June 2008

Adding Token Counting to Directory-Based Cache Coherence

Arun Raghavan
University of Pennsylvania, arraghav@cis.upenn.edu

Colin Blundell
University of Pennsylvania, blundell@cis.upenn.edu

Milo M.K. Martin
University of Pennsylvania, milom@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-22.

Adding Token Counting to Directory-Based Cache Coherence

Abstract
The coherence protocol is a first-order design concern in multicore designs. Directory protocols are naturally scalable, as they place no restrictions on the interconnect and have minimal bandwidth requirements; however, this scalability comes at the cost of increased sharing latency due to indirection. In contrast, broadcast-based systems such as snooping protocols and token coherence reduce latency of sharing misses by sending requests directly to other processors. Unfortunately, their reliance on totally ordered interconnects and/or broadcast limits their scalability.

This work introduces PATCH (Predictive/Adaptive Token Counting Hybrid), a coherence protocol that provides the scalability of directory protocols while opportunistically using available bandwidth to reduce sharing latency. PATCH extends a standard directory protocol to track tokens and use token counting rules for enforcing coherence permissions. Token counting allows PATCH to support direct requests on an unordered interconnect, while a novel mechanism called token tenure uses local processor timeouts and the directory's per-block point of ordering at the home node to guarantee forward progress without relying on broadcast.

PATCH makes three main contributions. First, PATCH uses direct request prioritization to match the performance of broadcast-based protocols without restricting scalability. Second, PATCH introduces token tenure, which provides broadcast-free forward progress for token counting protocols. Finally, PATCH provides greater scalability than directory protocols when using inexact encodings of sharers because only processors holding tokens need to acknowledge requests. Overall, PATCH is a "one-size-fits-all" coherence protocol that dynamically adapts to work well for small systems, large systems, and anywhere in between

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-22.

Adding Token Counting to Directory-Based Cache Coherence

Arun Raghavan, Colin Blundell, Milo M. K. Martin
University of Pennsylvania

UPenn CIS Technical Report TR-CIS-08-22
June 4, 2008

Abstract

The coherence protocol is a first-order design concern in multicore designs. Directory protocols are naturally scalable, as they place no restrictions on the interconnect and have minimal bandwidth requirements; however, this scalability comes at the cost of increased sharing latency due to indirection. In contrast, broadcast-based systems such as snooping protocols and token coherence reduce latency of sharing misses by sending requests directly to other processors. Unfortunately, their reliance on totally ordered interconnects and/or broadcast limits their scalability.

This work introduces PATCH (Predictive/Adaptive Token Counting Hybrid), a coherence protocol that provides the scalability of directory protocols while opportunistically using available bandwidth to reduce sharing latency. PATCH extends a standard directory protocol to track tokens and use token counting rules for enforcing coherence permissions. Token counting allows PATCH to support direct requests on an unordered interconnect, while a novel mechanism called token tenure uses local processor timeouts and the directory's per-block point of ordering at the home node to guarantee forward progress without relying on broadcast.

PATCH makes three main contributions. First, PATCH uses direct request prioritization to match the performance of broadcast-based protocols without restricting scalability. Second, PATCH introduces token tenure, which provides broadcast-free forward progress for token counting protocols. Finally, PATCH provides greater scalability than directory protocols when using inexact encodings of sharers because only processors holding tokens need to acknowledge requests. Overall, PATCH is a "one-size-fits-all" coherence protocol that dynamically adapts to work well for small systems, large systems, and anywhere in between.

1 Introduction

A multi-core chip's coherence protocol impacts both its scalability (*e.g.*, by requiring broadcast) and its miss latency (*e.g.*, by introducing a level of indirection for sharing misses). Traditional coherence protocols present a set of difficult tradeoffs. Snoopy protocols maintain coherence by having each processor send requests directly to all other processors (broadcast) to provide low sharing miss latency but do not scale due to excessive traffic. Conversely, directory protocols introduce a level of indirection to obtain scalability at the cost of increasing sharing miss latency.

Prior proposals have attempted to ease the tension between snoopy protocols and directory protocols. One approach aims to make snooping protocols more efficient by using destination-set prediction [4, 22] or bandwidth adaptivity [25] to send direct requests to fewer than all processors. These protocols suffer from their snooping heritage by sending all requests over a totally ordered interconnection network, limiting their scalability and complicating their implementation. An alternative approach adds direct requests to a directory protocol (*e.g.*, [1, 2, 6, 16]). These protocols use direct requests only in limited contexts and hence fail to capture all of their performance benefits.

In this paper we present PATCH (Predictive Adaptive Token Counting Hybrid), a novel protocol that achieves performance without sacrificing scalability. PATCH neither relies on broadcast nor an ordered interconnect for correctness. PATCH allows unconstrained predictive direct requests as performance hints that may be dropped at times

of interconnect contention. Finally, PATCH requires only true sharers to acknowledge requests. This property allows PATCH to be even more scalable than a directory protocol when directory encodings are inexact.

PATCH obtains these attributes by combining token counting and a standard directory protocol. Token counting [23] directly ensures safety of requests: processors pass tokens around the system and use rules based on the number of tokens that they currently have for a given block to determine when an access to that block is legal. Whereas the original token counting proposal focused on token counting as a mechanism of extending broadcast-based protocols to operate on unordered interconnects, we instead use it as a mechanism to enhance the performance of directory protocols by using direct requests to avoid adding indirection latency to sharing misses.

Adding token counting allows PATCH to naturally and simply support destination-set prediction and bandwidth adaptivity without requiring a non-scalable interconnect. PATCH's use of these mechanisms is enhanced by the fact that only token holders (*i.e.*, true sharers) have to respond to requests (unlike other proposals for adding direct requests to directory protocols [1, 2, 6, 16]). This property both allows PATCH to be more profligate in its destination-set predictions and enables a new form of bandwidth adaptivity called *prioritized direct requests*. Interconnect switches may de-prioritize direct requests and propagate them only when there is sufficient bandwidth to do so (discarding them if they become too stale). By de-prioritizing direct requests PATCH ensures that they never degrade performance relative to that of a directory protocol. The resulting protocol offers the scalability of directory protocols, the performance of broadcast-based protocols when bandwidth is plentiful, and the ability to smoothly transition between them as needs demand.

Although adding token counting to a directory protocol enables no-indirection sharing misses, the protocol also inherits the problem that forward progress is no longer guaranteed (Section 3 provides an illustrative example). Previously-proposed mechanisms to ensure forward progress in token counting-based protocols rely on broadcast, a requirement that we expressly want to avoid adding to our protocol. We propose a set of rules, the *token tenure* rules, that guarantee forward progress in a directory protocol augmented with token counting without introducing a broadcast requirement. Under the token tenure rules tokens can move in response to direct requests but must eventually be given up to the directory if they are not tenured: that is, if the processor does not see a message from the directory granting permission to retain the tokens within a certain amount of time. The process of tenuring tokens is not on the processor's critical path and thus has no effect on performance in the absence of races (the common case). When races exist, token tenure ensures forward progress because the directory will tenure the tokens of only one of the racing processors; tokens held at all other processors will timeout and eventually flow to the winning processor, allowing it to complete its request.

PATCH makes three main contributions:

- **Scalable direct requests via prioritization.** PATCH's direct requests enable it to match the performance of broadcast-based protocols when bandwidth is plentiful, while its prioritization mechanism provides scalability by guaranteeing that PATCH will never underperform directory protocols.
- **Scalable forward progress mechanism.** PATCH introduces token tenure, which provides broadcast-free forward progress for token counting protocols by using local timeouts to avoid a need for global consensus.
- **Increased scalability over directory protocol.** PATCH provides greater scalability than directory protocols when using inexact encodings of sharers because only processors holding tokens need to acknowledge requests.

