A Reactive Unobtrusive Prefetcher for Multicore and Manycore Architectures

Jason Mars
University of Virginia

Daniel Williams
University of Virginia

Dan Upton
University of Virginia

Sudeep Ghosh
University of Virginia

Kim Hazelwood
University of Virginia

ABSTRACT

Processor performance continues to out pace memory performance by a large margin. The growing popularity of multicore and manycore architectures further exacerbates this problem. The challenge of keeping the processor(s) fed with data becomes more difficult. One approach for mitigating this gap is to employ software-based speculative prefetching. Software dynamic prefetchers are able to identify more complex patterns than hardware prefetchers, while retaining the ability to respond to dynamic program behavior. However, modern techniques incur prohibitively high application overheads to detect and to exploit these data access patterns, and do little to accommodate multicore and manycore architectures.

In this work, we present an unobtrusive software prefetcher that takes advantage of underutilized cores to improve the performance of neighboring cores. We leverage multicore and manycore design to decouple the tasks of profiling, pattern detection and prefetching away from the application. Our approach takes advantage of cache coherence snooping mechanisms at the ISA level such that the cache miss patterns can be observed by a neighboring processor core. With this capability, it is possible to create a reactive solution that complements a hardware prefetcher, while isolating the tasks of pattern recognition and prefetching from altering the code or perturbing the performance of the running application. This allows our prefetching engine to be seamlessly deployed by the OS to any free core to assist neighboring cores, and terminated if those cores are needed. We call our approach *unobtrusive reactive prefetching*.

In this paper, we outline our system, discuss our hardware extensions, and present our unobtrusive speculative hot stream extraction and prefetching algorithms for detecting and mitigating recurring cache miss patterns. Using an aggressive hardware prefetcher baseline our unobtrusive core hopping prefetcher is able to reduce the number of cache misses by an average of 26% and in our best case our technique reduces the miss rate by 84%.

1. INTRODUCTION

Recent advances in processor design have resulted in a growing gap between processor and memory performance. The number of processor cores on a single chip is growing rapidly while our system memory architecture lags behind. The primary solutions to this growing problem have included hierarchical caches and hardware prefetching to overlap memory stalls with useful computation. While these solutions have been effective for certain applications (with the notable exception of pointer-chasing applications), we are now reaching a point of diminishing returns. We must develop novel approaches in order to realize any significant performance gains in the future or to handle complex data access patterns effectively.

Meanwhile, trends in computer systems and architecture, including multicore and manycore, are enabling new design approaches that were infeasible until now. The ubiquity of multicore processor designs means that new solutions can be designed that leverage idle processor resources rather than disrupting executing applications. And with manycore architectures on the horizon new techniques are needed to target the class of applications that exhibit thread level parallelism as they do not scale like applications that exhibit data level and task level parallelism. In fact, the shared cache configuration of processors that support shared memory has enabled an entirely new prefetching opportunity. A prefetching *lifeguard* can be executed on an idle core to support neighboring cores that share the same cache coherence mechanism. This enables performance gains on these neighboring core. This approach is particularly beneficial when executing on manycore systems where the OS can spawn and schedule a number of these prefetcher lifeguard to assist clusters of cores sharing a coherence bus. This prefetcher can then be deployed onto a processor core to assist other cores, or be seamlessly terminated if idle cores are reclaimed by other applications. This is possible because our software prefetching approach is *unobtrusive*. Software prefetching occurs without perturbing the application code to add instrumentation or insert prefetching instructions, thus the execution flow of the application remains uninterrupted.

In this paper, we discuss a *reactive* solution to the memory and processor performance divide that leverages the features and ubiquity of multicore and manycore processor design, the power of adaptive unobtrusive analysis software, combined with a simple extension to the cache coherence mechanism of existing and future architectures. We call our approach *unobtrusive reactive prefetching (URPref)*. We use a neighboring idle processor core to observe the *miss patterns* on the cache coherence mechanism that occur while executing an application. We then analyze the miss patterns using a novel software dynamic prefetching engine that continuously profiles and adapts to application behavior. Our

URPref engine uses Sequitur [16] to extract hot streams in cache misses (both fixed address patterns and strided patterns). We then perform prefetching whenever we observe the start (*prefix*) of a known miss pattern. We do not insert prefetch instructions into the application itself; rather, we perform the prefetching from the neighboring core directly into the application's cache via new sideline channels. These sideline channels build upon existing multicore and manycore cache coherence protocols.

URPref is dynamic, adaptive, unobtrusive, and reactive. We have chosen a *dynamic* approach because the application's memory access behavior presents prefetching opportunities that are unavailable with static prefetching algorithms. URPref is *adaptive* in that our analysis is constantly responding to application phases. It is *unobtrusive* because it requires no instrumentation or other static or dynamic code modifications to the running application, allowing it to be spawned and terminated seamlessly anywhere on the processor core grid. This is in contrast to current software dynamic prefetching techniques [6, 13, 20, 21] which interleave profiling with application code. Finally, URPref is *reactive* because it only engages when the miss rate is high. While other systems observe *cache accesses* in order to detect patterns and strides, our system observes *cache misses*, and thus only engages when patterns surface that the hardware is unable to identify. As other researchers have confirmed, prefetching for pointer-chasing applications is a difficult problem that requires sophisticated detection mechanisms that are very difficult to implement in hardware [8], thus our software detection component can help mitigate this complexity. In the case where the cache miss rate is low or no patterns can be found in those misses, our system allows the application to execute without disruptions.

The specific contributions of this paper include:

- An unobtrusive prefetching solution that profiles cache misses, detects patterns, and performs prefetching without directly perturbing the performance, or modifying the code of an executing application, allowing it to be spawned and terminated freely by the OS.
- Proposed hardware extensions that expose existing cache coherence snooping protocols, including cross-core prefetch instructions and a cache miss buffer.
- A pattern based approach to detect recurring miss patterns and to adapt to phase changes by removing expired miss patterns.

