Introducing Thread Criticality Awareness in Prefetcher Aggressiveness Control

Biswabandan Panda, Shankar Balachandran
Dept. of Computer Science and Engineering
Indian Institute of Technology Madras, Chennai
Email: {biswa,shankar}@cse.iitm.ac.in

Abstract*—A single parallel application running on a multi-core system shows sub-linear speedup because of slow progress of one or more threads known as critical threads. Some of the reasons for the slow progress of threads are (1) load imbalance, (2) frequent cache misses and (3) effect of synchronization primitives. Identifying critical threads and minimizing their cache miss latencies can improve the overall execution time of a program. One way to hide and tolerate the cache misses is through hardware prefetching. Hardware prefetching is one of the most commonly used memory latency hiding techniques. Previous studies have shown the effectiveness of hardware prefetchers for multiprogrammed workloads (multiple sequential applications running independently on different cores). In contrast to multiprogrammed workloads, the performance of a single parallel application depends on the progress of slow progress(critical) threads. This paper introduces a prefetcher aggressiveness control mechanism called Thread Criticality-aware Prefetcher Aggressiveness Control (TCPAC). TCPAC controls the aggressiveness of the prefetchers at the L2 prefetching controllers (known as TCPAC-P), DRAM controller (known as TCPAC-D) and at the Last Level Cache (LLC) controller (known as TCPAC-C) using prefetch accuracy and thread progress. Each TCPAC sub-technique outperforms the respective state-of-the-art techniques such as HPAC [2], PADC [4], and PACMan [3] and the combination of all the TCPAC sub-techniques named as TCPAC-PDC outperforms the combination of HPAC, PADC, and PACMan. On an average, on a 8 core system, in terms of improvement in execution time, TCPAC-PDC outperforms the combination of HPAC, PADC, and PACMan by 7.61%. For 12 and 16 cores, TCPAC-PDC beats the state-of-the-art combinations by 7.21% and 8.32% respectively.

I. Introduction

Multi-core systems employ hardware prefetchers at different levels of cache hierarchy. These prefetchers predict the future cache accesses based on the past cache access patterns. Hardware prefetchers such as stride and stream are commonly used in multi-core systems. The effectiveness of a hardware prefetcher depends on how accurately it predicts the future access patterns (quantified by prefetch accuracy) [1] and how timely these prefetch requests are. Prefetcher aggressiveness control techniques such as HPAC [2] and Bandwidth Efficient Prefetcher [6] have been proposed for multiprogrammed workloads. Also, techniques such as PACMan [3], PADC [4] have been proposed for prefetch awareness at the shared Last Level Cache(LLC) and shared DRAM controller respectively. Ebrahimi et al. proposed a technique [5] that achieves fairness and QOS in the presence of prefetchers. Recently, Ishii et al. proposed an architecture called Unified Memory Optimizing (UMO) [13] architecture, that changes the prefetch issue policies by making it DRAM aware. Their idea is based on memory access map based prefetching [12]. To the best of our knowledge, effectiveness of these techniques are not evaluated for a single parallel application running on a multi-core system. A single parallel application running on a multi-core system behaves differently. In parallel applications, multiple threads of a single application share and communicate data among themselves. In contrast to multiprogrammed workloads, the performance metric used for a parallel application is its execution time. Parallel applications use different synchronization primitives such as locks and barriers. Also, issues such as workload-imbalance and uneven LLC misses among the threads cause some of the threads (known as critical threads) to progress slowly that leads to increase in the overall execution time. Figure 1 shows the fraction of execution time affected [2] by slow (critical) threads. On an average, for SPLASH-2X [18] and PARSEC [14] benchmarks (run with 8 threads), more than 40% of the total execution time is affected by the critical threads.

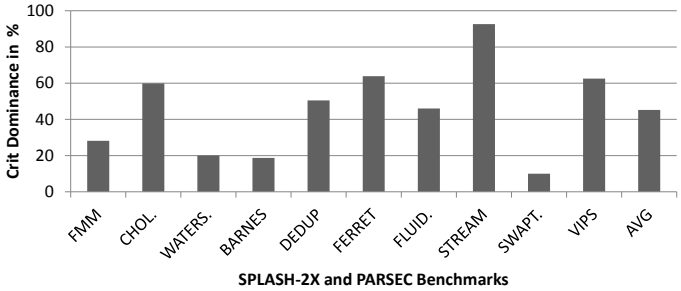

Fig. 1. **Dominance of Critical Threads on 8 core system.**

This paper proposes a prefetcher aggressiveness control framework called TCPAC. TCPAC consists of three sub-techniques named TCPAC-P, TCPAC-D and TCPAC-C that controls the aggressiveness of prefetchers at L2 prefetching controllers, DRAM controller and at the LLC controller respectively. Aggressiveness of prefetchers can be controlled at the producer (prefetching controller) by changing the prefetch degree and the prefetch distance and at the consumers (DRAM and LLC) by prioritizing/de-prioritizing the prefetch requests and responses. Overall TCPAC is a holistic design and its main

[1] $$PrefetchAccuracy = \frac{\#DemandHitstoPrefetchedCacheLines}{\#PrefetchesIssued}$$

[2] Progress of critical threads affect progress of the entire application.

