Purdue University
Purdue e-Pubs

Department of Electrical and Computer Engineering Technical Reports | Department of Electrical and Computer Engineering

11-12-2013

Automatic Sharing Classification and Timely Push for Cache-coherent Systems

Malek Musleh
Purdue University, musleh@purdue.edu

Vijay Pai
Purdue University, vpai@purdue.edu

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

Musleh, Malek and Pai, Vijay, "Automatic Sharing Classification and Timely Push for Cache-coherent Systems" (2013). *Department of Electrical and Computer Engineering Technical Reports.* Paper 460.
http://docs.lib.purdue.edu/ecetr/460

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Automatic Sharing Classification and Timely Push for Cache-coherent Systems

Malek Musleh

Vijay S. Pai

TR-ECE-13-14

November 12, 2013

Purdue University

School of Electrical and Computer Engineering

465 Northwestern Avenue

West Lafayette, IN 47907-1285

Automatic Sharing Classification and Timely Push for Cache-coherent Systems

Malek Musleh*, Vijay Pai*,
*School of Electrical and Computer Engineering
Purdue University West Lafayette, IN 47907
Email: {musleh,vpai}@purdue.edu

***Abstract*—This paper proposes and evaluates Sharing/Timing Adaptive Push (STAP), a dynamic scheme for preemptively sending data from producers to consumers to minimize critical-path communication latency. STAP uses small hardware buffers to dynamically detect sharing patterns and timing requirements. The scheme applies to both intra-node and inter-socket directory-based shared memory networks.**

We integrate STAP into a MOESI cache-coherence protocol using heuristics to detect different data sharing patterns, including broadcasts, producer/consumer, and migratory-data sharing. Using 12 benchmarks from the *PARSEC* and *SPLASH-2* suites in 3 different configurations, we show that our scheme significantly reduces communication latency in NUMA systems and achieves an average of 10% performance improvement (up to 46%), with at most 2% on-chip storage overhead. When combined with existing prefetch schemes, STAP either outperforms prefetching or combines with prefetching for improved performance (up to 15% extra) in most cases.

I. Introduction

Cache-coherent shared-memory systems range from rapidly-scaling chip multiprocessors (CMP or multicore) to multi-socket nodes that form components of HPC clusters. Applications and systems are both increasingly scaling, but conventional implementations of cache coherence limit their performance by depending on request-response data transfers initiated by a data consumer. In contrast, having producers send data to predicted consumer nodes before the actual demand requests (referred to as *push* or *post-store*) can greatly reduce the latency of cache-coherent data transfers. Although studies of post-store have shown significant potential to improving communication efficiency, it has not become mainstream in real hardware. Prior research in post-store has demonstrated benefits for certain application classes [1], [25], [26], [16], [11], [13], [27], [12]. However, implementing post-store communication inherits many of the potential drawbacks of write-update protocols, such as cache pollution, increased network traffic, and scalability bottlenecks caused by unnecessary data transfers. Thus, accurate predictions are essential to minimize negative side effects.

This paper makes the following contributions:

- Describes a coherence mechanism to recognize different data sharing patterns and to initiate post-stores automatically based on the pattern. The mechanism works by predicting data producers and data consumers for certain blocks, and is general enough to cover producer/-consumer, migratory, and multiple reader-writer sharing patterns
- Provides an adaptive heuristic that dynamically adjusts its predictions of nodes involved in a communication pattern according to a stability-set threshold
- Presents an effective timing analysis that can be used by the protocol to determine *when* the post-store requests should be initiated
- outperforms the current state-of-the-art design
- outperforms static timing + sharer predictors (each alone representing strategies of previous work)

To the best of our knowledge, the system presented in this paper, called *Sharing/Timing Adaptive Push (STAP)*, is the first to both classify and optimize automatically for different communication sharing patterns using post-store communication for shared-memory platforms. We evaluate STAP on a gem5-based full-system simulation platform that models the Alpha instruction set architecture [6]. Our baseline CC-NUMA mesh network systems have 16 or 32 cores using quad-core chips connected in a mesh network, with each core having private 32 kB split I- and D-caches and the cores on chip sharing an 8 MB L2 cache. STAP sees benefits for most of the SPLASH-2 and PARSEC applications evaluated, with an average of 10% improvement in execution time and up to 46%. We also evaluate alternative hardware configurations, such as single-core per chip and fully on-chip multiprocessor.

To explore the improvement beyond previous push proposals that do not capture dynamic behavior as thoroughly as STAP, we also evaluate STAP with static timing prediction and STAP with only stable sharing prediction. In each case we find that dynamic timing and dynamic sharing detection combine to enable STAP to achieve additional performance beyond the strategies supported by previous work. We additionally compare STAP with the ARMCO on-chip latency-reduction strategy, finding that STAP can exploit certain opportunities missed by ARMCO even within the chip-level; and with multi-stream prefetching, showing that STAP either outperforms prefetching or combines positively with prefetching in most cases studied. We identify one source of negative interaction with prefetching and suggest methods to overcome that.

II. Background and Related Work

This section presents an overview of the relevant background and discusses some of the limitations on which our de-

sign aims to improve. Previous work falls into four categories: timely data prefetching, hybrid cache coherence protocols, producer-initiated primitives, and data-sharing set predictions.

Timely prefetching. Prefetching is ideally timed when the prefetch request is issued neither so early that the data is replaced or invalidated before the demand request or so late that the data does not arrive on time for the demand request. Mowry et al. proposed a selective prefetching algorithm to calculate the number of iterations to prefetch ahead considering memory latency and loop body length [21]. Our work inherits Mowry's distance derivations, but targets a different data communication mechanism (from producers to consumers) and must deal with incomplete knowledge of when the consumer will need the data. Other prefetcher studies attempt to improve prefetch accuracy by introducing the notion of relative time [29], [24]. However, they either refer to time in the context of miss-address correlation, or categorize prefetches into coarse-grain time-interval length: *short, medium, long*. Furthermore, their evaluation is limited to 1–4 cores, which limits the impact of larger-scale coherence-sharing effects and non-uniform memory access times.

Hybrid write-invalidate/update protocols. Write-update protocols are effective at reducing memory request latencies for applications where multiple readers receive data from a single writer, but also have much higher network traffic than write-invalidate protocols when a block is rewritten by the same writer before being read by another sharer. However, hybrid write-update & write-invalidate protocols have been shown to outperform strictly write-invalidate protocols for a variety of applications and data sharing patterns [14]. Unlike true update protocols, the competitive update protocol counts the number of times that a block has been updated without being used and then self-invalidates the block [18], [19] (so as to lose interest in future updates) once a competitive threshold has been reached to reduce the need for unnecessary updates. Our scheme also employs self-invalidation to minimize possible network congestion.

Producer-initiated communication primitives. Abdel-Shafi et al. provide a *Write-Store* instruction to initiate a post-store for specific data accesses [1]. Their work provides static and programmatic initiation of pushes without any automatic hardware detection of candidate blocks. This work thus places the burden of deciding pushes on the programmer and/or compiler. Further, the work does not discuss the timing requirements of the speculative push.

Destination-Set Prediction. Martin et al. implement a destination set prediction scheme to predict processors involved in stable sharing patterns for multicast snooping coherence protocols [20]. They implement four static predictor policies: owner predictor, group predictor, broadcast-if-shared, and hybrid owner/group, ranging from least aggressive to most aggressive prediction model. Our work not only predicts destination sets, but also provides a runtime-adaptable prediction scheme that is better suited for handling less stable sharing (e.g. dynamically changing communication patterns such as Producer/Consumer $\rightarrow$ Migratory). We also track multiple different sharing patterns between different addresses/cores within the same prediction policy, thus obviating the need for separate static policies for different workloads.

