AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance

Muneeb Khan, Michael A. Laurenzano†, Jason Mars†, Erik Hagersten and David Black-Schaffer
Uppsala University and *University of Michigan, Ann Arbor*†
Email: {muneeb.khan, eh, david.black-schaffer}@it.uu.se, {mlaurenz, profmars}@umich.edu

***Abstract*—Modern processors widely use hardware prefetching to hide memory latency. While aggressive hardware prefetchers can improve performance significantly for some applications, they can limit the overall performance in highly-utilized multicore processors by saturating the offchip bandwidth and wasting last-level cache capacity. Co-executing applications can slowdown due to contention over these shared resources.**

This work introduces Adaptive Resource Efficient Prefetching (AREP) – a runtime framework that dynamically combines software prefetching and hardware prefetching to maximize throughput in highly utilized multicore processors. AREP achieves better performance by prefetching data in a resource efficient way – conserving offchip-bandwidth and last-level cache capacity with accurate prefetching and by applying cache-bypassing when possible. AREP dynamically explores a mix of hardware/software prefetching policies, then selects and applies the best performing policy. AREP is phase-aware and re-explores (at runtime) for the best prefetching policy at phase boundaries.

A multitude of experiments with workload mixes and parallel applications on a modern high performance multicore show that AREP can increase throughput by up to 49% (8.1% on average). This is complemented by improved fairness, resulting in average quality of service above 94%.

I. Introduction

High performance processors employ hardware prefetching to hide memory latency. While hardware prefetchers can improve single-thread performance considerably, aggressive prefetchers waste shared resources, such as offchip bandwidth and last-level cache capacity, which can impact overall processor performance [7, 8, 9, 17]. Aggressive hardware prefetching wastes shared resources due to the trade-off between prefetch accuracy and prefetch coverage – to prefetch more useful cache lines, the prefetcher also fetches more useless cache lines [17]. Figure 1 shows that hardware prefetchers on an Intel Sandybridge processor significantly increase offchip data volume (useless data prefetching) and offchip bandwidth across most of the 14 memory-intensive Spec CPU 2006 benchmarks [6]. When several such applications co-execute, the offchip bandwidth and LLC share per-core shrink. With little shared resources to spare, hardware prefetching can lead to worse performance. In fact, we observed that for a fully utilized 4-core Intel Sandybridge i7-2600K processor running 4 different threads (randomly chosen from benchmarks in Figure 1), there is a 40% chance that hardware prefetching hurts performance considerably. Using hardware prefetching as baseline in Figure 2, the *No-Prefetching* curve (shaded) shows the increase in processor throughput when all prefetching is turned off. In this paper we present a runtime system that dynamically adjusts prefetch settings for more economical prefetching when shared resources are constrained, maximizing multicore performance by avoiding overly-aggressive prefetching and efficiently utilizing those shared resources.

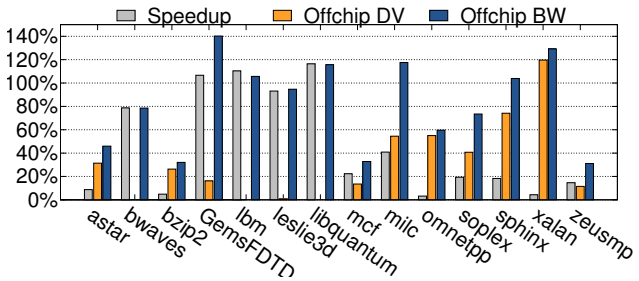


Figure 1. Aggressive hardware prefetchers (in Intel i7-2600K) increase offchip data volume (*Offchip DV*) due to useless prefetches. This wastes shared bandwidth and LLC capacity which can degrade the performance of co-executing threads.

Prior work has shown [8, 9] that a resource-efficient software-only prefetching method can avoid useless prefetches and improve throughput performance significantly in fully utilized multicores. They show that software-only prefetching (with intelligent cache bypassing) can help scale performance better than hardware prefetching by lowering the offchip bandwidth and LLC pollution. In this paper we show that hardware prefetching is not always harmful and performance can be scaled further by selectively combining it with software prefetching at runtime.

We present a runtime system that can dynamically combine any available software and hardware prefetching to scale multicore performance. The key idea is that instead of relying solely on static software prefetching [8], our approach dynamically combines it with hardware prefetching at runtime to maximize throughput in multicores. Several prefetch configurations can be created by combining software prefetching with hardware prefetching at differ-

1089-795X/15 $31.00 © 2015 IEEE
DOI 10.1109/PACT.2015.35

IEEE computer society

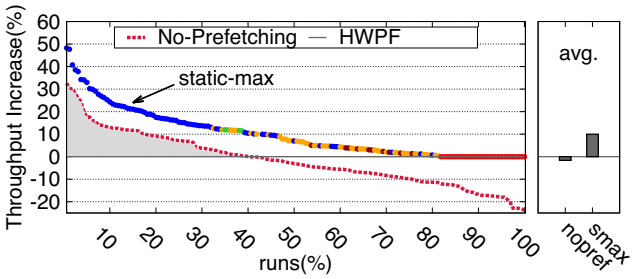


Figure 2. Hardware prefetching can hurt performance when shared resources are highly contended. When running 4 memory intensive benchmarks there is a 40% chance that hardware prefetching will hurt performance (*No-Prefetching* curve). The *static-max* curve shows that applying the right prefetch settings can improve performance up to 50% and 10% on average (*smax*). Each prefetching setting is shown with a different color.

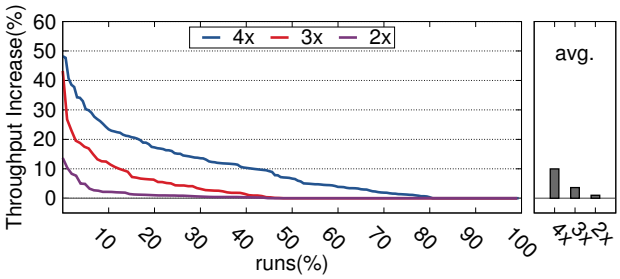


Figure 3. static-max performance across mixes of 2 (*2x*), 3 (*3x*) and 4 (*4x*) applications. As the number of active threads increase the contention in the shared resources grows, increasing the opportunity to improve performance with intelligent prefetching.

ent cache levels (Sections III-E and III-F). Our dynamic prefetching strategy explores overall processor performance across the various prefetch settings by quickly switching prefetching configurations across all running threads. The method applies the best prefetch configuration and re-explores at phase changes to keep exploration time low and to maximize the performance benefit.

In Figure 2, the color coded best prefetching curve called *static-max* shows the maximum performance improvement across 160 mixes of four applications. This curve is generated by picking the best strategy of the 5 different combinations of software prefetching and hardware prefetching (prefetch combinations described in Sections III-E and III-F). Each prefetching combination was applied to all mixes for the entire run and the best performance is shown. Prefetch configurations are color coded differently (color legend in Figure 10), for example hardware prefetching is shown with red points on the 0-axis. The static-max curve shows that with the right prefetch strategy the processor throughput can be improved (over hardware prefetching) for more than 80% of the mixes, and 10% on average (see averages on right in Figure 2).

