Adaptive Prefetching for Shared Cache Based Chip Multiprocessors

Mahmut Kandemir and Yuanrui Zhang
Computer Science and Engineering Department
Pennsylvania State University
University Park, PA 16802
Email: {kandemir,yuazhang}@cse.psu.edu

Ozcan Ozturk
Computer Engineering Department
Bilkent University
Bilkent, Ankara, Turkey
Email: ozturk@cs.bilkent.edu.tr

***Abstract*—Chip multiprocessors (CMPs) present a unique scenario for software data prefetching with subtle tradeoffs between memory bandwidth and performance. In a shared L2 based CMP, multiple cores compete for the shared on-chip cache space and limited off-chip pin bandwidth. Purely software based prefetching techniques tend to increase this contention, leading to degradation in performance. In some cases, prefetches can become harmful by kicking out useful data from the shared cache whose next usage is earlier than the prefetched data, and the fraction of such harmful prefetches usually increases when we increase the number of cores used for executing a multi-threaded application code. In this paper, we propose two complementary techniques to address the problem of harmful prefetches in the context of shared L2 based CMPs. These techniques, namely, suppressing select data prefetches (if they are found to be harmful) and pinning select data in the L2 cache (if they are found to be frequent victim of harmful prefetches), are evaluated in this paper using two embedded application codes. Our experiments demonstrate that these two techniques are very effective in mitigating the impact of harmful prefetches, and as a result, we extract significant benefits from software prefetching even with large core counts.**

I. Introduction

Prefetching has been shown to be a very effective technique in improving performance by hiding memory accesses latencies. However, timing and scheduling of prefetch instructions is a critical issue in software data prefetching and prefetch instructions must be issued in a timely manner for them to be useful. If a prefetch is issued too early, there is a chance that the prefetched data will be replaced from the cache before its use or it may also lead to replacement of other useful data from the higher levels of the memory hierarchy. If the prefetch is issued too late, the requested data may not arrive before the actual memory reference is made, thereby introducing processor stall cycles. Software data prefetching (e.g., implemented through compiler-inserted explicit prefetch calls) is inherently less speculative in nature than its hardware counterpart. However, scheduling prefetch instructions is the key for the success of any software prefetching algorithm. The criticality of scheduling the prefetch instructions increases in chip multiprocessors prefetching data to the shared L2 caches, due to the additional possibility of negative interactions among different processor cores.

The latest versions of many architectures have chip multiprocessors (CMPs) with a shared L2/L3 cache [9], [12], [22]. In these CMPs, the cores compete for the cache as any other shared resource. In the context of CMPs with shared on-chip caches, existing compiler algorithms for scheduling software prefetch instructions and static techniques to compute prefetch distances may not be very effective. The L2 cache itself is the last line of defense before off–chip memory accesses in these systems and therefore achieving a high accuracy of prefetches is of critical importance. Apart from useless prefetches, the impact of harmful prefetches is also high in case of chip multiprocessors with shared L2 cache. We refer to a prefetch as *harmful* if the prefetched data is used later than the data it displaces from the on-chip cache. Note that, harmful prefetches can occur among the accesses made by a single core, or among the accesses from different cores.

We can summarize the main contributions of this paper as follows:

- We quantify the impact of harmful prefetches in the context of shared L2 based CMPs. Our experiments with two data-intensive embedded applications indicate that the effectiveness of compiler-directed prefetching drops significantly as we move from single-core execution to multi-core execution.
- We show that the contribution of harmful prefetches also increases with the increased number of cores. And, therefore, there is a direct correlation between the degradation in the effectiveness of prefetching and the increase in the fraction of harmful prefetches.
- We further demonstrate that harmful prefetch patterns change dramatically across the different phases of a given application, and in many execution phases, the total number of processor cores that are involved in harmful prefetches is relatively small.
- Based on these observations, we propose two dynamic and adaptive complementary techniques to mitigate the impact of harmful prefetches. These techniques, namely, suppressing select prefetches (if they are found to be harmful) and pinning select data in the L2 cache (if they are found to be frequent victim of harmful prefetches), are evaluated in this paper using two embedded application codes. Our experiments demonstrate that these two techniques are very effective in mitigating the impact of harmful prefetches, and as a result, we are able to extract significant benefits from data prefetching even with large core counts.