The ARMCO coherence protocol targets several sharing patterns, including Producer/Consumer, Data-Migratory, and multiple readers/writers by storing predicted block-owners in a separate L1-prediction table [15]. Their goal is to directly access the owner of the data instead of performing several long-latency network hops to the directory. However, ARMCO requires direct access between L1-caches (rather than via regular coherence mechanisms) and L1-predictor tables for intra-CMP latency reduction. Neither mechanism scales to multi-socket (CC-NUMA). First, scaling would require greater access times between L1-caches on different nodes – in contrast to ARMCO's key requirement and goal. Second, CC-NUMAs have multiple directory caches, typically one per chip, and it would not be straightforward to place this block-owner information onto each L1 cache in the system. In addition, bypassing the coherence protocol for speculative messaging increases the complexity of ensuring correctness.These factors hinder ARMCO's ability to scale or support systems with different topologies.

Other related works. Besides the works mentioned above, many previous sharing detection schemes individually only target a subset of the sharing patterns detected by STAP, such as Data-Migratory [11], [14], producer-consumer [8], [17], or false-sharing [27], [12]. The effectiveness of certain schemes rely exclusively on programmer instrumentation to push, increasing programmer effort [1], [16]. Others evaluate their schemes using shared-bus architectures and thus scale less efficiently than modern point-to-point interconnects [12], [13].

Some cache management/data-replication proposals improve data movement without specifically targeting data sharing patterns [28], [7], [3], [9], [10], [22]. However, they are typically limited to either replicating read-only data or else suffering from increased coherence penalties and cache pollution if they speculatively replicate data that is being produced (and for which replicas must be invalidated).

Summary. Although there have been various categories of related work described above, each work has limitations that prevent it from acting as a generalized strategy for scalable and time-sensitive hardware-initiated push across a variety of dynamic sharing patterns. For example, schemes for data-migratory sharing apply only to a subset of workloads and only while that specific type of sharing exists, even if other read-write sharing exists in different program periods. Our work seeks to address these limitations using minimal hardware support. Although our test configurations do not lend themselves to direct comparison against a specific related work, we will perform comparisons that show the impact of certain decisions made in previous work, as well as the accuracy and overhead of STAP in comparison to other works.

III. Design Overview

In this section, we present Space/Time Adaptive Push (STAP), a producer-initiated communication protocol to reduce latency in shared-memory systems. STAP incorporates push into a baseline invalidation-based protocol, deciding the who, what, and when of sending data speculatively when appropriate. Unlike previous push schemes, STAP answers each of these questions dynamically, detecting the sharing pattern, the nodes involved, and the time at which the consumer will desire the data. Unlike update protocols, STAP avoids unnecessary pushes when cores are no longer interested in data and chooses the timing of pushes so as to avoid the likelihood of a premature push leading to a later replacement. The remainder of this section presents the architecture of STAP-based systems, the communication patterns detected, and the components of the STAP system.

A. Architectural Overview

Our baseline platform is a CC-NUMA shared-memory system with multiple cache-coherent CMP nodes connected via a 2-D mesh network. Each node contains N cores, where each core has private L1 (both I and D) caches, and all cores share a unified L2 Cache. There are also directory caches at the four corners of the network. We specifically consider NUMA-4 (quad-core chips) and NUMA-1 (1-core chips) configurations. We also consider a CMP configuration in which all cores are on a single chip.

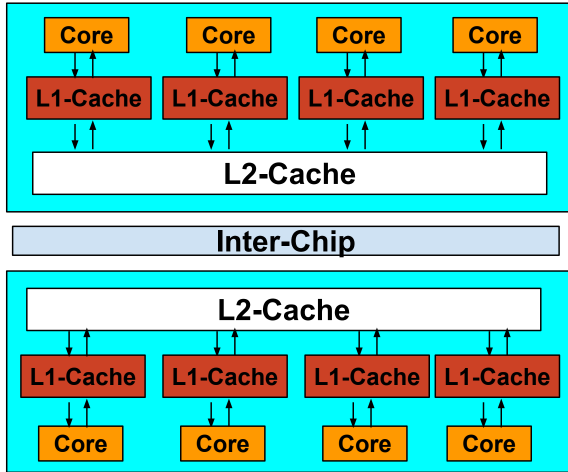


Fig. 1: NUMA-4 Layout

As illustrated in Figure 1, all coherence messages flow through the main on-chip and inter-socket interconnects. To ensure that any potential intra-chip communication latency-minimization is not missed and to minimize the need for memory-side requests from accessing the L1 Cache owner, we implement inclusion only for the specifically designated post-store tags.

B. Communication Overview

In order to facilitate low-latency fine-grain communication, we first attempt to recognize an application's sharing pattern. This assists the prediction scheme in selecting the data to send speculatively (*what*), the timing (*when*), and the destinations (*who*). STAP recognizes and predicts the following access patterns: 1) **Producer-Consumer** involves a single producer, and one or more consumer nodes. 2) **Broadcast** is a special case of producer-consumer in which all nodes (beside the producer) consume the data. 3) **Data-Migratory** has data elements repeatedly read and quickly modified by n different cores (most commonly $n = 2$ [4]). Exclusive ownership migrates among these n cores, requiring repeated coherence invalidations and acknowledgements to transfer exclusivity. In invalidation-based protocols, this type of sharing requires two separate requests during migration: a read-shared request followed by an exclusive request for write permission. However, once the pattern has been recognized, the write request can be speculated, coalescing the two requests into one and removing invalidations from the critical path [25]. 4) **Multiple Read/Write** arises whenever multiple nodes could be the writer or reader of a data section. This type of sharing pattern is in part an extension of migratory sharing where the object moves between different nodes. *False sharing* caused by the alignment of data structures can also cause this behavior. In typical invalidation-based protocols, the cache line bounces between caches, requiring a trip to the directory each time.

STAP does not attempt to handle Data-Migratory for N $>$ 2 cores, to avoid performance penalties from mispredicting highly-dynamic communication. However, as we will later discuss, STAP removes threads from the stability-sets once they are no longer contending, hereby decreasing the frequency of the un-handled case.

C. Required Components

In order to predict the data location and type of access pattern, STAP uses the following components: *Access-History Tags*, *Access-History Table*, *L1-PushBuffer*, and *Stability-Set Table*. Figures 2a-2b illustrate these structures' functionality.

Access-History Tags: This structure seeks to determine which cache blocks have true-sharing. L1-data cache lines are augmented with a 3-bit saturating repeat-counter. The repeat-counter can only start incrementing when a remote access occurs to a data block that has been locally written, or the cache requests data produced remotely.

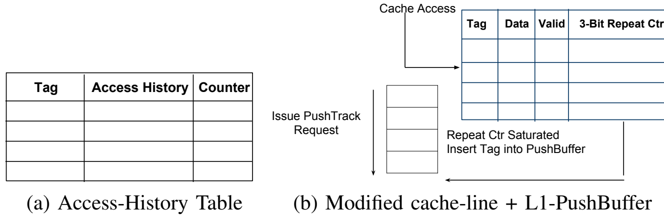
Once the counter has started to increment, a sharing pattern is confirmed only when it has saturated. In turn, the tag becomes a candidate for tracking and is placed in the push buffer to be sent to the lower level cache. The access history bits handle benchmarks that have a large fraction of replacements, in order to remember that a block exhibits read-write sharing. During replacement, the tag and current history state and counters are moved to the history table.

