Bundling: Reducing the Overhead of Multiprocessor Prefetchers

Dan Wallin and Erik Hagersten
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{dan.wallin, erik.hagersten}@it.uu.se

Abstract

Prefetching has proven useful for reducing cache misses in multiprocessors at the cost of increased coherence traffic. This is especially troublesome for snooping-based systems, where the available coherence bandwidth often is the scalability bottleneck.

The new ***bundling technique****, introduced in this paper, reduces the overhead caused by prefetching by two means: piggybacking prefetches with normal requests; and, requiring only one device on the "bus" to perform a snoop lookup for each prefetch transaction. This paper describes bundling implementations for three important transaction types: reads, upgrades and downgrades.*

While bundling could reduce the overhead of most existing prefetch schemes, the evaluation of bundling performed in this paper has been limited to two of them: sequential prefetching and Dahlgren's adaptive sequential prefetching.

Both schemes have their snoop bandwidth cut in about half for all the commercial and scientific benchmarks studied. The combined effect of bundling applied to these fairly naive prefetch schemes lowers the cache miss rate, the address bandwidth, as well as the snoop bandwidth compared with no prefetching for all applications – a result never demonstrated before.

Bundling, however, will not reduce the data bandwidth introduced by a prefetch scheme. We argue, however, that the data bandwidth is more easily scaled than the snoop bandwidth for snoop-based coherence systems.

1 Introduction

Many important applications spend a substantial part of their time waiting for memory transactions [5], [15]. A cache miss rate of 1 percent may add as much as 1.0 to the overall CPI number, assuming an access cost to memory of 400 CPU cycles and a 25 percent ratio of memory instructions. Much research effort has been devoted to reducing the number of cache misses using various latency-hiding and latency-avoiding techniques, such as prefetching.

While most existing prefetch techniques efficiently reduce the amount of cache misses, they also increase the amount of global coherence activity, which is a scarce resource in a shared-memory multiprocessor. This is especially true for systems based on snooping coherence, where each device has to perform a cache lookup for every global address transaction. The address networks of systems based on directory coherence are more scalable, since the address transactions are sent point-to-point. Still, systems based on snooping are often preferred due to their superior cache-to-cache transfer time. It should be noted that there is no difference in scalability of the data network between systems based on snooping coherence and systems based on directory coherence, since data packets can be sent point-to-point in both cases. Actually, commercial snooping-based systems have been built where the data network is over-designed with 50 percent compared to the available snoop bandwidth [6].

The goal of this research is to find a way to reduce the cache miss rate *without* increasing coherence activity. Our new proposal, *bundling*, piggybacks prefetch transactions on ordinary cache miss transactions. An alteration of the coherence protocol allows for a very selective snooping of the prefetch transactions, such that only one device needs to perform the snoop lookup. Bundling in combination with existing prefetch proposals may actually decrease the bandwidth of the address bus, and more importantly, also decrease the coherence activity (i.e., address snoops) compared with a system without prefetching.

This paper uses the simple but effective, adaptive sequential prefetching of read transactions, proposed by Dahlgren et al. [7], as a baseline system. The Dahlgren scheme is compared with a fixed sequential prefetch scheme and systems with various cache line sizes. Next, bundling techniques for read, upgrade and downgrade (i.e., write-back) transactions are introduced, followed by a study of the efficiency of bundled prefetch protocols. The bundled version of the original fixed sequential prefetch protocol has 54 percent less address snoops and 10 percent less cache misses on average for all studied applications. The bundled adaptive protocol reduces the address snoops with 47 percent and the cache misses with 5 percent compared with the original adaptive protocol. The data traffic is similar for bundled and non-bundled prefetchers. We also discuss the complexity of the proposal and provide a more detailed description of one possible implementation of bundling on an existing architecture.

We end the paper with a performance discussion of various prefetch protocols and make a comparison between the bundled prefetch protocols and non-prefetching protocols. Combining the adaptive prefetch scheme with bundling allows us to cheaply prefetch upgrade and downgrade transactions as well as read transactions, resulting in a miss reduction of 28 percent for the scientific benchmarks (SPLASH-2) and a miss reduction of 17 percent for the commercial benchmarks. It also *reduces* the snoop lookups by 25 percent and 16 percent respectively compared to a system without prefetching. It is interesting to note that we lower the miss rate *and* the snoop lookups for all studied applications.

While the evaluation section of this paper studies the effect of bundling on the Dahlgren sequential prefetching and pure sequential prefetching, it does not argue that these necessarily are the strategies of choice. However, it does make the case that various forms of bundling can help reduce the coherence overhead caused by most prefetch strategies. While bundling reads, upgrades and downgrades may seem obvious, it has not previously been proposed. Actually, we are not aware of any prefetch paper that have reported a *reduction* in the coherence activity for all studied applications.

2 Background: Multiprocessor Prefetching

Several prefetch techniques have been proposed for reducing the cache miss rate of multiprocessors. Software prefetching [18], [19], [20], [25] relies on inserting prefetch instructions in the code and results in an instruction overhead.

The hardware prefetching techniques [4], [7], [8], [12], [14], [16], [22] require hardware modification to the cache controller to speculatively bring additional data into the cache. Hardware prefetching often relies on detecting regularly accessed strides. A common approach to avoid unnecessary prefetches in multiprocessors is to make the prefetcher adapt the amount of prefetching at run time [12], [14], [22]. These proposals introduce small caches that detect the efficiency of prefetches based on the data structure accessed. Baer and Chen proposed to predict the instruction stream with a look-ahead program counter [4]. A cache-like reference predictor table is used to keep previous predictions of instructions. Correct branch prediction is needed for successful prefetching.

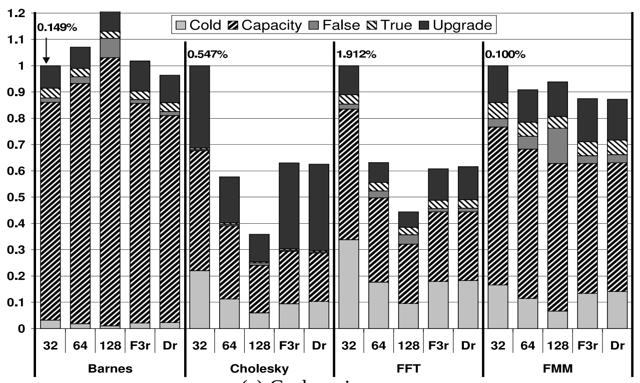
Another hardware prefetch approach is to take better advantage of spatial locality by fetching data close to the originally used cache line. A larger cache line size can achieve this. Unfortunately, enlarging the cache line size is not as efficient in multiprocessors as in uniprocessors since it can lead to a large amount of false sharing and an increase in data traffic. The influence of cache line size on cache miss rate and data traffic has been studied by several authors [9], [11], [13], [26], [27]. To avoid false sharing and at the same time take advantage of spatial locality, sequential prefetching fetches a number of cache lines having consecutive addresses on a read cache miss. The number of additional cache lines to fetch on each miss is called the prefetch degree.

Sequential prefetching in multiprocessors was first studied by Dahlgren [7]. The increased prefetch traffic tends to hurt multiprocessors more than uniprocessors. This is especially a problem in bus-based multiprocessors where the available snoop bandwidth is limited [24]. Dahlgren proposed two types of sequential prefetching schemes, a fixed version and an adaptive version [7]. Prefetches are only generated on read misses in these schemes and the focus on the study was on prefetching to the second level cache.