The remainder of the paper is organized as follows. In Section 2 we describe the necessary support needed for the unobtrusive reactive prefetching infrastructure, including the novel hardware extensions mentioned above. Next, in Section 3 we discuss our profiling and analysis algorithm for detecting hot cross-core L1 cache miss patterns using Sequitur. In Section 4, we describe the prefetching algorithm and our support for phase changes. Then, Section 5 describes our experimental framework and outlines our results in terms of misses avoided. Finally, Section 6 discusses and distinguishes our work from similar efforts and Section 7 concludes.

2. SUPPORT FOR URPREF

Both hardware and software must be altered slightly to support URPref and leverage spare processor time on multicore and manycore systems. This section describes the mechanisms that must be in place to use URPref.

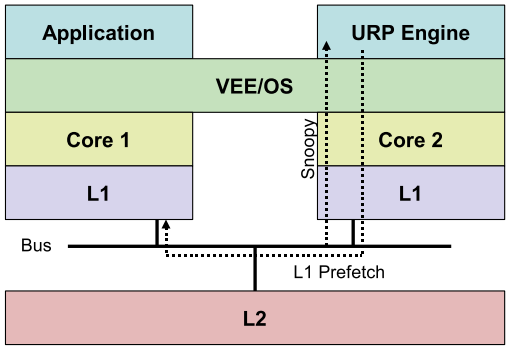

Figure 1: An overview of our system. Misses observed by snooping on the bus are passed upwards by Snoopy to our URPref engine. For example the URPref engine performs analysis and initiates prefetching into the application core's L1.

2.1 Hardware and ISA Support

As mentioned earlier, we propose exposing a cross-core cache snooping mechanism in multicore and manycore architectures. We build upon the cache coherence protocol that already exists in modern multicore chips. An example is shown in Figure 1, the bold line between core 1 and core 2 shows the existing bus used for cache coherence [11]. The dashed lines represent our proposed extensions.

First, we introduce a *Snoopy* buffer for relaying cache miss information to our URPref engine. Snoopy is a small hardware buffer that contains a short FIFO window of the most recent misses coming from the L1 of our neighboring core. Snoopy can be seen as a simplified version of the "event buffers" present in the work by Ganusov *et al.* [8]. Snoopy differs in that it provides only misses and no associated PC from the reorder buffer; thus we only need to connect our small Snoopy buffer into the cache bus line. Since we do not hold the PC associated with the miss or other bookkeeping information, we require less memory and hardware complexity.

The URPref engine sits on any underutilized core that shares the same cache coherence hardware as the application. The engine then profiles and analyzes the miss information received from Snoopy, and extracts the hot streams. Finally, our URPref engine prefetches directly into the application core's cache, as indicated by the dotted line labeled *L1 Prefetch*. Performing profiling, analysis, and prefetching on a separate core helps to mitigate the profiling overhead.

Our technique requires two ISA extensions. First, we introduce an instruction to read cache miss information from the Snoopy miss buffer, implemented in a manner similar to reading performance counters. The second ISA extension is an instruction that allows one core to prefetch into the L1 cache of its neighbor core, similar in form to existing prefetch instructions prevalent on current hardware [11].

2.2 Operating System Support

As indicated in Figure 1, the application and URPref engine run as separate processes (on separate processor cores) without explicit synchronization support. The host OS or virtual execution environment (VEE) controls both the UR-

Pref Engine and the target application and has several explicit responsibilities. First, the OS must be aware of the relationship between an application and a corresponding URPref engine. This is necessary in a multitasking environment to guarantee that the URPref engine and the application run concurrently. Second, the OS or VEE must ensure that the URPref component is executing on a core that is located on the same cache coherence bus as the application. Additionally, the OS must ensure that URPref does not usurp resources from other active processes. Therefore, if the OS is unable to find a spare core with a usable Snoopy channel, it should disable URPref. Fortunately the URPref algorithm can operate successfully even after an application has started, so benefits can resume once the neighboring core becomes free. If the OS chooses to halt the URPref engine, it is not necessary to store URPref miss histories or any other state. The system can tolerate gaps in cache misses, as it regularly retires miss patterns to adapt to phase changes.

2.3 Scalability

The scalability of URPref relies on the scalability of the underlying cache coherence protocols. In many systems, at least one level of cache is private to the core but is still kept coherent with other neighboring cores and with global shared memory state. Since URPref merely requires an interface to the existing coherence mechanism in order to snoop cache issues, the approach is applicable to any cache design that requires cache coherence information to be transmitted to neighboring cores.

Many of todays multicore systems have private L1 caches per core and a shared L2 cache [12, 17], so existing cache coherence protocols tend to communicate L1 values. For this reason, our current Snoopy prototype focuses on these systems. However, the approach scales to any level of cache that must be kept consistent between multiple cores. Our approach also scales with the number of cores assuming that the chip design supports localized shared memory between neighboring cores.

3. DETECTING HOT DATA STREAMS WITH SEQUITUR

The complete URPref solution leverages two key components: (1) the Snoopy miss buffer to carry out the task of profiling, and (2) a linear-time pattern-detection algorithm called *Sequitur* developed by Nevill-Mannings and Witten [16] to detect recurring miss patterns. Sequitur has been shown to be effective at detecting hot streams in data accesses by Chilimbi and Hirzel [5, 6]. In our work, we extend these observations to apply Sequitur to *cache misses* rather than *cache accesses*, which enables a more lightweight and reactive solution.

We use Sequitur to organize the cache-miss information into hot streams. A cache miss stream is considered *hot* based on two factors: the length of the particular miss stream and its frequency within the input sample window. Frequently, a large percentage of the total misses of an application are contained in a small portion of hot streams.