2 Background on Token Counting

The goal of an invalidation-based cache coherence protocol is to enforce the "single-writer or many-readers" cache coherence invariant. Token counting enables the direct enforcement of this invariant [23]. Instead of enforcing the coherence invariant using a distributed algorithm for providing coherence safety in the presence of subtle races, token counting uses explicit token counting to enforce the coherence invariant using only local rules. This characteristic simplifies the implementation of many potential performance optimizations by freeing implementers from the burden of worrying about coherence safety. Token counting also decouples enforcement of coherence from any specific interconnect ordering properties or ordering point in the system.

Token counting uses tokens to enforce coherence permissions and encode MOESI coherence states [30] (see Figure 1). At system initialization, the system assigns each block T tokens (where T is at least as large as the number of caches). One of the tokens is designated as the *owner token* that can be marked as either clean or dirty. Tokens are tracked per-block and can be held in processor caches, memory modules, and coherence messages (in-flight or buffered). Initially, the block's home memory module holds all tokens for a block. Tokens and data are allowed to move between system components as long as the system maintains the five token counting rules [21] given in Figure 2.

Tokens	Owner?	State
All	Dirty	M
Some	Dirty	O
All	Clean	E
Some	Clean	"F" [14]
Some	Not held	S
None	Not held	I

Figure 1: **Mapping of token counts to MOESI states.**

The invariants provided by these token counting rules are sufficient to allow a processor to enforce a memory consistency model (including sequential consistency) [21]. For example, together these locally enforceable rules are sufficient to enforce global invariants such as "No cache may have read permissions to a block while another cache has write permissions to the block" and "at a given time, all caches with read permissions to a block will have the same value for that block" [5]. These invariants hold no matter what request or writeback races occur. Races may still

Rule #1 (Conservation of Tokens): After system initialization, tokens may not be created or destroyed. One token for each block is the owner token. The owner token can be either clean or dirty, and whenever the memory receives the owner token, the memory sets the owner token to clean.
Rule #2 (Write Rule): A component can write a block only if it holds all T tokens for that block and has valid data. After writing the block, the writer sets the owner token to dirty.
Rule #3 (Read Rule): A component can read a block only if it holds at least one token for that block and has valid data.
Rule #4 (Data Transfer Rule): If a coherence message contains a dirty owner token, it must contain data.
Rule #5 (Valid-Data Bit Rule): A component sets its valid-data bit for a block when a message arrives with data and at least one token. A component clears the valid-data bit when it no longer holds any tokens. The home memory sets the valid-data bit whenever it receives a clean owner token (even if the message does not contain data).

Figure 2: **Token Counting Rules**

introduce protocol forward progress issues; we defer the discussion of forward progress in prior proposals that use token counting protocols to Section 4.2.

This work is not the first work to use token counting. Several other proposals have explored or exploited token counting [5, 7, 8, 11, 15, 23, 26, 27, 28, 29], including its use in the context of broadcast-based cache coherence [23], multi-socket multi-core systems [26], ring-based multiprocessors [27], virtual hierarchical cache coherence [28], fault tolerant coherence protocols [11, 29], and multicast interconnection networks [15].

3 PATCH Motivation and Overview

PATCH is a standard directory-based cache coherence protocol augmented with token counting. Whereas the original token coherence protocol used token counting to enable broadcast over an unordered interconnect, PATCH obtains high performance without sacrificing the scalability of the directory protocol on which it builds. To achieve this goal, PATCH uses several enabling features: predictive direct requests, avoidance of unnecessary acknowledgements, bandwidth adaptivity via best-effort direct requests, and a broadcast-free forward progress mechanism called token tenure.

Predictive direct requests. Token counting provides greater flexibility to transfer coherence permissions without incurring directory indirection. By adding token counting to a directory protocol, PATCH inherits the same flexibility. Whereas token coherence used this ability to directly broadcast requests, PATCH uses prediction to send direct requests to zero, some, or all other processors in the system. If the requester sends a direct request to all the necessary processors—the owner for read requests, all sharers for write requests—a higher-latency indirect (3-hop) sharing miss becomes a faster, direct (2-hop) miss. Coherence is easily enforced because the token counting rules govern coherence permissions.

Avoiding unnecessary acknowledgments. In protocols based on token counting, processors determine when a request has been satisfied by waiting for a certain number of tokens rather than by waiting for a certain number of acknowledgement messages. This change allows PATCH to elide those acknowledgement messages that would have contained zero tokens. By avoiding these *unnecessary acknowledgements*, PATCH enhances the appeal of direct requests by reducing the amount of traffic that they add to the system, which could otherwise dilute their benefits. In systems that employ bandwidth-efficient fan-out routing for multi-destination direct (or indirect) request messages, these unnecessary acknowledgements can cause "acknowledgement implosion", which can substantially reduce the effectiveness of direct requests by introducing a significant (non-scalable) amount of additional traffic into the system. In fact, PATCH also avoids unnecessary acknowledgments for indirect requests. For large systems that employ an inexact set of sharers at the directory, avoiding these unnecessary acknowledgements grants PATCH better scalability than even a standard directory protocol (Section 7).

Bandwidth adaptivity via best-effort direct requests. PATCH uses a novel form of bandwidth adaptivity to achieve high performance without sacrificing scalability. Because the directory continues to forward requests to the owner and/or sharers (as in the baseline protocol) and direct requests need not be acknowledged (as described above), PATCH can treat direct requests strictly as hints. Hence, PATCH's direct requests can now be delivered on a best-effort, lowest-priority basis. Interconnect switches forward direct requests only when they have sufficient bandwidth, simply dropping them if they become too stale. This property ensures the scalability of PATCH in that low-priority direct requests never degrade PATCH's performance relative to the baseline directory protocol even when PATCH is sending direct requests to many processors.

Broadcast-free forward progress via token tenure. The above attributes provide a framework for a fast, scalable, and adaptive protocol. However, races may cause starvation (as shown in Figure 3*a*). To prevent starvation without impeding scalability, PATCH introduces *token tenure*, a broadcast-free forward progress mechanism for token counting-based protocols. In token tenure, a processor that has received tokens for a block is required to discard these tokens after a bounded amount of time *unless* the directory has informed it that it is the active requester for the block. The home funnels all such discarded tokens to the active requester. Once the active requester completes, the directory activates the next queued request, ensuring that all requests eventually complete (as shown in Figure 3*b*).