978-3-9815370-2-4/DATE14/ ©2014 EDAA

contributions are:

- TCPAC-P: TCPAC-P controls the aggressiveness of the prefetchers by changing the prefetch degree and the prefetch distance. TCPAC-P is based on a throttling mechanism where an increase (decrease) in the prefetch degree and the prefetch distance corresponds to throttling up (down).
- TCPAC-D: TCPAC-D controls the aggressiveness of the prefetch requests by changing the DRAM scheduling policy. It assigns priorities to the DRAM requests based on prefetch accuracy, thread progress and the criticality of the thread.
- TCPAC-C: TCPAC-C controls the aggressiveness of the prefetch requests/responses by changing the LLC replacement policy. The replacement policy is based on the prefetch accuracy and the criticality of the thread.

To the best of our knowledge, for parallel applications, this is the first work that handles the prefetch requests/responses at different levels of memory hierarchy.

II. Basic Definitions and Motivation

This section defines some metrics we use in our study and describes the motivation behind introducing TCPAC. Sections IV, V and VI motivate and propose TCPAC-P, TCPAC-D and TCPAC-C respectively. The motivation behind introducing TCPAC is the absolute difference in the progress of the threads. Some of the reasons for this behavior are: First, workload imbalance and high LLC misses cause a particular thread to lag behind all other threads. Second, in case of pipelined parallel programs, if a particular pipeline stage is slow as compared to other stages then the threads associated with that particular stage becomes slow. Techniques such as Criticality Stacks [9], and Thread Criticality Predictor (TCP) [1] have been proposed to identify threads responsible for the slow progress of a parallel application. In this work, we use an extended version of TCP. To identify the critical threads, TCP uses cache misses and the corresponding cache miss latencies involved.

A. Definitions

In this section, we define criticality of a thread and introduce some new metrics that we use in this work. We use TCP [1], that correlates criticality of a thread to its cache statistics (hits and misses). TCP accounts for L1 and L2 cache misses and the miss penalty associated with it. A thread is more critical if it experiences more costly cache misses. We extend TCP for a three level cache hierarchy. Thread Criticality (TC) of thread i is defined as

$$TC(i) \quad = \quad L2_{hits}(i) + \frac{R_i + L_i + LL_i}{L1Penalty(i)} \tag{1}$$

where,

$$\begin{aligned} R_i &= Remote_{hits}(i) * RemotePenalty(i) \\ L_i &= L3_{hits}(i) * L1L2Penalty(i) \\ LL_i &= DRAM_{hits}(i) * L1L2L3Penalty(i) \end{aligned} \tag{2}$$

$L2_{hits}(i)$ is the # L1 misses that hit in the local L2. $L3_{hits}(i)$ is the # L1 misses that also miss in L2 and hit in the shared L3. $\text{Remote}_{hits}(i)$ is the # L1 or L2 misses that hit in the L1 or L2 of the remote cores. $\text{DRAM}_{hits}(i)$ is the # L1 misses that miss in L2 and L3 and hit in the DRAM. We also use a metric called Normalized Thread Criticality (NTC). NTC of thread i is defined as

$$NTC(i) = TC(i)/LS(i) \tag{3}$$

where, LS(i) is the # committed Load and Store instructions of thread i. We use LS(i) to minimize the variability in the # committed Load and Store instructions between two consecutive epochs. NTC helps in distinguishing threads that suffer more in terms of miss penalty. Next, we define a metric called Slack which tracks the difference in TC among the threads. For a given thread i,

$$Slack(i) \quad = \quad max(TC(j))_{\forall j=0 \ to \ n-1} - TC(i) \tag{4}$$

where n is the total # threads (0 to $n-1$). The thread with slack=0 is treated as the most critical thread. Slack helps in finding the most critical thread but in case of parallel applications, at a given epoch, multiple threads may become critical. To identify multiple critical threads if any, we use an algorithm that detects outliers [17] based on the slack values. The algorithm sorts the slack values in an ascending order. Then it finds the inner and outer fences based on the slack values. The outer fence provides a range in the form of [a,b]. Threads with slack value less than a are known as critical threads. If the # critical threads are more than $1/4$th of total # threads then we assume all the threads are non-critical. This scenario happens when the difference between the slacks is marginal. The objective of TCPAC is to speedup the critical threads which are prefetch friendly. To check whether a thread i is prefetch friendly or not we define a metric called Thread Progress with Prefetchers (TPP). TPP of a given thread i at a given epoch j is defined as

$$TPP_{ij} = \frac{NTC(i)_{j-1} - NTC(i)_j}{degree(i)_j - degree(i)_{j-1}} \tag{5}$$