The URPref engine will continually look for new hot streams even when other streams are being prefetched. Each new miss from Snoopy is sent through a pattern-detection system based on Sequitur. It is then analyzed to determine if it forms a prefix for a prefetching stream. When a matching prefix is detected, the remainder of the hot stream will be prefetched into the neighboring L1 cache. In the remainder of this section, we discuss the details of the Sequitur system for pattern detection.

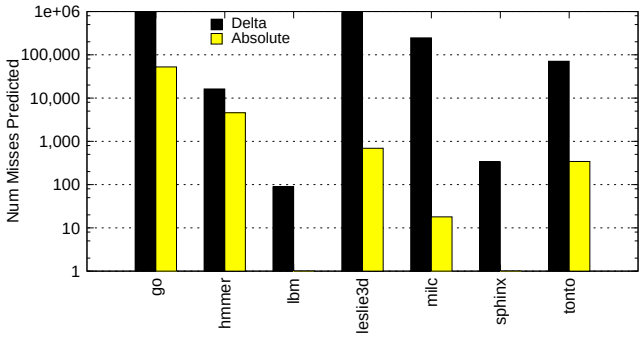

Figure 2: The effects of using the miss deltas versus absolute addresses on several benchmarks.

```
long data[5000000];

main() {
  int i;
  for(i=0; i<5000000; i++) {
    data[i] = data[i] + i;
  }
}
```

Figure 3: Pseudocode for a simple array traversal. This code is easily handled by both hardware prefetchers and our URPref algorithm.

3.1 Hot Data Miss Streams

A data miss stream is a sequence of data cache miss addresses that repeats throughout a given series of cache misses, called a *profiling window*. A miss stream is considered *hot* if it constitutes a given percentage of the profiling window. Sequitur builds a context-free grammar of the cache miss patterns. Each production in the grammar represents some sequence repeating itself two or more times throughout the sequence. The grammar productions are built miss-by-miss for some number of misses, determined by the window size. The hottest streams are then used to perform prefetching. We calculate hotness by multiplying the number of times a rule is used in the grammar (cold uses) by the number of terminal symbols (misses).

In addition to detecting patterns from actual cache addresses, we also investigate detecting recurring *deltas* in subsequent misses. In fact, we found that delta miss patterns are more effective than absolute addresses. Figure 2 shows that the number of misses or URPref engine is able to predict when using deltas versus when using absolute addresses. This graph shows that using delta gives us more opportunity when predicting misses. This makes sense given that absolute addresses patterns are captured within delta patterns, while the converse does not hold. In both cases, we are able to detect patterns that are more complex than arithmetic or geometric strides that are detected by hardware prefetchers.

To demonstrate the miss patterns that Sequitur can successfully track, we present three code snippets of increasing memory-access complexity (Figures 3–5). Figure 3 shows a

```
struct list_elm {
   long val;
   struct list_elm * next;
};
typedef struct list_elm item;

void main() {
   long i;
   item *cur = NULL, *head = NULL;

   for(i=1; i<=1000000; i++) {
    /* initialize list here assuming
       each element is allocated
       at an arbitrary location in mem */
   }
   cur = head;
   while (cur) {
      cur->val = rand() % 40;
      cur = cur->next ;
   }
   cur = head;
   while (cur) {
      cur->val *= 2;
      cur = cur->next ;
   }
}
```

Figure 4: Pseudocode for a simple linked list traversal. Hardware prefetchers are often ill-suited for arbitrarily complex stride patterns, while our URPref algorithm can detect and prefetch such patterns.

```
long mat[1000][1000];

/*arbitrary stride*/
inline int arb_stride(int base) {
  if(base > 500) return 2;
  if(base > 300) return -24;
  if(base % 2 == 0) return 8;
  return 1;
}

main() {
  int i,j;
  for(i=0; i<1000; i++)
    for(j=0; j<1000; j++)
      mat[i][(j+arb_stride(j))%1000] = i*j;
}
```

Figure 5: Pseudocode for an irregular access pattern. Hardware prefetchers are ill-suited to such patterns, while our URPref algorithm is often well suited.

simple array-traversal memory-access pattern that can be detected and prefetched by most hardware stream prefetchers, as well as our technique. Figure 4 shows a linked-list access pattern that revisits the same nodes repeatedly. This pattern is too complex for current hardware prefetchers, although it is handled by many software-based dynamic prefetchers. Finally, Figure 5 presents an irregular strided pattern that cannot be detected using absolute addresses. However, by examining the deltas between the accesses, Sequitur is able to effectively detect the miss pattern. Therefore, in the cases of highly-irregular access patterns, neither stride prefetcher nor absolute-address pattern detection are effective, while our URPref engine can successfully prefetch the correct data.

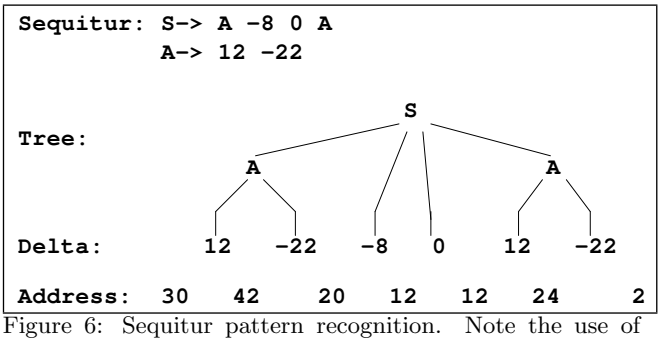

Figure 6: Sequitur pattern recognition. Note the use of address delta as a Sequitur key.

