Token Tenure and PATCH: A Predictive/Adaptive Token-Counting Hybrid

ARUN RAGHAVAN, COLIN BLUNDELL, and MILO M. K. MARTIN
University of Pennsylvania

Traditional coherence protocols present a set of difficult trade-offs: the reliance of snoopy protocols on broadcast and ordered interconnects limits their scalability, while directory protocols incur a performance penalty on sharing misses due to indirection. This work introduces PATCH (Predictive/Adaptive Token-Counting Hybrid), a coherence protocol that provides the scalability of directory protocols while opportunistically sending direct requests 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 mechanism called *token tenure* provides broadcast-free forward progress using the directory protocol's per-block point of ordering at the home along with either timeouts at requesters or explicit race notification messages.

PATCH makes three main contributions. First, PATCH introduces token tenure, which provides broadcast-free forward progress for token-counting protocols. Second, PATCH deprioritizes best-effort direct requests to match or exceed the performance of directory protocols without restricting scalability. 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.

Categories and Subject Descriptors: C.5.5 [**Computer Systems Implementation**]: Servers; C.4 [**Performance of Systems**]

General Terms: Design, Performance

Additional Key Words and Phrases: Cache coherence protocol, token coherence, predictive, adaptive, bandwidth-efficiency

ACM Reference Format:

Raghavan, A., Blundell, C., Martin,and M. M. K. 2010. Token Tenure and PATCH: A predictive/adaptive token-counting hybrid. ACM Trans. Architec. Code Optim. 7, 2, Article 6 (September 2010), 31 pages.
DOI = 10.1145/1839667.1839668 http://doi.acm.org/10.1145/1839667.1839668

This work is supported in part by NSF CAREER award CCF-0644197 and donations from Intel Corporation and Sun Microsystems.
This journal submission is an extension of Raghavan et al. [2008], *Token Tenure: PATCHing Token Counting Using Directory-Based Cache Coherence*, presented at the International Symposium on Microarchitecture (MICRO).
Author's address: A. Raghavan, C. Blundell, and M. Martin, Department of Computer and Information Science University of Pennsylvania, Philadelphia, PA 19104; email: arraghav@cis.upenn.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 1544-3566/2010/09-ART6 $10.00
DOI 10.1145/1839667.1839668 http://doi.acm.org/10.1145/1839667.1839668

1. INTRODUCTION

A multicore 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 trade-offs. Snoopy protocols maintain coherence by having each processor broadcast requests directly to all other processors using a totally ordered interconnect [Agarwal et al. 2009; Charlesworth 1998; Martin et al. 2000] or a physical or logical ring [Marty and Hill 2006; Strauss et al. 2006; Strauss et al. 2007; Tendler et al. 2002]. This approach can provide low sharing miss latencies but does not scale due to excessive traffic and interconnect constraints. 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 [Bilir et al. 1999; Martin et al. 2003] or bandwidth adaptivity [Martin et al. 2002] 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 constraining their implementation. An alternative approach adds direct requests to a directory protocol (e.g., Acacio et al. [2002a, 2000b], Cheng et al. [2007], Ros et al. [2008], Jerger et al. [2008b]), but these proposals target only specific sharing patterns or introduce significant implementation complexities for handling races.

This article describes PATCH (Predictive/Adaptive Token-Counting Hybrid), a protocol that achieves high performance without sacrificing scalability. PATCH relies neither on broadcast nor an ordered interconnect for correctness. PATCH uses unconstrained predictive direct requests as performance hints that may be dropped at times of interconnect congestion. Furthermore, PATCH requires only true sharers to acknowledge requests. This property can allow PATCH to scale better than a directory protocol when directory encodings are inexact.

PATCH obtains these attributes by combining token counting and a standard directory protocol. Token counting [Martin et al. 2003a, 2003b] directly ensures coherence safety: 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. Token counting allows PATCH to naturally and simply support direct requests and destination-set prediction without requiring a nonscalable interconnect [Martin 2003].

Although adding token counting to a directory protocol enables indirection-free sharing misses, PATCH also inherits the challenge of ensuring forward progress on introduction of direct cache-to-cache requests. Previously proposed mechanisms to ensure forward progress in token-counting protocols rely on broadcast, a requirement that we expressly want to avoid in PATCH.

To meet this challenge without relying on broadcast, this article proposes *token tenure*, which leverages the directory protocol underpinnings of PATCH. Token tenure allows tokens to move in response to direct requests, but it requires that processors must eventually relinquish any tokens to the directory if they are not tenured: that is, if the processor does not see an activation message from the directory granting permission to retain the tokens. When races occur, token tenure ensures forward progress because the directory activates only one of the racing processors; untenured tokens held at other processors will eventually

Table I. Token-Counting Rules

Rule
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).

flow to the winning processor, allowing it to complete its request. Untenured tokens can be identified either (i) implicitly, upon not receiving an activation message from the directory after a certain amount of time or (ii) explicitly, upon receiving a message from the directory when a racing request exists. In either case, untenured tokens are relinquished in favor of the active requester by the racing requesters.

The combination of token counting and a directory protocol allows PATCH to employ a novel form of bandwidth adaptivity called *best-effort direct requests*. In PATCH, the home forwards requests to the owner and/or sharers (as in all directory protocols). Furthermore, only token holders (i.e., true sharers) must respond to requests (as in all token-counting protocols). These properties together allow PATCH to treat direct requests strictly as performance hints. Interconnect networks and coherence controllers deprioritize direct requests and process them only when there is sufficient bandwidth to do so, discarding them if they become too stale. This approach allows PATCH to be profligate in its destination-set prediction without hindering scalability by ensuring that direct requests never delay other messages.

This article makes three main contributions. First, token tenure provides broadcast-free forward progress for token-counting protocols because it does not require global consensus. Second, PATCH's best-effort direct requests enable it to match the performance of broadcast-based protocols when bandwidth is plentiful, while its deprioritization mechanism maintains the scalability of a directory protocol. Finally, PATCH avoids unnecessary acknowledgements because only processors holding tokens send acknowledgements, which can allow PATCH to scale better than a baseline directory protocol when using inexact encodings of sharers.