3.1 Relationship to Prior Work

PATCH has several differences from prior protocols that either improve the scalability of broadcast-based protocols or add direct requests to directory protocols (summarized in Figure 4). There have been several protocols that aim to make broadcast snooping protocols more efficient by allowing requests to be sent to fewer than all processors

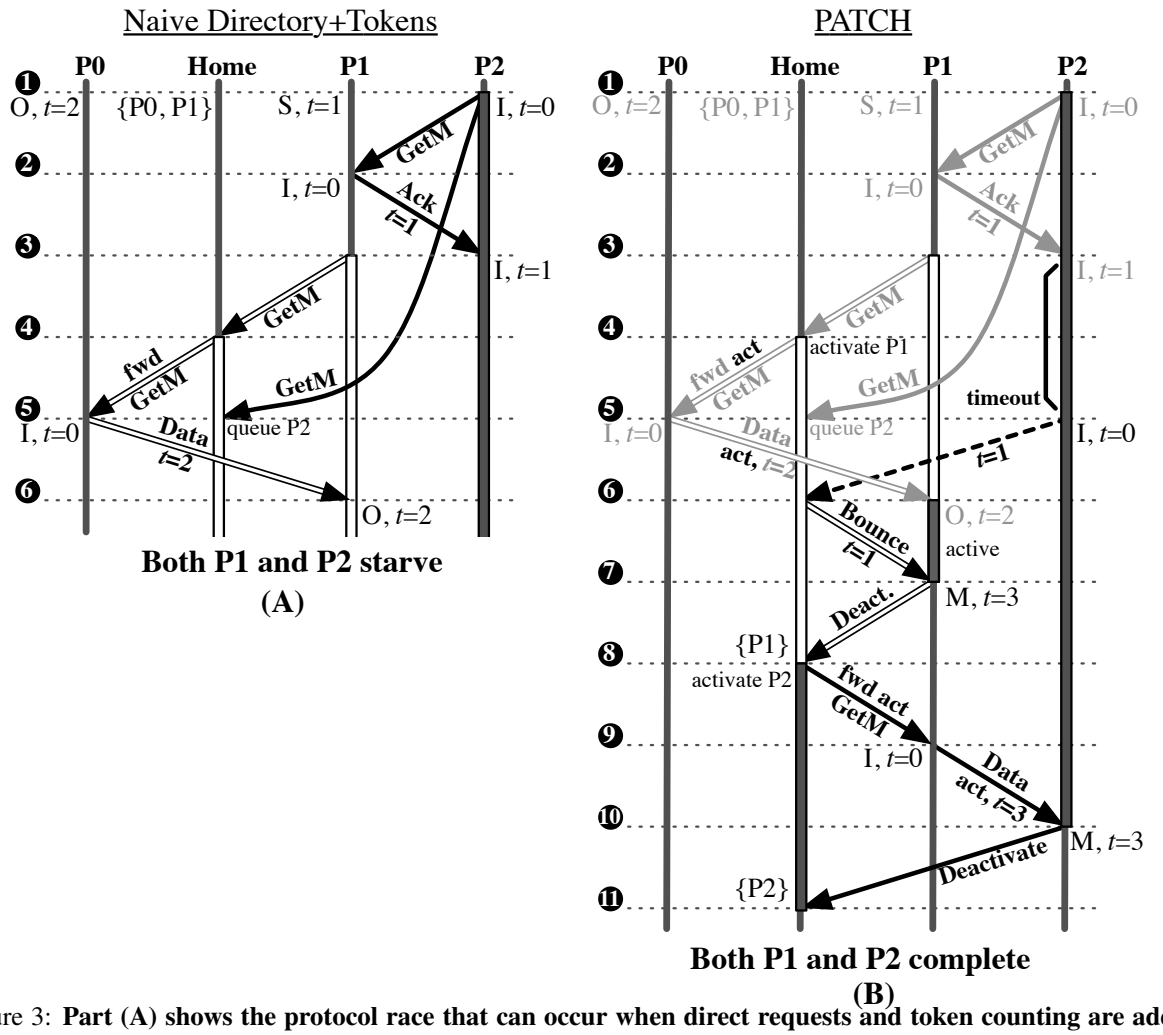

Figure 3: **Part (A) shows the protocol race that can occur when direct requests and token counting are added to a directory protocol, and part (B) shows how PATCH's token tenure resolves this race.** All messages are for a block A. A store miss ("GetM") is issued by both P2 (solid lines) and P1 (hollow lines). In part (A), initially, P0 has A in owned state with 2 tokens and P1 has A in shared state with 1 token. At time ❶, P2 sends a request for A in modified state to the directory as well as sending the request directly to P1. P1 sees this message at time ❷ and responds with its token, which P2 receives at time ❸. Also at time ❸, P1 sends a request for A in modified state to the directory. The directory sees P1's request at time ❹ and forwards it to P0 (the only other processor in the directory's sharers list). P0 receives the forwarded request at time ❺; also at this time, the directory receives P2's delayed request from time ❶ and queues it behind P1's request. At time ❻ P1 receives P0's data and tokens. At this point, both P1 and P2 are waiting for tokens that will never arrive. Part (B), the operation of PATCH in this scenario, proceeds initially as in part (A). However, when the directory activates the request of P1 at time ❹, it augments the forward to P0 with the "activated bit". At time ❺ P2 times out because it has not been activated and sends its token to the home. P1 receives P0's acknowledgement, including the activated bit, at time ❻, at which time the directory also receives P2's token. The directory forwards the token onto P1, which receives it at time ❼ and sends a deactivation message to the directory. When the directory receives this deactivation message at time ❽ it activates P2, sending a forward to P1 which includes the activation bit. At time ❾ P1 receives this forward and sends data and tokens to P2, which receives them at time ❿. At this time, P2 sends a deactivation message to the home, which is now ready to process other requests for A.

Protocol	Scalable interconnect?	Broadcast-free forward progress?	Direct requests?	Acks all requests?	Prediction?	Bandwidth adaptive?
Snooping	no	no	**yes**	**no**	no	no
Multicast Snooping [4, 22]	no	**yes**	**yes**	**no**	**yes**	no
BASH [25]	no	**yes**	**yes**	**no**	no	**bimodal**
Directory	**yes**	**yes**	no	yes	no	no
Acacio *et al.* [1, 2]	**yes**	**yes**	**yes**	yes	**yes**	no
Jerger *et al.* [16]	**yes**	**yes**	**yes**	yes	**yes**	no
Cheng *et al.* [6]	**yes**	**yes**	**yes**	yes	**yes**	no
TokenB [23]	**yes**	no	**yes**	**no**	no	no
PATCH	**yes**	**yes**	**yes**	**no**	**yes**	**priority**

Figure 4: **Comparison of PATCH to prior work on direct requests**

(*e.g.*, [4, 22, 25]). These protocols suffer from their snooping heritage by sending all requests over a totally ordered interconnection network, limiting their scalability and complicating their implementation. PATCH places no such requirements on the interconnect. The original token coherence proposal [23] used token counting to enable broadcast over an unordered interconnect. PATCH differs from token coherence, in that token coherence's forward-progress mechanism is broadcast-based and requires per-processor structures whose size grows linearly with system size, limiting its scalablity. Finally, prior proposals have added predictive direct requests to a directory protocol (*e.g.*, [1, 2, 6, 16]). In contrast to these prior works, PATCH can perform unconstrained destination-set prediction, has an easy-to-reason about safety property, and introduces only a few changes to the baseline directory protocol. Furthermore, PATCH's direct requests do not require acknowledgement messages, allowing PATCH to make them low-priority and thus be more profligate with its destination set predictions.

4 Token Tenure

The rules of token counting enforce coherence safety, but they say nothing about ensuring forward progress. This section introduces analogous local rules for ensuring forward progress in token counting protocols based on a novel mechanism called *token tenure*. In token tenure, tokens can be in two states: *tenured* and *untenured*. Tenured tokens are established and allowed to remain at a processor indefinitely or until requested by another processor. In contrast, untenured tokens must become tenured within a bounded amount of time. If not, the processor is required to discard the tokens by writing them back to the home memory module for the block. The tenured status is only used in providing forward progress; untenured tokens can be used to satisfy misses. Thus, the entire token tenure process is off the critical path for typical misses.