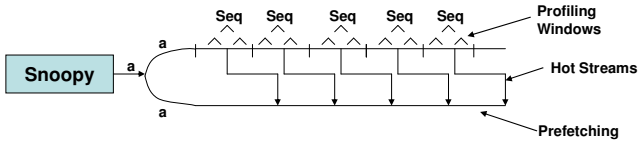

Figure 7: Simultaneous profiling and prefetching interaction. The symbol `a` represents a miss coming from Snoopy. This miss is processed by both the analysis and prefetching engines. The hot streams that are detected in the analysis is continuously fed to the prefetcher engine.

3.2 Using Sequitur

Sequitur is a linear-time algorithm that incrementally infers the compressibility and hierarchal nature of the miss patterns coming from Snoopy. As we show in Figure 6, we first determine the delta between adjacent misses. We use these deltas as the symbols for which Sequitur will build its grammar. In the figure, we show a tree representation of the information provided by Sequitur. Any repetition that occurs in the input results in a non-terminal being created in the grammar. Repetitions involving non-terminals create other non-terminals higher in the hierarchy. Each non-terminal encodes a potentially hot cache miss stream. Similar to previous work [5, 6], we measure the *hotness* of each stream using Equation 1. *Length* is defined as the sum of the number of terminals for a given non-terminal. For example, in Figure 6, `A`'s length is two. *Cold uses* is defined as the number of times the non-terminal appears in the grammar. In our example, `A`'s cold uses are two. The length times the cold uses of each non-terminal determines the stream hotness. `A`'s hotness value would be four.

$$hotness = length \times colduses \quad (1)$$

We use a prefix to prefetch the remainder of the stream speculatively. A prefix is composed of the first few symbols of the hot stream. As we watch for misses from Snoopy we look for prefixes and trigger prefetching whenever a prefix is observed. Our URPref engine uses a prefix size of two for hot streams of size four to six. For larger hot streams, we use a prefix size of four, which we have determined to be an effective size from careful tuning.

For our URPref engine to perform its profiling and pre-

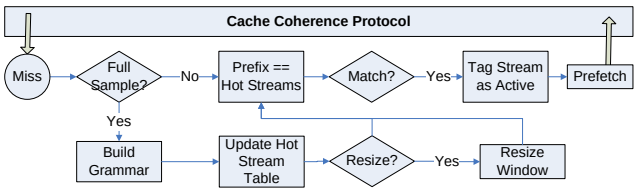

Figure 8: URPref Flow Diagram

fetching, we send every address received from Snoopy to both our profiling and prefetching engines. Our profiling engine accumulates a window of misses from Snoopy to build the Sequitur grammar and extract hot streams. The size of this window changes dynamically to normalize the number of hot streams detected per window. If we find too few hot streams in the current window, we double the window size; if we find too many, we divide the window size by two. Our prefetching engine keeps track of the last four misses and continually tests for a match in the *active hot stream table*. This table contains a set of extracted hot streams that remains active. As our engine receives misses from Snoopy a prefix window is maintained containing the last four misses. After each miss this prefix window is used to search our active hot stream table. If we find a match, the matching stream is prefetched. This stream in our table is also marked as *still active*. After a set window of misses all streams that are unmarked are retired from the table.

The criteria we use to determine the streams to place in our active hot stream table is that the stream must cover at least 2% of our window and contain at least four misses. We chose 2% after trying numerous alternatives. We want our streams to be sufficiently prominent to reduce our chance of prefetching unneeded addresses. However we must also be careful to extract as many useful patterns as possible.

In Figure 7, profiling windows are demarcated by tic marks within the profile stream. We build a Sequitur representation for each window. From this representation we extract hot streams based on our criteria and place them in the active hot stream table. This set of hot streams is then used by our prefetching engine to detect stream prefixes and prefetch while the profiler builds the next set of hot streams from the next window. Any new hot streams are added to the table and hot streams that are no longer active are retired. After each window is processed, the system decides whether to adapt the window size. This is determined by the number of hot streams we find. In our experiments, we found that a lower bound of 5 and an upper bound of 15 hot streams in our table worked well. A flowchart representation of our approach is depicted in Figure 8.

4. OFF CORE RESPONSE PREFETCHING

If no hot streams are present, our dynamic prefetcher remains dormant. If hot streams become cold, our prefetcher stops. This approach is very different than other software prefetching approaches. Other systems modify the application code statically, by way of the compiler, or dynamically, by way of a runtime compilation system. When inserting prefetching instructions into the application statically, these prefetch directives must be conservative because no runtime knowledge about the application is available. On the other hand, when prefetching instructions are injected and removed dynamically, overhead is incurred for these runtime code modifications. Our approach avoids both of these drawbacks.

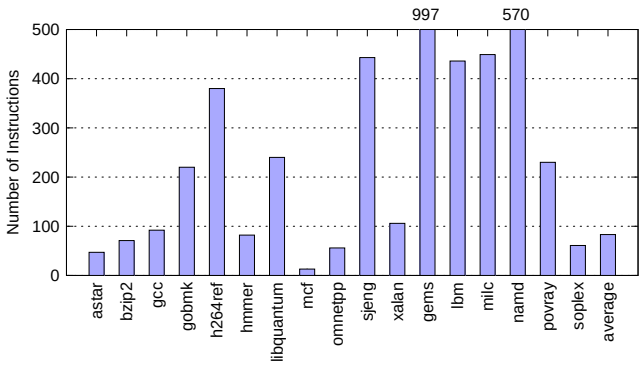

Figure 9: The average number of instructions between two misses. This shows the approximate slack we have in between misses to maximize the benefit when we react to miss patterns.

4.1 Non-Intrusive Dynamic Prefetching

Performing analysis on a separate core is only the first step; using the information obtained to carry out the prefetching presents the new challenge of how to prefetch the data to the other core. A simple solution would be to use the analysis to rewrite the application text with explicit prefetch instructions. Instead, we propose a remote prefetch instruction that can be executed on one core to prefetch directly into another core's cache. This allows the system to push data directly into the application core's cache without modifying the application code, thereby allowing the prefetcher to adapt to phase changes.