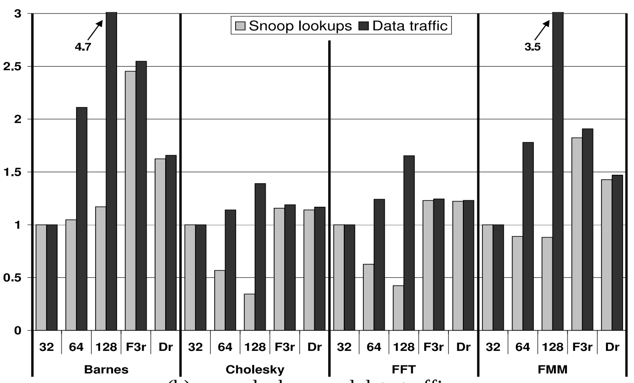
The fixed sequential prefetch scheme issues prefetches to the K consecutive cache lines on each cache read miss. If the consecutive cache lines are not already present in a correct state in the cache, a prefetch message for the missing cache line is generated on the interconnect. The prefetch degree K is fixed to a positive integer in this scheme. The fixed sequential prefetch scheme requires only small changes to the cache controller of the prefetching cache. In addition to this, a special prefetch request has to be handled by the interconnect and the memory system.

The adaptive sequential prefetch scheme is identical to the fixed sequential prefetch scheme, except that the prefetch degree K can be varied during run time. The prefetch degree is varied based on the success of previous prefetches. Dahlgren's approach to finding an optimal value of K is to count the number of useful prefetches. This is done by using two counters that keep track of the total number of prefetches and the number of useful accesses to prefetched cache lines. Prefetched cache lines are tagged for later detection. Every sixteenth prefetch, the useful prefetches are checked. If the number of useful prefetches is larger than twelve, K is incremented. K is decremented if the number of useful prefetches is lower than eight or divided by two if less than three prefetches are useful. The scheme also has a method of turning prefetching on, since no detection can be carried out if the prefetch degree is lowered so that no prefetches are performed.

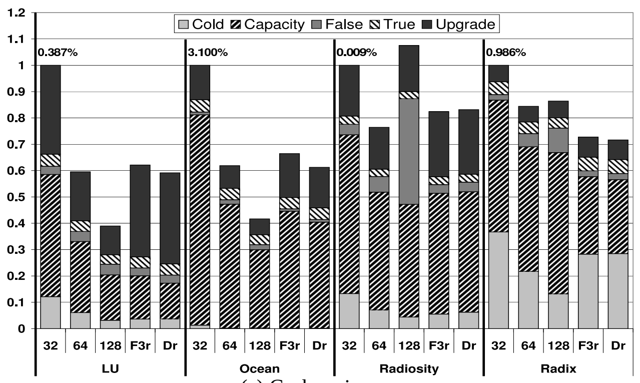
The behavior of fixed and adaptive sequential prefetch schemes on a single level of coherent caches is studied in Figure 1. The prefetch strategies are evaluated in terms of cache misses, snoop lockups and data traffic for fourteen commonly used scientific and commercial benchmarks. All results are normalized relative to the 32 B non-prefetching protocol. The numbers indicated in the cache miss figures are miss ratios, showing the percentage of cache accesses missing in the cache, for each application's 32 B configu-



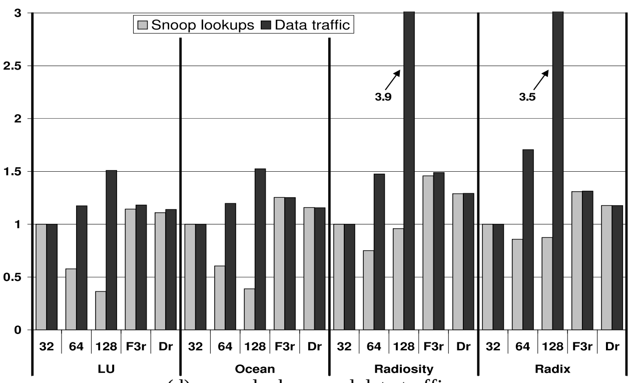
(a) Cache misses



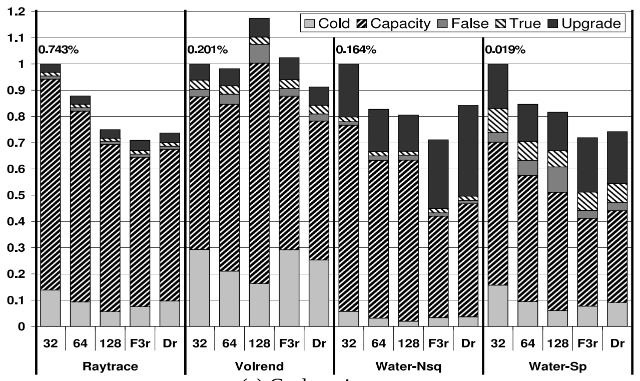
(b) snoop lockups and data traffic



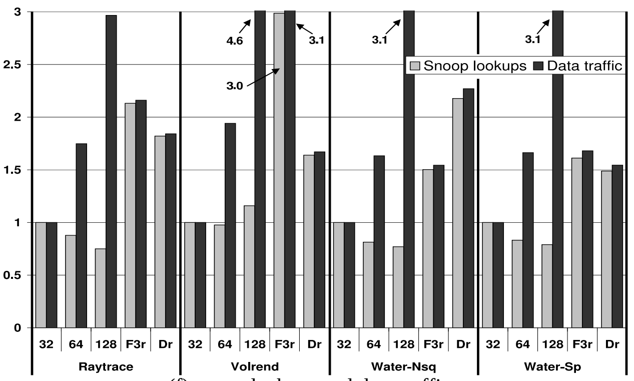
(c) Cache misses



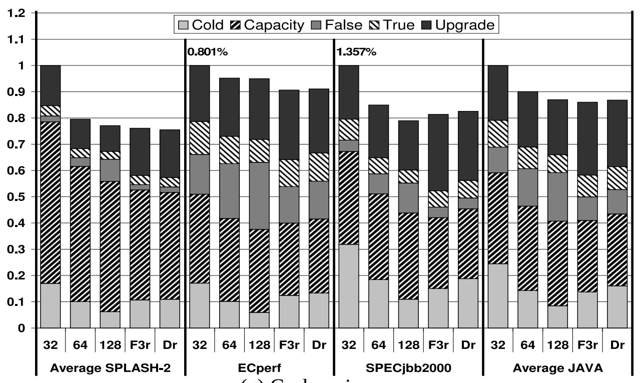
(d) snoop lockups and data traffic



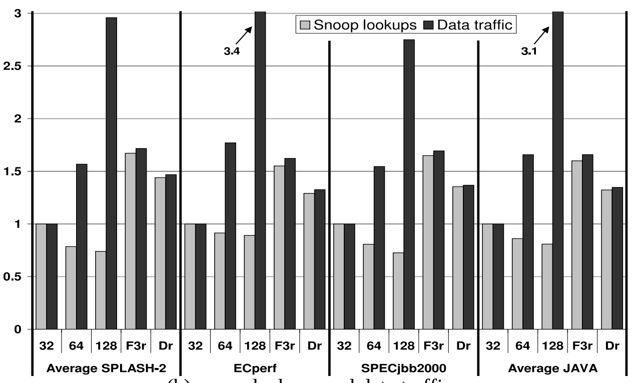
(e) Cache misses



(f) snoop lockups and data traffic



(g) Cache misses



(h) snoop lockups and data traffic

Figure 1. Cache misses, snoop lockups and data traffic for three different cache line sizes, *32*, *64*, *128*, a fixed sequential prefetch protocol with a prefetch degree of 3, *F3r*, and Dahlgren's adaptive prefetch protocol, *Dr*. The results are normalized relative to the 32 B non-prefetching configuration. The cache miss ratios are indicated for the 32 B configuration for each application.

ration. The adaptive protocol *Dr* is the one described by Dahlgren [7]. The abbreviation *Dr* indicates that this is the adaptive protocol proposed by *D*ahlgren, which only prefetches on *r*eads. The fixed sequential prefetch protocol *F3r* is a non-adaptive scheme that issues prefetches to the next three consecutive cache lines based on the address on all cache read misses. In addition to these protocols, the results of non-prefetching protocols are presented for various cache line sizes.