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 [Martin et al. 2003b]. Instead of enforcing the coherence invariant using a distributed algorithm for providing coherence safety in the presence of subtle races, token-based coherence protocols use token counting to enforce coherence permissions. At system initialization, the system assigns each block T tokens. One of the tokens is designated as the *owner token*, which can be marked as either clean or dirty. Tokens and data are allowed to move between system components as long as the system maintains the five token-counting rules given in Table I [Martin 2003]. Table II shows the correspondence

Table II. Mapping of Coherence States to Token Counts

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

between token counts and coherence states [Sweazey and Smith 1986; Hum and Goodman 2005; Tendler et al. 2002]. In particular, the token-counting rules require writers to hold all tokens and readers to hold at least one token. As the number of tokens per block is fixed, the token-counting rules ensure coherence safety irrespective of protocol races and without relying on interconnect ordering properties.

Races may still introduce protocol forward progress issues. Several mechanisms for guaranteeing forward progress in token-counting protocols have been proposed [Cuesta et al. 2007; Martin et al. 2003b; Marty et al. 2005; Marty and Hill 2006]. Token coherence uses persistent requests [Martin et al. 2003b; Marty et al. 2005] 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 [Martin et al. 2003b] or distributed arbitration [Marty et al. 2005] 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. To facilitate this mechanism, each processor must maintain a table of active persistent requests. Priority requests [Cuesta et al. 2007] are similar to persistent requests in that they are broadcast-based and use per-processor tables. Ring-Order [Marty and Hill 2006] uses broadcast on a unidirectional ring interconnect to ensure that 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.

Other proposals have explored token counting in the context of multisocket multicore systems [Marty et al. 2005], virtual hierarchical coherence [Marty and Hill 2007], fault-tolerant coherence [Fernandez-Pascual et al. 2007; Meixner and Sorin 2007], and multicast interconnection networks [Jerger et al. 2008a].

3. PATCH MOTIVATION AND OVERVIEW

PATCH is a directory-based cache coherence protocol augmented with token counting. The goal of PATCH is to obtain 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.

3.1 Predictive Direct Requests

Token counting provides flexibility to transfer coherence permissions without incurring directory indirection. By adding token counting to a directory protocol, PATCH inherits the

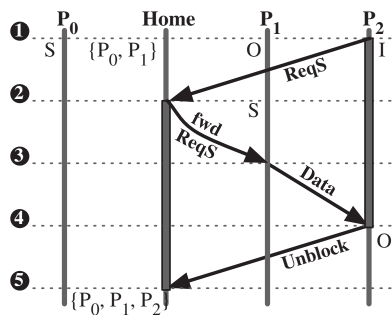


Fig. 1. Resolution of a load miss in a regular directory protocol. Initially, P_1 owns the block, with P_0 being a sharer. At time ❶, P_2 incurs a load miss and issues its "ReqS" to the home. On receiving this request at time ❷, the home forwards it to the owning processor P_1, which at time ❸ sends the data and transfers ownership to P_2. At time ❹, P_2 receives this message and may now cache the block in the Owned state and complete its outstanding load. In addition, it sends an "Unblock" message to the home, indicating that the request has been completed.

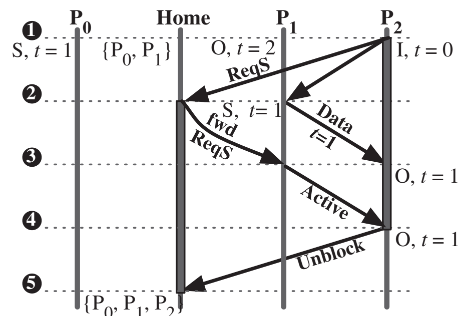


Fig. 2. Common case behavior of PATCH under the same conditions as Figure 1. The initial states of all caches and the memory are augmented with token counts, so that P_1 now holds two tokens while owning the block. At time ❶, P_2 issues its "ReqS" to P_1 in addition to the home. At time ❷, P_1 receives this and transfers data, ownership, and all but one of its tokens to P_2. Also at time ❷, the home forwards the request to P_1 per the DIRECTORY protocol. The home sets an additional bit in the forward message, indicating that P_1 is required to activate P_2. At time ❸, P_2 receives the response from P_1 and may cache the block and complete its load. Also at this time, P_1 receives the forwarded message from the directory, and on finding the activating field set, sends an "Activation" message to P_2 . At time ❹, P_2 receives the activation and deallocates any resources that might have been allocated for the load miss. P_2 also deactivates the home as with the DIRECTORY protocol.

same flexibility to use destination-set prediction to send direct requests to zero, some, or all other processors in the system [Martin 2003]. 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. Figure 1 shows the common-case behavior of a 3-hop load miss with a directory protocol. Figure 2 shows how PATCH satisfies the request in 2 hops, effectively removing the indirection latency from the critical path of the miss. Coherence is easily enforced with PATCH because the token-counting rules govern coherence permissions.

3.2 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 property allows protocols that use token counting to elide those acknowledgement messages that would have contained zero tokens. Avoiding these unnecessary acknowledgements 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 multidestination 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, token counting can also be used to avoid unnecessary acknowledgments for indirect requests [Marty and Hill 2007]. For large systems that employ an inexact set of sharers at the directory, avoiding these unnecessary acknowledgements enables PATCH to scale more gracefully than even a standard directory protocol.

3.3 Bandwidth Adaptivity via Best-Effort Direct Requests

PATCH uses a novel form of bandwidth adaptivity to achieve high performance without sacrificing scalability. Because the home continues to forward requests to the owner and/or sharers (as in the baseline protocol) and direct requests need not be acknowledged (as described earlier), 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 and coherence controllers deprioritize direct requests, simply dropping them if they become too stale. This approach ensures that direct requests never delay other messages, even when PATCH is sending direct requests to many processors.

3.4 Broadcast-Free Forward Progress via Token Tenure

The previously described attributes provide a framework for a fast, scalable, and adaptive protocol. However, races may prevent forward progress. Figure 3 provides an example of the problems caused by racing requests. To ensure forward progress without impeding scalability, PATCH introduces token tenure, a forward progress mechanism for token counting protocols that neither relies on broadcast nor requires any specific interconnect properties. Untenured tokens can be funneled to the active requester, using one of the following mechanisms.