In this paper we make the following contributions: 1) We investigate when hardware prefetching becomes sub-optimal and how performance can be improved, 2) We show that multicore performance can scale by combining resource-efficient software prefetching and hardware prefetching, and lastly 3) We propose AREP — a runtime system that dynamically combines software prefetching and hardware prefetching to maximize multicore performance. AREP can improve performance up to 49% (and by 8% on average) compared to hardware prefetching. This is complemented by high fairness with AREP maintaining above 94% *quality of service* (QoS) on average, whereas average QoS with hardware prefetching can be as low as 66%.

II. When Hardware Prefetching Hurts

Aggressive hardware prefetchers waste the shared offchip bandwidth and LLC capacity. Hardware prefetching works very well for improving single-thread performance [8]. However, as more cores are utilized and the effective memory bandwidth per core shrinks, hardware prefetching's wasteful use of shared resources can lead to worse performance.

Figure 3 shows the effect of choosing the best prefetching strategy when two (*2x*), three (*3x*) and four (*4x*) applications are co-executing. We applied the different software prefetching and hardware prefetching combinations (as for *static-max* in Figure 2) to mixes of two, three and four of the SPEC CPU 2006 applications shown in Figure 1 to see how the static-max performance scales as shared resources become more scarce (due to more co-executing applications). Note that the static-max curve in Figure 2 corresponds to the *4x* curve in Figure 3. As the number of threads increase the benefit of more intelligent prefetching choices also increases significantly. This happens because the per-core-share of the offchip bandwidth and LLC capacity decreases with increasing applications, and as a result the most resource-efficient prefetching choice wins. The opportunity is greatest when all cores are utilized: choosing the best of our prefetching approaches increases throughput compared to hardware prefetching in 80% of the mixes.

What resources are critical for scaling performance? To answer that question we make an argument using offchip bandwidth. Figure 4 shows how much the best performing prefetch policy reduces offchip bandwidth compared to hardware prefetching for mixes of 4. The diagonal represents the offchip bandwidth for hardware prefetching and the points along the diagonal are the workload mixes where hardware prefetching performs best. Points below the diagonal show that the best prefetch policy lowered the offchip bandwidth (w.r.t. hardware prefetching). Prefetch policies are colored differently (and same as Figure 2). With this graph we can get several insights. *First*, performance in a fully utilized processor improves when offchip bandwidth is reduced. The fact that there are no points above the diagonal, i.e.

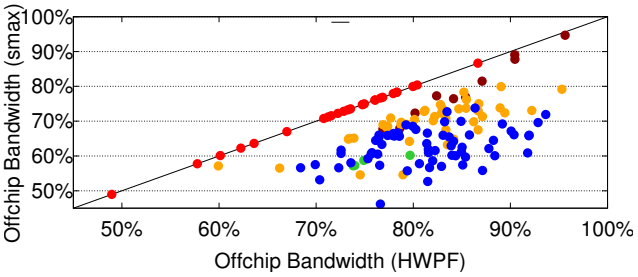


Figure 4. Offchip bandwidth of best prefetch configuration (static-max) versus hardware prefetching (HWPF). Each point represents an individual mix of 4 applications. Different colors (same as Figure 2) show the different prefetch settings. None of the winning policies is above the diagonal showing that better performance is correlated to conserving offchip bandwidth.

offchip bandwidth is never higher than the baseline hardware prefetcher bandwidth, confirms this observation. *Second*, in some cases performance is improved by further lowering offchip bandwidth even when hardware prefetching offchip bandwidth is already moderate (70% or lower). This shows that lowering contention at the memory controller and LLC also play a role in improving performance. *Third*, aggressive hardware prefetching does not do a good job of improving performance when about 75% of the total available offchip bandwidth has been saturated. Beyond this point prefetching needs to be scaled down to reduce the pressure on offchip bandwidth in order to scale performance.

III. Runtime Framework

The ability to dynamically control prefetching at runtime can help maximize performance by applying the right prefetching choices when they perform best. Since our method aims at choosing the best combination of software prefetching and hardware prefetching, it requires exploration across various hardware and software prefetching configurations. For the software prefetching part the runtime should be able to insert (and remove) software prefetches in the code on the fly. Moreover, to be able to dynamically adjust to different prefetch configurations and sample performance for those settings, the runtime must switch very quickly (to avoid performance overheads) between different hardware prefetcher and software prefetching configurations.

Hardware prefetchers can be configured on many modern processors by programming dedicated model-specific registers (MSRs). However, inserting (and removing) software prefetches in the instruction stream on the fly (when switching software prefetch configurations) is a major challenge. This can be achieved using JIT compilation, however, frequent JITing of code impacts performance and is not a feasible option. To that end we have extended the *Protean code* framework, originally developed by Laurenzano et al. [10], to maintain and use multiple versions of an application. Using *Protean code* we create two versions of the application binary at execution start up: the original binary and another one with software prefetches inserted. The runtime can then switch between these two versions to turn off/on software prefetching. The runtime can combine this capability with various hardware prefetcher configurations to create several prefetching options.

We now explain how our runtime framework functions and enables exploration of processor performance under varying prefetching settings. Applications compiled with *Protean code* instantiate a per-application runtime on a separate thread at startup. The runtime thread wakes up every 25 ms to monitor performance and (if required) change prefetch settings for its parent application. Our runtime framework consists of the applications' runtime threads and a separate *policy manager* (PM) that chooses when to change the prefetch policy.

A. Performance Monitoring

An application's runtime thread uses hardware counters to monitor the performance of its parent application. Hardware counters are configured to count branches, instructions, cycles and offchip memory requests. When the runtime thread wakes up it reports the performance values to the PM.

B. Supporting Multiple Variants

Controlling software prefetching requires inserting (and removing) software prefetches in the binary on the fly. We achieve this by maintaining two versions of the application binary: the original and one with software prefetches. At execution start up, the application's runtime thread (JIT) compiles the software prefetching binary version and saves it in a code cache. The PM can then request each application at runtime to switch versions as needed.

Protean code [10] is built on top of LLVM and applications compiled with *Protean code* have the LLVM-IR embedded in a dedicated section in the binary. At application startup the runtime thread is spawned which attaches to the embedded LLVM-IR in the binary, and uses it to JIT the software prefetching version. Since the runtime compilation of the software prefetching version happens independently on a separate thread context, the execution of the parent application thread does not stop. It should be noted that the need to JIT the software prefetching version is that of the underlying Protean code framework. It is not a necessary condition and can be removed completely if a pre-compiled software prefetching binary is embedded instead of the LLVM-IR. For this work we assume the software prefetch version has been cached and is available at startup.