The original adaptive Dahlgren scheme *Dr* lowers the cache misses for all studied applications compared with the 32 B non-prefetching configuration. The fixed prefetch protocol *F3r* lowers the cache misses for all applications except Barnes and Volrend.

Both fixed and adaptive prefetching increase the data traffic and address snoops rather heavily compared with a non-prefetching protocol. For some applications, e.g. Barnes and Raytrace, the increase is very large. However, for most applications the adaptive strategy has a better prefetch efficiency and thus results in less address snoops and data traffic than in the fixed version. The difference between the snoop lookups and the data traffic is caused by upgrades, which results in address snoops but no data traffic.

By looking at the snoop lookups and data traffic figures, we can identify algorithms with good prefetch efficiency. For such algorithms, e.g. Cholesky, FFT, LU and Ocean, the access pattern is very regular and the adaptive algorithm works well thus generating only a small increase in data traffic. These algorithms also have good spatial locality since the cache misses decrease substantially when a large cache line size is used.

Comparing a sequential prefetch protocol of prefetch degree *K* with a protocol with a large cache line size, e.g. comparing *F3r* with the 128 B non-prefetch protocol, we see that the fixed prefetch protocol generally generates less data traffic than the non-prefetch protocol. Dahlgren also showed this. One reason for this is that false sharing can be avoided in the prefetch protocol and another reason is that the prefetch protocol always fetches subsequent addresses, which is not the case in a protocol having a large cache line size.

3 Simulation Environment

The Simics [17] full-system simulator is used in all the experiments. The simulation is execution-driven and models the in-order SPARC v9 ISA. We have implemented an invalidation-based MOSI (Modified, Owner, Shared, Invalid) protocol extension to Simics as a baseline cache coherence protocol. In all experiments, a bus-based 16-processor system with one level of 4-way associative unified data and instruction caches per CPU is modeled. Since our goal is to reduce the second level cache misses and we assume an inclusive cache hierarchy with a write-through first level cache, we have chosen to only model one cache level. A single level simulation will yield the same number of cache misses as for a two level hierarchy, but the miss-ratio will be lower since more read accesses will reach the cache compared with a multi-level cache hierarchy.

Many different classification schemes of cache misses in multiprocessors have been proposed [9], [10], [23]. The cache miss characterization in our paper is influenced by Eggers and Jeremiassen [10]: The first reference to a given block by a processor is a cold miss. Subsequent misses to the same block by the same processor are either caused by invalidations and/or replacements. All misses caused by replacements are classified as capacity misses. The invalidation misses are either classified as false or true sharing misses. False sharing misses occur if another word in the cache line has been modified by another processor during the lifetime in the cache. All other invalidation misses are true sharing misses. Conflict misses are included in the capacity miss category.

Throughout the paper the terms *snoop lookups* and *address snoops* are used as measurements of the number of cache lookups needed by the coherence activities. The available snoop bandwidth is the major limitation of scalability, since it grows rapidly with the number of processors. The *data traffic* reported represents the number of bytes transferred on the data network.

3.1 Benchmark Programs and Working Sets

The studies are performed on the SPLASH-2 applications [27] and two commercial workloads, SPECjbb2000 [3] and ECperf [1] (a modified version has lately been adapted as SPECjAppServer2001 [2]).

The cache size is chosen to model realistic data footprints. The SPLASH-2 programs are rather old benchmark programs with small data footprints. Therefore, the cache size for the SPLASH-2 simulations is chosen accordingly to only 64 KB. At this cache size, the number of communication misses (false, true and upgrade misses) and non-communication misses (cold, capacity misses) are roughly equal and the different cache miss categorizes could be observed and evaluated for all applications. If a larger cache size is used, the diagrams are entirely dominated by cold misses and with a smaller cache size, very few communication misses occur. The workloads are chosen according to the default values specified in the SPLASH-2 release [27] with some minor changes: the Cholesky benchmark is optimized for the cache size, the FFT benchmark is run with 65536 data points, the Raytrace benchmark allocates a total of 64 MB global memory, and the Radiosity benchmark uses the small test scene provided in the distribution instead of the default room scene in order to limit the simulation time. All benchmarks are run using 16 parallel threads, and the measurements are started right after the child processes are created in all applications except Barnes and Ocean, where the measurements are started after two time steps.

ECperf and SPECjbb2000 are both commercial Java-based middleware benchmarks. ECperf is a benchmark modeling Java Enterprise Application Servers that uses a number of Java 2 Enterprise Edition (J2EE) APIs in a web application. ECperf is a complicated, multi-tier benchmark that runs on top of a database server and an application server. SPECjbb2000 evaluates the performance of server-side Java. It can be run on any Java Virtual Machine. Both are commercial benchmarks, which set heavy demands on the memory and the cache system. The SPECjbb2000 and ECperf workloads are chosen according to Karlsson [15]. The commercial benchmarks have larger data footprints and therefore the cache size is chosen to 1 MB for these applications for a more realistic mixture of cache misses.

ECperf models the number of successfully completed "benchmark business operations" during a time period. Such operations include business transactions such as a customer making an order, updating an order or checking the status of an order. The ECperf transactions take long time and a total of 10 transactions are run with a 3-transaction warm-up period. SPECjbb2000 transactions take much less time and we simulate 50,000 transactions including 10,000 transactions of warm-up time.

4 Reduction of Address Snoops Through Bundling

Snooping protocols result in a much lower latency for cache-to-cache transfers than directory-based protocols. However, the available snoop bandwidth of snoop-based systems limits their scalability. Data packets do not need the broadcast capabilities and can be returned on a general network, such as a crossbar switch or a point-to-point network, and do not suffer from such limitations. An example is the architecture of Sunfire 6800, which has a data interconnect capable of transferring 14.4 GB/s while its snooping address network only can support 9.6 GB/s worth of address snoops [6], i.e., the data network is over-designed by about 50 percent. The main goal of our proposal is to limit the snoop bandwidth consumed by the address network, while the amount of bandwidth used in the data network is considered to be less critical.

Traditionally, hardware prefetchers, e.g. sequential prefetchers, send address transactions for the original cache miss as well as for all prefetch transactions. This typically creates more snooping in the state memory – the major limiting factor for the scalability of such system. The snoop frequency is limited by the rate at which new cache state lookups can start. The number of address transactions sent on the network can be significantly reduced if the original transaction, and its associated prefetch transactions are bundled into a single transaction. Each original transaction has to be extended with an offset prefetch bit mask indicating which extra cache lines to prefetch beyond the original one according to Figure 2. While this would reduce the number of address transactions on the bus, it would not reduce the number of snoop lookups each cache has to perform. However, if we alter the semantics of the prefetching slightly using bundling, it is possible to limit the number of caches that each prefetch transaction needs to snoop.