Access-History Table: Each processor has a local history table to record the current history access state of the victim cache block during replacements to avoid losing access information for blocks that have read-write history but have not yet been marked as push candidates. To reduce overhead, the same tag does not exist in both the data cache and history table simultaneously. During allocation of a new cache line, the table

is checked to see if an entry for the new tag is valid, indicating that the tag's access history should be copied from the table to the cache line, and subsequently removed from the table. This structure's cost is comparable to ARMCO's L1 predictor-table, with the exception that this is more scalable. This is because our table is meant to temporarily store information lost due to the replacement policy, whereas ARMCO's predictor table is meant to permanently store the predicted owner of a block. Thus, any adjustment of their table, such as increasing its size to support larger systems, has a direct impact on ARMCO's overhead. In contrast, our table need not grow to support larger systems (as we later illustrate), and there is little benefit to storing a longer history pattern.

Extending the cache-line by 3 bits does not increase the L1 critical path latency. Moreover, transferring a line from the history access table to the L1-cache only occurs during a miss, thus not increasing the critical-path access time of cache hits.

L1-PushBuffer: A small 5-entry buffer used to temporarily hold post-store candidates tags until they can be issued to the lower level caches. The L1-cache then issues a special control message, *Push Track Request* to direct the directory to begin recording this tag's sharing accesses.



(a) Access-History Table (b) Modified cache-line + L1-PushBuffer

Fig. 2: L1 Cache Modifications

L	Inter-Chip Latency between Src & dest
l	Intra-Chip Latency
κ	Scalar Factor for non-static network latency
$TTFW$	Time to First Write
$TTFR$	Time to First Read
δ	Time to Send Interval
r	Maximum number of Retries

TABLE I: Parameter Definitions

Stability-Set Table: This structure tracks the cores whose confidence factor remains above the sharing pattern stability-set factor (called *active cores*), and subsequently decides whether to issue push requests. Once the directory receives the Push Track request from the higher level cache, it begins to record the stability sets of the workload in this table. Each entry consists of $2 \times P$-bits to indicate which cores are currently part of the stability set. STAP tracks two stability sets: readers and writers. Each entry also contains a 3-bit sharing-type field, last-op bit, indicating whether the last operation performed was a load or store, and a valid bit. The sharing-type field records the type of sharing pattern predicted, which determines the confidence factor used in maintaining active cores in the stability-set. The last-op bit helps determine whether the speculative request should be exclusive (producer given exclusive ownership and sharers invalidated) or shared (producer sends data to consumers) mode. If a directory tag is marked for invalidation, then the corresponding table entry is subsequently deleted as well. The system predicts the sharing classification based on the number of readers and writers, their intersection, and the last operation type (load/store). We experimented with a range of table sizes and found that a 84kB table augmented to each directory cache provides reasonable performance.

IV. Protocol and Implementation

This section explains how STAP addresses the critical concerns for a push protocol: destination prediction *(Who)*, cache tag prediction *(What)*, and push distance calculation *(When)*. Additionally, this section discusses the confidence factor of push, and how STAP deals with network limitations. Algorithm 1 and Figure 4 illustrate how STAP makes a prediction to minimize the critical path latency.

A. *Destination Prediction (*WHO*)*

Effective push requires an accurate prediction of the expected consumers, but this is non-trivial for diverse, dynamic sharing patterns. To facilitate thresholding for consumer prediction, we calculate a *Stability* factor for each core in the respective reader/writer sets. Following the methodology of Barrow-Williams et al., the stability value for a consumer represents the ratio of updates to a given address that are used by this consumer compared to the total number of updates to that given address [2]. A stability factor of 1 indicates that this node consumes all updates to an address, while a value of 0 indicates that this node never consumes updates of this address. Utilizing a stability factor in predicting destinations provides greater resilience to minor changes in sharing accesses than simpler prediction schemes, while providing sufficient flexibility for varying the aggressiveness of pushing.

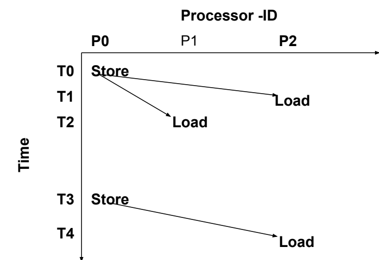


Fig. 3: Example Stability Set illustrating unstable sharing set. Procs.(1-2) both consume the 1st value produced by Proc. 0, while only Proc. 2 consumes the 2nd produced value. The resulting confidence factors for Proc. 1 and Proc. 2 are 50% and 100% respectively.

Choosing an appropriate stability threshold is critical: too high reduces the possibility of issuing pushes if the sharing stability never exceeds an absolute threshold value even for true consumers, but too low may degrade performance through cache pollution and network congestion induced by unnecessary communication.

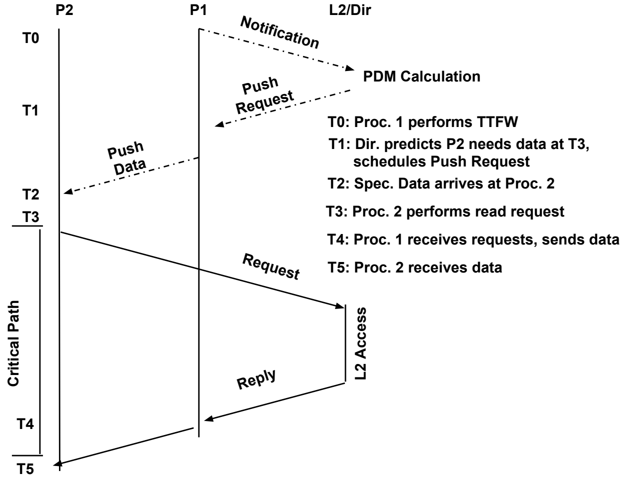


Fig. 4: Illustration of messages and activities of STAP in contrast to fetching from the directory.

Algorithm 1 STAP - Algorithm

1: T_{now} := current Time
2: PD := PushDistance
3: κ := kappa Factor
4: $T_{NR} = (PD + T_{now})$ ▷ Time of Next consumer read
5: D_{AB} := Shortest latency path nodes A $\leftrightarrow$ B
6: T_{sched} := Time to schedule push request
7: **procedure** STAP($TTFW, TTFR$)
8: L1-cache marks poststore-candidates tags
9: *PushTrack Request* issued to directory
10: Directory issues *Upgrade to Push State* to Caches ▷ 'S' -¿ 'SP'
11: Owner of Push Block (OP) sends TTFW to Directory
12: New Sharers (SP) send TTFR to Directory
13: **if** TTFW $\neq 0$ && TTFR $\neq 0$ **then** ▷ TTFW & TTFR have been recorded
14: calculate T_{NR}, PD
15: **if** $T_{NR} \leq PD$ **then** ▷ T_{NR} will occur soon
16: $T_{sched} = T_{now} + 1$
17: **else**$T_{sched} = T_{NR} - D_{AB} * \kappa$
18: **end if**
19: **end if**
20: **return** T_{sched}
21: **end procedure**

The common-case stability of a given set varies greatly depending on the sharing pattern, so it is ineffective to choose a single global stability threshold for all communication patterns. Instead, each communication pattern is assigned a static threshold value (determined via detailed experimentation) that a given core in the stability-set must exceed before it can be satisfied with a speculative push. By first classifying the sharing pattern, the complexity of implementing an accurate fine-grain destination-set predictor is reduced to a simple boolean check. This provides for a low-cost, but much-more fine-tuned and accurate mechanism than more general previous work [20]. Figure 3 depicts a simple example of the defined stability set for a Producer/Consumer communication pattern.