C. Inserting Software Prefetches

Deciding where to insert the software prefetches in the instruction stream is a considerable challenge. The runtime does not decide where to insert the software prefetches. This information is provided as input to the runtime and can be

generated with help of a profiling pass. There is a plethora of prior works that use various fast profiling methods to identify which memory instructions miss in the cache hierarchy, such as [1, 5, 13, 23]. Those memory instructions can be targeted for software prefetching as shown by [8, 13, 23].

In principle, any software prefetching strategy listed above can be used with our dynamic runtime framework. However, in this work we have focused on reuse-distance based methods to model prefetches as they can be targeted to enable improved use of shared resources. Reuse-distance based models such as [1, 5] can model miss ratios for individual memory instructions. This information can be used to decide which memory instructions should be targeted for software prefetching. Data-reuse based cache models can use reuse profiles to point at memory instructions whose accessed data never gets reused from the cache hierarchy. This information can be used to insert non-temporal software prefetches to cache-bypass the data that is not reused from the lower-level caches (L2 and LLC). This way data with good temporal reuse properties is maintained in the lower-level caches longer and reused from there.

Khan et al. [8] and Sandberg et al. [19] have extensively used this approach to develop software prefetching methods that conserve shared resources and help scale performance when offchip bandwidth and LLC space are scarce. They have also shown the approach to work well across different data inputs. In this work we leverage the resource-efficient software prefetching algorithm developed by Khan et al. [8] to guide software prefetch insertion. The information provided is 1) where to insert the software prefetches in the instruction stream, 2) how far to prefetch and 3) should cache-bypassing be used. The runtime thread takes this information, inserts the appropriate software prefetches at the IR level and then JITs the corresponding binary.

D. Enabling Software Prefetching

Once the software prefetching version has been created and deposited to the code cache, the runtime can switch between the original and the software prefetching version when requested. *Protean code* virtualizes function calls during the compilation pass to enable changing control flow at runtime. On function calls, control is redirected to the appropriate function via a Edge Virtualization Table (EVT). When the runtime needs to enable software prefetching, the control is redirected to the software prefetching version by changing the target addresses in the EVT to point to the corresponding function versions. This enables the runtime thread to turn on/off software prefetching dynamically on the fly.

E. Hardware Prefetchers Configuration

Many modern processors allow for software control over hardware prefetchers via MSRs. In this work we use an Intel Sandybridge processor whose hardware prefetchers at the LLC/L2 and L1 cache can be enabled (or disabled) independently. We look at three hardware prefetcher configurations that have different impacts on shared resources – 1) Full Hardware Prefetching, 2) L1 Hardware prefetching only (LLC/L2 hardware prefetching turned off), and 3) No Prefetching (all prefetching turned off) . For ease we refer to LLC/L2 prefetchers as LLC prefetcher. When requested, the runtime can configure the hardware prefetchers by programming the MSRs appropriately.

F. Policy Manager

An external agent, the *Policy Manager* (PM), monitors the overall system performance and oversees the operation of all applications' runtime threads. The PM performs the following operations:

1) The PM oversees the prefetch settings across all applications and can request their runtime threads to enable/disable software prefetching (Section III-D) and hardware prefetching on the different cores (Section III-E). The PM can combine Hardware and Software prefetching options to configure prefetch settings in a total of 5 different ways – a) *Hardware Prefetching* only, b) *Hardware Prefetching* + *Software Prefetching*, c) *L1 Hardware prefetching* + *Software Prefetching*, d) *Software prefetching* only (all hardware prefetchers disabled) and e) *No Prefetching*. The selected policy is applied uniformly across all cores.
2) When not operating in the above mode, the PM receives performance metrics from all applications' runtime threads. Based on the reported metrics the PM evaluates the overall system performance and may decide to change the prefetch configuration across all cores.

Together the runtime threads and the PM form a runtime system that can monitor performance, dynamically configure various prefetch settings to explore performance improvements and apply the one that works best. The prefetching policy exploration is detailed in the next section.

IV. Methodology

To make the best prefetching decision we need to determine the best prefetching configuration dynamically. Such an approach requires exploring different prefetching configurations at runtime and applying the best one. The effectiveness of such a method depends on its ability to make the right decisions with little overhead.

The runtime framework operates in two modes: 1) the exploration mode – when different prefetching configurations are tested for performance, and 2) the performance monitoring mode – when performance is monitored for the applied prefetch configuration. We use *branches-per-cycle* (BPC) metric to evaluate performance and avoid instructions-per-cycle (IPC) due to the bias introduced by the additional software prefetch instructions.

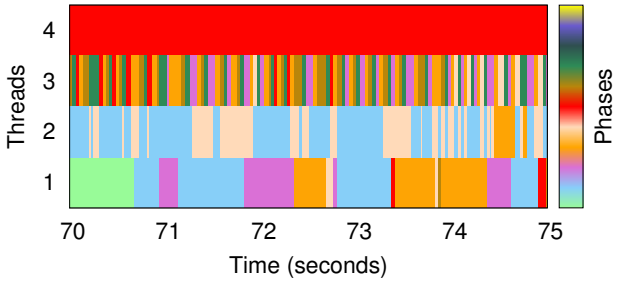


Figure 5. Phase changes across different threads in a mixed workload environment can affect performance in different ways. In this example although *thread 4* remains in the same phase, the combined affect is that of phases changing on the order of tens of milliseconds.

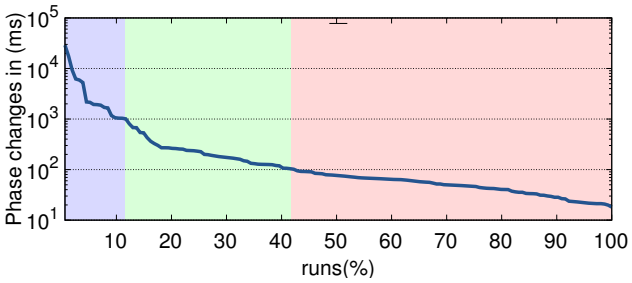


Figure 6. Average time (in milliseconds) before a phase change occurs in at least one of the threads in the workload mix. Phase changes may occur slowly - on the order of seconds (blue region), moderately fast - in hundreds of ms (green region) or fast - tens of ms (red region). Most commonly phases change in 10s of ms.

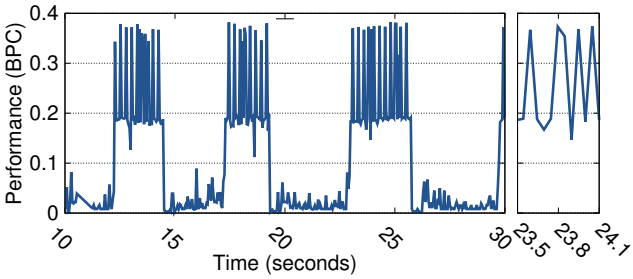


Figure 7. *bwaves* has regularly occurring microphases. Taking one sample at the granularity of tens of milliseconds will not give the right performance estimate. The behavior needs to be averaged.