The rest of this paper is structured as follows. The next section briefly explains the software based data prefetching scheme used in this work, and Section III presents an experimental evaluation of it using two embedded applications. Our two optimization techniques are detailed in Section IV, and their impact in reducing harmful prefetches and improving performance is quantified in Section V. Possible extensions to our baseline approach are discussed in Section VI, and the paper is concluded in Section VIII.

II. Software-Directed Prefetching

The baseline software based data prefetching approach used in this paper is similar to that proposed in [20]. In this

```
int $a[0..N-1]$;
int $b[0..N-1]$;
int $c[0..N-1]$;
for $i = 0$ to $N-1$ {
    $c[i] = a[i] \times b[i]$;
}
```

(a)

```
int $a[0..N-1]$, $b[0..N-1]$, $c[0..N-1]$;
prefetch($a$, 0, $D$);      prefetch($b$, 0, $D$);
for $t = 0$ to $\lfloor N/D \rfloor - 1$ {
    prefetch($a$, $(t+1) \times D$, $D$);
    prefetch($b$, $(t+1) \times D$, $D$);
    for $i = 0$ to $P-1$, 1
        $c[t \times D + i] = a[t \times D + i]$
                $\times b[t \times D + i]$;
}
for $j = \lfloor N/D \rfloor \times D$ to $N-1$
    $c[j] = a[j] \times b[j]$;
```

(b)

Fig. 1. An example that illustrates software prefetching. (a) Original code fragment. (b) Code with explicit prefetch calls inserted. D represents the unit for prefetching.

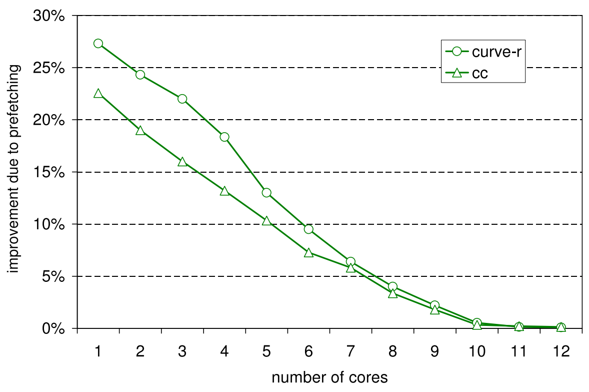


Fig. 2. Performance improvement due to software prefetching (original codes).

approach, prefetches are inserted into the code based on data reuse analysis. More specifically, an optimizing compiler analyzes the application code and identifies future accesses to data elements that are not likely to be in the data cache. It then inserts explicit prefetch instructions to bring such elements into the data cache ahead of time to ensure that data is in the cache when it is actually referenced. As a result, a successful data prefetch effectively hides the off-chip memory latency.

Figure 1 illustrates an example application of this prefetching scheme. In this example, three N-element arrays (a, b, and c) are referenced using three references ($a[i]$, $b[i]$, and $c[i]$). D denotes the block size, which is assumed to be the unit for data prefetching (i.e., a prefetch targets a data block of size D). Figure 1(a) shows the original loop (without any data prefetching), and Figure 1(b) illustrates the compiler-generated code with explicit prefetch calls embedded. The original loop is modified to operate on a block size granularity. As can be seen in the compiler generated code of Figure 1(b), the outermost loop iterates over individual blocks, whereas the innermost loop iterates over the elements within a block. This way, it is possible to prefetch a data block and operate on the data elements it contains. The first set of prefetch statements in the optimized code are used to load the first data block into the cache prior to loop execution. In the steady state, we first issue prefetch requests for the next set of blocks and then operate on the current set of blocks. The last loop nest is executed separately as the total number of remaining data elements may be smaller than a full block size.

We now briefly discuss the compiler analysis required for implementing this software based data prefetching in more detail. First, the compiler analyzes the given application code and predicts the future data access patterns. This is done using data reuse analysis, a technique developed originally in the context of cache locality optimization [29]. After that, potential cache misses are isolated through loop-splitting and prefetches are scheduled using software pipelining based on the data locality information generated by the compiler. In deciding the loop splitting point, the prefetching algorithm takes into account the estimated off-chip memory latencies as well (i.e., the time it takes to bring the data from the off-chip memory to the on-chip cache). In our implementation of this software prefetching algorithm, we also have a runtime layer that monitors the prefetch requests made by the application, and filters unnecessary prefetches as much as possible. This is done to minimize the cost of useless prefetch calls.