For less stable sharing behavior, the stability factor for any given node may not remain fixed. However, arbitrarily increasing the number of tracked accesses has negligible impact on the stability factor calculation. In fact, a larger depth may result in mispredictions as it becomes slower to detect changes in sharing behavior.

To avoid sending data to nodes no longer part of a communicating set, we implement a self-invalidation scheme and notify the home directory of the occurrence [18], [19].

*B. Cache Tag Prediction (*WHAT*)*

In order to determine *What* cache blocks to initiate post-store on, we implement a modified version of the directory history table used in the *2-level-Pattern Based Consumer Set Predictor* (2-PCSP) scheme of Somogyi et al. [23]. The 2-PCSP scheme maintains two tables at the directory level: 1) a history table of read/write sequences for each cache block, encoding entries to be either read or write sets; 2) a signature table that maintains the predicted consumer set using a 2-bit confidence counter. Similarly, we maintain a *Stability-Set Table* and initiate post-store communication from the directory. While we also utilize a 3-bit counter to select cache tags for post-store prediction, we do so at the L1-cache level instead. Figure 2a-2b illustrates the L1-Cache modifications.

Our reasoning for relying on the L1-cache to initially select prediction candidates instead of the directory cache is as follows. First, not all requests are sent initially to the directory cache. In certain situations the intermediate cache will forward the request to the owner or other local sharer (if known). Thus, the directory cache may not realize that another node had requested that block until it receives a revision message later. Second, it is more desirable to be able to detect block-level sharing as soon as possible to maximize the opportunity for speedup. Relying on an off-chip directory cache is likely to reduce the window of opportunity for pushes. The only issue of relying on an L1-cache to detect sharing is the increased likelihood of cache replacements. However, this issue is addressed by augmenting the L1-cache with a history table that serves as a temporary storage of a cache line's history access in the following scenarios:

noitemsep

- Data was produced by the local core
- First read of data produced by another core
- Current history size > 1
- 3-bit sharing-counter $>= 1$

To reduce the history length requirement, each line only records unique accesses (i.e. read request from logical core 0 and logical core 1 count as two unique accesses but redundant read requests from the same logical core count as 1 as long as the data has not been modified), and does not record loads performed by the local core. Our experiments have shown that the length of the history tags need not be more than a few bits, further reducing the storage overhead. Furthermore, to initially detect *any* communication sharing, it is enough

to simply compare the number of remote reads instead of recording detailed history pattern information. Once certain tags have been marked for speculation, the directory cache *then* begins to record the sharing pattern for the cores involved.

A large history table with high associativity may hold old information, which increases the likelihood of mispredictions. Our experiments with different predictor table sizes found that a 2-way set-associative 8 kB history table works well.

*C. Push Distance Calculation (*WHEN*)*

The directory cache has the most global perspective of coherence states, so it can perform the most cost-effective prediction of push timing. As long as the timing prediction takes into account the network latency of sending a message from the directory *to* the specified destination, then the difference in latency savings between prediction at a higher-level cache versus the directory is negligible. Furthermore, we must consider the required latency for the memory system to be able to effectively transfer data from any one node to another. Thus, each directory maintains a look-up table of inter-node latencies between each of the nodes in the system. In predicting *when* to Push, the directory uses the variables listed in Table I. We first define the *Time to Send Interval* (δ) of when the data can be sent using the time between when a new owner performs a write ($TTFW$), until the time the first sharer requests a copy ($TTFR$).

$$\delta \quad = \quad TTFR - TTFW \tag{1}$$

Once we determine when the first consuming node will need the data, we can expect the other sharers to issue coherence requests soon afterwards. To transform the theoretical upper bound to a more realistic one that accounts for system dependent artifacts. System dependent artifacts include intra (l) and inter-chip (L) latencies, as well as the potential for congestion along a predetermined routing path.

Using Equation (2), we can approximate the estimated distance in cycles that it will take to send data from Producer A to Consumer B.

$$Push\ Distance \quad = \quad L + (l \times \kappa) \times \delta \tag{2}$$

This range is dynamically fine-tuned during run-time via the adjustment of a scalar κ factor. This factor accounts for any additional current system latency, caused by increased network congestion or other pending requests to the same cache block. If it is determined that the scheduled Push has arrived too late (i.e. the consumer has already initiated the coherence request), a Push Nack is returned to the directory indicating so. In turn, if the scheduled Push requests continue to fail, κ is adjusted accordingly. If the Push request failed because it was issued too early, the directory applies an exponential backoff retry of up to a maximum of **r** times. r should be high enough to allow push opportunities despite early predictions but small enough to prevent excessive retries on failure.

Experimentation suggests that values of $\kappa \in [1, 5]$ and $r = 3$ seem to work well. The dynamic adjustment of κ allows the system to adapt to varying run-time conditions in the network and memory subsystems instead of relying on a static value from the start time.

Effective speculative push also requires certainty that the producing cache has completed its computation. In the simplest case, a producer need only perform a single store access, so computation completion time equals cache access latency. However, application and system dependent artifacts such as iterative refinement algorithms and register pressure often result in the data block being successively written before it is communicated. Speculatively sending the data *before* it is ready could create unnecessary coherence traffic and exacerbate the communication critical path since the producer would need to re-fetch the block and invalidate the speculated push receivers. This effect is partially countered by the fact that κ dynamically changes, allowing it to expand as needed to wait for the last store from the producer.

D. Push Mispredictions

As in any prediction-based scheme, a significant number of mispredictions may degrade performance. Post-Store mispredictions can prove to be costly in the following scenarios:

noitemsep

- Limited Bandwith
- Potential Increase in number of network hops

Push is well-suited to limited bandwidth if the speculations are correct, since push can reduce the number of coherence messages. Our scheme dynamically adjusts the timing prediction and frequency of requests to minimize additional bandwidth pressure from mispredictions. Second, in the case of a misprediction the coherence protocol would only realize more hops than in the baseline case if the Push request was issued too early, and consequently invalidated before it was used. In addition, a partial misprediction in the destination set for a shared block should not affect the critical path latency. The protocol would only realize more hops on the critical path when mispredicting an exclusive push, as the incorrect destination will now need to be invalidated, and exclusive ownership transferred to the correct destination. We minimize such situations by attempting exclusive Push predictions only when the sharing pattern is observed to be between two nodes and had met the associated confidence requirement. For less stable read and write sets, our scheme may speculatively request the data be sent to more nodes than could be needed, thus causing a greater number of required invalidations compared to the baseline. However, the additional invalidations should not severely impact communication latency, as the extra invalidations would be overlapped.

E. Storage Overhead

Section III-C describes the additional storage required by STAP. An alternative to adding the additional L1 storage space would be to have the directory track the history access patterns of all valid tags before being able to predict. However, this would restrict scalability by increasing link utilization and bandwidth demand. Second, adding the history table to the L1-level instead of the L2 decreases the storage requirement given the large difference in sizes between the L1 and L2 caches.

Each L1-cache requires a history table, small buffer, and extra bits in the cache lines. In addition, the directory is augmented with a predictor (stability set) table. In practice, both the 3-bits augmented to each cache line and push buffer are negligible overhead. The only significant hardware cost for the L1-cache resides in the history table storage. The cost of the history table can be equated to doubling the size of the tag given that it holds as many entries as the L1-cache. Including the counter bits of the history table incurs an additional (18 tag + 3 counter bits) = 21 bits per L1-cache line cost. For our L1-cache configuration, we have 21 bits $\times$ 512 lines = 10752 bits $\approx$ 1.3125 KB of overhead per L1-cache. Therefore, for our baseline of 32 processors, we have 32 x 1.3125 $\approx$ 42 kB.