Thread i progresses-es at the end of epoch j, if
(i) the denominator is 0 and the numerator is positive or
(ii) the denominator is $\neq 0$ and TPP_{ij} is non-negative. TPP_{ij} becomes non-negative in two cases.
Case1: Increase in degree(i)_j compared with degree(i)_{j-1} causes decrease in NTC(i)_j compared with NTC(i)_{j-1}.
Case2: Decrease in degree(i)_j compared with degree(i)_{j-1} causes increase in the NTC(i)_j compared with NTC(i)_{j-1}.

B. Thread Criticality and Hardware Prefetching

Hardware prefetchers play an important role in improving the execution time of a program by reducing and hiding the cache miss latencies. To further improve the execution time, intelligent prefetcher aggressiveness control policies have been proposed that changes the prefetch degree and the prefetch distance dynamically. On top of it, efficient LLC management techniques and DRAM scheduling techniques provide special care to the prefetch requests/responses at the LLC controller and DRAM controller respectively. These techniques further improve the execution time. In the literature, techniques such as HPAC [2], PADC [4], and PACMan [3] have been proposed for prefetching controllers, DRAM controller and LLC controller respectively. HPAC uses prefetch metrics such as prefetch accuracy, cache pollution and bandwidth consumption to control the aggressiveness of a prefetcher. Similarly, PADC

prioritizes the prefetch requests using prefetch accuracy at the DRAM controller, but in case of parallel applications, the accuracy of the prefetchers remain similar across non-critical threads as each thread uses shared data structures during the parallel phase. We find this trend in 8 out of the 10 applications from SPLASH-2X and PARSEC that we use in our study. Applications such as `ferret` and `waterspatial` do not fall in this trend. The above mentioned techniques are oblivious to the criticality of a thread and usually give more shared resources and prioritize threads with high prefetch accuracy that are progressing. Some of the examples from the literature are:

(i) Prefetcher aggressiveness control techniques such as HPAC [2] and Bandwidth Efficient Prefetcher [6] prioritize threads that have high prefetch accuracy, high prefetch coverage and less cache pollution.

(ii) DRAM scheduling policy such as TCM [11] prioritizes applications based on memory access behavior [3]. PADC [4] introduces prefetch awareness at the DRAM controller by prioritizing the prefetch requests based on the prefetch accuracy.

(iii) Shared LLC replacement policy such as TA-DRRIP [10] keeps the cache lines of the threads that are cache-friendly and cache-fitting for more time.

(iv) Similarly, PACMan [3], a technique that ensures prefetch awareness at the LLC, mostly de prioritizes prefetched cache lines (keeps them for small fraction of time at the LLC) by changing the LLC replacement policy.

For these reasons, recently, techniques such as CSHARP [8] and PAMS [16] have been proposed specifically for parallel applications. Applying the above state-of-the-art techniques meant for multiprogrammed workloads directly to a single parallel application may be effective but biasing them towards the critical threads may be more effective in improving the execution time. As Figure 1 shows the dominance of critical threads, we find there is a need for introducing thread criticality to the state-of-the-art techniques. We do this by identifying and allocating more shared resources to the critical threads and prioritizing the requests/responses of critical threads.