III. An Evaluation of Software Prefetching in Multicore Execution

In this work, we use two embedded applications. The first of these, called *curve-r*, is an image rendering code in which a sequence of images are captured by multiple cameras located at different positions along a curve. In this application, advanced radiosity estimation is coupled with a high quality ray tracing algorithm and therefore images exhibit convincing realism. The second application we use, called *cc*, is a motion compensation code that also uses compression. This application divides up the current frame into non-overlapping regions, and the motion compensation vector indicates where those regions come from.

We implemented the compiler-directed software prefetching algorithm explained in Section II, targeting the shared L2 cache of a CMP. We used the SUIF compiler infrastructure from Stanford University for this purpose (a Microsoft Phoenix based implementation is also underway). SUIF is an optimizing compiler infrastructure that can be used as a source-to-source translator. It consists of a small kernel and a suite of compiler passes built on top of the kernel. The SUIF kernel defines the intermediate representation, provides functions to access and manipulate the intermediate representation, and structures the interface between compiler passes. Our prefetch insertion algorithm is implemented as a standalone phase within SUIF. We observed that the impact of our prefetch implementation on compilation time was not too much (less than 8% for the two applications used in this work).

We used SIMICS [18], a full-system simulator, to implement our two schemes. In the simulated CMP architecture, the cores share a total on-chip L2 cache space of 2MB (with 8 cycle access latency), and each core (which is single-issue) has 16KB private instruction and data caches (each with 1 cycle access latency).

As stated earlier, we define a *harmful prefetch* as a prefetch that leads to the removal of a data block from the data cache and the prefetched data block is referenced only after the reference to the removed block.

Our goal in this section is to evaluate the impact of the software prefetching scheme explained above for our two embedded applications (curve-r and cc). The percentage improvements due to prefetching for different number of cores are given in Figure 2. These savings are with respect to the no-prefetch case. We see that the impact of prefetching suffers significantly as we increase the number of processor cores. For example, when we increase the number of cores from 1 to 12, the improvement the data prefetching brings drops from 27.3% to 0.09% in curve-r, and from 22.6% to 0.13% in cc. In fact, we can observe that the prefetching does not bring any tangible benefits beyond nine cores in our CMP. Figure 3

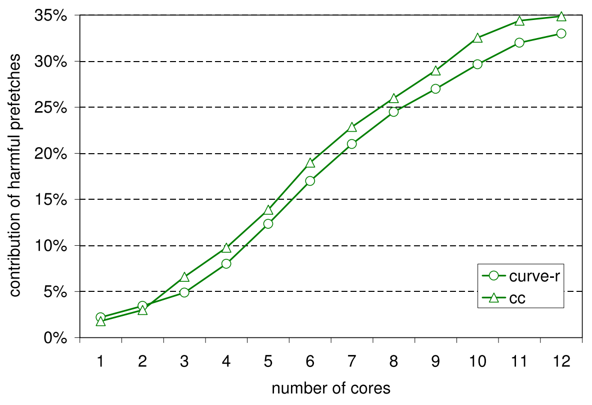


Fig. 3. Fraction of harmful prefetches.

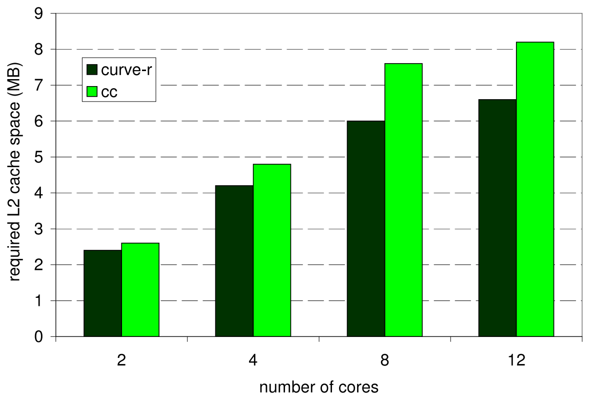


Fig. 4. Amount of L2 space required to achieve the same prefetch performance in the single core case.

gives the fraction of harmful prefetches as we increase the number of cores. We note from this plot that the contribution of harmful prefetches increases with increasing number of cores. Considering Figures 2 and 3 together, we can see a direct *correlation* between the increase in the contribution of harmful prefetches (to overall prefetches) and the reduction in the effectiveness of data prefetching.