To tenure tokens, the system selects one request at a time on a per-block basis to be the block's current *active request*. Only the home node and active requester need to know which request is active. Once a processor has become

Rule #1 (Activation Rule): The home fairly designates one requester as the block's current active requester. The home informs a requester when its request has been activated.
Rule #2 (Token Arrival Rule): Tokens that arrive at a processor are by default untenured.
Rule #3 (Promotion Rule): Only the active requester may tenure tokens, and it tenures all tokens it possesses or receives.
Rule #4 (Probationary Period Rule): A processor may hold untenured tokens only for a bounded duration before discarding the tokens by sending them to the home.
Rule #5 (Home Forward Rule): The home node forwards any discarded tokens to the active requester.
Rule #6 (Processor Response Rule): (*a*) The active requester hoards tokens by ignoring incoming requests until its request completes. (*b*) All other processors with tokens respond to forwarded requests. (*c*) Processors with untenured tokens ignore direct requests.
Rule #7 (Deactivation Rule): Once the active requester has collected sufficient tenured tokens, it gives up its active status by informing the home.

Figure 5: **Token Tenure Rules**

the active requester, any tokens that it holds become tenured, as well as any further tokens that it receives while its request is active. We formalize these rules in Figure 5.

4.1 On the Correctness of Token Tenure

In addition to the token tenure rules in Figure 5, the system places a two-part requirement on the home: first, as part of activating a request, the home must send forwarded requests to at least those processors holding tenured tokens and second, this is the only time that the home may send forwarded requests. Together, this requirement and the above rules guarantee that all requests eventually complete.

Consider many racing requests to the same block. The home activates one of these requests (rule #1). The home forwards any tokens that it receives to the active requester (rule #5), thus any tokens that arrive at the home will eventually arrive at the active requester.

Tokens that are not at the home or tenured at the active requester may be either in-flight, untenured, or tenured at a non-active processor. Any in-flight tokens that arrive at a non-active processor become untenured (rule #2). Untenured tokens may not move via direct requests (rule #6*c*). As such, they either (*i*) timeout and are sent to the active requester via the home (rules #4 and #5) or (*ii*) move in response to a forwarded request (rule #6*b*). The forwarded request will likely send the tokens to the active requester, but could be a lingering (stale) forwarded request from a prior activation. However, the number of such lingering requests is bounded, so untenured tokens may move due to such requests only a finite number of times, after which they will move to the active requester either directly via a forwarded request or indirectly via a timeout. Finally, all tenured tokens either move to the active requester via a direct or forwarded request or move to another (non-active) processor, in which case they become untenured and the above reasoning applies.

Until the active requester has received its notification of activation, it acts like any other non-active requester, and may inadvertently send tokens to the home or another processor; any such tokens will eventually be returned to the active requester as described above. Once the active requester learns that it has been activated, it will tenure (rule #3) and hoard (rule #6*a*) tokens, and thus it will eventually collect sufficient tokens. By rule #7, once the active requester has been satisfied, it will then deactivate by informing the home to fairly select another requester to activate, thus ensuring that all pending requests eventually become the active requester (and thus eventually complete).

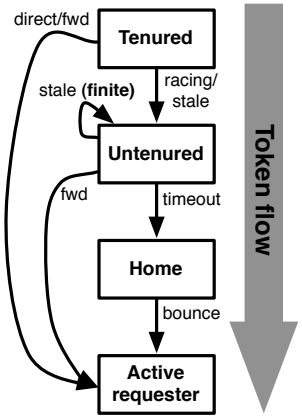

Figure 6: **Illustration of the flow of tokens to the active requester.**

The above discussion assumes that the request desires all tokens (*i.e.*, it is a write request). For read requests, which require only data and one token, the same reasoning can be applied to just the owner token.

Although the correctness reasoning for token tenure is subtle, token tenure is simple to implement on top of a directory protocol. Token tenure requires three mechanisms, the first two of which are already provided by a directory protocol: (1) forwarding request to processors with tenured tokens, (2) fairly selecting and notifying the active requester, and (3) timing out untenured tokens using a timer per outstanding request at the processor. In the next section, we describe the specifics of PATCH's implementation of the token tenure rules. First, however, we compare token tenure to prior proposals for guaranteeing forward progress in token counting-based protocols.

4.2 Prior Forward Progress Mechanisms for Token Counting Protocols

Other protocols that use token counting for various purposes also have tackled similar forward progress issues [23, 26, 27]. Figure 7 highlights their different attributes. The philosophy of token tenure is different from these prior proposals in that: (1) the directory protocol foundation provides a scalable way to activate a single request for a block and (2) token tenure's timeouts provide forward progress without global consensus (*i.e.*, broadcast) or interconnect constraints.

Token coherence uses *persistent requests* [23, 26] to ensure forward progress. A processor invokes a persistent request after its *transient requests* have repeatedly failed to collect sufficient tokens during a timeout interval. Persistent requests are broadcast to all processors, and the system uses centralized [23] or distributed arbitration [26] to achieve a global consensus of the identity of the highest priority requester. All processors then forward data and tokens to that highest priority requester. The persistent request mechanism differs from token tenure in that all processors must agree on—and remember—which request is highest priority, necessitating both broadcast and a persistent request table at

Mechanism	Broadcast-free?	Interconnect	Reissues?	State at home	State at processor
Persistent/priority requests [7, 23, 26]	no	any	yes	tokens & P.R. table	tokens
RingOrder [27]	no	ring	no	tokens	1 bit
Token tenure	yes	any	no	tokens	tokens & sharers set

Figure 7: **Comparison of forward progress mechanisms proposed for token counting protocols.**

each processor. In contrast, in token tenure only the home and the active requester need to know which request is active; all the other requesters infer that they are not the active requester by timing out and releasing their tokens to the home. Because of this property, token tenure does not require either broadcast or persistent request tables, both of which scale poorly as the number of cores grows. Priority requests [7] are similar to persistent requests in that they are broadcast-based and uses per-processor tables.

Ring-Order [27] uses token counting on a unidirectional ring interconnect to ensure initial requests always succeed (without reissue or invoking persistent requests). Ring-Order introduces a priority token to prioritize different requesters as the priority token moves around the ring. In contrast to Ring-Order, token tenure requires neither broadcast nor a ring interconnect.

5 Implementation and Operation

This section describes the operation of one specific implementation of PATCH. Although the conceptual framework of PATCH could likely be applied to any directory protocol, for concreteness we first describe a specific baseline directory protocol on which our implementation of PATCH is built. We then describe the modifications PATCH adds to this baseline protocol to support direct requests via token counting and token tenure.

5.1 Baseline Directory Protocol

Our baseline protocol is based on the directory protocol distributed as part of GEMS [24]. This protocol resolves races without negative acknowledgement messages (nacks) by using a busy/active state at the directory for every request. Although this approach is different from DASH [19] and SGI Origin 2000 [18], it is reflective of more recent protocols such as Sun's WildFire [13], the Alpha 21364/GS1280 [9], and AMD's Opteron/Hypertransport protocol [17]. In this approach, the arrival order at the directory unambiguously determines the order in which racing requests are serviced. The baseline protocol uses three-phase writebacks (writeback request to directory, ack from directory, writeback to directory). The protocol does not depend upon any ordering properties of the interconnect.

When a request arrives at the directory, it sets the block's state to active. All subsequent requests for that block are queued (at the directory or in the interconnect) until the active request is deactivated. If the request is a write request, the directory sends invalidation messages to all sharers, which respond with invalidation acknowledgments

directly to the requester. To make these invalidation messages more bandwidth-efficient, the interconnect supports sending them as a single fan-out multicast. For both read and write requests, if the directory is the owner, the directory responds with data, otherwise the directory forwards the request to the processor that owns the block. The owner's data response message includes a field informing the requester how many acknowledgements to expect (the directory forwards this information to the owner in the case where the block is not owned by the directory itself). Once the original requester has received all the expected acknowledgements, it sends a deactivation message to the home node to update the directory based on the requester's new coherence state and deactivate the request. This deactivation unblocks the directory for that block, allowing the next queued request for that block (if any) to proceed.