The stability set table, consisting of 8192 entries, utilizes a static bit-vector to record which caches are in the stability sets. Each entry contains a three-bit sharing type field, last operation bit, valid bit, and an integer for the delta value. Hence, each entry requires ((2x number of processors) + 2 + 8) + 3 bits. The per-directory storage is ((2x(8+10))+3) x 8192 $\approx$ (39 x 8192) = 78 kB. Accounting for both directory and L1-cache overhead, the storage overhead overhead for a system with 32 processors, private 32 kB L1s, 2 MB L2 caches shared by groups of 4 cores, and 4 directories would be: (4 x 78 kB + 42 kB)/(8x2048kB) $\approx$ 2% of the on-chip cache-hierarchy.

V. EXPERIMENTAL METHODOLOGY

We implement STAP as an extension to the existing MOESI directory coherence protocol supplied by the Ruby Memory Module in conjuction with the gem5 simulator [6]. Table II illustrates the system configuration parameters used for our experiments; these parameters closely mirror recent works [15], [20]. The latency of non-memory instructions is 1 cycle; the latency of memory instructions is fully modeled. NUMA-4 refers to a multi-socket system with quad-core chips, NUMA-1 uses 1-core chips, and CMP has all cores on 1 chip (with private 2 MB L2 caches per core). The network uses static X–Y dimension-ordered routing, modeling the links at each hop (but not switch internals). The network models infinite buffering at the switches to ensure that the performance gains of push come from actual network latency reduction rather than buffer stall reduction. Our simulation model is consistent with that of previous works in utilizing In-Order processors with a less detailed network model so that our scheme's impact can be explored across a broader range of applications and systems [15], [5], [9], [28]. We evaluate STAP using 12 benchmarks from the Splash-2 and PARSEC suites, including all benchmarks that have non-negligible read-write sharing. STAP does not issue pushes in the other benchmarks, as they exhibit either read-only sharing or no substantial sharing. Thus, STAP does not help or hurt their performance.

Table III indicates the minimum percentage threshold requirement for communicating nodes to remain in the read-/write sets. The threshold values were chosen considering the cost of speculation and the stability factors noted in previous work [2].

Processors	(16/32) 2.0 GHz in-order single-issue, non-mem IPC=1, Alpha ISA
L1 (I and D) caches	each 32 KB 2-way, 64 byte-block, 2-cycle
Shared L2 cache (NUMA-4/NUMA-1)	8 MB, 4-way unified, 4 banks, 64-byte blocks, 8-cycle
Directory Cache	14 cycles
Memory	2 GB, 200 cycle latency
L1-cache PushBuffer	5 entries
L1 Access-History Table	8kB 2-way
Dir. Stability-Set Table	(8k entries) 78kB size
Topology:	4-row Mesh w/ Corner Dir. Caches
Individual Link Width:	64 bytes
Interconn. link lat. (CMP,NUMA):	(1,20) Cycles

TABLE II: System Configuration

	Comm. Type	**Percentage**
1	Unknown	0%
2	BroadcastL2 only	10%
3	BroadcastAll	20%
4	Data Migratory	30%
5	Producer/Consumer	40%

TABLE III: Stability Set Confidence Requirement

We employ several sanity check mechanisms to ensure correctness. We use the built-in Ruby Network Tester to generate synthetic coherence traffic along with randomized post-store requests to stress the protocol design. Additional periodic checks are used to make sure neither deadlock nor livelock occurs by ensuring all outstanding requests are serviced within a predetermined period.

VI. PERFORMANCE EVALUATION

A. *Effectiveness of STAP*

Figures 5 and 6 show the percentage improvement in execution time brought about by STAP for the 16-core and 32-core experiments. For all measurements we use simulated ticks as the metric for execution time and run all the benchmarks to completion. All results are normalized against the standard MOESI Invalidation-based directory protocol, since most current systems employ some variation of MOESI. The X-axis gives the benchmark name, and the Y-axis illustrates the corresponding percentage change in execution time. Each

Benchmark	**Major data access pattern**
barnes	Producer-Consumer/Read-Shared
cholesky	Producer-Consumer
fmm	Producer-Consumer/Data-Mig
lu-noncontig/contig	Producer-Consumer
ocean-noncontig/contig	Producer-Consumer/Data-Mig
water-spatial	Producer-Consumer
water-nsquared	Data-Mig
bodytrack	Migratory Sharing (Bursty)
dedup	Producer-Consumer
raytrace	Producer-Consumer/Read-Shared
streamcluster	Producer-Consumer (Low-Sharing)
swaptions	Migratory Sharing