Application behavior can change significantly during different phases which can have varying impact on the shared resources. Phases can change on the order of tens of milliseconds (or even faster) and require to re-explore the prefetch settings very frequently. Even more challenging is when several threads co-execute and frequently change phases. Figure 5 illustrates phase changes across four independent threads in a mix during a period of 5 seconds. Even though the phase in *thread 4* remains constant, the frequently changing phases on other threads can have varying impact on the processor's shared resources and overall performance. Fast varying phases require more frequent sampling which increases the overhead.

Figure 6 presents a summary of how quickly phases change when running 4 independent applications on 4 cores on an Intel Sandybridge i7-2600K processor[1]. We looked at more than 150 randomly generated workload mixes of 4 benchmarks from Figure 1. In more than 50% of the mixes a phase change (across all threads) can be expected on the order of tens of milliseconds (red region in Figure 6). The fastest case being an expected phase change every 18 ms. We observed that to keep the runtime system's average processor utilization below 5%, it has to operate at a granularity of at least 25 ms. This can make our runtime method oblivious to phases shorter than 25 ms. To implement an efficient runtime method we start with a simple periodic exploration approach.

A. Periodic Sampling

Sampling performance with period of 25 ms raises the issue of micro-phases. Figure 7 shows the execution of *bwaves* where frequently changing micro-phases occur repeatedly and performance shifts significantly. Figure 6 also shows that there are many cases where phases change in less than 25 ms. So, relying on a single sample window (spanning 25 ms) to evaluate the performance of a prefetching policy during exploration can lead to erroneous results. Therefore we average the behavior over several sample windows.

We implement a simple periodic sampling mechanism – the PM enters exploration phase after every d seconds and explores (i.e. records performance for) each prefetch configuration for duration of s seconds. After exploration, the prefetch configuration with the highest throughput (evaluated using average weighted speedup over hardware prefetching) is selected and applied across all cores for the next d seconds. We vary the parameters d and s and test on a small set of 35 workload mixes of 4 applications. In addition, we test a second configuration where a moving average over the whole sampled history is used to select the best prefetching policy. Figure 8 shows the average increase in throughput (over hardware prefetching) across the 35 mixes when varying d between 1–20 seconds and s between 0.125–0.5 seconds. Note that a single sample window spans 25 ms, so the parameter s=0.125 means that 5 windows are consecutively sampled and averaged for a single prefetch setting. The figure also shows the standard deviation of the difference of the runtime method's performance and static-max across all mixes. This is to evaluate how close in performance our method is to static-max. A lower standard deviation means the performance achieved by the runtime exploration is closer to the static-max performance. Performance for both

[1]Phase information was generated using Scarphase tool [20]

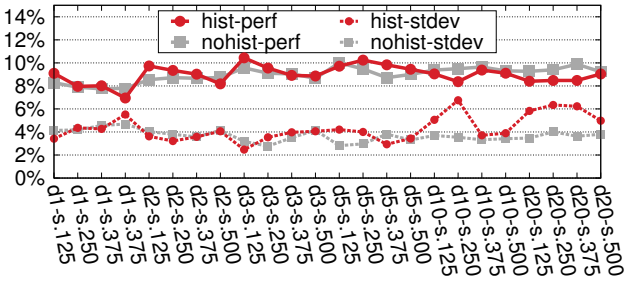


Figure 8. Average performance across a set of 35 workload mixes for the two different periodic sampling approaches: with moving-average (*hist*) and without (*nohist*). The average performance improvement over hardware prefetching and standard deviation w.r.t. *static-max* is shown. The sampling parameters giving best performance and smallest standard deviation are (hist, sample distance – d=3s, sample size – s=0.125s).

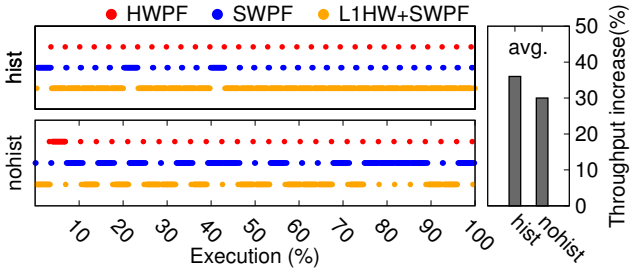


Figure 9. An example workload mix where moving-average (*hist*) helps avoid sub-optimal prefetch settings. This results in considerable performance improvements for the entire run (averages on right). We explore 3 prefetch settings here to simplify this example.

configurations, moving-average (hist) and without history (nohist) are shown.

Figure 8 shows that the performance is mostly stable across the various sampling parameters and can help increase throughput between 8-10% over hardware prefetching on average. The sampling parameters with highest performance and smallest standard deviation are (hist, d=3, s=0.125). As we move left from this point and d is further decreased to 2 and 1 we see diminishing returns due to increasing useless exploration.

Figure 9 shows an example mix (of 4 applications) demonstrating how performance history can perform better than *nohist* configuration. The figure shows when (during the execution) different prefetch configurations are applied to the mix. *nohist* applies the sub-optimal baseline (HWPF) configuration at 4% into the execution and applies the software-only prefetching (SWPF) several times during the execution, which proves to be sub-optimal as well. This results in a smaller throughput increase compared to *hist* (see averages on right of Figure 9). The moving average approach (*hist*) avoids immediately adapting a sub-optimal prefetch setting when it performs better for a short duration.

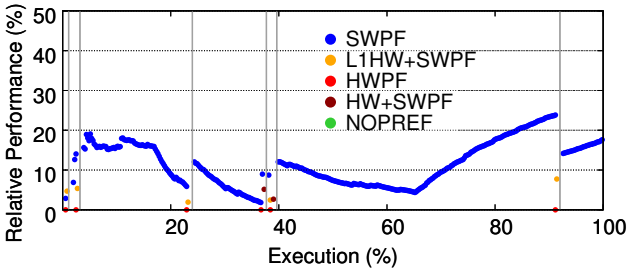


Figure 10. The Phase-adaptive exploration approach re-explores for the best prefetch setting when the overall performance changes significantly. Re-exploration (marked with vertical lines) takes place when relative performance changes by 11%. The jitters in performance are because of re-calibration of hardware prefetching (baseline) performance at exploration points. Phase-adaptive approach explores only 6 times in total, whereas the periodic exploration approach explores 32 times for this mix. The color scheme used here is the same as Figure 2.

B. Phase Adaptive Exploration

The exploration time for periodic sampling can be considerable, for example, the best periodic exploration configuration in Section IV-A (hist, d=3, s=0.125) spends almost 14% of the time testing different prefetch settings. To lower the time spent in exploration we develop a phase adaptive performance sampling approach that re-explores only when performance changes considerably (due to phase changes).