The main reason behind the poor performance of prefetching, observed from Figure 2, when the number of cores is increased is the increasing pressure on the limited L2 capacity. We also performed an additional set of experiments where we recorded the required amount of L2 space (as the number of cores is increased) to *maintain the same prefetching performance as in the single core execution case.* Recall that the default L2 capacity used in our experiments in 2MB. The results of these experiments are presented in Figure 4 for core counts of 2, 4, 8 and 12. We see that, for example, when 8 cores are used, we need 6MB L2 cache space to achieve the same percentage benefits from the baseline software prefetching in application curve-r when it is used on a single core with 2MB cache space. That is, to maintain the same percentage benefits of prefetching, the L2 cache capacity should be increased by three times. Obviously, one cannot keep increasing the on-chip L2 capacity continuously. Therefore, in this paper, we take an alternate path where we try to optimize the performance of prefetching rather than increasing L2 cache capacity. We believe this is a particularly promising option for embedded systems where area concerns are an important factor.

In our next set of plots, we take a closer look at the patterns exhibited by harmful prefetches. We can divide harmful prefetches into two categories: *intra-core harmful prefetches* and *inter-core harmful prefetches.* In the first case, the prefetch discards the data used by the same core, and in the second case the prefetching core and the core that makes reference to the discarded data are different. To understand these intra-core and inter-core harmful prefetch patterns, we study four different scenarios plotted in Figure 5. In each of these plots, we illustrate how harmful prefetches happen. In these graphs, ”prefetching core ” is the core that performs the prefetch to the shared L2 cache and ”affected core” is the core whose data got removed from the cache as a result of this prefetching, i.e., whose data is the victim of a harmful prefetch.

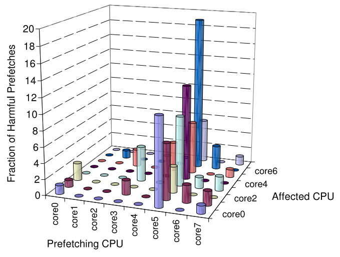
We start our discussion with Figure 5(a) which gives us the breakdown of harmful prefetches from an epoch at the beginning of execution of application curve-r. We see that one particular core (core5) causes an overwhelming majority of harmful prefetches (nearly about 74% of total harmful prefetches). An entirely different pattern can be observed in Figure 5(b), that belongs to one of the epochs toward the end of the same application. In this case, there is not a single culprit core that causes most of the harmful prefetches, but, there is a single (victim) core that is affected from a large majority of harmful prefetches (caused probably by different cores). Specifically, nearly 73% of total harmful prefetches affect core3.

The next two patterns in Figure 5 are taken from two different epochs of application cc that occur around the half way through its execution. In Figure 5(c), we can see a small clustering where a large majority of harmful prefetches involve three prefetching cores (core3, core4 and core5) and two affected cores (core5 and core6). Finally, in Figure 5(d), we see a pattern which looks like a combination of the two patterns shown in Figures 5(a) and (b).

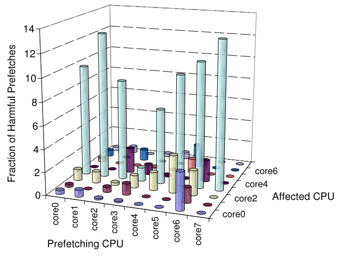
Overall, we can conclude from these results that harmful prefetches are *not* uniformly distributed across all available cores. In a given program execution phase, there is typically a small set of cores that are causing and/or being affected by the harmful prefetches. Another important observation is that the observed harmful prefetch patterns can change dramatically from one execution phase to another (consider for example the graphs in Figures 5(a) and (b) which belong to the same application). In fact, during our experiments with these two application codes, many other interesting harmful prefetching patterns were observed. If we can develop an adaptive scheme that can take advantage of these two observations, we may be able to eliminate a large fraction of harmful prefetches, without affecting useful prefetches.