Transaction type	**Original address**	**Prefetch bit mask**

Figure 2. Bundled transaction.

Several hardware coherence protocols and software DSMs have used piggyback techniques to transfer multiple requests in one packet. None of these protocols have used piggybacking as a method of reducing the address snoops for prefetches. A simple form of bundling applied only to read transactions has previously been studied together with the capacity prefetching technique [26]. However, no evaluation of the possible performance gains of read bundling has previously been performed. This paper also extends the previous publication with a thorough discussion on how to implement bundled reads on an existing architecture, the SunFire 6800, as well as to also introduce bundling for upgrades and write-backs. Bundling is also studied together with an adaptive prefetch scheme. We show that bundling makes the studied prefetchers more efficient by reducing the cache misses and address snoops without increasing the data traffic.

4.1 Read Bundling

Some prefetch schemes may decide to prefetch K cache lines for each cache miss. The prefetch cache lines can either be to the consecutive addresses, or separated by some stride, depending on the prefetch scheme. However, in our study, only sequential cache lines are prefetched.

For any of the prefetch cache lines not present in the cache, a global prefetch request should be issued. However, instead of generating a new bus transaction for each requested cache line, we bundle the transaction caused by the cache miss with prefetch information. A single transaction is transmitted, consisting of the address A of the cache miss and information about the K prefetches encoded in a prefetch bit mask. All devices on the bus need to perform a snoop lookup for address A, but only the owner of cache line A needs to performs lookups for the prefetched addresses. It will reply with data for each prefetch address for which it is also the owner. Otherwise, an empty NACK data packet will be supplied for the cache lines. Since the state of the other caches are not affected by the prefetch transaction, they do not need to snoop the prefetch addresses.

The memory is also extended with a one-bit state: Owner. The Owner bit is cleared on a *ReadExclusive* request and is set again on a *Write-back* request. This will allow the memory controller to perform the bundling optimization as well.

4.2 Upgrade Bundling

Upgrade prefetch transactions can also be bundled to limit the number of snoop lockups required with some changes to the original invalidate MOSI-protocol. First, we have to introduce one additional flavor of the Owner state in the cache coherence protocol. Second, a similar extension has to be made to the memory states.

The two flavors of the Owner state, Owner$_2$ (Owner 2) and Owner$_m$ (Owner many) are used to keep track of how many shared copies that are available in the system. The first time a cache line becomes downgraded from the Modified state, the cache line will enter the Owner$_2$ state. In the Owner$_2$ state, we know that there is at most one other cache sharing the data. If additional read requests are issued to the same address by another processor, cache lines in the Owner$_2$ state will change their state to the Owner$_m$ state. Cache lines in this state can be shared by an unknown number of caches. At most one cache line, being in either the Owner$_2$ or Owner$_m$ state, is responsible for updating the memory on a cache line replacement. A transition state diagram can be found in Figure 4.

On each upgrade cache miss, i.e., a write request to state Shared, cache lookups are also carried out to the prefetch addresses. For each of the prefetch cache lines in a Shared state, a prefetch request will be bundled with the original upgrade request for the address *A*. Address *A* is snooped by all devices, possibly causing a cache invalidation. If a device has address *A* in the Owner$_2$ state, it will also invalidate each of the prefetch cache lines currently in the Owner$_2$ state.

Since cache lines in the Owner$_2$ state are shared by at most one other device, i.e., the requesting device, we know that the copy in the requesting device now is the only copy left. The device owning address *A* will send a reply to the requesting node indicating which of the bundled upgrade cache lines it now safely can put into state Modified.

Cache lines being invalidated in the Owner$_m$ state can not be handled the same way since we do not know the number of sharers. In this case, only the original address will be invalidated.

Upgrade bundling should work well in all programs where at most two processors share a cache line. This behavior occurs in programs experiencing migratory sharing, which has been identified as one of the major sources of global invalidations in multiprocessors [13].

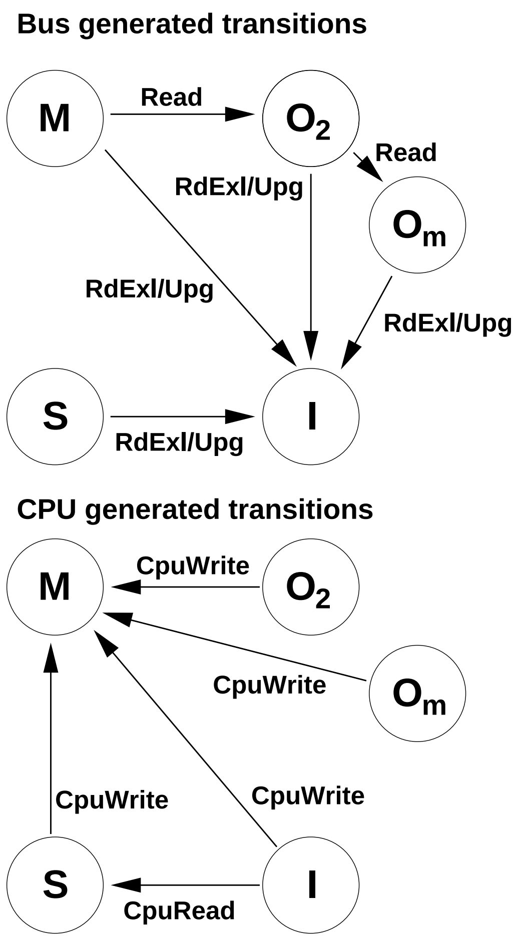
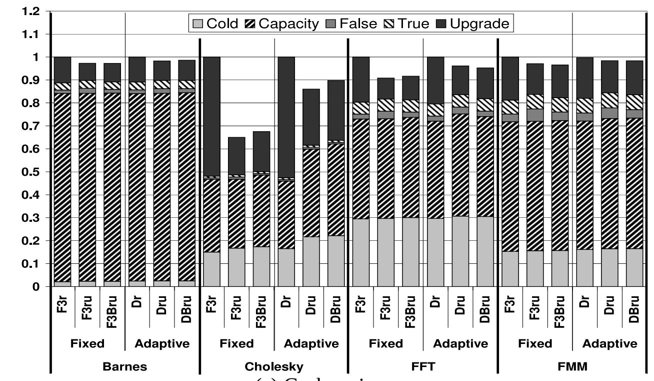


Figure 4. Transition state diagrams for bus and CPU generated transitions.

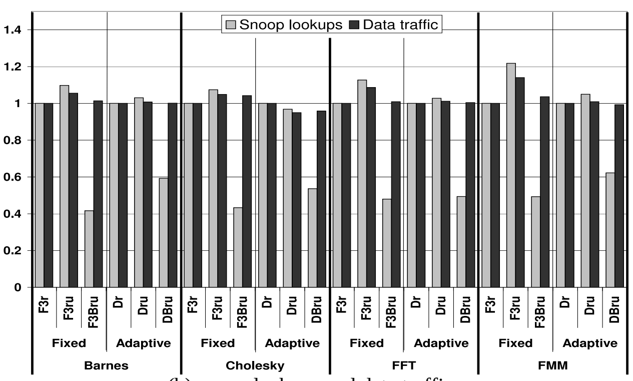
4.3 Downgrade Bundling

The last bundling technique is used for downgrade transactions. The downgrade bundling uses speculation on write-backs. On a write-back caused by a replacement from a bundled read request, the original cache line is put in the write-back buffer. The cache then checks whether the bundled prefetch addresses are in the Modified or Owner state in the cache. In that case, these cache lines also perform speculative write-back transactions and will be downgraded to the Shared state in the cache. This could potentially avoid future write-back requests for the cache line, but will still allow the cache lines to be read in the cache.