The PM continuously monitors performance (across all threads) when not in exploration mode and uses performance change as proxy for phase change. The PM enters re-exploration when it observes the overall relative performance change by $\pm c$. This change in overall performance can occur because 1) one (or more) executing thread(s) enters a new phase and its performance metric (BPC) changes, or 2) contention at the shared-resources affects performance of one (or more) executing thread(s). In either case re-exploration is required to adjust the prefetch configuration. Figure 10 illustrates the exploration process with an example assuming c=11%. The marked points in the figure show that exploration to find the best prefetch settings happen only when the relative performance changes by 11%. The jitters in the relative performance are because of re-calibration of hardware prefetching (baseline) performance at re-exploration points.

To evaluate the effectiveness of this method we explore the phase change sensitivity (c) between 5-25% on 35 workload mixes (as we did for periodic sampling Section IV-A). Figure 11 shows the average increase in throughput (over hardware prefetching) across 35 mixes for varying values of c. The figure also shows the standard deviation w.r.t. static-max, showing how close our method's performance is to static-max on average. The average performance and standard deviation (w.r.t. static-max) for the best periodic exploration settings (hist, d=3, s=0.125) is marked with a line for reference. The average performance for the

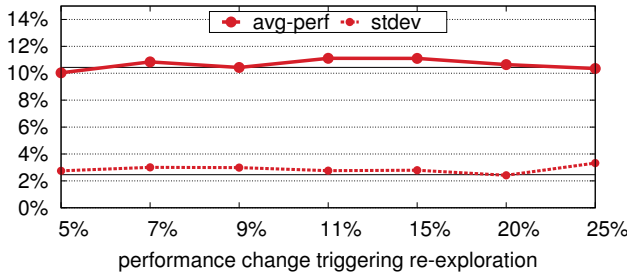


Figure 11. Average performance improvement over hardware prefetching across a set of 35 workloads mixes when varying phase sensitivity for re-exploration. The average performance and standard deviation (w.r.t. static-max) for the best performance in periodic sampling are marked with lines.

Figure 12. Performance and exploration time of the three different policies for an example mix of 3 applications (*bzip2*, *libquantum* and *milc*). The time spent in exploration clearly has an impact on performance for some workload mixes.

phase-adaptive approach is only slightly better than periodic sampling, indicating that the best prefetching policy usually remains the same through the execution. However, time spent in exploration can impact performance in some cases. Figure 12 shows an example mix of 3 applications where lowering the exploration time with the phase-adaptive approach benefits performance. For the rest of the paper we evaluate performance using phase-adaptive exploration with c=11%. With this configuration the runtime spends on average only 3.4% of the time exploring prefetching options. We call this phase-adaptive approach *adaptive resource-efficient prefetching* (AREP).

V. Evaluation

In this section we evaluate the performance of our phase-adaptive runtime prefetching method AREP (Section IV-B) on a modern x86 multicore processor. We use a 4-core Intel Sandybridge i7-2600K processor, with all cores clocked at 3.4 GHz. The DRAM (DDR3) frequency is 1333 MHz and the processor has 2 memory channels. *streams* benchmarking measures the offchip bandwidth at 13 GB/s. The L1, L2 and LLC sizes are 32 KB, 256 KB and 8 MB, respectively.

To evaluate our approach we ran mixes of single-threaded applications, parallel workloads and mixes of parallel workloads. We selected 14 benchmarks from SPEC CPU 2006 suite [6] whose dataset does not fit in the LLC, shown in Figure 1. All benchmarks were compiled using the LLVM-3.3 compiler with *-O3* optimization flag. We use randomly generated set of 160 mixes of 4 applications and 105 mixes of 3 applications. The mixes are run for two minutes and throughput is reported, excluding the initial JIT duration. We also look at 10 parallel workloads and mixes of two parallel applications, each running two threads.

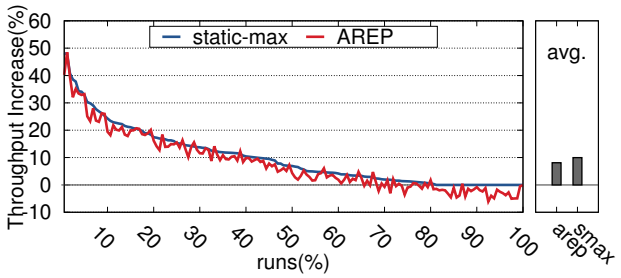
A. Mixed Workloads Performance

Figure 13a shows the performance of AREP for mixes of 4 applications. The performance increase is reported as weighted speedup, computed as

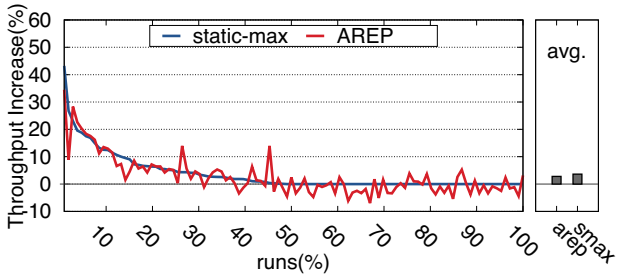
$$WS = \left(\sum_{i=1}^{N} \frac{Performance_{app_i(AREP)}}{Performance_{app_i(HWPF)}} \right) / N$$

The mixes are sorted in descending order of static-max performance. The static-max shows that there is an opportunity to improve the throughput by 10% on average by statically choosing the best prefetching choice. Note that the static-max incurs no exploration. AREP improves the performance by 8.1% on average and up to 49% in the best case. Figure 13b shows the performance for mixes of 3 applications, where AREP improves performance by 3% and static-max improves performance by 3.6% on average. In some cases AREP performs better than the static-max because it is dynamic and can better adjust the prefetching configuration to some slower changes in phase behavior.

Varying Inputs: The information about software prefetch insertion used by the runtime system is generated by a profiler pass. To evaluate AREP's sensitivity to varying inputs we used different application inputs for modeling the software prefetches and for the actual runs. The *train* curve in Figures 14a and 14b shows the performance of AREP for mixes of 4 and 3 applications when software prefetching is modeled using the *train* input set and *ref* input is used for the actual runs. The mixes are sorted by performance in descending order. The *ref* curve shows AREP's performance when the same input set (*ref*) was used for modeling software prefetches and the actual runs. There is near perfect overlap in performance when using the different inputs showing that AREP is efficient at improving performance across different inputs. Figure 15 summarizes the performance of the phase adaptive runtime exploration (AREP), periodic runtime exploration (Section IV-A) and software prefetching only (SWPF) across all mixes. Using software prefetching (SWPF) alone clearly does not perform as well as runtime exploration and is sub-optimal for mixes of 3 applications. This shows that even in highly utilized multicores the right strategy is to choose the prefetch policy at runtime instead of relying on a static policy. The static-max is 2% better than AREP, though this does not take its profiling overhead into consideration, which involves running each workload mix with each of the 5 prefetching policies to determine the best.