IV. Our Approach

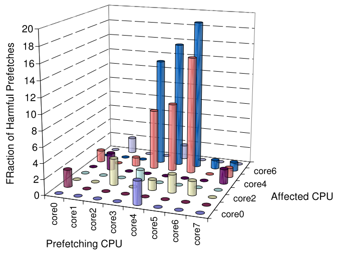
We propose two complementary techniques to cope with harmful prefetches in shared cache SPMs. Both of these techniques are *history based,* that is, the execution of the application is divided into epochs and the observations made during the execution of the current epoch are used to optimize the behavior of the next epoch. The first technique suppresses prefetches from select cores. Consider once more the pattern illustrated in Figure 5(a). In this figure, one core is responsible for most of the harmful prefetches. If this pattern is detected in epoch K, we can prevent this processor from issuing prefetches in epoch K+1. Assuming that the similar pattern will occur in epoch K+1 as well (i.e., assuming that there is a kind



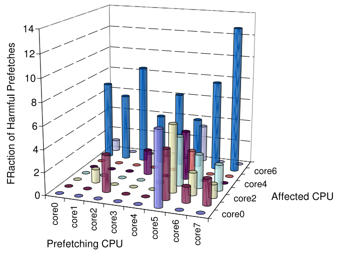
(a)



(b)



(c)



(d)

Fig. 5. Statistics collected at different points during the course of execution of our two applications. Each bar-chart shows a distribution of the harmful prefetches for the execution that uses eight cores. (a), (b) are from curve-r, and (c) and (d) are from cc.

of locality regarding harmful prefetches), we can expect this scheme to reduce a large number of harmful prefetches and hence improve performance. It has to be noticed however that it is very likely that this core is also issuing some useful prefetches. Therefore, there is an associated cost of suppressing prefetches issued by a core. To address this potential problem, our approach suppresses data prefetches from a core only if its contribution to the total harmful prefetches (i.e., sum of the harmful prefetches issued by all cores combined) exceeds a certain (preset) threshold. The suppressed core is allowed to prefetch in epoch K+2. In other words, we suppress a core if it really contributes to a large fraction of harmful prefetches, and we release it in the next epoch (later we will discuss what happens if we do not release it in the next epoch).

To implement this scheme, we have to keep track of the harmful prefetches issued by each core. Specifically, when a data block is prefetched into the cache, we record the block it discards, and then later check whether the prefetched block or the discarded block is accessed first. If it is the latter, we increase a counter (which counts the number of harmful prefetches) attached to the prefetching core by one. In addition to these core-local harmful prefetch counters, we also keep track of the total number of harmful prefetches using a global counter. At the end of each epoch, the contents of the local counters and the global counter are used for calculating the contribution of each core to the total number of harmful prefetches. The cores whose contributions to harmful prefetches are above a preset threshold value are prevented from issuing further prefetches in the next epoch. In addition, the counters (including the global one) are reset to 0 before the next epoch starts to ensure that we capture the dynamic variations in the behavior of the application.

Our second technique *pins* data blocks brought to the shared cache by select processing cores, and is designed to handle the type of patterns illustrated in Figure 5(b). In this technique, select data blocks brought to the shared L2 cache by a core are marked as non-removable (i.e., pinned in the shared cache) for a certain period of time. This helps those cores which are affected from harmful prefetches significantly (as illustrated in Figure 5(b)). We implement data pinning using a preset *threshold value,* similar to the one used in the case of our first technique. This time, however, we keep record – for each core – the fraction of cache misses it incurs because of harmful prefetches. This is done by employing a counter for each core that records the number of cache misses it incurs due to harmful prefetches and another counter which keeps track of all misses (across all processor cores) due to harmful prefetches. When, for a given core, the fraction of misses due to harmful prefetches exceeds the threshold value in epoch K, the data blocks brought by that core to the cache are pinned during the entire epoch K+1. In this case, when a prefetch tries to kick out a data block of this core, another victim (from another core) is selected, again based on the LRU policy. In other words, the new victim is the block that has not been brought into the cache by that core and has the lowest LRU value among all such blocks. As in the case of our first technique, the counter values are reset to 0 at the beginning of each epoch. Note also that the data blocks brought into the cache in epoch K+2 (by the core whose blocks are pinned in epoch K+1) are not pinned.

