend of the pipeline is stalled until some instructions in the backend commits and releases precious PRF entries. In a single-thread superscalar processor, PRF should be kept large enough to support multiple names for each architectural register, and the size of the PRF might be the sole design problem that we can face. In a SMT processor, though, there are other problems originating from the nature of the processor. Instructions from multiple threads race on the same datapath and allocate PRF entries. A harmful thread, which holds too many resources but executes too few instructions, may easily render the whole system into a thrashing state.

In an earlier study, the authors propose the fixed capping of the PRF after a detailed analysis. But, a fixed cap size becomes a problem when a genuine thread looks for extra resources for improving its throughput or when a harmless thread does not make use of any allocated resource to itself. We believe that such inefficiencies can be easily avoided with a mechanism that can move the CAP size to an appropriate position when it is needed. In our proposed design, we set the lower and the upper bound for the CAP size to 8 and 64, respectively. In our experiments, we also selected a delta value of 8 that defines the atomic size of PRF entries a thread can receive or lose. Then, we apply a simple hill-climbing algorithm on the IPC curve for various CAP sizes. Our main goal is to find the CAP size that gives us the near-optimal performance. To achieve this, we define a time period in 1 M-cycle granularity, and we check the performance trend in terms of instructions per cycle (IPC) at the end of each period. If the IPC for the current period becomes higher than the IPC for the previous period within a certain threshold (0.01 in our tests), we conclude that we can apply hill-climbing and move to a consequent CAP size. Otherwise, if the current IPC becomes smaller than the previous one, than we wait at that CAP size for a grace period (5 periods, which is chosen empirically, in our current design) assuming that the current CAP size and its corresponding IPC reside at the top of the hill, which we are already climbing. After that grace period, all the performance geography might be changed since all threads can change their behavior at any instant of time, and, then, we restart the hill-climbing algorithm assuming that there are new performance peaks around the current CAP size.

Algorithm 1. Pseudo Code of Hill-Climbing

```
Initialize: CAP ← 40, wait ← 0, delta ← 8
 1: diffIPC ← currIPC – prevIPC
 2: delta ← (wait = 0 and (CAP = 8 or CAP = 64) ? -delta : delta)
 3: if wait = 0 then
 4:    if firstTime = true then
 5:       prevIPC ← currIPC
 6:       CAP ← CAP + delta
 7:       firstTime ← false
 8:    elseif diffIPC > 0.01 then
 9:       prevIPC ← currIPC
10:       CAP ← CAP + delta
11:    else
12:       wait ← 5
13:    end if
14: else
15:    wait ← wait - 1
16: end if
```

Algorithm 1 given above shows the details of our hill-climber. The most important feature of the algorithm is its bouncing delta value. It is initially, set to +8, which enables us to search cap sizes in one direction starting from 8 up to 64. But, when the algorithm reaches the CAP size of 64, the delta value becomes -8 to enable a hill-climb process in the reverse direction.

3 Experimental Methodology

In this paper, we use M-sim simulator to run SPEC CPU2006 benchmarks [7]. Configuration details of the simulated processor are given in Table 1. Our hill-climbing algorithm is run every one million cycles. We also tested other periods but one million cycle period gives the best results. We fast-forwarded each benchmark for 100 million cycles and run cycle-accurate simulations for 200 million cycles. The workload mixtures that we use in our simulations are given in Table 2.

Table 1. Configuration of the simulated processor

Parameter	Configuration
Machine width	4-wide fetch/dispatch/issue/commit
L/S Queue size	48 Load/Store queue
ROB & IQ size	128 entry ROB, 32-entry IQ
L1 I-cache	64 KB, 2-way set-associative 64-byte line
L1 D-cache	64 KB, 4-way set-associative 64-byte line, write-back, 1-cycle access latency
L2 Cache unified	512 KB, 16-way set-associative 64-byte line, write-back, 10-cycle access latency
BTB	512 entry, 4-way set-associative
Branch predictor	Bimod: 2 K entry
Memory	32-bit wide, 300 cycles access latency

Table 2. 4-threaded workloads

Workloads	Benchmarks
Mix 1	libquantum, dealII, gromacs, namd
Mix 2	hmmer, sjeng, gobmk, gcc
Mix 3	libquantum, dealII, gobmk, gcc
Mix 4	gromacs, dealII, lbm, cactusADM
Mix 5	hmmer, sjeng, lbm, bzip2
Mix 6	libquantum, sjeng, lbm, gcc
Mix 7	namd, dealII, hmmer, milc
Mix 8	namd, gcc, lbm, milc
Mix 9	gobmk, gromacs, cactusADM, bzip2
Mix 10	sjeng, namd, cactusADM, hmmer

4 Tests and Results

In our tests, we test the adaptive nature of our algorithm, first. Figure 2 shows the percentage of time each cap size is visited across all workload mixtures in an elevation-map form. From the figure, we learn that cap size of 32 and above are quite popular among workloads. We also see that there is a variety of cap sizes utilized in each workload proving the dynamic tracking ability of the algorithm.