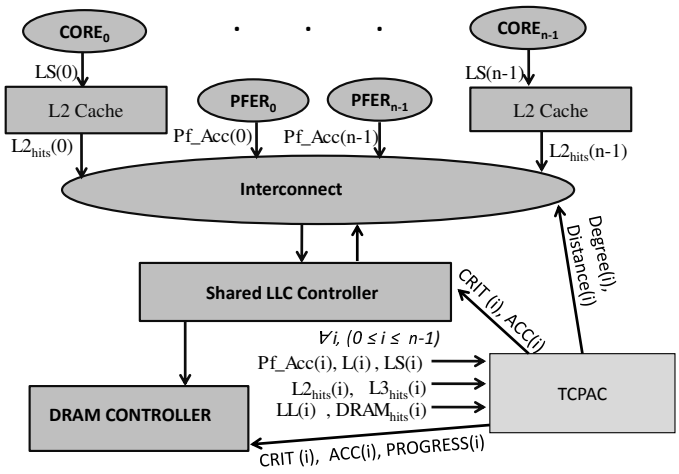

Fig. 2. **Organization of TCPAC (Private L1 caches are not shown.)**

III. TCPAC

We use TPP(i), Slack(i) and Pf_{Acc}(i) (prefetch accuracy) of a thread i to control the aggressiveness. TCPAC collects and uses the above metrics at a regular epoch (epoch length of $100K$ cycles) and resets it to zero after making a decision. Each thread maintains a set of counters to calculate the relevant metrics across two consecutive epochs. Algorithm 1 explains the basic flow of TCPAC. The first part of the algorithm(line number 2 to 7) detects change in the program phase. It is based on the MPKI[4] of a particular thread i at the LLC. In the line number 12, TCPAC tries to identify the critical threads and it counts them. If the if condition is satisfied then TCPAC policies kick in, else TCPAC uses the state-of-the-art policies.

Algorithm 1 TCPAC

1: **for all** i, where i is the the thread id, at a given epoch j **do**
2: new_phase=0 // variable used to detect changes in the program phase at the end of every epoch.
3: **if** ($\mid avg\_MPKI(i) - MPKI(i)_j \mid \geq 0.03$) **then**
4: new_phase=1
5: **else**
6: new_phase=0
7: **end if**
8: compute $Slack(i)$, TPP_{ij}
9: **end for**
10: sort the Slack(i)s' in the ascending order
11: Find the thread ids' which are outliers [17] and count them
12: **if** ((# outliers $\leq$ $1/4$th of total # threads) and (new_phase==0)) **then**
13: Use TCPAC-P at the prefetching controllers, TCPAC-D at the DRAM controller and TCPAC-C at the LLC controller
14: **else**
15: Use HPAC at the prefetching controllers, PADC at the DRAM controller and PACMan at the LLC controller
16: **end if**

Organization of TCPAC: We incorporate TCPAC on multi-core systems with a three level cache hierarchy and hardware prefetchers enabled at the private L2 caches. Figure 2 shows the organization of TCPAC which is placed beside LLC with an access latency equivalent to the LLC latency. At regular epochs, TCPAC calculates various metrics, finds out the critical threads and their progress. Then it responds to the L2 prefetching controllers, the DRAM controller and the LLC controller. TCPAC calculates metrics such as progress (PROGRESS), criticality (CRIT) and uses Pf_{Acc} (ACC) of the threads. In the interconnect, bus, and in the entries of DRAM Request Buffer (MRB), TCPAC augments the requests/responses with extra bits for the above metrics. At a given epoch j, for a thread i, PROGRESS=1, if thread i progresses as per the conditions mentioned in section II-A, else PROGRESS=0. Similarly CRIT=1, if a particular thread is identified as critical. ACC=1, if the $\text{Pf}_{Acc} \geq 0.55$. TCPAC collects the above metrics just before the end of an epoch and makes decisions by the end of an epoch. Please note, all these calculations are not in the critical path of program in execution.

IV. TCPAC for Prefetchers - (TCPAC-P)

TCPAC-P controls the aggressiveness of L2 prefetchers (at the prefetching controllers) based on the prefetch metrics

[3]Latency sensitive or bandwidth sensitive.

[4]Misses Per Kilo Instructions.

TABLE I. **Decisions based on TCPAC-P**

CASE	CRIT	ACC	PROGRESS	THROTTLING	RATIONALE
1	1	1	1	Degree $+ = 2$	Minimize criticality
2	1	1	0	Degree $- = 2$	Minimize criticality
3	1	0	1	No Change	Prefetcher Insensitive
4	1	0	0	Degree $- = 1$	Minimize criticality
5	0	1	1	Degree $+ = 1$	Minimize pollution
6	0	1	0	Degree $- = 1$	Minimize pollution
7	0	0	1	Degree $- = 2$	Cache friendly
8	0	0	0	Degree $- = 2$	Minimize pollution

– prefetch accuracy, progress and criticality of the threads as communicated by TCPAC framework. At the prefetching controllers, we throttle the prefetcher by changing the prefetch degree and the prefetch distance. Our aggressiveness control mechanism speeds up the critical threads that leads to reduction in the cache misses.

Limitations in Prior Approaches: Prior technique such as HPAC [2] controls the aggressiveness of a prefetcher by using metrics such as prefetch accuracy, cache pollution and bandwidth consumption at the DRAM. Based on these metrics, the local prefetching controller throttles a particular prefetcher. It also uses a global controller which enforces throttling decisions based on bandwidth consumption at the DRAM information from other threads. HPAC works well for most of the multiprogrammed workloads but thread criticality oblivious HPAC is less effective in case of a single parallel application running on a multi-core system. Figure 3 shows the correlation between the prefetch metrics used in the HPAC and the thread progress in terms of improvement in execution time on an 8 core system. We use multivariate linear regression modeling to find this correlation. On an average, the correlation is just above 40%. Applications such as `streamcluster` and `swaptions` have high correlation because these applications are mostly dominated by a single thread throughout the entire execution of the program and all the throttling decisions are based on a particular thread only. For other applications, multiple threads are critical. Also, HPAC does not evaluate the utility [5] of each throttling decision because of which, on an average, for an eight core system, 31% of the decisions made by HPAC is back-lashed.