Another challenge to performing prefetching on a separate core is the asynchrony between application execution and the prefetcher. In particular, URPref prefetches the entire tail of a stream once a prefix is identified, without respect to the expected latency of future misses. If the data is fetched too late, the instruction using the memory might have already requested it, thus making the prefetch redundant. However, as shown by Figure 9, this issue arises infrequently in practice because many instructions occur between misses on average. Even if the prefetch has not completed by the time the instruction requests the memory, the load latency will be decreased due to the prefetching of the entire tail of the miss stream at once, thus many subsequent prefetched addresses will be available in the cache.

4.2 Adaptive Response to Phase Changes

Detecting and responding to phase changes in a timely manner is critical to achieving maximum performance and reducing cache pollution due to incorrect prefetching. Failing to notice a phase change can cause the prefetching engine to retain unnecessary state, and more importantly, to prefetch incorrect data while evicting useful data. For instance, consider the simple case where an application accesses memory in a pattern `ABCDEF` in the first phase, and that the URPref engine prefetches `DEF` when addresses `A`, `B` and `C` miss. In a later phase, the application may see the pattern `ABCGHI`, where `DEF` and `GHI` map to the same cache lines. If the prefetching engine fails to adapt, it will incorrectly prefetch `DEF`; in the extreme case, the accesses may

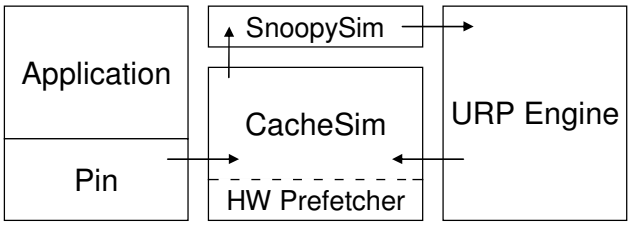

Figure 10: The URPref Simulation Framework. We use Pin, a binary instrumentation tool, and a heavily modified version of Pin's dcache simulator with hardware prefetching as our simulation infrastructure.

happen in a loop and cause `GHI` to miss every iteration.

Some dynamic optimization systems have approached phase detection by sampling the program counter and determining that a phase change has occurred if the sampled PC deviates from the previously-sampled range by more than a given tolerance [13]. Later work [7] suggested better phase detection could be accomplished by targeting sampling around specifically optimized regions.

The URPref engine automatically handles changing phases via continuous profiling. In each execution window, all inactive hot streams are removed from the active hot stream table. The engine then processes the current grammar and repopulates the active hot stream table. The new table is then used to drive prefetching in the next window. Because of this, there is at most a one-window latency between entering a new phase and being able to prefetch.

One potential side-effect of prefetching based on profiling data from the last window of the previous phase is incorrect prefetching leading to cache pollution. This may happen if the data streams in both phases share a common prefix; if, as in the example above, the first phase has a hot data stream `ABCDEF` and the second phase has a memory access pattern `ABCGHI`, the prefetching engine would incorrectly prefetch addresses `DEF`. However, as we will show later, the low percentage of mispredicted streams on average suggests this is not a common case. Meanwhile, in the case where the prefixes are different, the engine will not issue any prefetch instructions.

5. EXPERIMENTAL RESULTS

To test the effectiveness of the URPref Engine, we used the Pin dynamic instrumentation framework to implement our simulation infrastructure. Pin [15] is a binary instrumentation system that examines and possibly instruments all instructions immediately before they execute. As we show in Figure 10, we implement our Snoopy mechanism using a cache simulator built as a plug-in to Pin. We used Pin's API to instrument all loads and stores within the running application; we then feed that information into our cache simulator, which is a plug-in Pintool. If we miss in the L1 our cache simulator sends that miss information through Snoopy to our URPref Engine. Our URPref Engine continually analyzes these misses and sends any resulting prefetch directives to our cache simulator. The cache simulator reports the accuracy (hit rate) of our approach while our URPref engine measures other important information, such as the hot stream extraction rate and the prefetch predication accuracy.

To gather our results, we executed 17 applications from

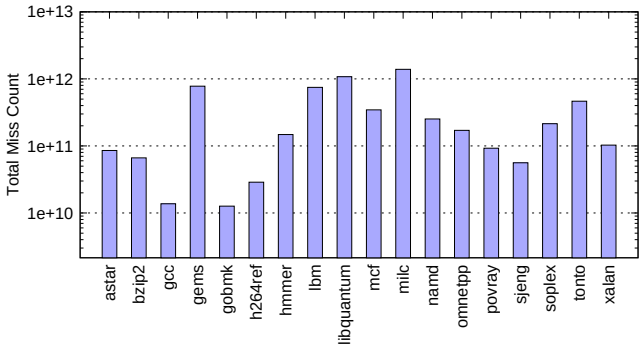

Figure 11: The total number of misses per benchmark.

Table 1: Baseline Cache Hierarchy

Cache	Size	Associativity
L1 Data	256KB	256
L2 Data	2MB	4
Prefetch Buffer	2 Streams	

the SPEC2006 benchmark suite [9]. Each benchmark was compiled with `gcc` optimization level 2 (-O2) and run to completion on of our simulation infrastructure using the reference inputs. In the case were there were multiple inputs, we used the first input listed. Figure 11 shows the baseline number of cache misses that occur during the execution of each of our benchmarks without prefetching.