TABLE IV: Benchmark Sharing Patterns

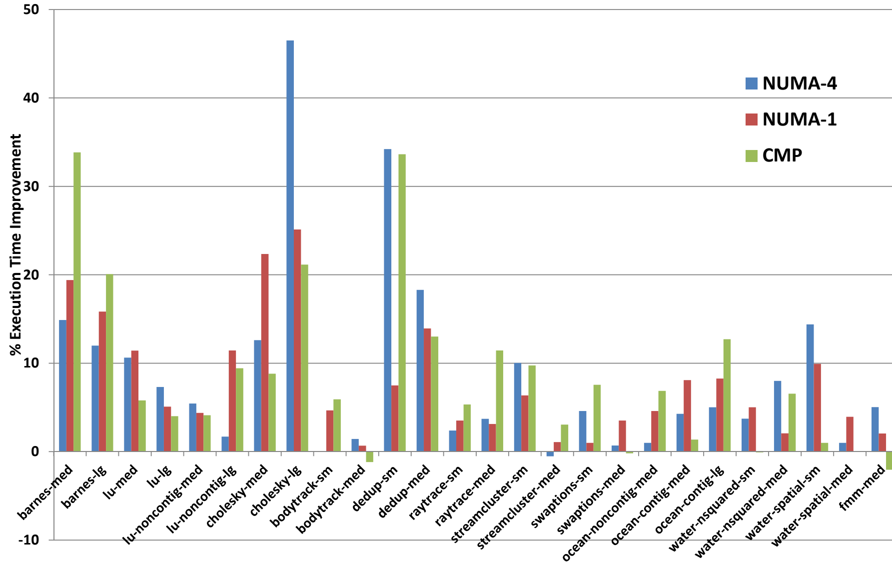


Fig. 5: Performance of STAP on 16 Cores

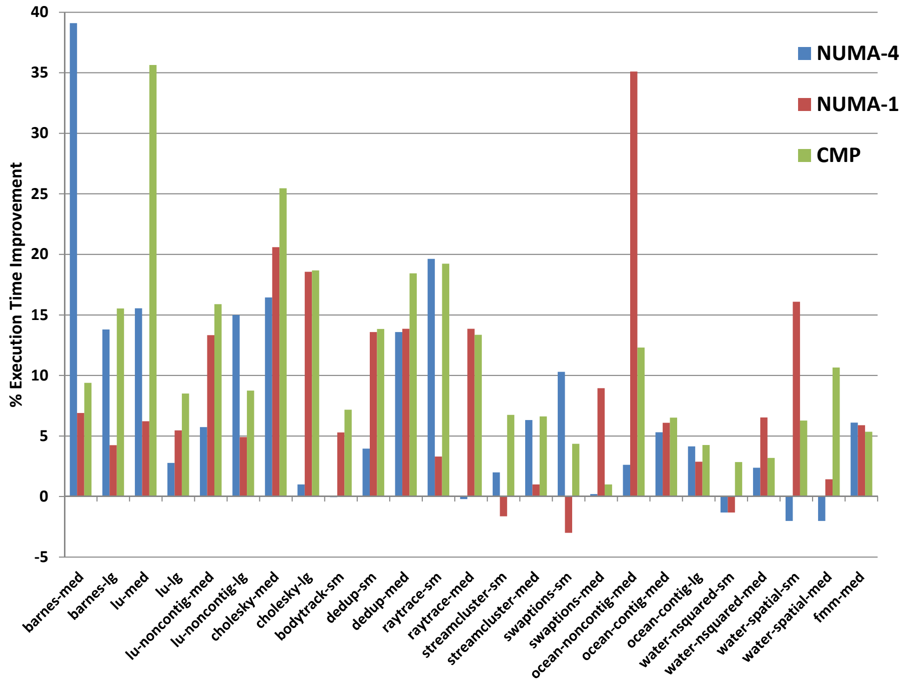


Fig. 6: Performance of STAP on 32 Cores

graph illustrates 3 bars for each benchmark, where NUMA-4, NUMA-1, and CMP results are depicted in blue, red, and green respectively. Across all configurations, STAP realizes performance speedup between 1% and 46% above the baseline.

Cholesky, a sparse matrix multiplication application, employs a working-set optimization based on the specified L1-cache capacity, allowing it to control the frequency of communication required. This allows STAP to more accurately predict timing and issue requests. We simulate both versions of LU and Ocean: contiguous and non-contiguous. Ocean-contiguous demonstrates producer-consumer communication sharing between processors that own adjacent grid blocks. The non-contiguous version ensures that the data blocks are not laid out in a contiguous fashion, thus enforcing additional communication. The resulting trade-off is that the contiguous version realizes a lower shared-write ratio (3-5x) for a higher miss rate [4]. STAP provides less benefit to Ocean as it goes from 16 to 32 cores. Although increasing the core count would normally increase sharing traffic, in this case, the per-core working set is too small to properly exploit STAP. In contrast, STAP provides more substantial benefits to the LU benchmarks, largely because 25% of the execution time is spent in synchronization, providing a larger window of opportunity for post-store.

Dedup exhibits consistent thread-group communication

sharing, providing ample opportunity for performance gains. Specifically, Dedup utilizes a pipeline model such that most read-write communication corresponds to the sharing between pipeline stages. This is one situation in which sending data to most members of a read stability-set is less likely to incur high invalidation cost associated with invalidating incorrect destinations. Swaptions exhibits a high-degree of migratory sharing in which a majority of its address space is used by more than two cores. STAP performs well in this case primarily because 1) it predicts a push in exclusive state, avoiding future invalidation costs from pushing in shared state, and 2) it attempts to make a more accurate prediction of the next writer by selecting the node with the highest confidence that has not written recently. Bodytrack exhibits almost no communication for roughly two-thirds of its parallel section, followed by a period of bursty communication. This short period of bursty communication constrains STAP's ability to achieve significant speedup. Streamcluster is a low-sharing communication application with 85% of the execution time characterized as little to no sharing. The remaining portion has fixed producer/consumer sharing between just a few cores. Our results illustrate a similar trend as streamcluster's speedup is not comparable to the other high-communicating applications.

In general, the NUMA variants see higher absolute performance gains because of their higher communication latencies, but the CMP tests still see benefit from STAP.

B. STAP With Static Timing

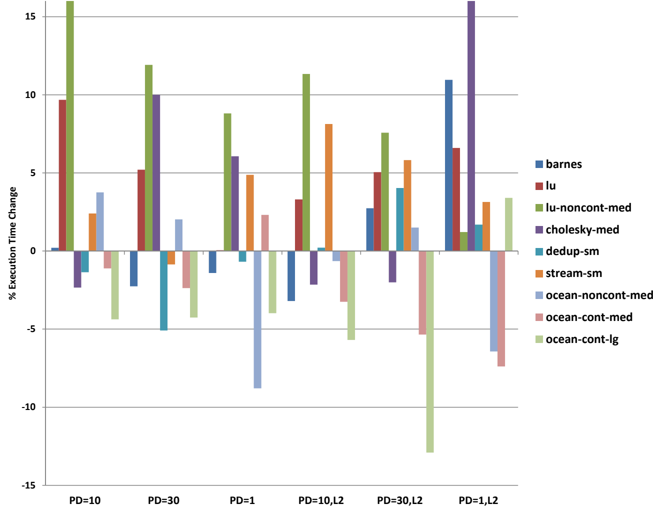


Fig. 7: Performance of STAP-ST - 32 Core NUMA-4

As discussed in Section II, many related proposals offer piecemeal or outdated solutions that either target a small subset of applications, require specific hardware features or address only one of the three (*Who, What, Where*) questions. Thus, instead of performing a direct reimplementation of any one previous work, this section evaluates STAP that excludes the dynamic timing analysis component of STAP, and uses a static Push Delay offset (PD) to represent the offset in time with respect to the current system clock for when the post-store request should be initiated. Doing so provides a reference for how the push model of Abdel-Shafi et al. in conjunction with a destination set predictor model such as that of Martin et al. is likely to perform in modern, scalable multi-core configurations [20], [1]. We believe this to be a more balanced approach than evaluating several previous proposals, each of which would only be applicable to a subset of benchmarks.

Figure 7 show the performance of STAP with static timing analysis (STAP-ST), using a static PD offset factor and bit-vector mask filter for allowable destinations. We experiment with three different PD delays (PD=1, 10, 30), and two destination masks (All Dest. vs. L2 only) for a total of 6 runs per benchmark. PD=1 represents pushes being initiated immediately assuming that the consumer(s) need the data immediately. The values of PD=10 and PD=30 were chosen to provide a slight cushion to account for both algorithmic and system considerations (e.g., iterative refinement algorithms and network link congestion, respectively).

Overall, comparing the STAP and STAP-ST results exhibit three significant advantages of dynamic timing analysis:

noitemsep

- Reduced performance variance across benchmarks and configurations
- Significantly better worst-case performance
- Significantly better best-case performance

Large variances in STAP-ST performance illustrate the sensitivity of communication to minor changes in PD, suggesting that a one-size-fits-all approach is insufficient. STAP limits the performance degradation in the worst case, with only a few benchmark runs showing 1–4% performance slowdown. In contrast, a greater number of STAP-ST runs show slowdown, and those slowdowns are substantial for many. STAP also realizes comparatively better speedups in most cases. This is due to the dynamic adjustment that avoids repeated mispredictions by factoring in instantaneous system conditions or variations in an application's communication pattern. For example, the static offset of STAP-ST does not recognize cases where there are repeated writes to the same cache block that require repeated invalidations on early pushes, thus incurring a high penalty cost not otherwise present in the base case.

C. STAP with Stable Sharing Predictor

To evaluate the performance impact of using a set predictor that detects both unstable and stable sharing sets, we implement a stable-set predictor similar to the one proposed by Martin et al. [20], [5]. In the stability-set table, each core has an associated threshold sharing-counter (SC), which must saturate prior to being considered as part of the predicted destinations. Each time a core requests a shared copy of a new block (TTFR), it issues a control message to the directory. In contrast to our standard unstable-predictor, a consumer would have to issue N TTFR messages instead of just 1. Once entered into the stability-set, a predicted sharer is never removed, unless of course the entire table entry is replaced due to replacement policy. Thus, the stable-predictor has a stronger

train-up mechanism, but also a slower *train-down* method. For this predictor, we experiment with values of SC=1,3.