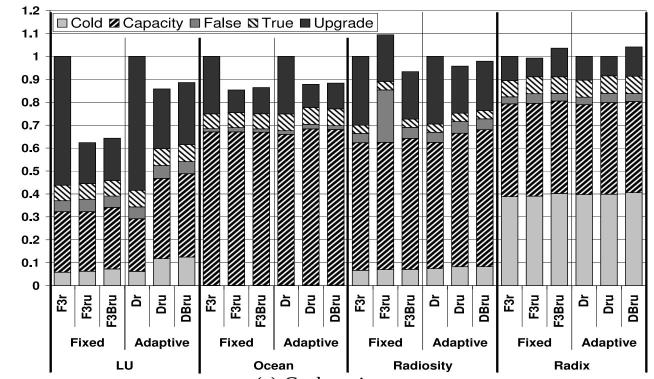
Downgrade bundling may increase the data traffic since data packets are speculatively written to memory. They may also increase the number of upgrade misses, for cases where the speculative downgrade was a bad choice. However, the cache capacity misses should not increase since the



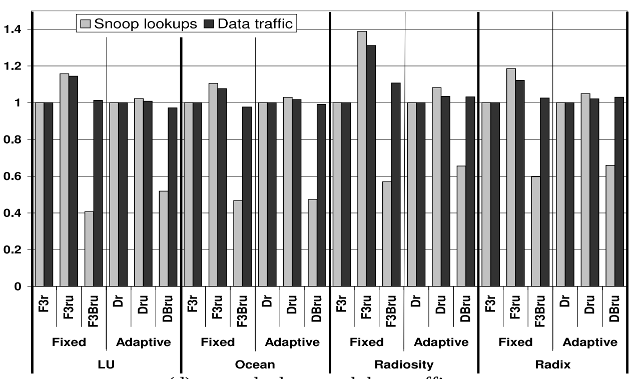
(a) Cache misses



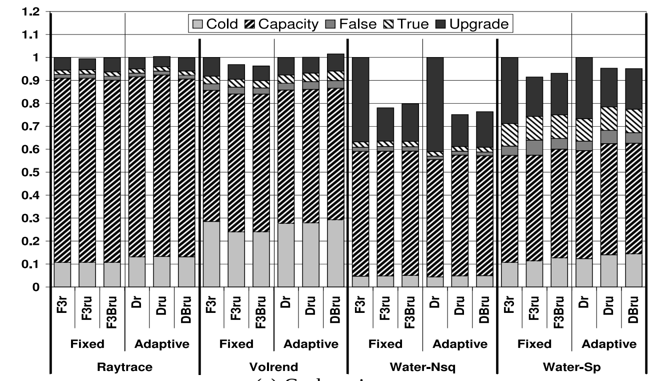
(b) snoop lockups and data traffic



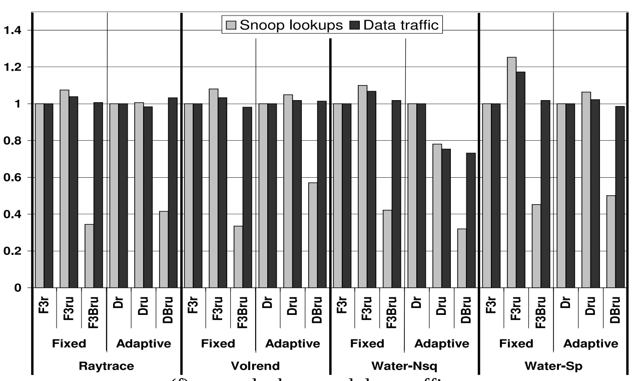
(c) Cache misses



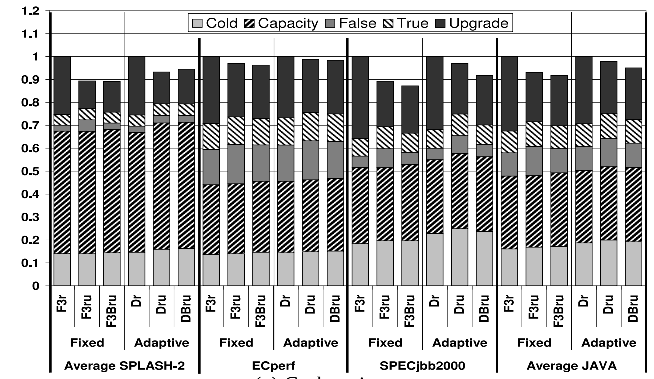
(d) snoop lockups and data traffic



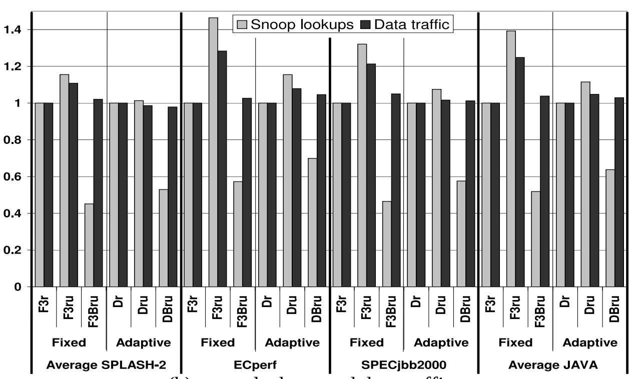
(e) Cache misses



(f) snoop lookups and data traffic



(g) Cache misses



(h) snoop lockups and data traffic

Figure 3. Influence of bundling on a fixed sequential prefetch scheme and the Dahlgren adaptive scheme. Cache misses, snoop lockups and data traffic for three fixed, *F3r*, *F3ru*, *F3Bru* and three adaptive, *Dr*, *Dru*, *DBru*, prefetch schemes are presented. The fixed schemes are normalized relative to the baseline fixed prefetch scheme *F3r* and the adaptive schemes relative to the baseline adaptive prefetch scheme *Dr*.

speculative downgrades do not invalidate the downgraded copies. The speculative downgrades may reduce the address snoops by sending a single address transaction for several data packets.

4.4 Efficiency of Bundling in Prefetch Protocols

Figure 3 shows the efficiency of bundling for the fixed sequential prefetch scheme and the Dahlgren adaptive prefetch scheme. The *F3r* configuration prefetches the three consecutive addresses on each read miss, while the *F3ru* configuration prefetches the three consecutive addresses on each read and on each write generating upgrades. Upgrade prefetching can be useful since many access patterns first read a variable and then update the variable with a new value. The *F3Bru* configuration is the bundled protocol prefetching three cache lines on each read and upgrade miss. The *Dr*, *Dru* and *DBru* are similar protocols except that they use adaptive prefetch degrees. The cache misses, data traffic and snoop lookups are normalized relative to the baseline fixed and adaptive schemes *F3r* and *Dr* in the figure. Hence, we can easily study the efficiency of the bundling proposals on the prefetch schemes.

The figure shows that in terms of cache misses, the bundled and non-bundled prefetchers prefetching on both reads and upgrades perform similarly for most applications. The behavior is similar both for the fixed and adaptive schemes. The cache misses are generally lower for protocols prefetching on both reads and upgrades than the baseline read prefetch protocols.