5.1 Cache Simulation

The structure of our simulated cache is summarized in Table 1. The L2 is a 2MB, 4-way associative data cache that uses a round robin replacement policy. The L1 is a 256KB 256-way associative cache. We use this highly associative cache as an upper bound on performance attainable through the hardware cache by removing the effect of conflict misses as much as possible. In order to fairly compare against modern microprocessors, we also implemented a hardware prefetch engine that consisted of a strided prefetcher that keeps track of two arithmetic strides within recent memory accesses. Because the cache simulation is not cycle accurate, our baseline prefetch engine is optimistic, meaning that regardless of the timing between strided accesses, the data in the prefetch buffer is assumed to be available. We compare our URPref approach against the above aggressive baseline hardware configurations.

5.2 URPref Engine Simulation

When simulating our URPref Engine, we begin with a minimum window size to analyze and to extract hot streams. We then adaptively scale the size of the window if we receive too few or too many hot streams. This adaptation allows us to determine a suitable granularity automatically to detect hot miss streams. By default URPref doubles the current window if it observes less than five hot streams, and halves the window if it sees more than fifteen hot streams. We also set the upper and lower bounds on the window size to be no less than 50 and no more than 3000.

A stream is identified as hot based on its coverage over the entire window. Recall that we defined the stream's hotness as the product of its length and cold uses (occurrences in

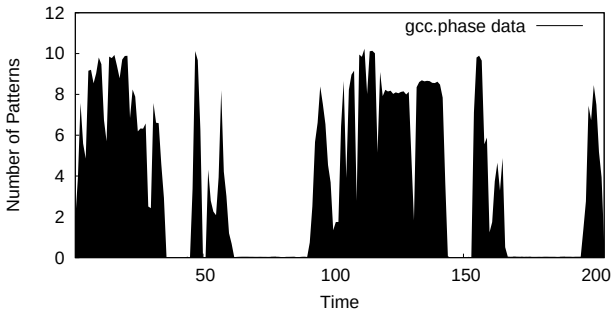

Figure 13: Number of hot streams detected per window over the execution of `gcc`.

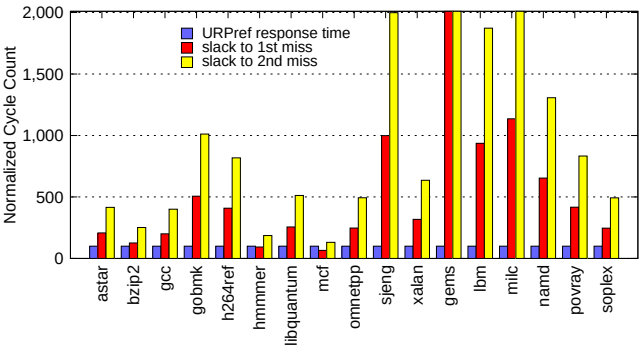

Figure 14: Response time to successfully prefetch.

Sequitur's grammar). We identify a stream to be hot if it covers at least 2% of the window size.

To measure the effectiveness of the URPref Engine, we examine the reduction in miss rate due to URPref prefetching. We also examine the accuracy of our predictions and and the cache pollution that results from miss predictions. Additionally we present data on the observed phases in cache miss behavior using URPref hot stream extraction rates.

5.3 Miss Reduction

Figure 12 shows the effectiveness of our approach on SPEC-2006 integer and floating point benchmarks. This graph contrasts our aggressive hardware baseline with and without use of a hardware stride prefetcher, to this same baseline when using our URPref approach. In every case, URPref's performance equals or exceeds that of the hardware prefetcher, in many cases, by a large margin. To calculate our overall improvement we use the harmonic mean. As the last cluster of bars show, just using that hardware prefetcher alone gives us a 6% improvement. However, when using our URPref approach the mean miss rate reduction is 26% when using the hardware prefetcher, and 30% when only using the URPref engine.

At first glance, it may seem counter intuitive to have a better improvement when using URPref without the hardware prefetcher. However, when the hardware prefetcher is enabled, the quantity and quality of cache miss information that is sent to the URPref is reduced. This results in fewer prefetches from the URPref engine and slightly more cache pollution, as will be shown in the next section.

Our URPref approach works particularly well on libquantum with a 84% reduction in cache misses. This suggests that libquantum has data access behavior that contains patterns that are too complex for the hardware prefetcher. Libquantum is the simulation of a quantum computer [9]. It has some non-trivial access patterns through matrices and we suspect this is the source of the problematic miss patterns; our research into specifically identifying these patterns is ongoing.

5.4 Phases Observed

In Figure 13, we show the phase changes in miss predictability of `gcc`, a benchmark representing the type of application for which our approach is effective. The phases are based on the number of hot streams our URPref engine was able to extract from Sequitur per cache miss window, therefore points on the x-axis point in the graph represents the average number of streams over 10,000 windows. From this data we can see the regularity and predictability of that our URPref engine can detect throughout the execution of the application. A consistently high number of patterns implies we are in a hot code region.

5.5 Pattern Recognition and Prefetch Response

As mentioned earlier, the application and our URPref engine run simultaneously. Therefore it is important to determine whether the URPref prefetching occurs in time for the application to use those data elements. To answer this question, we measure the number of cycles it takes our URPref engine to go from reading a miss to prefetching the first few elements of the hot stream. We use PAPI [2] on the Intel Xeon architecture to read the processor's performance counters. To simulate the timing of a read from the snoopy FIFO, we executed a `RDTSC` instruction. Because this is a read from a register that is not on the common execution path, we believe this provides an upper bound for timing the snoopy FIFO. Timing this microbenchmark, the URPref engine takes just under 100 cycles to progress from the miss to the prefetch of the first element. We call this the response time of the URPref engine to detect and to respond to observing a hot pattern. To see how this compares to the average slack of each benchmark, we use PAPI to calculate the average CPI for that benchmark and multiply it by the average number of instructions between two misses as it is shown in Figure 9. This gives us the average number of cycles between two misses for each benchmark. In Figure 14 we compare the URPrefs response time to this slack for each benchmark.