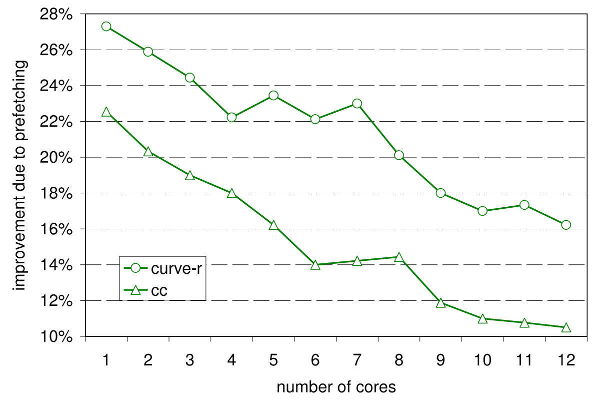


Fig. 6. Percentage improvements in execution cycles when our two optimization schemes are used with prefetching.

Note that many CPU architectures today provide performance counters and thus the proposed counters are not difficult to implement in practice.

V. RESULTS

The results presented below include the overheads incurred in maintaining the counters used by our schemes (e.g., counter updates, comparison operations, etc). The performance improvements brought by our techniques are presented in Figure 6 with varying number of cores. That is, this graph plots the benefits software prefetching brings (over the case with no prefetching) when it is supported by our two optimization techniques explained above. Comparing this graph with that in Figure 2, we see that our schemes boost the performance of prefetching significantly. For example, when 6 (resp. 12) cores are used, the percentage improvements brought by data prefetching to shared L2 supported by our two techniques are 22.1% (resp. 16.2%) and 14% (resp. 10.5%) with applications

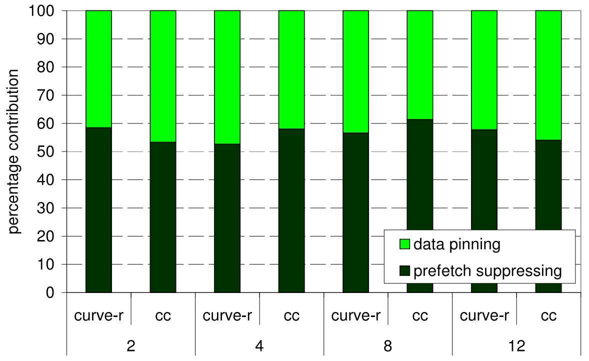


Fig. 7. Breakdown of the percentage benefits brought by our schemes. The results are given for core counts of 2, 4, 8 and 12 (on the x-axis).

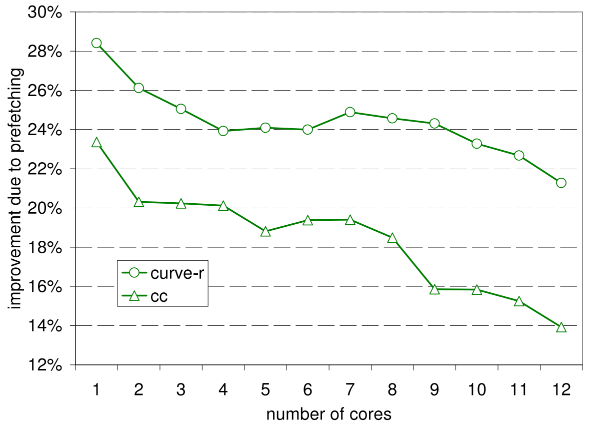


Fig. 8. Performance improvement due to software prefetching (with the fine grain version of our approach).

curve-r and cc, respectively. These results clearly demonstrate the impact of our techniques in rendering compiler-directed prefetching an effective approach in the CMP context.

We now present the breakdown of the performance benefits brought by our approach. In Figure 7, each bar represents the total benefits brought by suppressing prefetches and data pinning, and is set to 100. The lower portion of a bar gives the percentage benefits brought by suppressing select prefetches, while the upper portion captures those obtained through data pinning. We see that suppressing prefetches seems more beneficial than data pinning for these two applications.

VI. Discussion

Our two techniques that are designed to cope with harmful prefetches in shared L2 based CMPs can be considered as *coarse grain* as they both keep track of harmful prefetches and misses due to them from an individual core perspective rather than in a core-pair centric manner. In other words, if the prefetches of a core are suppressed, some useful prefetches issued by that core will also be suppressed, and this can affect the overall performance negatively in some cases. Similarly, when data blocks of a CPU are pinned, they are pinned against all prefetches from all processor cores. Some of these prefetches may in fact be useful and can improve performance if enabled. At this point, one can envision a *fine-grain* version of our schemes, which is oriented to address the problems associated with the coarse grain version explained above, we keep track of harmful prefetches and misses due to harmful prefetches for each core-pair. In a sense, we try to capture – at runtime – the information represented in bar-charts of the type shown in Figure 5. Note that this level of information can allow us to perform some detailed optimizations which could not be possible under the coarse grain implementation. However, we also note that this fine grain version requires more counters than the course grain version.