All bundled prefetch requests will not return data. Will not the effect on the cache miss rate be negative compared with non-bundled prefetching, since less prefetches are issued? There seems to be a fairly small difference. Actually, sometimes bundling seems to have a positive effect. If the owner of the original transaction is not also the owner of the prefetch data, this may indicate that they do not have a common history and do not belong to the same software object. Not prefetching could therefore be the action of choice. A small positive effect can be seen by comparing the miss rate for *F3ru* and *F3Bru* or *Dru* and *DBru* in ECperf and SPECjbb2000. A negative effect can be observed in Radix and Cholesky.

There are more cold misses in the adaptive protocol *Dru* than in *Dr* for some applications, e.g. Cholesky, LU , Water-Sp and SPECjbb2000. The reason for this is that useful prefetches are detected also for upgrades in the *Dru* scheme. Upgrades generally take advantage of a smaller prefetch degree than reads. This makes the *Dru* more restrictive at prefetching and causes the cold misses to increase in this protocol compared to the *Dr* protocol.

For most applications, prefetching leads to an increased data traffic when upgrade prefetches are issued. Looking at the data traffic for the non-bundled protocols prefetching only on reads and the bundled protocols, we can conclude that the difference in cache misses is very small. The bundled protocols issue prefetches also on upgrades but here the negative cannot be observed. This is an effect of the bundling since unnecessary prefetches for data belonging to separate software objects are avoided. Using bundling, prefetches can be issued on upgrades, thus decreasing the cache misses, without the negative effect on data traffic. There is also a positive effect of bundling on false sharing since the more restrictive prefetching prevents unnecessary prefetches in e.g. Radiosity, Water-Sp and SPECjbb2000.

The large difference between the bundled and non-bundled protocols is in address snoops. The bundled adaptive protocol *DBru* requires much less address snoops than the *Dr* and *Dru* protocols for all applications. The decrease in address snoops is 45 percent between *Dr* and *DBru* on average for all applications. The decrease is even larger compared with the non-bundled protocol prefetching also on upgrades. The fixed protocol always issues three additional prefetches on each miss. This makes the total number of prefetches much larger in this protocol. Bundling is therefore more efficient at reducing address snoops in this protocol. The average reduction is 54 percent between the *F3r* and *F3Bru* protocols. Generally, the bundling technique is more efficient at reducing address snoops in protocols which issue a large amount of prefetches per cache miss.

4.5 Details of the Bundled Snooping Protocol Implementation

While the implementation of upgrade and downgrade bundling is rather obvious, in the following section we will go further into an implementation of read bundling on a specific architecture. Although this section assumes a cache coherence implementation similar to that of Sunfire 6800 [6], read bundling implementations should be fairly similar in other modern snooping architectures too.

In Sunfire 6800, snooping cache coherence is implemented using logically duplicated cache state tags: the snoop state (OSI) and the access state (MSI). A similar scheme was also used in the Sun E6000 family of servers [21]. The action taken for each snooped transaction depends on the snoop state. A service request may be put in the service queue for the cache as the result of the snoop state lookup, e.g., an *Invalidate* request or a *Copy-Back* request. The snoop state is changed before the next address transaction lookup is started. Also when a cache snoops its own transaction, entries may be added to the queue, e.g., a *My-Read* request or a *My-ReadExclusive* request. Eventually, each service request will access the cache and change its access state accordingly. The cache's own requests are not removed from the head of the queue until the corresponding data reply is received and can thus temporarily block the completion of later service requests [21].

The UltraSPARC III processor, used in Sunfire 6800, implements the two logically duplicated states as a combination of the snoop result from the lookup in the main cache state and in the much smaller transition state table, which contains the new state caused by the snooped transactions still in the request queue. The hit in the transitional state table has precedence over a hit in the main cache state.

Bundled *Read* prefetches in Sunfire 6800 will only retrieve the data if the owner of the original transaction also is the owner of the prefetched data. All caches snoop the address of the original read transaction address in order to determine if they are the owner. Only the owner will add an entry in its service queue: a *Read-Prefetch* request. Thus, the snoop bandwidth will not increase for the other caches. When the transaction reaches the head of the request queue, it is expanded to behave like one *Copy-Back* request for each prefetch cache line. If the cache is not the owner of a prefetch line, it will reply with a null-data NACK packet to the requesting CPU, which will simply remove the *My-Read* request from its service queue. The requesting cache must assume a shared snoop state for each prefetched cache line when the original request is sent out. This may create false invalidate requests if the null data is returned from the owner cache.

The owner state bit must be added to each cache line in memory. The Enterprise 6800 already has some state (gI, gS and gO) associated with each cache line for other reasons [6]. The extra bits used for these states is retrieved by calculating the ECC code over a larger data unit, and come ”for free”. There is one such unused bit in memory that comes handy for the owner state bit. That bit should be cleared on the first *ReadExclusive* request to the cache line and set again on its *Write-Back* request and speculative downgrade requests.

4.6 Implementation complexity

Our experience from designing commercial shared-memory systems has taught us that much of the implementation complexity lies in the details of an implementation – often at a level of details far below the description found in research papers. Here, we will nevertheless try to carry out a complexity discussion at a higher level.

Read bundling will introduce most complexity of the three bundling schemes. That is also why we covered it in some more details in Section 4.5. While this kind of bundling will neither alter the core of the coherence protocol nor add new states, it will introduce more corner cases in its implementation, e.g., the invalidation of prefetched data that are NACKed. However, we feel that our detailed description would solve such corner cases at the cost of a reasonable amount of logic adjacent to the service queue.

Another major cost for read bundling could be adding the one Owner state to the memory if the existing system not already has a memory state associated with each cache line in memory. The upgrade bundling does add more state to the caches and memory. However, it does not alter the core coherence scheme, since both Owner states behave the same way from a global coherence point of view. Adding one state may force you to add one extra state bit, if there should not be any unused pattern using the current bits. The mechanisms to handle the corner cases for read bundling should be sufficient for implementing also upgrade bundling.

The downgrade bundling is the simplest of the three techniques to implement and will not introduce much extra cost for its implementation.

5 Bundling Performance Discussion

So what is the performance effects of bundling on a real system?

The net effect of prefetching on execution time can be shown to be either positive or negative depending on the parameters chosen for the simulation study. A system with plentiful of coherence bandwidth will be dominated by the positive effects of the lower cache miss rate, while the negative effects from increased queuing delay in the interconnect will dominate if the coherence traffic is close to the available bandwidth of the system.

The contention bottleneck makes it very difficult to estimate the potential performance gain of prefetchers based on simulation. In applications that spend a lot of their time waiting for memory transactions, the simulated execution time will vary very much depending on the bandwidth assumptions. For these applications, the wall clock time will follow the amount of cache misses if the available bandwidth is chosen large. If the simulated bandwidth is small, the execution time will instead depend on a combination of the required bandwidth and the amount of cache misses.