—In PATCH-TIMEOUT, a processor that has received tokens for a block is required to discard these tokens after a bounded amount of time unless the home 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 home activates the next queued request, ensuring that all requests eventually complete. Figure 4 illustrates PATCH-TIMEOUT resolving the race from Figure 3.
—In PATCH-NOTIFY and PATCH-SPLIT, whenever the home receives a racing request while an active request is already pending to the same block, the home sends the racing requester an explicit notification message identifying the currently active requester. The racing requester then gives up any untenured tokens it may have received until such time it receives its own activation message, as illustrated in Figure 5. Whereas PATCH-NOTIFY requires worst-case endpoint buffering, PATCH-SPLIT splits each request into redundant queued and nonqueued request messages. The nonqueued requests are

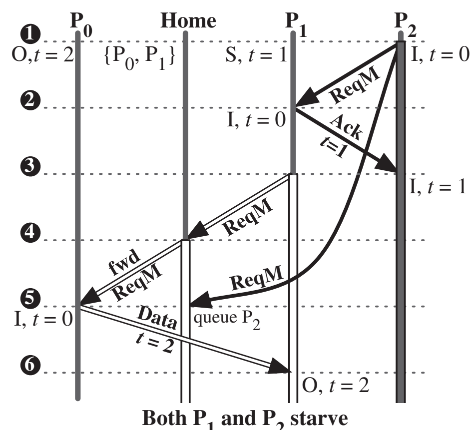


Fig. 3. Protocol race that can occur when direct requests and token counting are naïvely added to a directory protocol. Initially, P_0 has the owner token and one other token (placing it in the O state) and P_1 has one token (placing it in the S state). At time ❶, P_2 (solid lines) initiates a "ReqM" (a request for modified) to the home as well as sending a direct request to P_1. P_1 observes this message at time ❷ and responds with its token, which P_2 receives at time ❸. Also at time ❸, P_1 (hollow lines) sends a ReqM request to the home. The home observes P_1's indirect request at time ❹ and forwards it to P_0 (the only other processor in the directory's sharers list). P_0 receives the forwarded request at time ❺; also at time ❺, the home receives P_2's delayed request from time ❶ and queues it behind P_1's request. At time ❻, P_1 receives P_0's data and tokens. At this point, both P_1 and P_2 are waiting indefinitely for tokens that will never arrive.

always processed immediately upon receipt (i.e., without blocking) and they are sent on a separate virtual network, hence no additional buffering is required.

—To improve PATCH's performance when many requesters race for the same block, PATCH-CHAIN extends PATCH-NOTIFY by chaining the explicit activation messages to provide one-hop direct notification of subsequent active requests.

The next section elaborates on token tenure and the advantages and disadvantages of these variants.

4. TOKEN TENURE

The rules of token counting enforce coherence safety, but they say nothing about ensuring forward progress. In directory-based protocols, the directory ensures forward progress by tracking all sharing caches in the system. In such a setting, token movement between processors can only take place in response to forwarded requests from the directory. Introducing direct cache-to-cache requests invalidates this invariant by moving tokens between processors without the knowledge of the directory. Racing requests between two processors could be seen by the directory in a different order from other processors. Figure 3 gives an example of how such a race could result in starvation. This section introduces rules for restoring the directory's ability to ensure forward progress in token-counting protocols, based on a 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 processors until requested by another processor. In contrast, untenured tokens must become tenured. If not, the processor is required to discard the tokens (e.g., by writing them back to

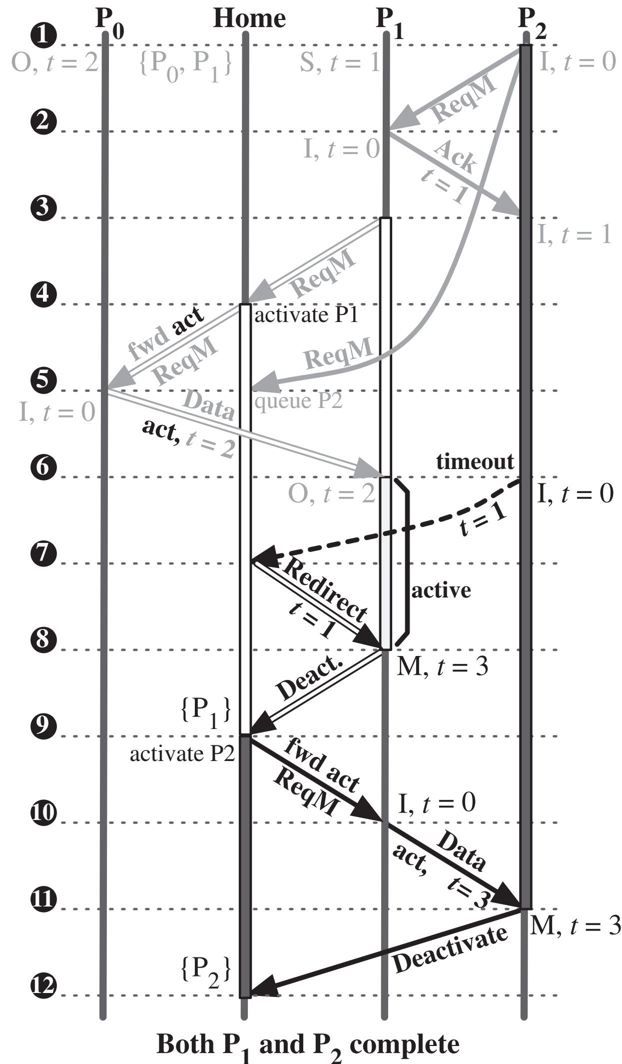


Fig. 4. Implicit denial of tenure with timeout. The operation of PATCH-TIMEOUT in this scenario proceeds initially as in Figure 3. However, when the home activates the request of P_1 at time ❹, it augments the forward to P_0 with the "activated bit." At time ❻, P_2 times out because it has not been activated and sends its token to the home. Also at time ❻, P_1 becomes active when it receives P_0's acknowledgement, which includes the activated bit. At time ❼, the home receives P_2's token. The home redirects the token to P_1. At time ❽, P_1 receives the redirected token, completes its request, and sends a deactivation message to the home. When the home receives this deactivation message at time ❾, it activates P_2, sending a forward to P_1 that includes the activation bit destined for P_2. At time ❿, P_1 receives this message and sends data, tokens, and the activation bit to P_2. At this time, P_2 completes its miss and sends a deactivation message to the home.