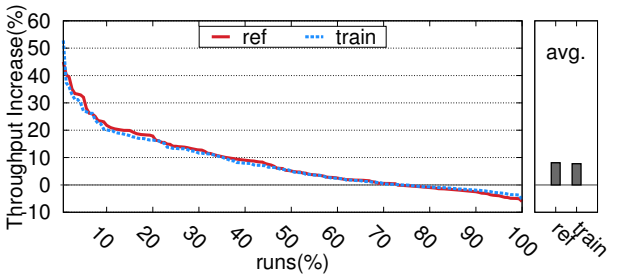


(a) Performance across 160 mixed workloads of 4 applications.

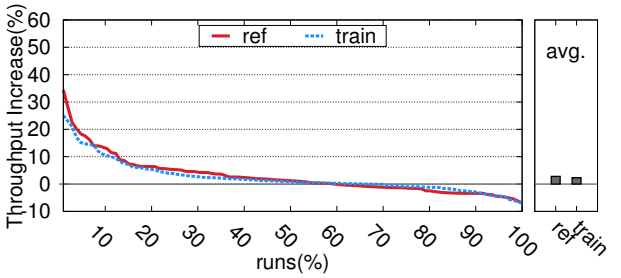


(b) Performance across 105 mixed workloads of 3 applications.

Figure 13. Performance of AREP compared to hardware prefetching (0-axis) and static-max.



(a) Performance across 160 mixed workloads of 4 applications.



(b) Performance across 105 mixed workloads of 3 applications.

Figure 14. Performance of AREP when using the *train* dataset for generating software prefetches. In both cases performance is similar to optimal case using the same input (*ref*).

B. Offchip Bandwidth

In Section II we showed that increasing offchip bandwidth utilization can have a significant impact on performance in a highly utilized multicore processor. Figure 4 showed that in such cases performance improvement is correlated to reduced offchip bandwidth (compared to hardware prefetching). In Figure 16 we compare the offchip bandwidth of AREP to hardware prefetching for mixes of 4. The mixes are sorted in ascending order of relative offchip bandwidth increase. Note that here we use *No Prefetching* as the baseline to compare how offchip bandwidth increases for AREP and for hardware prefetching. Hardware prefetching increases the offchip bandwidth pressure significantly in all cases, typically above 70% of the total available bandwidth (not shown here). Whereas, AREP consistently maintains lower offchip bandwidth than hardware prefetching. In more than 50% of the runs AREP maintains lower offchip bandwidth than *No Prefetching*. This helps explain the performance improvement achieved by AREP over hardware prefetching.

C. Last Level Cache Pollution

Besides offchip bandwidth, performance is also affected by shared LLC capacity. Aggressive hardware prefetching often brings in data that is not useful for the application. This wastes LLC space and can hurt performance of the running applications. It is important to note that an increase in offchip data volume[2] can also be the effect of a single thread executing faster due to prefetching and kicking out

[2]measured as memory requests per kilo branches

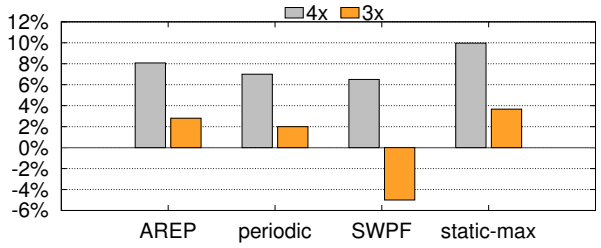


Figure 15. Average throughput increase for mixes of 4 (4x) and mixes of 3 (3x) with different prefetching policies. The phase-adaptive (AREP) approach is closest in performance to *static-max*. Software-only prefetching (SWPF) is worse in performance than both runtime approaches (AREP and periodic exploration). Also, it performs 5% worse compared to hardware prefetching when using 3 cores.

useful data for the other threads earlier. While it is hard to isolate such effects in a real system, we can simply look at the increase in offchip data volume to compare the efficiency of AREP. We use *No Prefetching* as the baseline here to compare AREP with hardware prefetching. Using the *No Prefetching* baseline gives us insight into how much "additional" (useless) data is fetched from the memory when (any) prefetching is enabled. Figure 17 compares the offchip data volume increase of AREP and hardware prefetching with the baseline. The mixes are sorted in ascending order of data volume increase. Cache bypassing can lower offchip data requests by up to 50%. On average AREP improves (lowers) offchip data volume by 22% compared to hardware

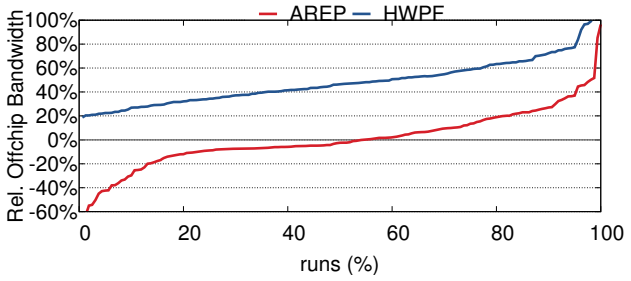


Figure 16. Offchip bandwidth for AREP remains significantly lower than hardware prefetching, and lower than *No Prefetching* (0-axis) in more than 50% of the runs. Figure 4 shows improved performance is correlated to reduced offchip bandwidth.

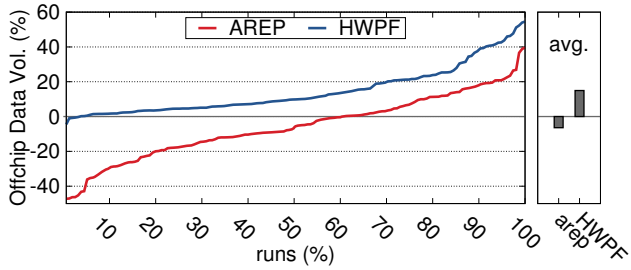


Figure 17. AREP consistently maintains lower offchip traffic than hardware prefetching, as a result causing less LLC pollution and reduced offchip bandwidth (Figure 16).

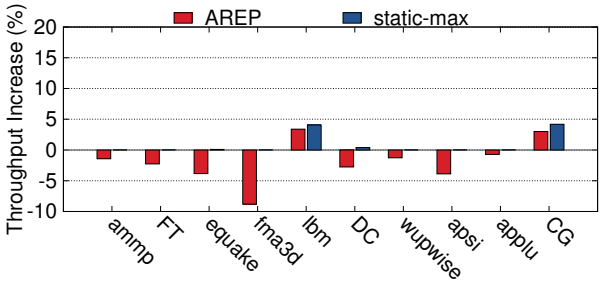


Figure 18. Throughput increase for parallel workloads running 4 threads. Prefetching improves performance in only two benchmarks.