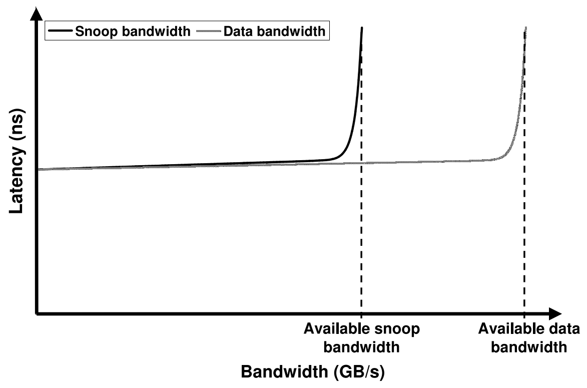
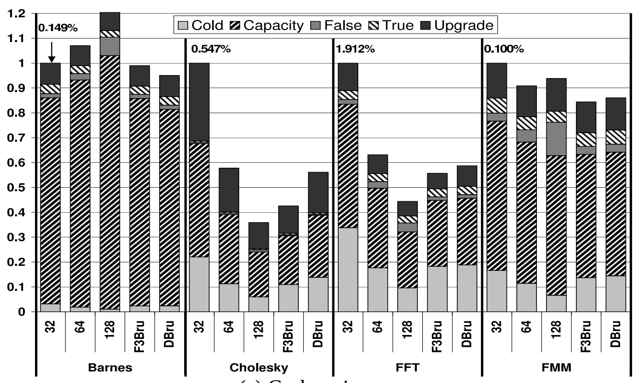
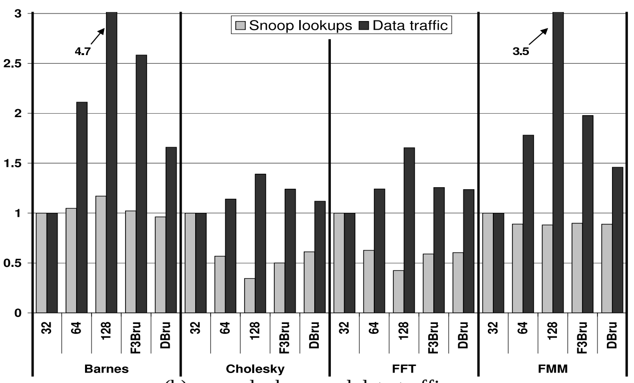


Figure 5. Effects of contention on latency.

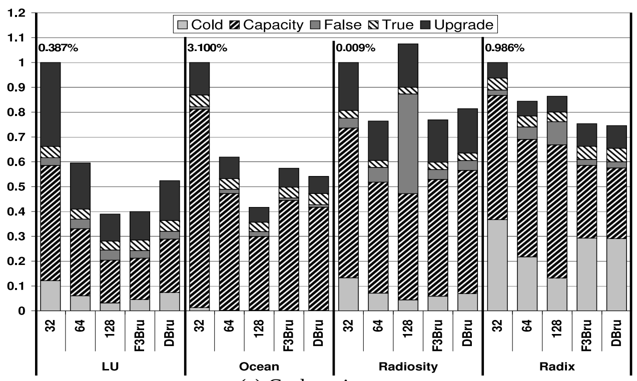
Singhal et al. performed measurements of the impact of contention on memory latency in the snoop-based Sun



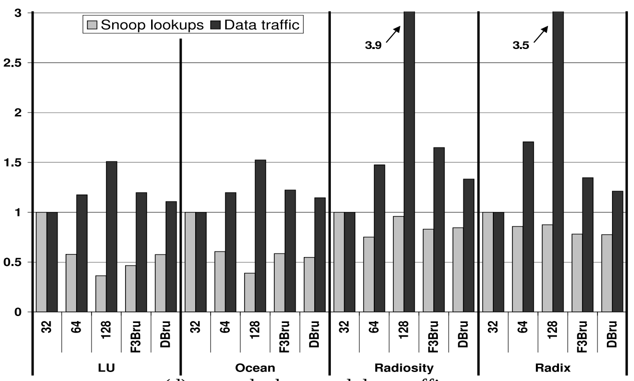
(a) Cache misses



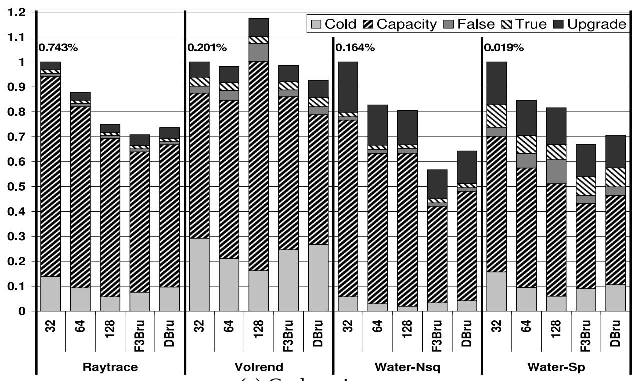
(b) snoop lockups and data traffic



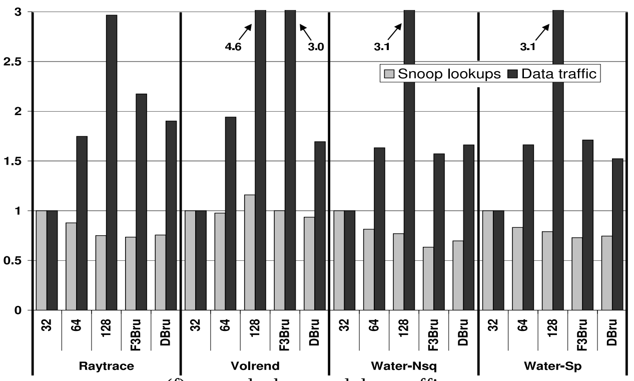
(c) Cache misses



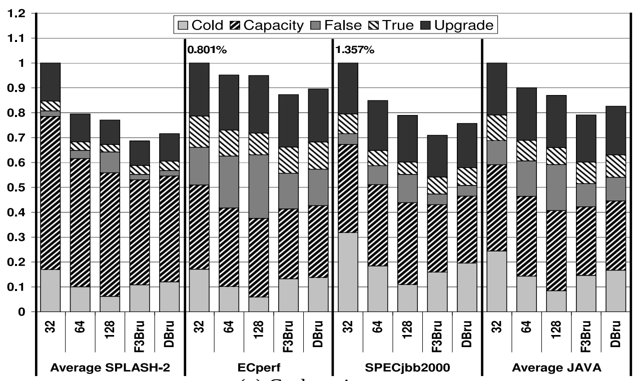
(d) snoop lockups and data traffic



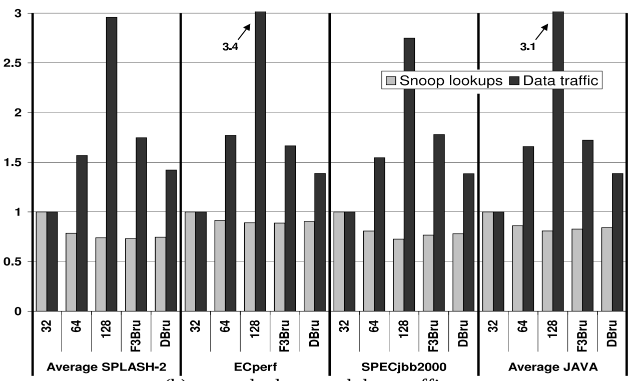
(e) Cache misses



(f) snoop lockups and data traffic



(g) Cache misses



(h) snoop lockups and data traffic

Figure 6. Cache misses, snoop lockups and data traffic for three non-prefetching protocols with different cache line sizes and the bundled fixed and adaptive sequential prefetchers. The results are normalized relative to the 32 B non-prefetching configuration. The cache miss ratios are indicated for the 32 B configuration for each application.

E6000 family of servers [21]. A similar diagram is presented in Figure 5. The figure shows that contention only has a modest influence on memory access time as long as the systems available bandwidth has not been reached. However, when the interconnect is contended, the access time increases largely. This makes it difficult to come up with one unbiased speedup number associated with a good-news/bad-news proposal, such as prefetching [1].