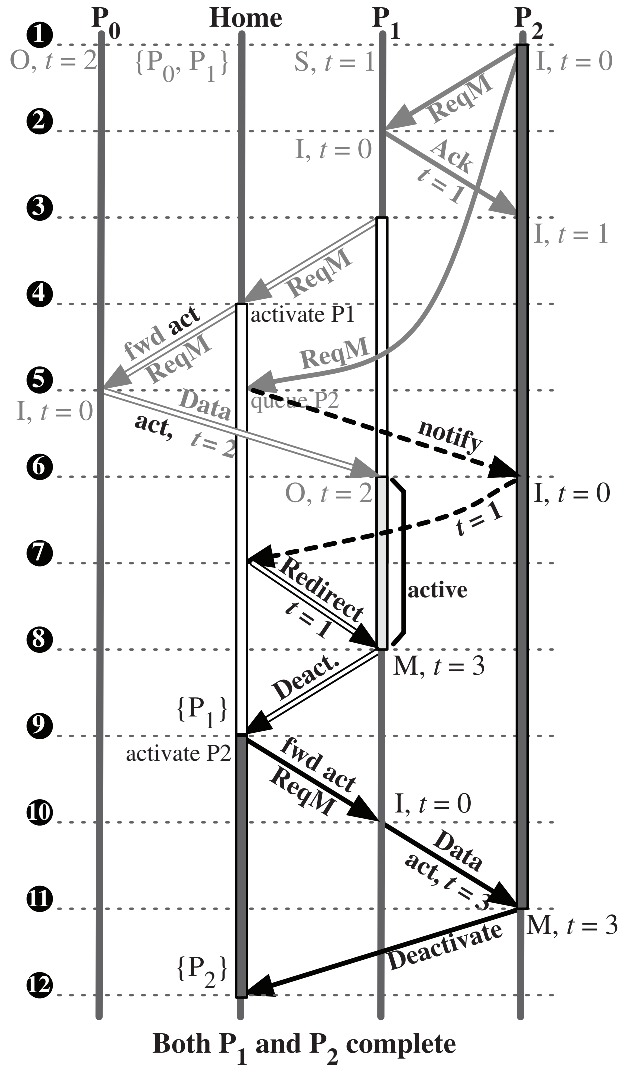


Fig. 5. Explicit denial of tenure with notification. In PATCH-NOTIFY, when the home receives P_2's request at time ❺, in addition to queuing it, the home sends an explicit "denial of tenure" notification message to P_2. At time ❻, P_2 receives this notification message and sends its transiently received token to the home. At time ❼, the home redirects this token to the active requester P_1, which receives the token at time ❽. The actions from time ❽ proceed just as with Figure 4, with P_1 and P_2 subsequently completing their requests.

the block's home memory module). This denial of tenure is inferred either implicitly after a bounded amount of time (PATCH-TIMEOUT), or from receiving an explicit notification message from the home (PATCH-NOTIFY). The tenured status is used only in providing forward progress; untenured tokens can be used to satisfy misses. Thus, the token tenure process is off the critical path for typical misses.

To tenure tokens, the home 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

Table III. Implict (PATCH-TIMEOUT) and Explicit (PATCH-NOTIFY) Token Tenure

<table>
<tr><td>Token tenure with timeout (PATCH-TIMEOUT)</td><td>Tenure with notification (PATCH-NOTIFY)</td></tr>
<tr><td colspan="2">Rule #1 Activation Rule: (a) The home fairly designates one request as the block's current active request and informs that requester of its activation, also responding to the request with tokens if present. (b) When activating a request (and only when activating a request), the home forwards the request to a superset of caches holding tenured tokens.</td></tr>
<tr><td colspan="2">Rule #2 (Token Arrival Rule): Tokens that arrive at a processor are by default untenured.</td></tr>
<tr><td colspan="2">Rule #3 (Promotion Rule): Only the active requester may tenure tokens, and it tenures all tokens it possesses or receives.</td></tr>
<tr><td>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.</td><td>Rule #4′ (Explicit Notification Rule): (a) When the home receives a request to a block with an active requester, the home sends an explicit race notification message to the conflicting requester. (b) If a requester receives an explicit race notification message, it discards its tokens (if any) to the home, and it continues to discard any tokens it receives until it is notified by the home that it has become the active requester. (c) Whenever a processor without an outstanding request receives tokens, it discards them to the home.</td></tr>
<tr><td colspan="2">Rule #5 (Home Redirect Rule): The home node redirects any discarded tokens to the active requester.</td></tr>
<tr><td colspan="2">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.</td></tr>
<tr><td colspan="2">Rule #7 (Deactivation Rule): Once the active requester has collected sufficient tenured tokens, it gives up its active status by informing the home.</td></tr>
</table>

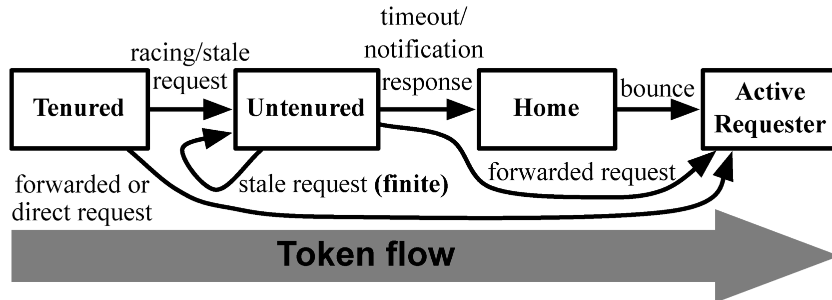


Fig. 6. Flow of tokens to the active requester.

which request is active. Once a processor has become the active requester, any tokens that it holds become tenured, as well as any further tokens that it receives while its request is active. Table III contains the complete set of rules for token tenure. Rules #4 and #4′ differentiate the timeout-based mechanism and the notification-based mechanism.

To verify that token tenure ensures that all requests eventually complete, we describe the flow of tokens it enforces (this flow is represented pictorially in Figure 6). Consider many racing requests to the same block. The home activates one of these requests (Rule #1*a*). The below discussion assumes this request is a write request (i.e., it seeks all tokens); for a read request, the same reasoning can be applied to just the owner token. Tokens that are not already tenured at the active requester may be either at the home, in-flight, untenured, or tenured at one or more nonactive processors.

Table IV. Comparison of Forward Progress Mechanisms Proposed for Token-Counting Protocols

Mechanism	Broadcast-free?	Interconnect	Reissues?	State at Processor	State at Home
Persistent/priority requests [Cuesta et al. 2007; Martin et al. 2003b; Marty et al. 2005]	no	any	yes	tokens & P.R. table	tokens
RingOrder [Marty and Hill 2006]	no	ring	no	tokens	1 bit
Token tenure	yes	any	no	tokens	tokens & sharers set

The home forwards all tokens that it holds to the active requester (Rule #1*a*) and also redirects any tokens that it subsequently receives to the active requester (Rule #5). Thus, any tokens that begin or arrive at the home will eventually arrive at the active requester.