This baseline protocol supports the MOESI states [30] plus a migratory sharing optimization. To reduce accesses to DRAM, this protocol uses the dirty-owner (O) state and a clean-owner state (F) [14]. To increase the frequency with which some cache is the owner, ownership of the block transfers to the most recent requester on both read and write misses. The protocol uses the exclusive-clean (E) state to avoid upgrade misses to non-shared data (but it does not support silent eviction of blocks in the E state).

5.2 PATCH Implementation

As discussed in Section 3, PATCH adds token counting to the baseline directory protocol to enable direct requests and avoid unnecessary acknowledgement messages. PATCH makes four changes to the baseline directory protocol: (1) adding token state, (2) enforcing coherence via token counting, (3) supporting direct requests, and (4) providing a token timeout mechanism to support token tenure. We discuss each of these changes below.

Adding token state. PATCH adds an additional *token count* field to directory entries, cache tags, data response messages and data-less acknowledgement response messages. When responding to requests, processors use this token count field to send their tokens to the requester. The token count is encoded using $logN$ bits for N cores plus a few bits for identifying the owner token and its clean/dirty status. Ten bits would comfortably hold the token state for a 256-core system. For 64-byte cache blocks, this adds only about 2% overhead to caches and data response messages. To ensure conservation of tokens, processors may not silently evict clean blocks, so they instead send a data-less message with a token count back to the directory.

Enforcing coherence via token counting. PATCH uses token counting for completing requests (Section 2). Data responses always contain the owner token plus zero or more additional tokens, and a single token is sufficient for completing a read miss. Instead of waiting for a specific number of invalidation acknowledgements to arrive to complete a write miss, the requester counts tokens and completes the miss when all tokens have arrived. Because

token counting (and not ack counting) is used to complete misses, the protocol does not send acknowledgement messages that would have a zero token count.

Supporting direct requests. In addition to its regular request sent to the directory, a requester may also send request messages directly to one or more other processors. Processors that have a miss outstanding to the block always ignore these direct requests. Otherwise, the processors respond to direct requests exactly as they would respond to forwarded requests. When activating a request, the directory responds and/or forwards the request to the owner and/or sharers exactly as in the baseline directory protocol (independent of whatever direct messages were sent for the request). Processors always respond to requests forwarded from the directory, even if they have an outstanding miss to the block.

Token tenure mechanism. As discussed in the previous section, token tenure requires three system mechanisms: (1) a mechanism for fairly activating requests one-at-a-time on a per-block basis (and informing a requester that it has been activated), (2) the ability to send a forwarded message to (at least) the set of processors holding tenured tokens on activating a block, and (3) a mechanism for processors to determine when to give up untenured tokens to the home. To support the first two mechanisms PATCH leverages existing properties of the baseline protocol. To support the third mechanism PATCH adds a timer per outstanding request at the processors.

To activate requests one-at-a-time, PATCH leverages the directory's property of processing requests serially on a per-block basis. PATCH informs a requester that it has been activated by reusing the "acks to expect" field (which is not necessary in PATCH) to inform the requester has been activated. **Becoming activated is typically not on the critical path of misses because the processor can access a requested block as soon as it has enough tokens to do so.** Once the requester is both active and has sufficient tokens, it sends a deactivation message to the directory (as would the baseline directory protocol).

PATCH uses the directory to track which caches have tenured tokens, ensuring that it can forward requests to all processors with tenured tokens when activating a request. When the directory receives a deactivation message, it uses the included coherence state of the processor to update the block's directory entry. Because only active processors tenure tokens, the sharers set is a (possibly non-strict) superset of the set of caches holding tenured tokens at any given point in time.

Finally, each processor adds a timer per outstanding request to implement token tenure's timeout mechanism. To reduce the performance impact of racing requests, PATCH does two things. First, it sets the value of the bounce timeout to twice the average round trip latency; if the processor has not seen an activation request after this amount of time, then it is likely that another processor has been activated for the block. Second, PATCH reuses the timer when a processor sends its deactivation message. During this "use timeout" interval, processors continue to ignore direct

requests. This second interval is strictly a performance optimization, but during times of extreme contention, it gives the directory the ability to direct the movement without interference from direct requests.

6 Prediction and Bandwidth Adaptation in PATCH

This section describes PATCH's predictive and adaptive use of direct requests. PATCH uses previously proposed destination-set predictors for selecting the recipients of direct requests. PATCH, however, enhances the utility of destination-set prediction by sending all direct requests as low-priority, best-effort messages to achieve a natural bandwidth adaptivity. This optimization prevents direct request messages from ever harming performance, enabling PATCH to be more profligate in its prediction.

The goal of destination-set prediction [4, 22] is to send direct requests only to those processors that need to see the request (in PATCH's case, all token holders for a write and the owner token holder for a read). Processors determine what predictions to make by tracking the past behavior of blocks (recording which processors have sent incoming requests or responses). Predictors can make different bandwidth/latency tradeoffs ranging from predicting a single additional destination (*i.e.*, the owner) to all processors that have requested the block in the recent past. Our goal in this work is not to devise new predictors. In fact, PATCH uses predictors taken directly from prior work [22]

One challenge with destination-set prediction, however, is that the predictor that obtains the optimal bandwidth/latency tradeoff varies based on the specific system configuration and workload characteristics [25]. BASH [25] proposed using all-or-nothing *throttling* to disable the broadcasting of direct requests when a local estimate of the global interconnect utilization indicates the interconnect is highly utilized. BASH was shown to be effective in adapting to system configuration and workload in the context of a multicast snooping protocol on a totally ordered crossbar interconnect, but the interconnect congestion caused by direct requests can reduce performance to less than that of a directory protocol [25].

Instead of deciding between all or nothing at the time a processor issues a request, PATCH uses a form of bandwidth adaptivity that operates via low-priority best-effort messages. The interconnect gives direct requests strictly lower priority than all other messages. If a direct message has been queued at a switch for a long time, the interconnect eventually drops the stale message. This best-effort approach to bandwidth adaptivity is enabled by the property that PATCH's direct requests are simply performance hints that are not necessary for correctness; the indirect request sent through the directory ensures forward progress independent of any additional direct requests.

De-prioritizing direct requests allows the system to benefit from whatever bandwidth is available for delivering them *without* slowing down other messages, including indirect requests. Thus, to the first order, PATCH with best-

effort delivery will perform no worse than the baseline directory protocol (a "do no harm" guarantee not provided by prior bandwidth adaptivity proposals).

Best-effort delivery may also simplify the interconnect. Guaranteed multicast (or broadcast) delivery requires additional virtual channels and sophisticated buffer management to prevent routing deadlocks [10, 15]. In contrast, best-effort multicast avoids many of these issues, because deadlock can be avoided by simply dropping best-effort messages. Furthermore, whereas general multicast is necessary for destination-set prediction to be most effective, our experimental results indicate that broadcasting best-effort direct requests is highly effective (see Section 8.4). This implies that the interconnect design may eschew support for generalized multicast in favor of just the simpler case of efficient broadcast.

7 More Scalable Directory Protocol

PATCH is more tolerant of inexact directory state than the baseline directory protocol because PATCH avoids unnecessary acknowledgments. A full-map bit vector (one bit per core) becomes too much state per directory entry as the number of core grows. For this reason, inexact encodings of sharer information (*i.e.*, conservative over-approximations) have been proposed [12, 18]. Inexact encodings result in additional traffic due to an increased number of forwarded requests and acknowledgment messages.