Figure 8 presents the results with this finer grain version of our approach. We see that these results are better than the corresponding results with the coarse grain version (Figure 2), and the gap between this fine grain version and optimal savings is not too high. Consequently, we can conclude that, if the resulting hardware (counter) increase and complexity can be tolerated, the fine-grain version can be chosen for maximizing benefits extracted from the compiler-directed data prefetching.

We now summarize the results obtained when a simpler prefetching algorithm is used. Recall that the baseline data prefetching approach used so far in our experimental evaluation is a compiler based one. As explained earlier in Section II, this approach makes use of data reuse analysis to identify the data blocks for which to issue prefetches and most suitable points in the code to insert explicit prefetch instructions. As a result, this software prefetching scheme is careful in inserting prefetches and this in turn helps minimize the number of unnecessary or useless prefetches. To quantify the effectiveness of our approach under a different data prefetch scheme, we also implemented a simpler prefetching scheme, whereby whenever a cache block is fetched (not through prefetching) from the off-chip memory to the on-chip cache, the next block in the memory is prefetched automatically. Note that this is a hardware-based data prefetching scheme. Clearly, as compared to the compiler based scheme, this simpler scheme can issue many more prefetches. We collected the results with this scheme as well. These results showed that our approach generates better savings with the simple prefetch scheme for all the core counts tested (on average 11% better as compared to the compiler-based prefetching case). The main reason for this is the fact that, as the number of prefetches increases, the percentage of harmful prefetches also increases. For example, although we do not present here in detail, for the 8 core case, when we move from the compiler based prefetching scheme to the simple scheme, we observed that the percentage of harmful prefetches increased by about 13% and 12% for our two applications. Since suppressing prefetches and data pinning target harmful prefetches, their effectiveness increases with the simple scheme (the hardware-based prefetching scheme).

As explained earlier, when our approach decides, during epoch K, suppressing prefetches or data pinning for epoch K+1, in the next epoch (K+2), these optimizations are disabled. However, this can potentially lead to miss some optimization opportunities. In particular, epoch K+2 (and possibly the next several epochs that follow it) could also benefit from the prefetch suppressing and data pinning decisions taken for epoch K+1. To test this, we performed another set of experiments where the decisions taken during epoch K are applied to epochs K+1 through K+s, where s is a parameter that can be changed. Note that in our experiments discussed up to this point s is set to 1. We performed experiments with 6 and 12 cores using the coarse grain version of our approach. Our results showed that, for both applications, the s value which generated the best results was 3. This can be explained as follows. As the value of s is increased, the percentage improvements first increase, but then beyond a point, they start to decrease. This means that the same harmful prefetch pattern lasts a few consecutive epochs but does not go beyond that. Therefore, setting the value of s to 3 seems to be the right choice for our two benchmarks. In our current work, we are investigating whether it is possible to determine the best s value dynamically during execution.

VII. Related Work

Data prefetching has received considerable attention in the literature as a potential way of boosting performance in uniprocessor systems and loosely-coupled multiprocessor systems. A comprehensive survey of most popular prefetching schemes for multiprocessors (also applicable to uniprocessors) is presented in [27]. This survey paper discusses various prefetching schemes including pure hardware [2], [10], [4], [30], [11], software [5], [8], [19], [20], [15] and integrated schemes.

Among the integrated schemes, Gornish et al [6] describe an integrated prefetching scheme, which is a variation of tagged hardware prefetching, in which the degree of prefetching for a particular reference stream is calculated at compile time and passed on to the prefetch hardware. To implement this scheme, a prefetching degree field is associated with every cache entry. A special prefetch instruction is provided that prefetches the specified block into the cache and then sets the tag bit and the value of the prefetch degree field of the cache entry holding the prefetched block. VanderWiel and Lilja [26] propose a prefetch engine that is external to the processor. This engine is a simple processor that executes its own program to prefetch data for the CPU. Through a shared second-level cache, a producer–consumer relationship is established in which the engine prefetches new data blocks into the cache, but only after previously prefetched data is accessed by the processor. The processor also partially directs the actions of the prefetch engine by writing control information to memory-mapped registers within the prefetch engine's support logic.