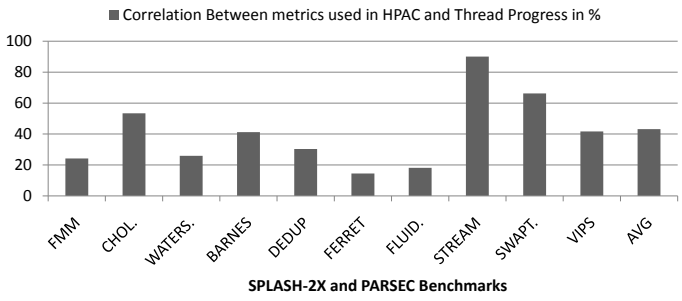

Fig. 3. **Correlation between Prefetch Metrics and Thread Progress on 8 core system.**

TCPAC-P: To address these issues, we propose TCPAC-P. TCPAC-P increases the prefetch degree of critical threads more aggressively as compared to non-critical threads. Increase in the aggressiveness improves the execution time but it also comes with its own cost. Frequent increase in the prefetch degree saturates the off-chip bandwidth and causes shared LLC pollution. To minimize the negative interference because of the prefetch requests, we use metrics such as PROGRESS, CRIT and ACC of the threads. Table I [6] shows the throttling decisions based on TCPAC-P. In an 8 core system, among the 8 cases outlined in the table I, case 1, case 3 and case 6 are the dominant cases across all the applications (on an average, 71% of the time TCPAC-P stays at these cases) followed by case 4 and case 7 (24%). In case of `ferret`, a prefetcher-unfriendly application, 63% of the time TCPAC-P stays at case 8 and rest at case 6.

[5]At a particular epoch, effect of throttling on the execution time.

Algorithm 2 TCPAC-D

1: **Priority** 1 - Demand requests from memory non-intensive[7]non-progressed threads followed by memory intensive critical threads
2: **Priority** 2 - Row hits
3: **Priority** 3 - Prefetch requests from critical and progressed threads followed by prefetch requests from non-critical threads with ACC=1
4: **Priority** 4 - FCFS

V. TCPAC for DRAM Controller (TCPAC-D)

At the DRAM controller, all the memory requests are entered in to the MRB where based on the DRAM scheduling policy, these requests are scheduled. Most of DRAM controllers take advantage of row buffers and the schedulers use FR-FCFS scheduling policy where the FR corresponds to servicing row hit memory requests first. Our baseline scheduling algorithm uses PADC.

Limitations in Prior Approaches: Prior Prefetch aware DRAM policies such as PADC [4] and BAPI [7] are oblivious to the criticality and progress of the threads. In the presence of hardware prefetchers, PADC tries to improve the row buffer locality. With PADC, on an average, on an 8 core system, 53% of total scheduling decisions are biased towards progressed threads, but prioritizing the requests from non-progressed threads helps in improving the execution time. Requests from non-progressed threads are in the critical path of the program in execution. These policies prioritize the prefetch requests based on the accuracy of the prefetchers but as discussed in section II-B, in a given epoch, the accuracy of the prefetchers remain similar across non-critical threads. Making scheduling decisions based on accuracy of a prefetcher does not improve the execution time significantly. Also, to maintain the effectiveness of TCPAC-P, the DRAM controller should be made aware of TCPAC-P.

TCPAC-D: We propose a DRAM scheduling policy called TCPAC-D that takes care of both the demand and the prefetch requests based on the progress and criticality of the threads. Algorithm 2 explains a self-contained TCPAC-D. Similar to PADC, from the MRB, TCPAC-D drops the old prefetch requests [8] that are critical. Please note, we do not consider PAMS as the baseline scheduling policy. PAMS [16], a memory scheduling technique have been proposed for parallel applications. PAMS identifies the code segments that are source of bottlenecks mainly because of locks and barriers. For our study, we do not use PAMS directly as we do not consider effect of synchronization primitives explicitly. We aim to investigate the effect of synchronization primitives along

[6]prefetch distance = prefetch degree * 16. Base prefetch degree is 4.

[7]Misses Per Kilo Instruction (MPKI) < 1.

[8]Prefetch requests which are issued $100K$ cycles before.