Employing fan-out multicast (as our baseline protocol does) reduces the traffic incurred by extra forwarded requests. Unfortunately, this optimization has no impact on unnecessary acknowledgments. On an N-processor D-dimensional torus interconnect supporting fan-out multicast, for example, the worst-case traffic cost of unnecessary acknowledgments in our baseline directory protocol is $N \times \sqrt[D]{N}$ while that of unnecessary forwarded requests is only N.

In PATCH only token holders (*i.e.*, true sharers) send acknowledgment messages on receiving a forwarded request from the directory. Thus, PATCH avoids the unnecessary acknowledgments created by inexact directory encodings. As a result, PATCH's worst-case unnecessary forward+invalidation traffic scales more gracefully than that of the baseline directory protocol (N rather than $N \times \sqrt[D]{N}$ in the above example). We experimentally confirm the impact of this property in Section 8.6.

8 Experiments

This section experimentally shows that PATCH's use of prediction and best-effort direct requests coupled with its elimination of unnecessary acknowledgement messages allows it to (1) obtain higher performance than a directory protocol without sacrificing scalability and (2) out-scale directory protocols for inexact directory encodings.

8.1 Methods

We use the Simics full-system multiprocessor simulator [20] and GEMS [24]. GEMS/Ruby builds on Simics' full-system simulation to model simple single-issue cores with a detailed cache coherent memory system timing model. Each core's instruction and data caches are 64KB, and each core has a 12-cycle private 1MB second-level cache. All caches have 64-byte blocks and are 4-way set associative. Off-chip memory access latency is the 80-cycle DRAM lookup latency plus multiple interconnection link traversals to reach the block's home memory controller. We assume an on-chip directory with a lookup latency of 16 cycles. The interconnect is a 2D-torus with adaptive routing, efficient multicast routing, and a total link latency of 15 cycles. If not otherwise specified, the throughput of each link bandwidth is 16 bytes per cycle.

We use two scientific workloads from the SPLASH2 suite [31] (barnes and ocean) and three commercial workloads from the Wisconsin Commercial Workload Suite [3] (oltp, apache, and jbb). We simulate these workloads on a 64-core SPARC system by running four 16-core copies of the same workload concurrently. We perform multiple runs with small random perturbations and different random seeds to plot 95% confidence intervals [3]. To evaluate scalability up to 512 cores, we use a simple microbenchmark wherein each core updates a random entry in a shared, fixed-size table (16k locations) 30% of the time and reads a random entry 70% of the time.

8.2 Comparison of PATCH to DIRECTORY and TokenB

The first two bars of each group in Figure 8 show that PATCH configured not to send any direct requests (PATCH-NONE) and DIRECTORY perform similarly, which shows that there is no common-case performance penalty introduced by PATCH's token counting and token tenure mechanism. The interconnect link traffic (the first two bars in each group of Figure 9) shows that PATCH's data and request traffic are the same as DIRECTORY. PATCH's overall traffic is somewhat higher (only 2.2% on average) because of its non-silent writebacks of clean shared blocks and its few home-to-requester messages for activation on owner upgrade misses.

The final two bars in each group of Figure 8 show that PATCH configured to sends direct requests to all other cores on each miss (PATCH-ALL) generally performs the same as token coherence's broadcast-based TokenB [23]. Overall traffic (Figure 9) is also similar because of two largely offsetting effects: TokenB's reissued requests increase its traffic and PATCH's indirect requests, forwarded requests, and activations increase its traffic.

8.3 Impact of Direct Requests and Destination-Set Prediction

Comparing PATCH-NONE and PATCH-ALL (in Figure 8 and Figure 9) highlights the impact and cost of direct requests: direct requests improve runtime (22% for oltp, 19% for apache, and by 14% on average), at the cost of increasing traffic