Our work can be thought of as an extension of single core based data prefetching to the CMP domain. Observing that harmful prefetches can be an important issue with large number of processor cores, this paper proposes and evaluate two techniques to cope with them.

VIII. Conclusions

Due to the huge and continuously increasing disparity between processor speeds and off-chip memory access latencies, data intensive applications that frequently exercise off-chip memory components tend to waste a disproportionate percentage of their execution times waiting for memory requests to complete. Data prefetching has been employed in the past as one of the mechanisms to hide memory access latencies. However, prefetching requires an accurate timing to be effective in practice. This timing problem is particularly problematic when multiple processors share the same on-chip cache (L2/L3) space, which is the case in some of the emerging chip multiprocessors (CMPs). In this paper, we (i) quantify the impact of software prefetching on shared L2 cache in a CMP; (ii) identify inter-core misses due to harmful prefetches as one of the main sources of this reduction in performance with increased number of cores; and (iii) propose and evaluate two complementary techniques to mitigate the negative impact of harmful prefetches. Our experiments with two data-intensive embedded applications reveal that the proposed techniques can improve the performance of a baseline software prefetching scheme significantly by reducing the number of harmful prefetches. In fact, our results show that the proposed two optimization schemes render software prefetching a very effective technique for CMP based execution platforms.

Acknowledgment

This research is supported in part by NSF Grants 0811687, 0720645, 0720749, and 0702519.

References

[1] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 1–10, July 2001.

[2] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, 1994.

[3] Y. Choi, A. Knies, G. Vedaraman, J. Williamson, and I. Esmer. Design and experience: Using the intel itanium-2 processor. In *Proceedings of the EPIC2 Workshop*, 2002.

[4] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, page 5663., St. Charles, IL, 1993.

[5] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the Intel IA-64 compiler. *Intel Technology Journal*, 1999.

[6] E. H. Gornish and A. Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. *International Journal of Parallel Programming*, 27(1):35–70, 1999.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach; Second Edition.* Morgan Kaufmann, 1996. HEN j2 96:1 1.Ex.

[8] T. Horel and G. Lauterbach. Ultrasparc-iii: Designing third-generation 64-bit performance. *IEEE Micro*, 19(3):73–85, 1999.

[9] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell. Exploring the cache design space for large scale CMPs. *SIGARCH Computer Architecture News*, 33(4):24–33, 2005.

[10] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

[11] A. Ki and A. E. Knowles. Adaptive data prefetching using cache information. In *Proceedings of the 11th International Conference on Supercomputing*, pages 204–212, Vienna, Austria, 1997.

[12] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, 2005.

[13] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[14] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the SUN ultrasparc CMP processor. In *Proceedings of the 38th International Symposium on Microarchitecture*, 2005.

[15] C.-K. Luk. Tolerating memory latency through software-controlled pre- execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, 2001.

[16] C.-K. Luk and T. C. Mowry. Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 182–194, 1998.

[17] C.-K. Luk and T. C. Mowry. Architectural and compiler support for effective instruction prefetching: a cooperative approach. *ACM Transactions on Computer Systems*, 19(1):71–109, 2001.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[19] T. C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems*, 16(1):55–92, 1998.

[20] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.

[21] X. Shi, Z. Yang, J. Peir, L. Peng, Y.-K. Chen, Lee, and Liang. Coterminous locality and coterminous group data prefetching on chip multiprocessors. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, 2006.

[22] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.

[23] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[24] L. Spracklen, Y. Chou, and S. G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *Proc. of International Symposium on High-Performance Computer Architecture*, February 2005.

[25] V. Srinivasan, E. S. Davidson, and G. S. Tyson. A prefetch taxonomy. *IEEE Transactions on Computers*, 53(2):126–140, 2004.

[26] S. P. Vanderwiel and D. J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings of the IEEE International Conference on Computer Design*, 1999.

[27] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[28] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 388–398, 2003.

[29] M. E. Wolf. Improving Locality and Parallelism in Nested Loops. *Ph.D. thesis,* Stanford University, Computer Systems Laboratory, August, 1992

[30] D. Wood and A. Alameldeen. Interactions between compression and prefetching in chip multiprocessors. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, February 2007.