Any in-flight tokens that arrive at a nonactive processor become untenured (Rule #2). Untenured tokens may not move via direct requests (Rule #6*c*). As such, they either (i) are denied tenure—via timeout (PATCH-TIMEOUT's Rule #4), an explicit race notification message (PATCH-NOTIFY's Rule #4′*a* and #4′*b*), or by not having an outstanding request for the block (PATCH-NOTIFY's Rule #4′*c*)—and are thus sent to the active requester via the home (Rules #4 and #5) or (ii) move in response to a forwarded request (Rules #1*b* and #6*b*). Forwarded requests usually cause tokens to move to the active requester, but a lingering (stale) forwarded request from a prior activation can move tokens to a nonactive processor. However, the number of such stale requests is bounded (Rule #1*b*), so 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 denial of tenure.

Finally, all tenured tokens either (i) move to the active requester via a direct or forwarded request or (ii) move to another (nonactive) processor, in which case they become untenured and the previously described reasoning applies.

Until the active requester has received its notification of activation, it acts like any other nonactive 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 earlier. 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. Once the active requester has been satisfied, it will then deactivate by informing the home (Rule #7). The home will then fairly select another requester to activate (Rule #1*a*), thus ensuring that all pending requests eventually become the active requester (and thus eventually complete).

Although the correctness reasoning for token tenure is somewhat subtle, the next section shows that its implementation requires only small extensions to a directory protocol. In particular, unlike prior proposals for ensuring forward progress in token-counting protocols (see Table IV), token tenure does not require broadcast [Cuesta et al. 2007; Martin et al. 2003b; Marty et al. 2005; Marty and Hill 2006], per-processor tables [Cuesta et al. 2007; Martin et al. 2003b; Marty et al. 2005], or a ring-based interconnect [Marty and Hill 2006].

5. IMPLEMENTATION AND OPERATION

This section describes the operation of specific implementations of PATCH. Although the conceptual framework of PATCH could likely be applied to any directory protocol, for concreteness, this section first describes a specific baseline directory protocol on which

our implementation of PATCH is built. This section then describes the modifications PATCH makes to this baseline protocol to support best-effort direct requests via token counting and token tenure. Section 6 describes the predictive and adaptive policies that use this direct request mechanism.

5.1 Baseline Directory Protocol

Our baseline protocol, DIRECTORY, is based on the blocking directory protocol distributed as part of GEMS [Martin et al. 2005]. DIRECTORY resolves races without negative acknowledgement messages (nacks) by using a busy/active state at the home for every request. Although this approach is different from the SGI Origin 2000 [Laudon and Lenoski 1997], it is reflective of more recent protocols such as Sun's WildFire [Hagersten and Koster 1999]. In this approach, the arrival order at the home unambiguously determines the order in which racing requests are serviced. DIRECTORY does not depend on any ordering properties of the interconnect.

When a request arrives at the home, it sets the block's state to active. All subsequent requests for that block are queued (at the home or in the interconnect) until the active request is deactivated. If the request is a write request, the home 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 home is the owner, the home responds with data; otherwise, the home forwards the request and the number of acknowledgements to expect to the processor that owns the block. The owner (be that either the home or a processor) includes the number of acknowledgements to expect in its data response message sent to the requester. 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. This deactivation also unblocks the home for that block, allowing the next queued request for that block (if any) to proceed.

DIRECTORY supports the MOESI [Sweazey and Smith 1986] and F [Hum and Goodman 2005] states plus a migratory-sharing optimization. To reduce memory accesses, DIRECTORY uses the dirty-owner (O) state and a clean-owner state (F) [Hum and Goodman 2005]. 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. DIRECTORY uses the exclusive-clean (E) state to avoid upgrade misses to nonshared data, but it does not support silent eviction of blocks in the E state.

5.2 PATCH's Modifications to DIRECTORY

PATCH adds token counting to the baseline directory protocol to enable best-effort direct requests and avoid unnecessary acknowledgement messages. PATCH makes four changes to DIRECTORY: (i) adding token state, (ii) enforcing coherence via token counting, (iii) supporting direct requests, and (iv) implementing token tenure. We discuss each of these changes below.

5.2.1 *Adding Token State.* PATCH adds an additional token count field to directory entries, cache tags, data response messages, and dataless acknowledgement messages. When responding to requests, processors use this token count field to send their tokens

to the requester. The token count is encoded using log N 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 dataless message with a token count back to the home.

5.2.2 *Enforcing Coherence via Token Counting.* PATCH uses token counting for completing requests. Only the owner supplies data, and these data responses always contain the owner token plus zero or more additional tokens. 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.

5.2.3 *Supporting Direct Requests.* In addition to its regular *indirect request* sent to the home, a requester may also send direct 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 home responds and/or forwards the request to the owner and/or sharers exactly as DIRECTORY (independent of whatever direct messages were sent for the request). Processors always respond to requests forwarded from the home, even if they have an outstanding miss to the block.

5.3 Token Tenure Mechanism

Token tenure requires three mechanisms: (i) a mechanism for fairly activating requests one-at-a-time on a per-block basis (and informing a requester that it has been activated), (ii) the ability to send a forwarded message to (at least) the set of processors holding tenured tokens upon activating a block, and (iii) a mechanism for processors to determine when to give up untenured tokens.

5.3.1 *Fair Activation.* To activate requests one at a time, PATCH leverages 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). 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 home (as would DIRECTORY).

5.3.2 *Forwarding Requests to Processor with Tenured Tokens.* 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 home 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 nonstrict superset of the set of caches holding tenured tokens at any given point in time.

5.3.3 *Denial of Tenure.* Token tenure requires a mechanism to redirect untenured tokens (held by processors with racing requests) to the active requester. PATCH-TIMEOUT

implements this by adding a timer per outstanding request at each processor. PATCH adaptively sets the value of the tenure timeout to twice the dynamic 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.

PATCH-NOTIFY relies on the home observing all racing requests and explicitly notifying the (nonactive) racing requests that they must discard their tokens. If the system has sufficient buffering to allow any home controller to ingest all incoming requests bound for it (i.e., each home can handle $n \times k$ pending requests for a system with n processors and k outstanding misses per processor), the home is guaranteed to observe all racing requests. Without such buffering, queueing at coherence controllers or in the interconnect could block racing requests and lead to deadlock.