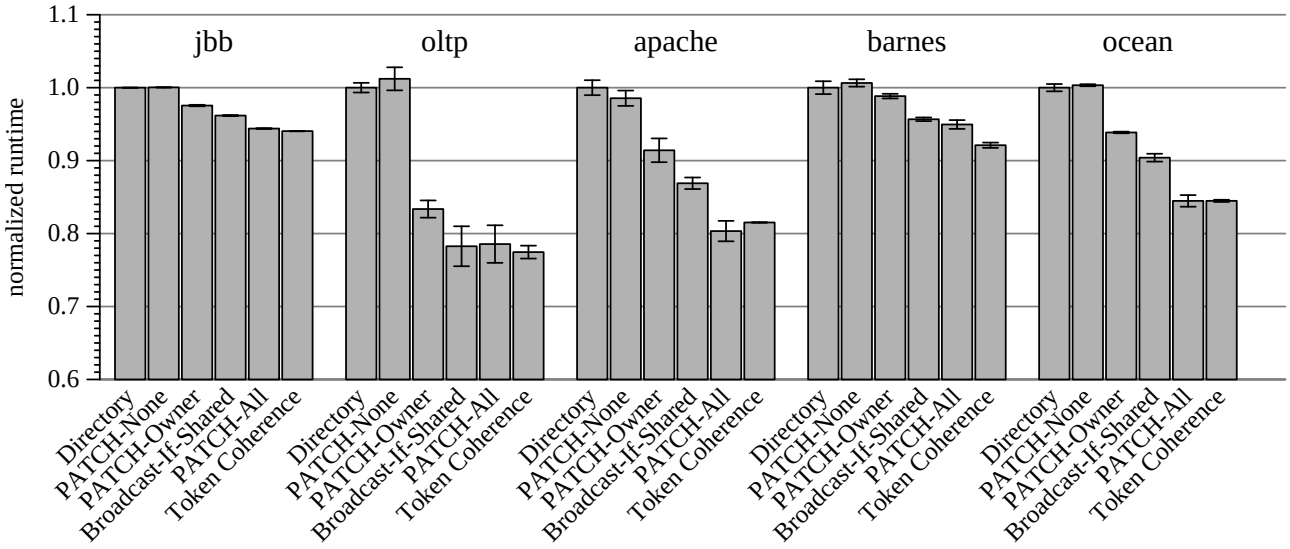

Figure 8: PATCH runtime

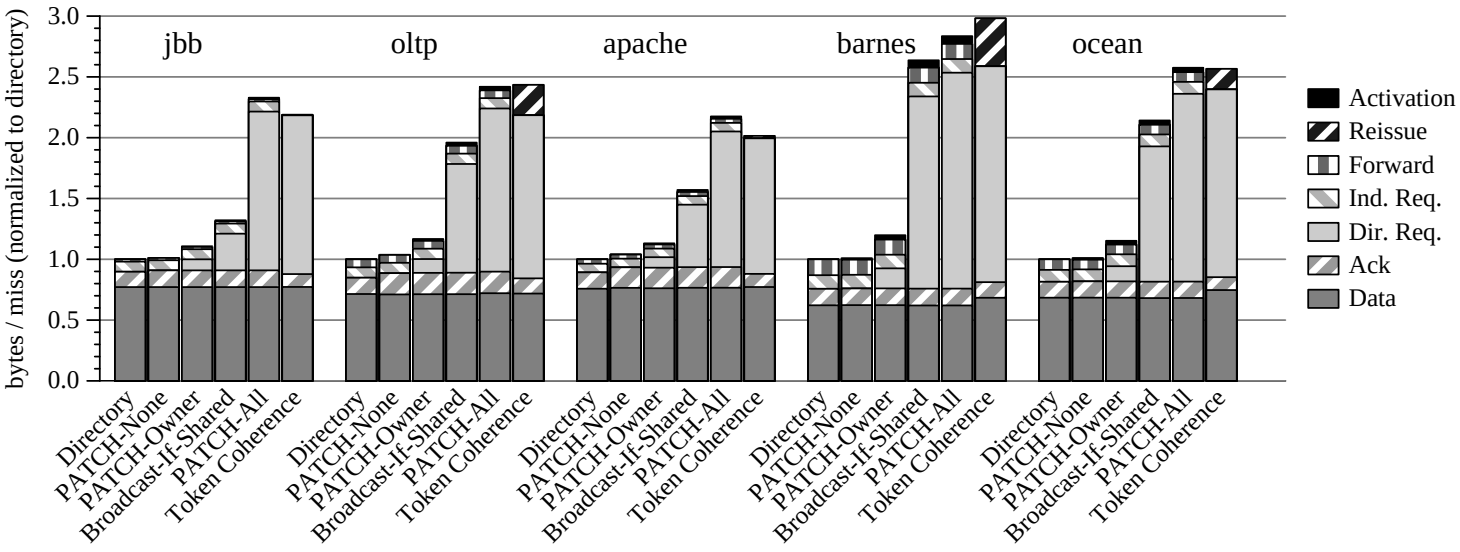

Figure 9: PATCH traffic

by 145% on average. For our bandwidth-rich baseline system configuration, PATCH-ALL's additional traffic results in little actual queuing in the interconnect. Thus, direct requests provide a significant performance improvement.

To explore different latency/bandwidth trade-offs, the middle bar of these figures show the effects of using PATCH with two previously-published destination-set predictors [22]. In the first (PATCH-OWNER), PATCH sends a direct request to a single core (*i.e.*, the predicted owner) in addition to the indirect request to the directory; in the second (PATCH-BROADCASTIFSHARED), PATCH sends direct requests to all cores for recently shared blocks.

PATCH-OWNER achieves speedups over PATCH-NONE that are about half those of PATCH-ALL (7% on average), while its additional direct requests cause only a 20% increase in traffic on average (versus PATCH-ALL's 145%).

PATCH-BROADCASTIFSHARED uses 22% less traffic on average than PATCH-ALL while achieving a runtime within 4% of PATCH-ALL. These results show that destination-set prediction can be valuable in cases of constrained bandwidth or where the extra power consumed by direct requests is a concern.

8.4 Bandwidth Adaptivity Under Constrained Bandwidth

We next study the impact of bandwidth adaptivity via best-effort requests in PATCH. Figure 10 and Figure 11 show the impact of limiting the available interconnect bandwidth by varying the link bandwidth for two representative benchmarks (the other three are qualitatively similar). The three lines show the runtime normalized to DIRECTORY for three configurations: DIRECTORY, PATCH-ALL, and a variant of PATCH-ALL that uses guaranteed delivery for all messages (PATCH-ALL-NONADAPTIVE). When bandwidth is plentiful, PATCH-ALL-NONADAPTIVE and PATCH-ALL identically outperform DIRECTORY. When bandwidth is scarce, however, PATCH-ALL-NONADAPTIVE's runtime quickly increases to be worse than that of DIRECTORY. PATCH-ALL's runtime, in contrast, always stays at or better than DIRECTORY. Furthermore, in the middle of the graph, where there is enough bandwidth for *some* but not *many* direct requests, PATCH-ALL is actually faster than the other configurations (by as much as 6.3% for ocean and 5.2% for jbb).

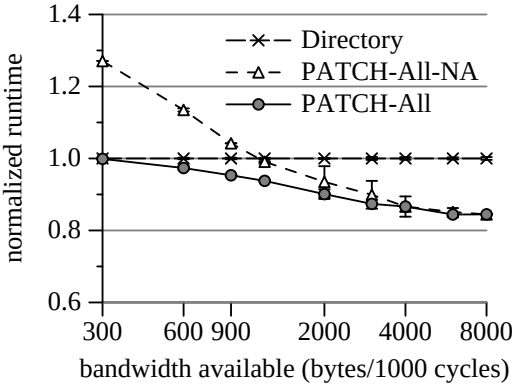

Figure 10: ocean

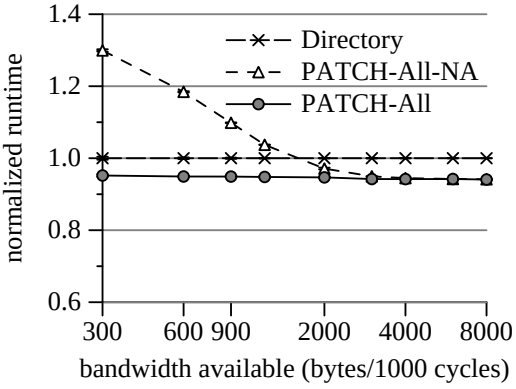

Figure 11: jbb

8.5 PATCH's Scalability

We next show that PATCH's best-effort requests enable it to match the scalability of DIRECTORY. Figure 12 show the microbenchmark's runtime of DIRECTORY, PATCH-ALL, and PATCH-ALL-NONADAPTIVE (as defined above) with a link bandwidth of two bytes per cycle on four cores to 512 cores. PATCH-ALL-NONADAPTIVE performs significantly better than DIRECTORY up to 64 cores but sharply worse from