Table V illustrate the performance speedup of using the stable-predictor over the MOESI baseline. For some benchmarks, the stable-predictor performs similarly to the unstable-predictor (lu-contig, ocean-contig), while performs worse for others: (streamcluster, swaptions). The stable-predictor performs suboptimally when there is low-sharing or when communicating sets periodically change. Due to the stronger train-up method, the stable-predictor is less responsive to changes in communication behavior. As a result, the opportunity window for issuing timely requests decreases. Second, the slower train-down method allows cores to remain in the stability set for a time period (T_{extra}) that exceeds the end of the communication phase. Thus, the timing analysis would continue to predict useless poststore requests during the T_{extra} period. In a push-based scheme, it is thus important to have a dynamic sense of the sharing destination, in order to more accurately account for runtime changes, such as different program phases and variation in communication intensity.

Benchmark	SC=1	SC=3
barnes-med	-8.16%	14.6%
lu-contig-med	9.27%	7.73%
lu-noncontig-med	7.76% 6.	73%
cholesky-med	6.63%	10.511%
streamcluster-sm	-7.68%	-7.26%
swaptions-sm	-0.06%	-1.41%
ocean-contig-sm	8.9%	-9.24%
water-nsq-sm	6.09%	6%

TABLE V: STAP+Stable Predictor NUMA-4 Results

D. Comparison with ARMCO

To estimate how well ARMCO compares to STAP, we implemented an approximate upper-bound of ARMCO's performance for the CMP configuration. Using ARMCO's accuracy (75%), we probabilistically determine during a Load/Store request to an invalid cache block, if ARMCO's L1-owner prediction table is correct when forwarding the request. When predicted correctly, we assume a fixed latency cost that represents an upper-bound of the network-hop latency from source to destination node. For incorrect predictions, the penalty is still incurred, before which the destination node forwards the request to the L2 cache (defaults to baseline).

The key program feature for which ARMCO is well suited for is nearest-neighbor communication, as seen in ocean-contig. For three out of the four benchmarks that overlap with those reported by the ARMCO paper (LU-contig, Cholesky, barnes), their results show minor speedup (4-7%), while ocean-contig realizes 20% speedup as a result of remotely accessing nearest-neighbor cache.

We test 4 benchmarks (LU-contig, LU-noncontig, ocean-noncontig, and ocean-contig) in our comparison. Our approximation of ARMCO validates ARMCO's benefit of improving L2-missrate through a reduction of total L2-accesses for benchmarks with nearest neighbor communication (e.g. LU-contig, Ocean-contig). Overall execution time does not improve significantly in ARMCO due to an already low L1-missrate (as noted in their paper as well). However, the drop in L2-missrate for the noncontiguous benchmarks appears to be negliglble, thus highlighting one of STAP's advantages over ARMCO. In particular, ARMCO works well for nearest-neighbor communication for boundary row communication when per-core data sets are small enough to fit largely in private L1. STAP, on the other hand, can support both L1 and L2 working sets, as well as any pair of communicating cores (e.g. lu-noncontig, ocean-noncontig).

Benchmark	Input-size	Perf. Δ	L2-miss Δ
lu-contig	512x512	3.8%	6%
lu-noncontig	512x512	3.2%	3%
ocean-contig	258x258	0.5%	2%
ocean-noncontig	258x258	1.2%	1%

TABLE VI: ARMCO Performance Approximation CMP Results

E. STAP + Prefetching

To evaluate how STAP interacts with other latency-tolerance mechanisms, we conduct a set of experiments that combines STAP with an adaptive prefetcher [6]. This helps us determine how these two techniques work together, as well as identify any negative interactions and possible solutions.

The adaptive prefetcher initializes three distinct miss streams: 1) positive unit-stride 2) negative unit-stride and 3) non-uniform stride in order of decreasing priority. On a miss, it chooses to prefetch the next stride address from the highest priority stream. This type of adaptive prefetching was chosen as it illustrates a smarter (yet not overly complex) method that can be viewed as a natural extension to the widely-used unit-stride prefetching schemes.

Benchmark	Layout	STAP	Pref	STAP+PF
barnes-med	NUMA-4	39.1%	-3.43%	1.04%
	NUMA-1	6.9%	-2.20%	-8.30%
	CMP	9.4%	33.70%	23.7%
water-nsq-med	NUMA-4	-1.32%	4.76%	**10.5%**
	NUMA-1	-1.3%	8.3%	-9.66%
	CMP	2.8%	-5.3%	-6.72%
water-spa-sm	NUMA-4	16.1%	6.61%	10.97%
	NUMA-1	-2.02%	0.2%	-0.20%
	CMP	6.3%	-5.1%	-2.38%
ocean-cg-med	NUMA-4	5.31%	14.50%	14.13%
	NUMA-1	6.1%	6.3%	**9.45%**
	CMP	6.5%	-0.2%	3.07%
lu-noncontig-lg	NUMA-4	14%	13.1%	**15.5%**
	NUMA-1	4.9%	-1.5%	-2.1%
	CMP	8.7%	-2.1%	0.31%

TABLE VII: 32-Core Prefetch+STAP Analysis

Table VII compares the speedup results of STAP-only, Prefetch-Only, and a combination of STAP+Prefetch normalized against the baseline MOESI protocol for several representative benchmarks. In the ideal case, the performance benefit of prefetch+push would be additive, or at least better than either alone. These cases are shown in bold. However, the two speculative schemes can also have problematic interactions such that the combination of the two performs worse than

either individually. This is illustrated with the underlined entries. Focusing on NUMA-4 alone, STAP+PF works best for 2 of 5 benchmarks, STAP alone for 2 others, and PF alone for one (though STAP+PF works nearly as well in that case).

The coverage overlap between the two schemes can cause redundant speculative messages to be issued (e.g., consumer issues prefetch-read to producer while producer issues post-store to consumer). The redundant messages are handled by nacking the latter speculative request to arrive. The main concern of the redundancy is not necessarily the extra network congestion (as the total maximum network utilization does not exceed 65%), but rather the latency expansion that ensues when both cache lines are blocked and then nacked to avoid deadlock. There is also an indirect consequence of uncoordinated STAP+Prefetch, illustrated in some of the NUMA-4 results (e.g. barnes). In this particular case, prefetching is interfering with the replacement of pushed blocks, causing the L1-caches to successively reacquire the push block. Transiting from $I \rightarrow S$ states requires the L1-cache to issue a TTFR message to the directory. As a result, the stability-set table recalculates δ as a result of conflict misses.

These negative interactions can be solved by tighter integration of post-store and prefetch mechanisms to improve collision handling by 1) setting a message priority ordering where either prefetches are dropped in favor of push requests or vice-versa and 2) allow post-store data messages to satisfy prefetch requests to shared data rather than nacking.

F. Sensitivity and Accuracy Analysis

STAP's decisions depend on destination set correctness and accuracy of the timing predictions, which may vary based on table sizes, and parameters: κ, and r. However, our sensitivity experiments indicate that (not shown due to space constraints) most benchmarks realize similar performance behavior independent of the parameter values studied. *Across all benchmarks, STAP's accuracy ranges between 65%-90%*. Transmitting speculative messages may also increase bandwidth pressure and congestion in the network. Few benchmarks realize minimal speedups due to the state of the network for certain high communication phases. We determine this to be the cause as even though the network remains underutilized across all benchmarks and input combinations, varying the amount of total bandwidth for these select benchmarks results in notable performance improvement. Note that varying the bandwidth for other benchmarks had minimal impact on performance. We attribute this effect to the routing policy chosen: fixed shortest path, which does not factor in the instantaneous state of the network-links. However, as previously mentioned these periods of network congestion are somewhat mitigated by adjusting the κ factor, as STAP recognizes the message roundtrip time as increased. Utilizing a more intelligent routing scheme would make it harder to analyze STAP's impact on memory performance, while developing a integrated push-routing policy is outside the scope of this work.