As this worst-case buffering is likely expensive for large systems, PATCH-SPLIT is an alternative implementation that avoids this restriction at the cost of increasing interconnect traffic. To ensure the home observes all racing requests, in PATCH-SPLIT requesters send a nonqueued request to the home in addition to the regular request. Whereas the regular requests are queued and activated as normal, nonqueued requests are sent on their own virtual network and are never queued or otherwise blocked by the home, ensuring all nonqueued requests are eventually observed by the home. In PATCH-SPLIT, the home responds to a nonqueued request in one of three ways: (i) if the nonqueued request is from the currently active requester, the home can simply discard it; (ii) if the nonqueued request is from a processor other than the currently active requester, the home responds with an explicit race notification message; (iii) if there is no currently active requester, the home also sends a notification message. This conservative response is necessary to avoid deadlock, because another processor's request could arrive (and become active) and prevent the nonqueued request's corresponding request from being processed by the home. In all the three cases, the nonqueued request is immediately deallocated by the home. To avoid unnecessary denial of tenure in the common case when there are no racing requests, the requester sends the nonqueued request a few cycles after the original request.

These three token tenure mechanisms present different implementation trade-offs. PATCH-TIMEOUT has the overhead of a timer per outstanding request and, as with the token coherence's original forward progress mechanisms, PATCH-TIMEOUT inherits the performance stability and protocol verification concerns raised by timeouts. PATCH-NOTIFY and PATCH-SPLIT both avoid tenure timeouts, but have other overheads. PATCH-NOTIFY introduces hardware overhead at the home by requiring buffering sized to the maximum number of outstanding requests per processor. PATCH-SPLIT avoids extra buffering, but instead increases interconnect traffic by sending an extra nonqueued request per miss.

5.4 Reducing the Performance Impact of Races

To reduce the performance impact of racing requests, PATCH introduces a few mechanisms to mitigate the impact of racing requests.

5.4.1 *Use Timeouts.* Because many racing direct requests to the same block may cause tokens to inefficiently flow to processors other than the next active requester, for a short interval of time after completing a request, all variants of PATCH ignore direct requests (but not indirect requests, which are sent by the home). This use timeout reduces the impact of racing requests by providing a window for the home to orchestrate the movement of tokens to the next active requester without interference from direct requests. Such use

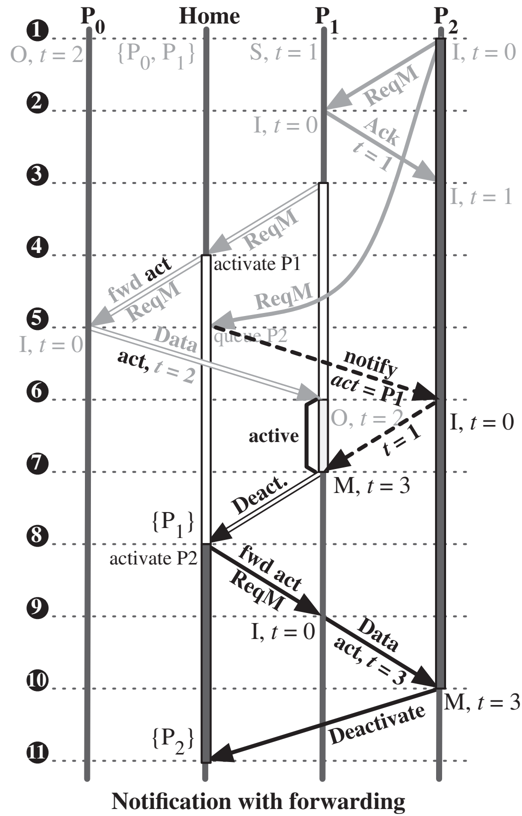


Fig. 7. PATCH-NOTIFY's notification with forwarding. In the example in Figure 5, the home issued a notification to P_2 at time ❺, resulting in writing back of tokens from P_2 to the home and redirection from the home to P_1. This can be optimized by including the active requester's identity in the notification message at time ❺. P_2 responds to this message by sending its token to P_1 at time ❻, allowing P_1's request to complete at time ❼. Consequently, P_1's deactivation at time ❽ allows P_2's request to be activated at this time and completed at time ❿.

timeouts mimic the round-trip delay present in DIRECTORY from when a requester sends its unblock request to when the home forwards the next request.

5.4.2 PATCH-NOTIFY *with Forwarding.* As described in Section 4 and Figure 5, requesters in PATCH-NOTIFY discard tokens to the home upon receipt of notification that another processor is the active requester. One simple enhancement to PATCH-NOTIFY is for the home to include the current active requester in the notification message, and then when a requester receives a notification message, it sends its tokens directly to the specified active requester (illustrated in Figure 7). This direct transfer of tokens avoids the latency of redirecting these discarded tokens through the home in most cases. As a requester will receive at most one notification message per request, this one-time forward to another processor retains token tenure's forward progress guarantee. Because of its simplicity of implementation, both PATCH-NOTIFY and PATCH-SPLIT incorporate this optimization.

5.4.3 PATCH-CHAIN . To further improve the performance impact of racing requests in PATCH, PATCH-CHAIN is a variant of PATCH-NOTIFY that chains consecutive requests

for the same block to provide direct transfer of tokens during periods in which many processors are repeatedly requesting the same block.[1] As before, there is at most one activation message for a given block in the system. However, instead of the home activating the next requester for the block, in PATCH-CHAIN each requester directly activates the next conflicting requester (if any), immediately once its own request has both received sufficient tokens and has been activated. Thus, the completion message to the home and the message to activate the next requester are sent in parallel (rather than being serialized).

The home establishes this per-block chain of successive active requesters by tracking the tail of the chain and then whenever the home receives a racing request it: (i) sends a message to the tail with the next requester's identity and (ii) records this most-recently received request as the new tail of the chain. These next activate messages contain the requester's identity and type of request, which the requester records upon receipt of the message. If the message recipient has no pending request for the block, it immediately sends an activate message to the specified next requester. Otherwise (i.e., a request is pending at the receipt of the next active message), upon completion of its own request (i.e., when the requester receives sufficient tokens), it sends an activation message to the next active requester, which includes zero or more tokens and data as appropriate based on the request type. Figure 8 illustrates the operation of PATCH-CHAIN.