As we can see in Figure 14 in most cases the URPref engine is able to recognize and respond to hot streams with plenty of time to spare. And although we are not able to respond in time for the first miss with `mcf` or `hmmer`, by the second miss in the stream we are able to prefetch in time. Considering that streams range from 6 elements to as much as 100+ elements, our URPref engine is efficient enough to prefetch effectively.

5.6 Cache Pollution

Figures 15 and 16 show the L1 data elements prefetched by the URPref engine. They are separated into two groups. The first group contains successful prefetches, data elements that were used shortly after being prefetched. The second group shows data elements that were not used shortly after being prefetched. We call these prefetch 'hits' and prefetch 'misses', and they indicate how much our technique pollutes the cache. Figure 15 shows on average 95% of the

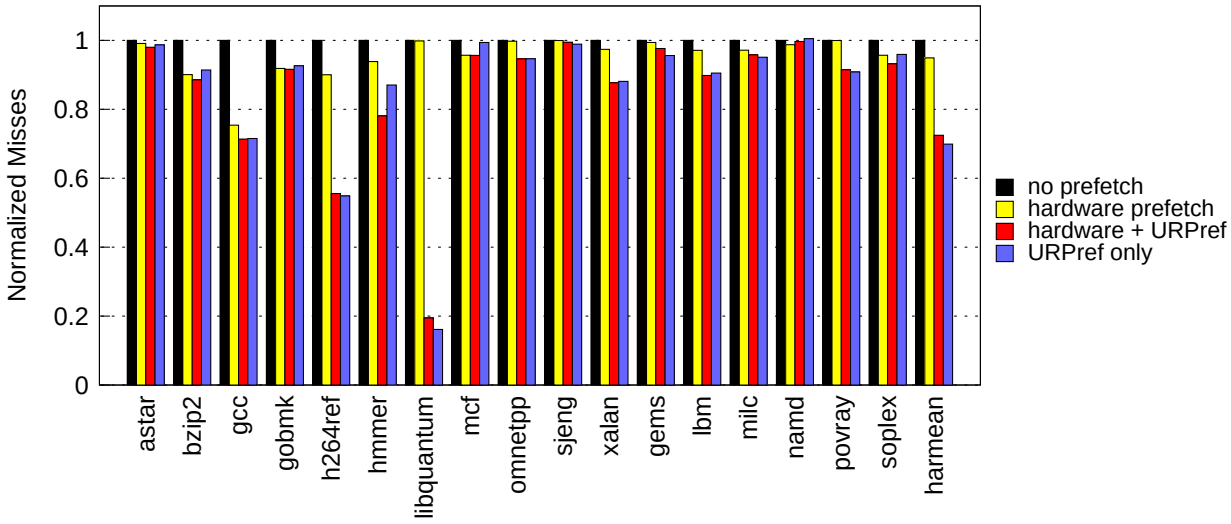

Figure 12: Reduction in misses. All results are normalized to the miss rate without prefetching.

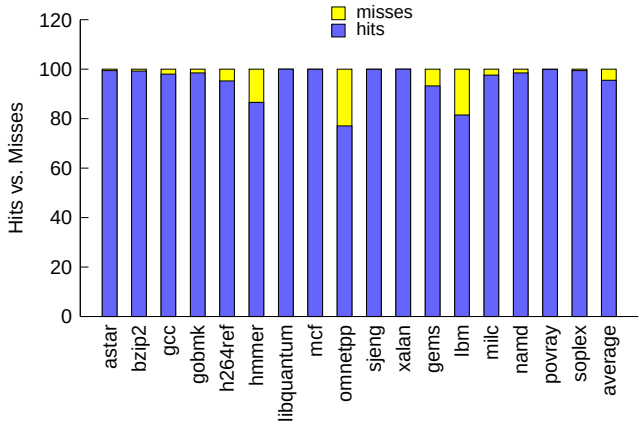

Figure 15: Addresses successfully vs unsuccessfully predicted by URPref.

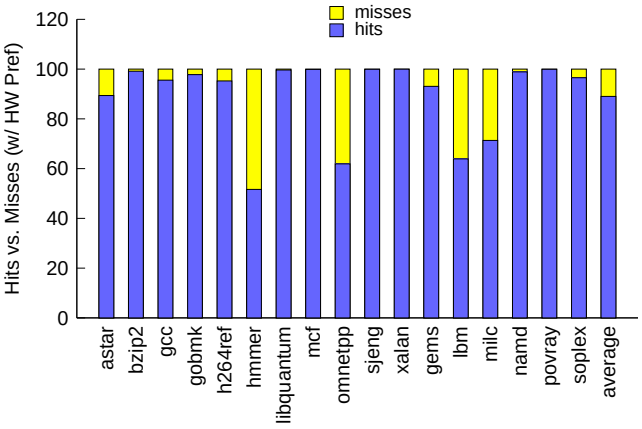

Figure 16: Accuracy of URPref when hardware prefetching is enabled.

data prefetched by our engine is used by our application shortly after it is prefetched, when not using a hardware prefetcher. Similarly Figure 16 shows that when coupled with a hardware prefetcher, 90% of our prefetched data is used. This helps to explain why Figure 12 demonstrated a performance degradation when using the URPref engine with versus without a hardware prefetcher. However, the overall performance improvement of URPref is over 25%, either way.

6. RELATED WORK

The work presented in our paper combines two research areas, prefetching and run-time optimization. Software prefetching methods can be divided into two main solutions, inserting explicit prefetch instructions [4, 6, 13] or using precomputation or helper threads [13, 14, 20, 21]. Run-time optimization systems [1, 3, 15] provide opportunities for instrumenting and modifying code while it executes; this opens opportunities for adaptive behavior not available at compile time.