Table VIII illustrates the impact of varying the total available bandwidth for representative benchmarks. We perform two additional experiments: halving the total bandwidth and doubling it relative to the baseline case. In Table VIII, there are several instances where performance improves when the bandwidth is halved and decreases when the bandwidth is doubled. Push (vs. pull) typically reduces bandwidth pressure for critical requests, so STAP is typically more effective (relatively) in more bandwidth-constrained systems: baselines execution time increases far more than STAPs. The results demonstrate that 1) decreasing available link bandwidth does not eliminate performance gains as STAP still achieves speedups, and 2) increasing link bandwidth can provide even greater speedup.

	Benchmark	% Execution Time Change		
		1/2 BW	Default	2xBW
NUMA-4	lu-contig-med	4.02%	10.63%	12.14%
	lu-contig-lg	-3.78%	7.3%	-5.59%
	water-nsq-sm	-8.0%	3.72%	-5.13%
	water-spatial-sm	-0.8%	14.39%	0.55%
NUMA-1	lu-contig-med	27.5%	11.4%	35.22%
	lu-contig-lg	0.97%	5.1%	5.22%
	water-nsq-sm	6.33%	6.33%	2.13%
	water-spatial-sm	3.85%	9.9%	3.53%
CMP	lu-contig-med	11.9%	5.8%	-1.75%
	lu-contig-lg	2.8%	4%	7.72%
	water-nsq-sm	-0.15 %	-0.10%	12.5%
	water-spatial-sm	-0.0.1%	1%	0.09%

TABLE VIII: Impact of bandwidth on 16 Core STAP

VII. CONCLUSION

In this paper, we present a coherence protocol that identifies communication patterns to effectively predict various forms of data sharing, accurately predicts when and where the data will be needed, and then pushes that data from producers to consumers. Compared to previous sharer predictor/replication schemes, this work is broader in applicability, more effective at targeting scalability, and more general in supporting dynamic sharing patterns.

STAP outperforms stable predictor-sharing and static timing, indicating that simply gluing different previously proposed strategies does not achieve the same gains as STAP, or can lead to a significant performance degradation.

Furthermore, our design builds upon widely implemented invalidation protocols without requiring a clean design, invasive changes at the L1, or repeated modifications for future systems, providing hardware designers with a cost-effective solution for scaling shared-memory systems. Results show up to 46% improvement in overall execution time for applications that demonstrate both stable and dynamically-changing sharing patterns, at a cost of very little hardware overhead. We show that push can integrate well with prefetching, and also identify a possible negative interaction and a possible solution. These findings suggest that push can be an important and well-integrated component of a broader solution to reduce exposed latency in cache-coherent shared-memory systems.

REFERENCES

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve, "An evaluation of fine-grain producer-initiated communication in cache-coherent mul-

tiprocessors," in *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, ser. HPCA '97, Washington, DC, USA, 1997, pp. 204–.

[2] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of splash-2 and parsec," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09, Washington, DC, USA, 2009, pp. 86–97.

[3] B. Beckmann, M. Marty, and D. Wood, "Asr: Adaptive selective replication for cmp caches," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2006, pp. 443–454.

[4] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on Workload Characterization*, sept. 2008, pp. 47 –56.

[5] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood, in *ISCA*, A. Gottlieb and W. J. Dally, Eds. IEEE Computer Society, pp. 294–304.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[7] J. Chang and G. Sohi, "Cooperative caching for chip multiprocessors," in *Computer Architecture, 2006. ISCA'06. 33rd International Symposium on Computer Architecture*. IEEE, 2006, pp. 264–276.

[8] L. Cheng, J. Carter, and D. Dai, "An adaptive cache coherence protocol optimized for producer-consumer sharing," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on Computer Architecture*. IEEE, 2007, pp. 328–339.

[9] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps." in *ISCA*, 2005, pp. 357–368.

[10] A. chow Lai and B. Falsafi, "Memory sharing predictor: The key to a speculative coherent dsm," in *Proceedings of the 26th annual international symposium on Computer architecture*, 1999, pp. 172–183.

[11] A. L. Cox and R. J. Fowler, "Adaptive cache coherency for detecting migratory shared data," in *Proceedings of the 20th annual international symposium on computer architecture*, ser. ISCA '93, New York, NY, USA, 1993, pp. 98–108.

[12] P. Foglia, "An algorithm for the classification of coherence related overhead in shared-bus shared-memory multiprocessors," *IEEE TCCA Newsletter*, pp. 40–46, 2001.

[13] P. Foglia, R. Giorgi, and A. C. Prete, "Reducing coherence overhead and boosting performance of high-end smp multiprocessors running a dss workload abstract."

[14] H. Grahn and P. Stenstrm, "Evaluation of a competitive-update cache coherence protocol with migratory data detection," *Journal of Parallel and Distributed Computing*, vol. 39, pp. 39–2, 1996.

[15] H. Hossain, S. Dwarkadas, and M. C. Huang, "Improving support for locality and fine-grain sharing in chip multiprocessors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08, New York, NY, USA, 2008, pp. 155–165.

[16] B. Kahhaleh, "Analysis of memory latency factors and their impact on ksr1 performance," in *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, apr 1994, pp. 649 –656.

[17] A. Kayi and T. El-Ghazawi, "An adaptive cache coherence protocol for chip multiprocessors," in *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, ser. IFMT '10. New York, NY, USA: ACM, 2010, pp. 4:1–4:10.

[18] A.-C. Lai and B. Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction," in *Proceedings of the 27th annual international symposium on Computer architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 139–148.

[19] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors," in *Proceedings of the 22nd annual international symposium on Computer architecture*, ser. ISCA '95, New York, NY, USA, 1995, pp. 48–59.

[20] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood, "Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 206–217, May 2003.

[21] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *SIGPLAN Not.*, vol. 27, no. 9, pp. 62–73, Sep. 1992.

[22] S. S. Mukherjee and M. D. Hill, "Using prediction to accelerate coherence protocols," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 179–190.

[23] S. Somogyi, "Memory coherence activity prediction in commercial workloads," in *In Proceedings of the Third Workshop on Memory Performance Issues (WMPI-2004*, 2004.

[24] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proceedings of the 33rd annual international symposium on Computer Architecture*, ser. ISCA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 252–263.

[25] P. Stenström, M. Brorsson, and L. Sandberg, "An adaptive cache coherence protocol optimized for migratory sharing," in *Proceedings of the 20th annual international symposium on computer architecture*, ser. ISCA '93, New York, NY, USA, 1993, pp. 109–118.

[26] C. Tumuluri and A. Choudhary, "Scalable software latency hiding schemes: Evaluation of the poststore and prefetch options," in *Euro-Par'96 Parallel Processing*, ser. Lecture Notes in Computer Science, 1996, vol. 1124, pp. 486–491.

[27] G. Venkataramani, C. J. Hughes, S. Kumar, and M. Prvulovic, "Deft: Design space exploration for on-the-fly detection of coherence misses," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 2, pp. 8:1–8:27, Jun. 2011.

[28] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2. IEEE Computer Society, 2005, pp. 336–345.

[29] H. Zhu, Y. Chen, and X.-H. Sun, "Timing local streams: improving timeliness in data prefetching," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 169–178.