However, this paper is not about evaluating prefetching per se, but to evaluate the effects bundling has on prefetch algorithms. We have shown that bundling can cut the snoop bandwidth roughly in half and the cache misses by ten percent for the two baseline prefetch algorithms, while the data traffic is largely unaffected (Figure 3). We could easily have chosen simulation parameters to demonstrate a 100 percent speedup by limiting the snoop bandwidth. We could also have shown no speedup if the data bandwidth had been made the major bottleneck, or just a modest speedup if there is plentiful of both.

One could argue that we have chosen very primitive prefetch algorithms as our baseline systems, and this is why we can demonstrate the large cut in bandwidth. This may be a valid argument, which may lead us to compare the performance effects of bundling in combination with these primitive prefetch algorithms compared with the non-prefetching 32 B protocol. Looking for example at Ocean in Figure 6, its drop in miss ratio from 3.1 to 1.7 percent for *DBru* would result in a CPI reduction from 4.1 to 2.7 (34 percent) using the rough memory access latency numbers given in the introduction. If the available snoop bandwidth had been the dominating bottleneck, a speedup of 45 percent could have been achieved, and a data bandwidth bottleneck would have yielded a slowdown of 15 percent. In Table 1, similar values are presented for all the applications. The table shows that if the available snoop and data bandwidths are large enough, the prefetching scheme will reduce the execution time for all applications. Even larger speedup can be expected if the scarce resource is snoop bandwidth. If instead the data bandwidth is the limiting factor, the performance will decrease with the bundled adaptive scheme.

In Figure 6, three non-prefetching protocols with cache line sizes of 32, 64 and 128 B are compared to the bundled fixed and adaptive prefetchers, *F3Bru* and *DBru*. The figures are normalized relative the 32 B configuration since this is the cache line size used in the prefetching schemes. However, based on the simulation results for protocols of different cache line sizes, we can conclude that the most probable choice of an “optimum” cache line for the design of a multiprocessor using our benchmarks is 64 B. Using a 64 B cache line size, the cache misses and address snoops on average are significantly decreased compared with a 32 B cache line, while the data traffic is much smaller than in the 128 B configuration. The 64 B cache line size is also the design choice of the SunFire 6800 servers.

[1]Or rather, it is quite easy to come up with any desired number depending on what you would like to prove.

	Cache miss	Snoop BW	Data BW
Barnes	0.6	3.8	-65.9
Cholesky	18.6	38.8	-11.8
FFT	27.4	39.6	-23.6
FMM	1.2	11.2	-45.9
LU	13.4	42.4	-10.7
Ocean	34.1	45.3	-14.5
Radiosity	0.2	15.5	-33.3
Radix	12.3	22.5	-21.2
Raytrace	11.5	24.4	-90.1
Volrend	1.2	6.5	-69.4
Water-Nsq	5.0	30.3	-66.2
Water-Sp	0.5	25.5	-52.3
ECperf	4.5	9.8	-38.7
SPECjbb2000	13.8	22.0	-38.5

Table 1. The performance difference in percent between the bundled adaptive protocol *DBru* and the original non-prefetching 32 B protocol depending on whether the performance bottleneck is the number of cache misses, the snoop bandwidth or the data bandwidth. The cache miss bottleneck assumes 25 percent memory references and a 400 CPU cycle miss penalty.

The bundled adaptive prefetch configuration, *DBru*, has 10 percent less cache misses, 5 percent less snoop traffic and 10 percent less data traffic than the 64 B configuration on average for all applications. The bundled fixed prefetcher, *F3Bru*, reduces the cache misses with 14 percent, reduces the snoop bandwidth with 6 percent but increases the data traffic with 10 percent compared with the 64 B configuration on average for all applications.

6 Conclusion

Prefetching is useful for reducing cache misses in multiprocessors. Also rather small cache miss rates of less than 1.0 percent can harm the overall performance severely in multiprocessors as the gap between processor speed and memory access time grows. However, many prefetching schemes largely increase the address snoops and data traffic.

Snoop-based systems are generally limited by how fast snoop lookups can be performed. By using the three bundling techniques proposed in this paper for read, upgrade and downgrade transactions, the address snoops in prefetch protocols can be largely reduced. Bundling, lumps several snoop transactions together in a way that requires most of the caches to snoop only one of the transactions. We have investigated the efficiency of bundling in two different prefetchers, one fixed sequential prefetch scheme and one adaptive prefetch scheme proposed by Dahlgren [7]. Compared with the original adaptive Dahlgren proposal, the bundled adaptive Dahlgren protocol decreases the cache misses with 5 percent, the address snoops with 47 percent and the data traffic with 2 percent on average for all studied applications. Bundling is even more efficient in the fixed sequential prefetch protocol, where the average reduction in cache misses is 10 percent, while the address snoops is reduced by 54 percent and the data traffic increases 2 percent on average for all applications. Bundling requires only small changes to the coherence protocol.

We show that combining bundling with the adaptive scheme gives a protocol that reduces the cache misses with 10 percent, the snoop lookups with 5 percent and the data traffic with 10 percent compared with the most efficient non-prefetching protocol with 64 B cache line size on average for all studied applications.

This is the first prefetch paper that reports a *reduction* in the coherence activity (e.g. snoop lookups) as well as cache misses for all studied applications. Bundling is not limited to sequential prefetch schemes but could be used together with more sophisticated prefetch proposals.

References

[1] http://ecperf.theserverside.com/ecperf/.

[2] http://www.spec.org/osg/jappserver2001/.

[3] http://www.spec.org/osg/jbb2000/.

[4] J.-L. Baer and T.-F. Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the 1991 Conference on Supercomputing*, pages 176–186, 1991.

[5] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14. IEEE Press, 1998.

[6] A. Charlesworth. The Sun Fireplane System Interconnect. In *Proceedings of the 2001 Conference on Supercomputing*, 2001.

[7] F. Dahlgren, M. Dubois, and P. Stenström. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, 1995.

[8] F. Dahlgren and P. Stenström. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):385–398, 1996.

[9] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 88–97, 1993.

[10] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 377–381, 1991.

[11] S. J. Eggers and R. H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, 1989.

[12] E. H. Gornish. *Adaptive and Integrated Data Cache Prefetching for Shared-Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[13] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, 1992.

[14] E. Hagersten. *Toward Scalable Cache-Only Memory Architectures*. PhD thesis, Royal Institute of Technology, Stockholm, 1992.

[15] M. Karlsson, K. Moore, E. Hagersten, and D. A. Wood. Memory System Behavior of Java-Based Middleware. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, 2003.

[16] D. M. Koppelman. Neighborhood Prefetching on Multiprocessors Using Instruction History. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, 2000.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[18] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.

[19] T. C. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *ACM Transactions on Computer Systems (TOCS)*, 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 fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73. ACM Press, 1992.

[21] A. Singhal, D. Broniarchyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, and E. Hagersten. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of IEEE Hot Interconnects*, pages 41–52, 1996.

[22] M. K. Tcheun, H. Yoon, and S. R. Maeng. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the International Conference on Parallel Processing*, pages 306–313, 1997.

[23] J. Torrellas, M. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[24] D. M. Tullsen and S. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proceedings of 20th Annual International Symposium on Computer Architecture*, pages 278–288, 1993.

[25] D. M. Tullsen and S. J. Eggers. Effective Cache Prefetching on Bus-Based Multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 13(1):57–88, 1995.

[26] D. Wallin and E. Hagersten. Miss Penalty Reduction Using Bundled Capacity Prefetching in Multiprocessors. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.