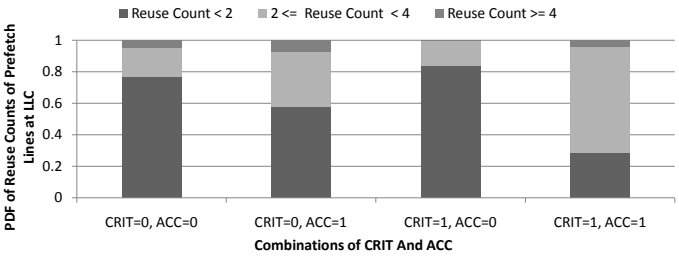

Fig. 4. **Probability Distribution Function(PDF) of Reuse counts of Prefetched Lines at the LLC.**

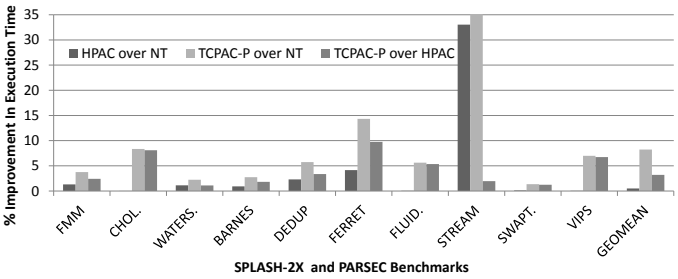

Fig. 5. **Performance of TCPAC-P on 8 core system.**

with TCPAC as part of future work.

VI. TCPAC for LLC Controller (TCPAC-C)

TCPAC-C works at the LLC controller by prioritizing/de-prioritizing the prefetch requests and the responses going to and coming from the DRAM by changing the insertion and promotion policy. We use a baseline cache replacement policy named TA-DRRIP [10] and make it prefetch aware. DRRIP stands for Dynamic Re-Reference Interval Prediction. For each cache line, DRRIP uses a N bit register to store Re-Reference Prediction Value (RRPV). For a cache line, higher the value of RRPV, the lesser is the chance of getting re-reference. For a 2 bit RRPV register, a cache line can have values from 0 to 3, where 0 denotes high chance of re-reference and 3 denotes low chance of re-reference. In case of TA-DRRIP, on a cache miss, the insertion policy inserts a new cache line with RRPV = 2 or 3. On a cache hit, the promotion policy updates the value of RRPV to 0.

Limitations in Prior Approaches: PACMan [3] is a prefetch aware LLC policy which provides prefetch awareness to DRRIP. PACMan inserts the prefetched cache lines mostly with RRPV=3 and it does not promote them frequently. So the prefetched cache lines that can get future hits are evicted before they get their first hit after insertion. Overall, PACMan is too aggressive in handling the prefetch requests with an intention to reduce shared LLC pollution. Figure 4 shows the Probability Distribution Function (PDF) of reuse counts (# re-references between insertion and eviction of a cache line) of prefetched cache lines at the LLC categorized based on the CRIT and ACC. It can be seen, prefetched cache lines that belong to threads with high prefetch accuracy are reused more often.

TABLE II. **Decisions for prefetched cache lines under TCPAC-C**

	Policy	PACMan	TCPAC-C
ACC=1, CRIT= -	Insertion	RRPV=3	RRPV=2, PF=1
	Promotion	RRPV=3	RRPV=0, PF=0
ACC=0, CRIT=1	Insertion	RRPV=3	RRPV=2, PF=1
	Promotion	RRPV=3	RRPV=1, PF=0
ACC=0, CRIT=0	Insertion	RRPV=3	RRPV=3. PF=1
	Promotion	RRPV=3	RRPV=3, PF=1

TCPAC-C: Irrespective of thread criticality, TCPAC-C inserts the prefetched cache lines from a thread with high prefetch accuracy with RRPV=2. To differentiate the prefetched cache lines from the demand cache lines, we use a PF bit. At the time of insertion, PF bit is set to 1 for a cache line inserted because of prefetching and PF bit is reset to 0 at the time of promotion of prefetched cache line. During eviction, less reused prefetched cache lines (PF=1 with RRPV=3) are prioritized over demand cache lines with RRPV=3. Table II summarizes the changes incorporated into TA-DRRIP only for the prefetch requests/responses.

VII. Experimental Methodology and Results

We use gem5 Full System (FS) [15] simulator to simulate a three level cache hierarchy. Our baseline system setup has a shared L3 cache and per core private L2 caches with HPAC controlling the L2 prefetchers that prefetches into L2 from L3. At the LLC, we use PACMan as the baseline policy and at the DRAM controller, PADC is the baseline policy. We use PARSEC and SPLASH-2X benchmarks. We run the parallel phase of these benchmarks with `sim-large` as the input size. Please note, we select the applications based on their memory footprints, prefetcher friendliness,and variations in criticality among threads. We pinned the threads into hardware cores to eliminate the variability in the execution time. In table III, we provide the details of system parameters used in our evaluation.