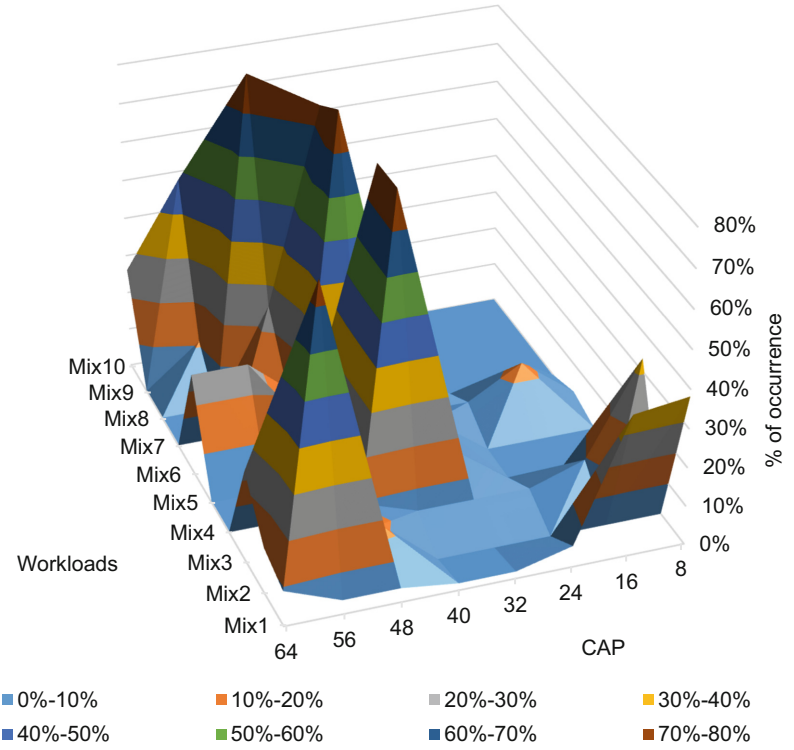

Fig. 2. 3D Histogram showing number of visits of each cap size across simulated workload mixtures

We also studied how well the algorithm tracks the actual peak performance of the workloads given in Fig. 1. Figure 3 shows the percentage of time our algorithm dynamically selects the cap sizes corresponding to within 1%, 2%, 3% and 5% tolerance range of the peak performance value. For instance, from the figure we see that our hill-climber successfully locates 55% of the cap sizes within 1% tolerance range of the peak performance, on the average. But, when the tolerance range is within 2%, the algorithm sweeps more than 90% of the near-optimal cap sizes, and we believe that this is a great success for pinpointing peak performance points of workloads at runtime. Tolerance ranges of 3% and 5% have 94.2% and 95.5% coverage of performance at peak cap sizes, respectively. From the figure, we also see that our algorithm is not perfect for all type of workloads. For instance, when we observe Mix 3 from Fig. 1, we see that it shows a very low performance for cap size of 8. However, our adaptive algorithm spends its 33% of time at this cap size, and, therefore, its coverage is constant at 67% for all tolerance ranges that we study.

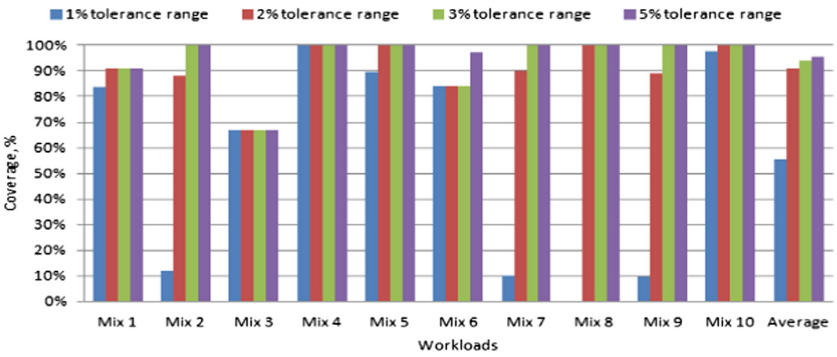

Fig. 3. Coverage percentage of cap sizes with peak performance points for various tolerance ranges