6.1 Software Prefetching

Ganusov and Burtscher [8] propose a mechanism to use cache miss events derived from hardware to drive software prefetching. Their approach involves leveraging hardware prefetching techniques at the software level to drive prefetching helper threads, by adding an event buffer. Our work is similar to theirs in that we use hardware information to drive software prefetching; however, rather than adding structures to the hardware, our approach simply leverages existing information. Further, we can drive prefetching for more complex access patterns than can be detected using standard hardware techniques.

Chilimbi and Hirzel [6] also used a Sequitur-based method in a dynamic optimization system to direct prefetching. Our work differs from theirs in that while they sample small parts of the execution using bursty tracing [10], we are able to profile the execution of the entire program. This increases opportunities for detecting and adapting to phase changes. In addition, because our approach does not require instrumenting the program and duplicating code, we are able to obtain lower overheads in terms of time and space usage. Finally, we have shown that we can successfully detect patterns in cache misses, while previous approaches focused on

patterns in cache accesses.

Zhang *et al.* [21] propose using small hardware modifications along with the Trident dynamic optimization system [19] to guide prefetching threads. Our work differs from theirs in several ways. First, whereas they propose adding additional hardware to collect information about delinquent loads, our work leverages existing hardware for cache coherence on a chip multiprocessor (CMP). While their approach focuses on improving the performance of frequently-delinquent loads in hot traces, we profile and make decisions based on longer memory access patterns across the whole application. Finally, because we use a reactive method based on these patterns rather than looking into the future with precomputation threads, we are able to avoid the overhead of generating specialized precomputation code and monitoring it to ensure that it stays closely synchronized with the main thread.

Solihin *et al.* [18] proposed using user-level memory threads to do correlational prefetching into the L2 cache. However their work was focused on using correlational prefetching as opposed to using pattern recognition on sequences of misses. Also the hardware changes required for their approach differs from those required by our URPref approach. We take advantage of already existing hardware to do cross-core introspection.

6.2 Dynamic Optimization

Dynamic optimization systems such as Dynamo [1], DynamoRIO [3], ADORE [13], Trident [19], and Pin [15] provide run-time opportunities for code instrumentation and modification. In our work, by moving monitoring for prefetching into a dynamic optimization environment, we can implement more sophisticated policies than could be implemented in hardware alone.

Zhang *et al.* [20] also presented work on analyzing hot traces in a dynamic optimization environment to insert prefetching instructions. Like our work, their work involves hardware feedback to help guide decisions. However, their proposal requires adding additional hardware, whereas we only require a modification to expose the existing information on a hardware CMP. In addition, we are able to run our prefetching separately instead of having to modify the application's code to include prefetch instructions.

Lu *et al.* [13] analyzed load access patterns in the ADORE dynamic optimization system. They also perform detection on a core separate from the main application core. Similarly to Zhang's work, and unlike our system, they focus primarily on code regions with delinquent loads and inserting prefetch instructions. Again, our work is able to detect patterns across the entire application and handle prefetching without modifying the application's code.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented our unobtrusive reactive prefetching engine which utilizes multicore architectures to perform direct prefetching from one processor core to another. We accomplish this using our Snoopy interface, which exposes the cache coherence protocols to the software layer, and by using the Sequitur algorithm, which performs memory access pattern analysis using an adaptive dynamic prefetching algorithm. By offloading analysis and prefetching to a separate core, we avoid rewriting the application's code and can profile continuously without introducing any overhead in the main application. We are able to prefetch with a high accuracy based on knowledge of previous misses, including automatic reaction to phase changes, and can successfully predict more complex patterns than a hardware stride prefetcher. Our prefetching solution reduces the strain on the memory subsystem, thus increasing data cache hit rates and reducing overall processor stalls.

The current version of our URPref engine detects patterns and performs prefetching based on misses. Future extensions include more sophisticated analysis based on miss context, or the pattern of misses with respect to hits in surrounding cache accesses. In addition, we would like to leverage our interface for applications beyond prefetching. For instance, the patterns dynamically extracted from the application may be used to guide re-layout of memory.

8. REFERENCES

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int'l Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[3] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *First Int'l Symposium on Code Generation and Optimization*, pages 265–275, March 2003.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 40–52, Santa Clara, California, April 1991.

[5] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 191–202, New York, NY, USA, 2001. ACM Press.

[6] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. *SIGPLAN Not.*, 37(5):199–209, 2002.

[7] A. Das, J. Lu, and W.-C. Hsu. Region monitoring for local phase detection in dynamic optimization systems. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 124–134, Manhattan, NY, USA, March 2006.

[8] I. Ganusov and M. Burtscher. Efficient emulation of hardware prefetchers via event-driven helper threading. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 144–153, Seattle, Washington, September 2006.

[9] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[10] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic*

Optimization (FDDO-4), December 2001.

[11] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Order #253668-019, March 2006.

[12] R. Kumar, V. Zyuban, and D. M. Tullsen. Inteconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *ISCA '05: Proceedings of the 32nd International Symposium on Computer Architecture*, Madison, Wisconsin, June 2005.

[13] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Barcelona, Spain, November 2005.

[14] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 40–51, Göteborg, Sweden, 2001. ACM Press.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[16] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[17] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, Cambridge, Massachusetts, United States, 1996. ACM Press.

[18] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching, 2002.

[19] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *14th Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, St. Louis, Missouri, 2005.

[20] W. Zhang, B. Calder, and D. M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *6th Int'l Symposium on Code Generation and Optimization*, pages 50–64, New York, New York, 2006.

[21] W. Zhang, D. M. Tullsen, and B. Calder. Accelerating and adapating precomputation threads for efficient prefetching. In *13th Int'l Conference on High Performance Computer Architecture*, Phoenix, Arizona, February 2007.