Unlike PATCH-NOTIFY, in PATCH-CHAIN the order in which deactivation messages arrive at the home may not match the activating order. To ensure the home updates the directory state (owner and sharers) in the activation order, the home records the expected order of activations and reorders any out-of-order deactivation messages it may receive.

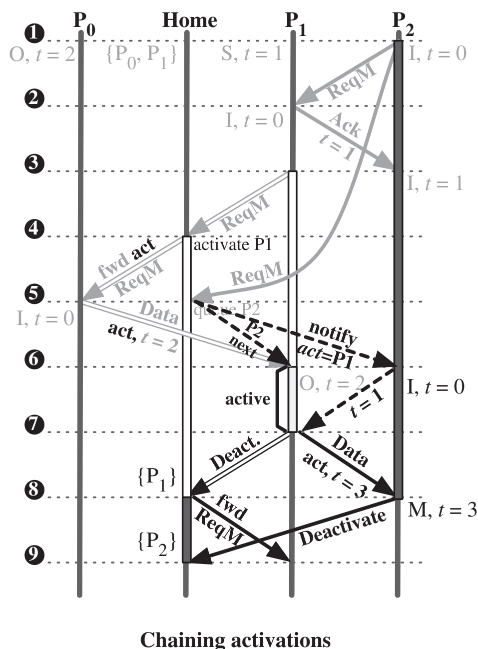
6. PREDICTION AND BANDWIDTH ADAPTATION

Inspired by a multicast variant of token coherence [Martin 2003], PATCH uses destination-set prediction 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 introducing harmful interconnect congestion.

The goal of destination-set prediction [Bilir et al. 1999; Martin et al. 2003] 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 trade-offs 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 [Martin et al. 2003].

One challenge with destination-set prediction, however, is that the predictor that obtains the optimal bandwidth/latency trade-off varies based on the specific system configuration and workload characteristics [Martin et al. 2002, 2003]. BASH used all-or-nothing throttling to disable the broadcasting of direct requests when a local estimate of the global

[1] PATCH-CHAIN can also be applied to PATCH-SPLIT in a straightforward manner. Although similar ideas could likely be applied to PATCH-TIMEOUT, it would likely require introducing some sort of activation notification messages—perhaps used selectively only when contention manifests—but we have not explored such an approach.



Chaining activations

Fig. 8. PATCH-CHAIN's chaining of activations. At time ❺, when the home receives P_2's racing request, along with the notification to P_2, the home also sends P_1 a message indicating that P_2's ReqM is the next active request. P_1 receives this message at time ❻ and stores the next active identifiers. At time ❼, when P_1 completes its request, it deactivates the home as before, but also activates P_2 and sends it data and all tokens. P_2 receives this message at time ❽, completes its request, and deactivates the home. At time ❽, the home recognizes P_2 to be the next active requester and forwards P_2's request to all sharers (P_1). P_1 receives this request at time ❾, but ignores it since it now holds no tokens.

interconnect utilization indicates the interconnect is highly utilized [Martin et al. 2002]. This approach 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 intermittent interconnect congestion caused by direct requests reduces performance to less than that of a directory protocol for some system configurations.

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 and cache snoop controllers give direct requests strictly lower priority than all other messages. If a switch or endpoint has queued a direct message for a long time, it 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. Deprioritizing 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 proposals).

Although adding a single low-priority virtual network introduces some design complexity, more sophisticated schemes than that assumed by PATCH have been implemented in high-performance systems. For example, the SGI Spider chip used in the Origin 2000 supports 256 levels of priority on a per-message basis [Galles 1997]. Furthermore, best-effort delivery has the potential to simplify the interconnect. Guaranteed multicast (or broadcast) delivery requires additional virtual channels and sophisticated buffer management to prevent routing deadlocks [Dally and Towles 2004; Jerger et al. 2008a]. In contrast, best-effort multicast avoids many of these issues, because deadlock can be avoided by simply dropping best-effort messages. Finally, our experimental results indicate that broadcasting best-effort direct requests is highly effective (see Section 8.7). This implies that the interconnect design could eschew support for generalized multicast in favor of the simpler case of best-effort broadcast.

7. SCALING BETTER THAN DIRECTORY

Protocols that use token counting are more tolerant of inexact directory state than DIRECTORY because such protocols avoid unnecessary acknowledgments. A full-map bit vector (one bit per core) becomes too much state per directory entry as the number of cores grows. For this reason, many inexact encodings of sharer information (i.e., conservative over-approximations) have been proposed (e.g., Gupta et al. [1990], and Laudon and Lenoski [1997]). Inexact encodings result in additional traffic due to an increased number of forwarded requests and acknowledgment messages. This additional traffic may be mitigated in several ways: for example, fan-out multicast (which reduces forwarded request traffic), acknowledgement combining (which reduces acknowledgment traffic), cruise missile invalidations [Barroso et al. 2000] (which reduces both), or using a totally ordered interconnection network to avoid explicit acknowledgements [Gharachorloo et al. 2000; Bilir et al. 1999].

DIRECTORY employs fan-out multicast to reduce the traffic incurred by forwarded requests. Unfortunately, this optimization does not reduce acknowledgment traffic. On an N-processor D-dimensional torus interconnect supporting fan-out multicast, for example, the worst-case traffic cost of unnecessary acknowledgments in DIRECTORY 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 DIRECTORY (N rather than $N \times \sqrt[D]{N}$ in the previous example).

The Virtual Hierarchies protocol exploited the same properties of token counting to avoid unnecessary acknowledgements in the context of a directory protocol [Marty and Hill 2007]. This prior work used the extreme case of a single-bit directory. In the next section, we experimentally explore this phenomenon over a range of directory encodings.

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 (i) achieve higher performance than DIRECTORY without sacrificing scalability, (ii) adapt

to varying amounts of available interconnect bandwidth, and (iii) out-scale DIRECTORY for inexact directory encodings.

8.1 Methods

We use the Simics full-system multiprocessor simulator [Magnusson et al. 2002] and GEMS [Martin et al. 2005]. 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 look-up latency plus multiple interconnection link traversals to reach the block's home memory controller. We assume an on-chip directory with a look-up 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. The interconnect deprioritizes direct requests and drops them if they have been queued for more than 100 cycles. Cache and directory controllers process, at most, one incoming message each cycle.