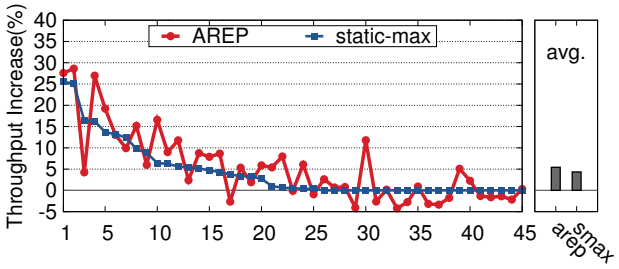


Figure 19. Throughput increase for mixes of 2 parallel workloads. Two workloads, each running two threads were co-scheduled on the processor. Overall, AREP performs better than hardware prefetching.

prefetching and 7% compared to the baseline. This significant reduction in offchip data volume is a good indicator of efficient use of LLC space and is very visibly reflected in reduced offchip bandwidth (Figure 16).

D. Parallel Workloads

To investigate performance for parallel workloads we looked at 9 benchmarks from SPEC OMP 2000 and NAS parallel benchmark suite, and parallelized *lbm* from the SPEC CPU 2006 benchmark suite (Figure 18). Interestingly, most of the parallel workloads do not saturate the offchip bandwidth, even with 4 threads. Only 3, namely *applu*, *cg* and *equake*, use more than 9 GB/s (70%) of offchip bandwidth when running with 4 threads. All runs were made using input set other than the one used for generating software prefetch information. Here it should be noted that using *branches per cycle* (BPC) is a poor metric for investigating throughput in parallel workloads, as an application spinning for a contended lock can increase this metric without making any forward progress. So, we instead used FLOPS (*floating-point operations per second*) as a measure of progress. This is an appropriate measure here since all studied workloads are floating point applications.

Across the 10 benchmarks (running with 4 threads) AREP improved performance for only two (*lbm* and *CG*), 3% for each. Both these benchmarks reuse data from the lower level caches and benefit from cache bypassing. The slowdowns for *FT, fma3d, DC* and *apsi* is because of their short running duration (less than 40 seconds at 4 threads) with several phase changes requiring re-exploration. However, we found mixed workloads of multi-threaded applications more interesting. Two workloads in Figure 18 were co-scheduled, each running with two threads, and all threads were pinned to separate cores. Figure 19 shows the performance of AREP across the 45 mixes (10 choose 2) sorted in descending order of static-max performance. On average, AREP improves throughput by 5.4%, whereas static-max improves by 4.3%. AREP performs significantly better than the static-max across several mixes. Figure 20 illustrates the case of one such example, when *equake* and *FT* are run in a mix and hardware prefetching is the static-max (mix 39 in Figure 19). When running alone with two threads the workloads require 12 GB/s (*equake*) and 4.2 GB/s (*FT*) in offchip bandwidth (Figure 20a). When run together in a mix the offchip bandwidth remains at 12 GB/s (ideally it should add up to 16.2 GB/s) which means that both applications fight for this resource (*smax* in Figure 20b). AREP adapts a policy with software prefetching enabled for 30% of the execution and lowers the offchip bandwidth considerably (*arep* in Figure 20b). That results in 5.1% improvement in performance over hardware prefetching (Figure 20c).

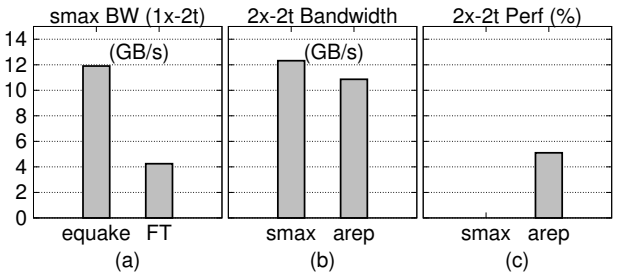


Figure 20. *equake* and *FT* fight for the offchip bandwidth when run together (each running 2 threads). AREP lowers the offchip bandwidth demand and improves throughput for this mix by 5%.

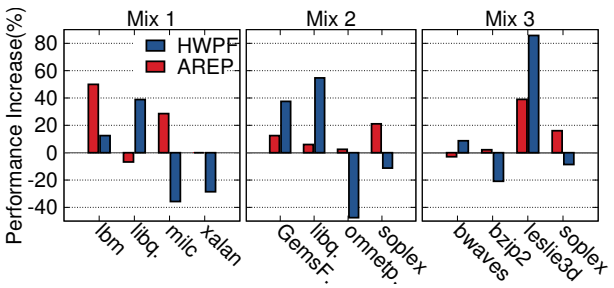


Figure 21. Example mixes showing performance across individual threads when using AREP and hardware prefetching. The baseline is *No Prefetching*. AREP does not penalize performance considerably for the individual threads and is more fair. Hardware prefetching on the other hand can slow down individual threads by more than 40%.

E. Quality of Service

In a multicore environment, shared resource hungry applications can penalize the performance of threads executing on neighboring cores by using more offchip bandwidth and LLC space (Sections V-B and V-C). This is especially the case when workloads that benefit significantly from prefetching co-execute with workloads that don't benefit much. To evaluate the fairness of AREP we look at *Quality of Service* (QoS) – the cumulative application slowdown per workload mix, computed as

$$QoS = 1 - \sum_{i=1}^{N} min\left(0, \frac{Performance_{app_i(prefetching)}}{Performance_{app_i(baseline)}} - 1\right)$$

where N is the number of co-executing threads. The QoS metric gives us insight in to how balanced the throughput increase is compared to the cumulative slowdown of running applications. To compare AREP with the hardware prefetcher we assume the original mix with *No Prefetching* as the baseline. The *No Prefetching* baseline helps compare the unfairness of any prefetching scheme that is introduced in the system. We look at an example of three integer benchmarks (*bzip2*, *omnetpp*, *xalan*) which do not benefit much from prefetching (Figure 1). AREP ensures that when they co-execute with prefetcher-friendly applications they do not starve on shared resources and consequently experience significant slowdowns. Figure 21 shows these benchmarks executing in three different mixes with shared-resource hungry benchmarks. Despite improved performance of the other applications, these benchmarks do no not see significant slowdown with AREP. Hardware prefetching on the other hand penalizes the performance of these benchmarks significantly, while increasing the other threads' performance disproportionately. For the three mixes, hardware prefetching degrades QoS by more than 40% in mix 1 and mix 2, and more than 25% in mix 3, whereas AREP degrades QoS by 6% at worst. The *QoS* for AREP is above 94% on average (Figure 22) across all workloads. Average QoS for parallel workloads running with 4 threads (*p-1x-4t* in Figure 22), and for mixes of 2 with 2 threads each (*p-2x-2t* in Figure 22) are also shown. The high QoS means that applications in the mix do not experience significant slowdowns most of the times. This shows that AREP is very effective at maintaining a high degree of fairness across the executing threads.