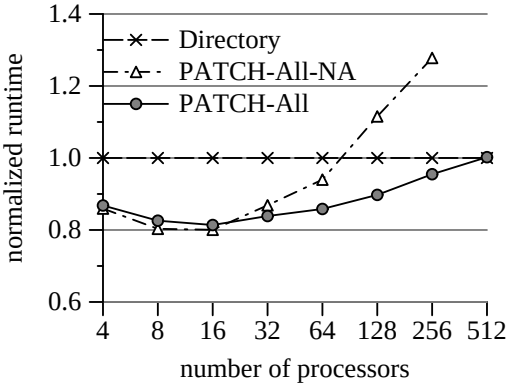

Figure 12: microbenchmark scalability

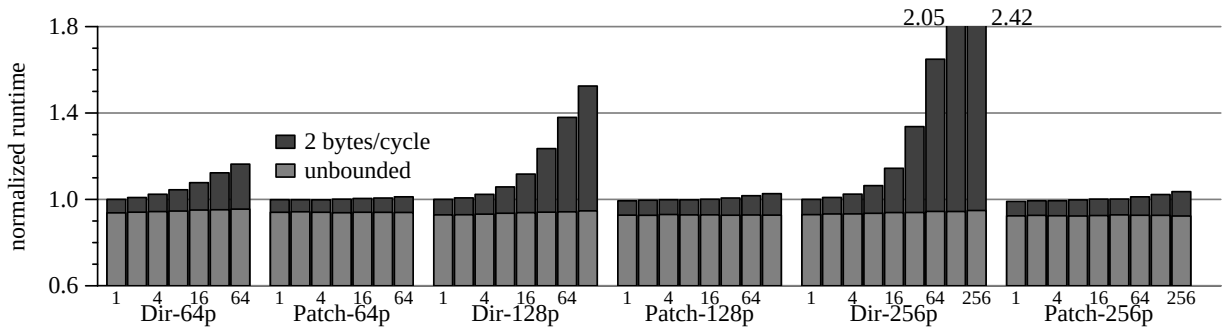

Figure 13: Scalability of PATCH vs DIRECTORY: runtime

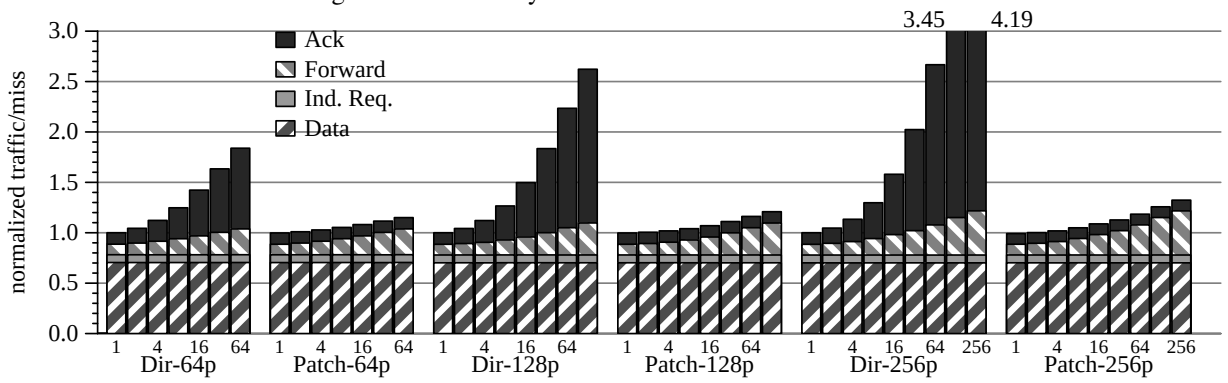

Figure 14: Scalability of PATCH vs DIRECTORY: traffic

128 cores onward (our simulation of PATCH-ALL-NONADAPTIVE on 512 cores did not complete). PATCH-ALL, by contrast, matches both the performance of PATCH-ALL on low numbers of cores and the scalability of DIRECTORY on a large number of cores. PATCH-ALL outperforms directory up to 256 cores, which shows that even for reasonably large systems, direct requests provide some benefit (without sacrificing scalability).

8.6 Scalability with Inexact Directory Encodings

The above experiments show that PATCH scales as well as DIRECTORY when using a full-map encoding of directory state. We now experimentally show that PATCH can scale better than DIRECTORY, when the directory encoding is inexact (as discussed earlier in Section 7).

To isolate this effect, we compare PATCH-NONE and DIRECTORY on the microbenchmark using varying degrees of inexactness of directory encoding. In the simulated inexact encoding, the owner is always recorded precisely (using $\log n$ bits), making all read requests exact. Encoding of additional sharers uses a coarse bit vector that maps 1-bit to K-cores, and the experiment varies K from 1 (which is a full map) to N (which is a single bit for all the cores). Figure 13 show the runtime for varying levels of coarseness normalized to a full-map bit vector for 64, 128, and 256 cores. With unbounded link bandwidth (the lower portion of each bar) the runtimes are all similar. When link bandwidth is bound to 2 bytes per cycle (the total bar height), the runtime of DIRECTORY for 128 and 256 cores show substantial increases

(up to 142%); in contrast, PATCH's runtime increases by only 3.6% in the most extreme configuration of a single bit to encode the sharers for 256 cores. Figure 14 shows the traffic of DIRECTORY is dominated by acknowledgements message under extreme coarseness (319% more traffic than full-map directory on 256 cores). PATCH's elimination of unnecessary acknowledgements prevents them from dominating the overall traffic (a maximum of only 32% more traffic than the full-map baseline).

9 Conclusion

In this paper we have introduced PATCH (Predictive Adaptive Token Coherence Hybrid), a protocol that obtains performance without sacrificing scalability by augmenting a standard directory protocol with token counting and a novel broadcast-free forward progress mechanism called *token tenure*. The combination of token counting and token tenure allows PATCH to easily support direct requests over an unordered interconnect *without* relying on broadcast messages for any part of correctness. When bandwidth is plentiful, PATCH displays the performance characteristics of high-performance broadcast-based protocols such as TokenB. When bandwidth is scarce, PATCH displays the scalability of a directory protocol. In fact, PATCH out-scales a directory protocol when using inexact directory encodings. PATCH employs a novel form of bandwidth adaptation, *best-effort requests*, wherein the interconnect deprioritizes direct requests during times of contention. The combination of best-effort requests and destination-set prediction allows PATCH to transition smoothly between the extremes of broadcast and directory protocols for different system configurations. These properties result in a protocol that is both high-performance and highly scalable.

References

[1] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner Prediction for Accelerating Cache-to-Cache Transfers in a cc-NUMA Architecture. In *Proceedings of SC2002*, Nov. 2002.

[2] M. E. Acacio, J. González, J. M. García, and J. Duato. The Use of Prediction for Accelerating Upgrade Misses in cc-NUMA Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 155–164, Sept. 2002.

[3] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[4] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Network. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 294–304, May 1999.

[5] S. Burckhardt, R. Alur, and M. M. K. Martin. Verifying Safety of a Token Coherence Implementation by Parametric Compositional Refinement. In *Proceedings of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, Jan. 2005.

[6] L. Cheng, J. B. Carter, and D. Dai. An Adaptive Cache Coherence Protocol Optimized for Producer-Consumer Sharing. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, pages 328–339, Feb. 2007.

[7] B. Cuesta, A. Robles, and J. Duato. An Effective Starvation Avoidance Mechanism to Enhance the Token Coherence Protocol. In *Proceedings of the 15th EUROMICRO International Conference on Parallel, Distributed, and Network-Based Processing (PDP'07)*, 2007.

[8] B. Cuesta, A. Robles, and J. Duato. Improving Token Coherence by Multicast Coherence Messages. In *Proceedings of the 16th EUROMICRO International Conference on Parallel, Distributed, and Network-Based Processing (PDP'08)*, 2008.

[9] Z. Cvetanovic. Performance analysis of the Alpha 21364-based HP GS1280 multiprocessor. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 218–229, June 2003.

[10] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[11] R. Fernandez-Pascual, J. M. Garcia, M. E. Acacio, and J. Duato. A Low Overhead Fault Tolerant Coherence Protocol for CMP Architectures. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, Feb. 2007.

[12] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *International Conference on Parallel Processing (ICPP)*, volume I, pages 312–321, 1990.

[13] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the Fifth Symposium on High-Performance Computer Architecture*, pages 172–181, Jan. 1999.

[14] H. H. J. Hum and J. R. Goodman. Forward State for use in Cache Coherency in a Multiprocessor System, July 2005. U.S. Patent 6,922,756.

[15] N. E. Jerger, L.-S. Peh, and M. Lipasti. Virtual Circuit Tree Multicasting: A Case for On-Chip Hardware Multicast Support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, June 2008.

[16] N. E. Jerger, L.-S. Peh, and M. H. Lipasti. Circuit-Switched Coherence. In *Proceedings of the IEEE International Symposium on Networks-on-Chip*, Apr. 2008.

[17] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, March-April 2003.

[18] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[20] P. S. Magnusson *et al.* Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[21] M. M. K. Martin. *Token Coherence*. PhD thesis, University of Wisconsin, 2003.

[22] 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. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 206–217, June 2003.

[23] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193, June 2003.

[24] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[25] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth Adaptive Snooping. In *Proceedings of the Eighth Symposium on High-Performance Computer Architecture*, pages 251–262, Feb. 2002.

[26] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*, Feb. 2005.

[27] M. R. Marty and M. D. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006.

[28] M. R. Marty and M. D. Hill. Virtual Hierarchies to Support Server Consolidation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[29] A. Meixner and D. J. Sorin. Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, Feb. 2007.

[30] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[31] 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 24–37, June 1995.