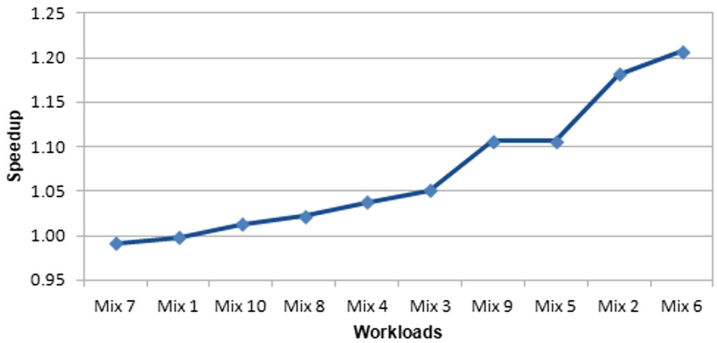

Fig. 4. Speedup of our proposed algorithm compared to the original algorithm proposed in [6].

In Fig. 4, we report the speedup of our hill-climber compared to the fix cap size algorithm suggested in [6]. Here, we see that Mix 7 shows a worse performance compared to the original algorithm. When we observe Fig. 1, we see that Mix 7 has a low sensitivity to PRF cap size. But, it still shows slightly better performance values for cap sizes less than 32. Unfortunately, as we examine Fig. 2, we find that our algorithm spends most of its time on the opposite range (i.e. cap sizes 32 to and up) assuming that it is already close to the peak of the performance curve. Therefore, its coverage percentage becomes quite high (90%) even with a tolerance range of 2%. However, the coverage result (10%) with the tolerance range of 1% tells us the real story. Mix 7 fools our algorithm and its performance becomes worse than the original fixed cap size approach. But, the good news is we successfully track the actual peak performance points in the rest of the workloads, and achieve 7.2% better performance, on the average. Our best results are observed with Mix 2 (18%) and Mix 6 (21%), which are two genuine class workloads always hungry for resources.

5 Conclusion

We show that various workload mixtures have changing needs on physical register file use, and application of a fixed cap size on this precious resource has performance problems. In this study, we propose a hill-climbing algorithm, which climbs the performance curve and locates the near-optimal cap size at runtime. The performance geography dynamically changes with the behavior changes of each running thread, and this makes our strategy a challenging task. Our test results show that we can successfully track down the moving peak performance in the performance curve, and achieve up to 21% speedup relative to a fixed capping scheme proposed in an earlier study. We also show that our proposed algorithm is not perfect. We plan to work on individual cap sizes that are tailored to the needs of each running thread, in a future study. We believe that this new strategy may solve the problems that we encountered and left unsolved in this study.

Acknowledgments. This work is supported by the Scientific and Technical Research Council of Turkey (TUBITAK) under Grant No:117E866.

References

1. Lo, J.L., Parekh, S.S., Eggers, S.J., Levy, H.M., Tullsen, D.M.: Software-directed register deallocation for simultaneous multithreaded processors. IEEE Trans. Parallel Distrib. Syst. **10** (9), 922–933 (1999)
2. Monreal, T., González, A., Valero, M., González, J., Viñals, V.: Dynamic register renaming through virtual-physical registers. J. Instr. Level Parallelism **2**, 1–20 (2000)
3. Choi, S., Yeung, D.: Learning-based SMT processor resource distribution via hill-climbing. In: ACM SIGARCH Computer Architecture News, vol. 34, no. 2, pp. 239–251. IEEE Computer Society (2006)
4. Wang, H., Koren, I., Krishna, C.M.: An adaptive resource partitioning algorithm for SMT processors. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 230–239. ACM, Oct 2008
5. Cazorla, F.J., Ramirez, A., Valero, M., Fernandez, E.: Dynamically controlled resource allocation in SMT processors. In: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 171–182. IEEE Computer Society (2004)
6. Zhang, Y., Lin, W.M.: Efficient resource sharing algorithm for physical register file in simultaneous multi-threading processors. Microprocess. Microsyst. **45**, 270–282 (2016)
7. Sharkey, J., Ponomarev, D., Ghose, K.: M-Sim: a flexible, multithreaded architectural simulation environment. Technical Report CS-TR-05-DP01, Department of CS, SUNY-Binghamton (2005)