TABLE III. **Simulated Parameters**

Processor	ALPHA 21264
Fetch/Decode/Commit width	8
ROB/LQ/SQ/Issue Queue	192/96/64/64 entries
L1 D/I Cache	32KB, 4 way, 2 cycle latency
L2 Unified Cache	256KB, 8 way, 16 cycle latency
L3 Unified Cache	4MB/6MB/8MB for 8/12/16 cores, 16 way, 32 cycle latency
MSHRs	8/16 at L1/L2, 120 at L3
Cache Line size	64B in L1, L2 and L3
Prefetcher	Stream (32 streams), Degree=4, Distance=64
Coherence Protocol	MOESI
DRAM Controller and DRAM	On-chip, Open row, MRB entries=120, DDR2, Row hits = 168 cycles, Row conflicts = 408 cycles, 4 DRAM banks, 2 KB row buffer

A. Results

We show the effectiveness of TCPAC-P, TCPAC-D, TCPAC-C and TCPAC-PDC as a complete framework.

TCPAC-P: Figure 5 shows the effectiveness of TCPAC-P on 8 core system. We compare TCPAC-P with HPAC [2] and with No Throttling (NT) at the L2 prefetchers. In terms of improvement in execution time, on an average (Geomean), TCPAC-P beats the baseline prefetching framework HPAC by 3.20% and outperforms NT by 8.22% for 8 cores. Benchmarks such as `cholesky`, `vips` and `fluidanimate` are the major gainers. In case of `streamcluster`, the performance benefit from HPAC alone is very high and the effect of TCPAC-P is marginal. Most often, TCPAC-P throttles down application such as `dedup` because of low prefetch accuracy (< 0.38). Across all the applications, TCPAC-P steals prefetch degree from non-critical threads and allocates it to prefetch friendly progressed critical threads.

TCPAC-D: Figure 6 shows the effect of TCPAC-D compared to PADC. We also show the combined effect of TCPAC-P and TCPAC-D (TCPAC-PD) over HPAC used along with PADC. For 8 cores, TCPAC-D beats PADC by 2.95%. Applications

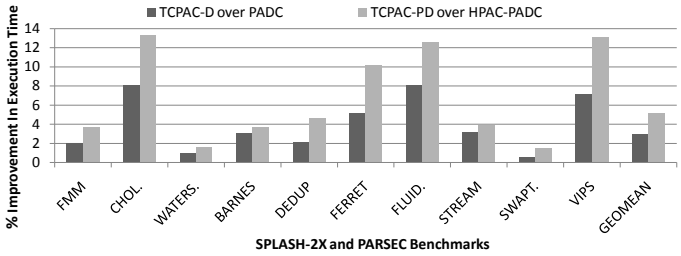

Fig. 6. **Performance of TCPAC-D on 8 core system.**

such as `fluidanimate`, `ferret` and `dedup` get the maximum benefit from TCPAC-D. In case of `fluidanimate`, only 5% of the L2 prefetch requests get hit at the LLC. So optimization at the DRAM controller helps `fluidanimate`. In case of `cholesky`, 37% of the DRAM requests come from critical threads when the MRB is $1/8th$ full that results in improvement in the execution time. For, `ferret`, the row buffer hit rate increases up to 21% with the use of TCPAC-D. When combined with TCPAC-P, TCPAC-PD beats the combination of HPAC and PADC by 5.14%.

TCPAC-C: For 8 cores, TCPAC-C improves the execution time by 4.15% over PACMan [3]. `swaptions` is the only benchmark with minimal improvement. When combined with TCPAC-P and TCPAC-D (TCPAC-PDC) improves the execution time by 7.61% compared to the combination of three state-of-the-art policies (HPAC, PADC and PACMan). Figure 8 shows the effectiveness of TCPAC as a complete framework.

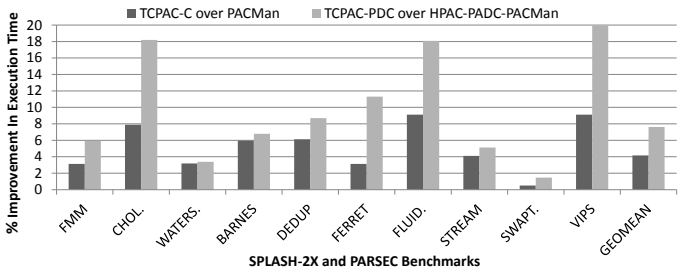

Fig. 7. **Performance of TCPAC-C on 8 core system.**

Scalability: TCPAC, as a framework is scalable on multi-core systems with 16 cores. On an average (Geomean), in terms of improvement in execution time, TCPAC-PDC outperforms the combination of HPAC, PADC, and PACMan by 7.21% and 8.32% on 12 and 16 cores respectively. Due to space constraints, we do not present detailed graphs. For many-core based systems (more than 16 cores), effectiveness of TCPAC is marginal in terms of performance. To improve the performance, TCPAC can be applied to the on chip congestion control techniques.