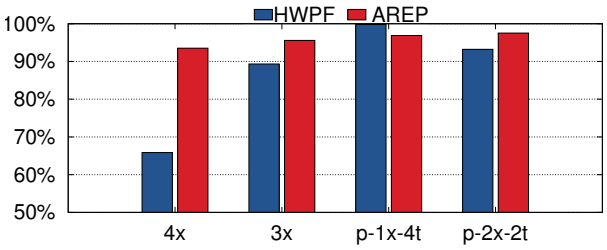


Figure 22. QoS (compared to *No Prefetching* – higher is better) for mixes of 4 (4x), mixes of 3 (3x), parallel workloads with 4 threads, and mixes of 2 parallel workloads (2 threads each). AREP maintains QoS above 94% on average, whereas hardware prefetching degrades QoS considerably in mixed workloads.

VI. Related Work

Several works in recent years have proposed novel hardware prefetching schemes for improving the utilization of shared resources (offchip bandwidth and LLC capacity) in multicores [2, 3, 4, 7, 17, 22]. Liu and Solihin [12] have proposed analytical models for bandwidth partitioning to identify when prefetching can help in improving system performance. Several other works have used software prefetching effectively to improve single-thread performance [13, 14, 15, 18, 23, 24, 25]. The list of prefetching work is too detailed to cover here, so we discuss the most relevant prior work that relates to our runtime prefetching approach.

Khan et al. developed a resource-efficient software prefetching method to scale performance in multicores when shared resources are constrained [8]. They showed that by using software prefetching (with cache-bypass hints) instead of hardware prefetching, performance scales better on a fully loaded system. However, their method relies entirely on software prefetching and does not make any use of

hardware prefetchers at all. We have shown that this is sub-optimal and performance improves significantly by applying hardware prefetching. Instead of relying on a static approach our method dynamically combines hardware prefetching and applies the best prefetch setting.

Sandberg et al. used reuse-distance based cache modeling to insert non-temporal prefetch instructions to cache bypass the data that is not reused from the lower level caches [19]. Similarly, Laurenzano et al. [10] proposed a runtime mechanism to find opportunities to insert non-temporal prefetch instructions in batch applications to conserve LLC space so that user-facing applications' performance in datacenters remains predictable. Lee et al. [11] investigated combining hardware prefetching and software prefetching for single-threaded applications, concluding that caution should be exercised when mixing the two. In contrast to their work we have shown that hardware prefetching can be combined with software prefetching in a useful way to increase throughput performance in multicores.

Jiménez et al. implemented a runtime mechanism for exploring and adjusting hardware prefetcher configuration on a POWER7 processor to maximize performance [7]. The POWER7 processor allows the prefetcher aggressiveness to be configured at 7 different levels. Their runtime method explores the best hardware prefetcher settings on per-core basis (for two cores only) and applies the one that performs best. Unlike our work, their method avoids interaction with software prefetching by explicitly disabling software prefetch insertion.

Several works have proposed hardware solutions for resource friendly prefetching [3, 17, 21] and shared resource management [16, 22]. In this work we propose a runtime system with the ability to dynamically combine any implementation of software/hardware prefetching to improve performance on existing commodity processors instead of introducing a new hardware method.

VII. Future Work

Our work is focused towards finding a single best prefetching configuration that maximizes throughput performance for the multicore. However, the selected prefetch setting may not be the best for all running threads. Prefetching may be configured at the per-core level to tap even more performance. Our framework can support such an exploration, however, there is a large exploration space that needs to be traversed at runtime. For example, in case of a 4-core processor, there will be 625 prefetch combinations to be explored. Exploring too many states at runtime is not feasible with the exploration method we present in this work. This problem can perhaps be solved efficiently with a machine-learning algorithm that is trained on the fly and then used to predict the right prefetch settings. However, such a study is out of the scope of this paper.

VIII. Conclusion

This paper presents a software based approach to alleviate the issue of inefficient prefetching on modern commodity processors. Hardware prefetchers on modern high performance processors can prefetch significant amount of useless data and increase the pressure on shared resources. This can degrade performance significantly when several threads run in parallel and the shared resources become constrained. To alleviate this problem we proposed AREP $-$ a method that improves performance by prefetching data in a resource efficient way. This is achieved by dynamically choosing combinations of hardware prefetching and software prefetching at runtime. We propose a phase-aware framework that explores the various prefetch configurations (hardware prefetching and software prefetching) at runtime and applies the best performing policy across the processor. Our results show that AREP can improve throughput by up to 49% at best and 8.1% on average. In addition to increasing throughput our method is also fair and maintains high QoS, above 94%.

References

[1] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *ISPASS*, 2004.

[2] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *ISCA*, 2001.

[3] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*, 2009.

[4] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Prefetch-aware shared resource management for multicore systems. In *ISCA*, 2011.

[5] D. Eklov and E. Hagersten. StatStack: Efficient Modeling of LRU caches. In *ISPASS*, 2010.

[6] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 2006.

[7] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell. Making data prefetch smarter: Adaptive prefetching on power7. In *PACT*, 2012.

[8] M. Khan, A. Sandberg, and E. Hagersten. A case for resource efficient prefetching in multicores. In *ICPP*, 2014.

[9] M. Khan, A. Sandberg, and E. Hagersten. A case for resource efficient prefetching in multicores. In *ISPASS*, 2014.

[10] M. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *MICRO*, 2014.

[11] J. Lee, H. Kim, and R. Vuduc. When Prefetching

Works, When It Doesn't, and Why. *ACM TACO*, 9 (1), Mar. 2012.

[12] F. Liu and Y. Solihin. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *SIGMETRICS*, 2011.

[13] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn. Profile-Guided Post-Link Stride Prefetching. In *ICS*, 2002.

[14] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In *CGO*, 2004.

[15] J. Mars and R. Hundt. Scenario Based Optimization: A Framework for Statically Enabling Online Optimizations. In *CGO*, pages 169–179, 2009.

[16] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim. Location-aware cache management for many-core processors with deep cache hierarchy. In *SC*, 2013.

[17] S. Pugsley, Z. Chishti, C. Wilkerson, P. fei Chuang, R. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *HPCA*, 2014.

[18] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler Orchestrated Prefetching via Speculation and Predication. In *ASPLOS*, 2004.

[19] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses. In *SC*, 2010.

[20] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *IISWC*, 2011.

[21] S. Srinath, O. Mutlu, H. Kim, and Y. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[22] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer. Pacman: Prefetch-aware cache management for high performance caching. In *MICRO*, 2011.

[23] Y. Wu. Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching. In *PLDI*, 2002.

[24] Y. Wu, M. J. Serrano, R. Krishnaiyer, W. Li, and J. Fang. Value-Profile Guided Stride Prefetching for Irregular Code. In *International Conference on Compiler Construction (CC)*, 2002.

[25] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous Memory Introspection. In *CGO*, 2007.