DRAM Traffic: In terms of DRAM traffic, on an average, TCPAC injects 2.15%, 2.27%, 2.5% extra DRAM traffic [9] for 8, 12 and 16 core systems. Benchmarks such as `ferret`, `barnes` and `dedup` inject less memory traffic compared to the combination of baseline policies.

Hardware Overhead: The hardware overhead for the basic TCPAC framework is less than 400 bytes for 8, 12 and 16 core systems as compared to the combination of state-of-the-art policies. We assume the presence of PF bit in the baseline policy (as PADC uses it). At the DRAM controller, each

[9]In terms of memory Bus Accesses Per Kilo Instruction (BPKI).

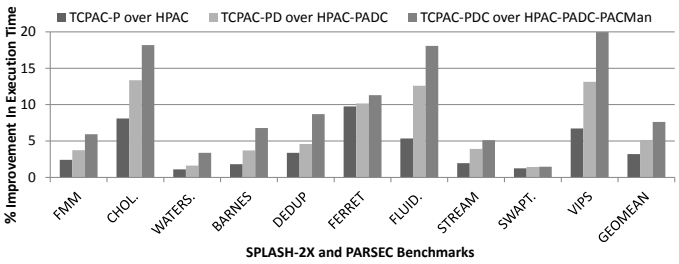

Fig. 8. **Performance of TCPAC-PDC on 8 core system.**

DRAM request buffer entry uses extra 4 bits (total 480 bits= 120 entries * 4). Also for TCPAC, extra logic circuits are used for the outliers detection algorithm.

VIII. Conclusion

In this paper, we introduced TCPAC, a prefetcher aggressiveness control mechanism that prioritizes prefetch friendly critical threads throughout the memory hierarchy. Our evaluation showed that with minimum hardware overhead, TCPAC improves the execution time significantly for parallel applications.

IX. Acknowledgments

The first author was supported by TCS India Research Scholar Program. This work was supported by IBM India Shared University Research (SUR) Grant. The authors would like to thank the anonymous reviewers for their feedback. The authors also thank Neel Gala, Prasanna Venkatesh, Anju Moosad, Raghavendra K, and Anil Krishna for their valuable feedback on earlier drafts.

References

[1] Bhattacharjee et al., "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," ISCA 2009, pages 290-301.

[2] Ebrahimi et al., "Coordinated control of multiple prefetchers in multi-core systems," MICRO 2009, pages 316-326.

[3] Wu et al. "PACMan: Prefetch-Aware Cache Management for High Performance Caching," MICRO 2011, pages 442-453.

[4] Lee et al. "Prefetch-aware DRAM controllers," In MICRO 2008, pages 200-209.

[5] Ebrahimi et al. "Prefetch-aware shared resource management for multi-core systems," ISCA 2011, Pages 141-152.

[6] Ebrahimi et al. "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," HPCA 2009, pages 7-17.

[7] Lee et al. "Improving memory bank-level parallelism in the presence of prefetching," MICRO 2009, Pages 327-336.

[8] Panda et al, "CSHARP: Coherence and SHaring Aware Cache Replacement Policies for Parallel Applications," SBAC-PAD 2012, pages 252-259.

[9] Bois et al, "Criticality stacks: identifying critical threads in parallel programs using synchronization behavior," ISCA 2013, pages 511-522.

[10] Jaleel et al, "High performance cache replacement using re-reference interval prediction (RRIP)," ISCA 2010, pages 60-71.

[11] Yoongu et al, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," MICRO 2010, pages 65-76.

[12] Ishii et al, "Access map pattern matching for high performance data cache prefetch". Special Issue: The First JILP Data Prefetching Championship (DPC-1), 2011.

[13] Ishii et al, "Unified memory optimizing architecture: memory subsystem control with a unified predictor," ICS 2012, pages 267-278.

[14] Bienia et al, "The PARSEC benchmark suite: characterization and architectural implications," PACT 2008, pages 72-81.

[15] Binkert et al, "The gem5 simulator," ACM SIGARCH Computer Architecture News, Volume 39 Issue 2, May 2011, pages 1-7.

[16] Ebrahimi et al. "Parallel application memory scheduling," MICRO 2011, pages 362-373.

[17] "http://www.ehow.com/how_5201412_calculate-outliers.html"

[18] PARSEC Group. "A Memo on Exploration of SPLASH-2 Input Sets"