We compare PATCH to the original token coherence protocols. TOKENB is a broadcast-based token-counting protocol [Martin et al. 2003b]. TOKENM [Martin 2003] extends the original TOKENB to support destination-set prediction [Bilir et al. 1999; Martin et al. 2003] by: (i) adding a best-effort directory at the home to track likely sharers and owner of the block, (ii) using completion messages to inform the home to update its directory state, and (iii) sending transient requests to none, some, or all other processors based on a destination set predictor. TOKENM audits each transient request at the home to forward the request to any sharers and/or owner that must receive the request but that were not included in the predicted destination set. In contrast, PATCH forwards the request, as would a normal directory (irrespective of whatever direct requests were sent). In our experiments, TOKENB and TOKENM attempt two transient requests before issuing a persistent request, and they use persistent requests with distributed activation [Martin 2003; Marty et al. 2005], which allows for efficient transfer of tokens even when many processors are requesting the same block [Marty et al. 2005]. To provide a fair comparison, TOKENB and TOKENM also use PATCH-TIMEOUT's adaptive timeout mechanism (Section 5.3.3).

We use two scientific workloads (`barnes` and `ocean`) from the SPLASH2 suite [Woo et al. 1995] and three commercial workloads (`oltp`, `apache`, and `jbb`) from the Wisconsin Commercial Workload Suite [Alameldeen et al. 2003]. 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 [Alameldeen et al. 2003]. To evaluate scalability up to 512 cores, we use a simple microbenchmark wherein each core writes a random entry in a fixed-size table (16K Locations) 30% of the time and reads a random entry 70% of the time.

8.2 PATCH without Direct Requests

PATCH augments a directory protocol with the ability to support direct requests, so when PATCH's use of direct requests is disabled, it should perform similarly to the baseline DIRECTORY protocol. The first two bars of each group in Figure 9 experimentally validate

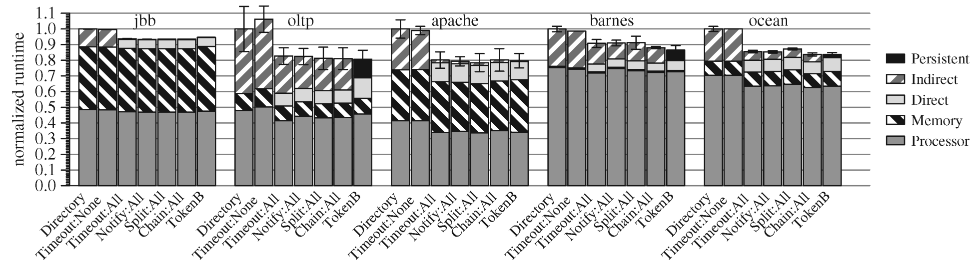


Fig. 9. PATCH runtime.

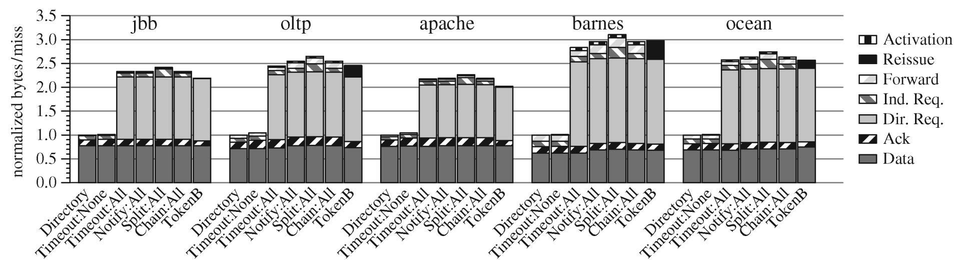


Fig. 10. PATCH traffic.

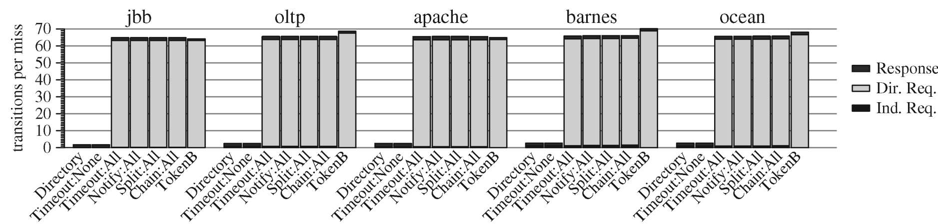


Fig. 11. PATCH cache-controller throughput.

this assumption by showing that PATCH-TIMEOUT configured not to send any direct requests (PATCH-TIMEOUT:NONE) and DIRECTORY perform similarly across a set of benchmarks. This result 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 10) shows that PATCH's data and request traffic are the same as DIRECTORY. PATCH's overall traffic is somewhat higher (only 2%, on average) because of PATCH's nonsilent writebacks of clean shared blocks and its few home-to-requester messages for activation on owner upgrade misses. Similarly, the normalized coherence controller traffic results (Figures 11 and 12) show that PATCH-TIMEOUT:NONE and DIRECTORY process similar a number of messages at the cache and directory controllers.

8.3 PATCH with Broadcast Direct Requests

The simplest use of direct requests is to emulate the low-latency sharing of a broadcast snooping system (but without the interconnect restrictions of snooping) by broadcasting

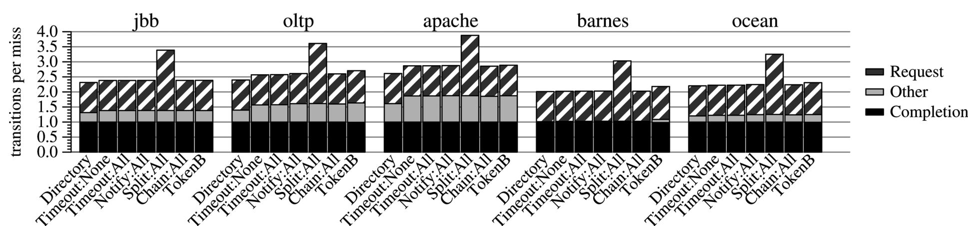


Fig. 12. PATCH directory-controller throughput.

direct requests to all processors for all misses [Martin et al. 2003b]. The third bar from the left in each group (labeled PATCH-TIMEOUT:ALL) of Figures 9 and 10 represents PATCH configured to broadcast direct requests. Comparing PATCH-TIMEOUT:ALL to PATCH-TIMEOUT:NONE (and DIRECTORY) highlights the benefit and cost of direct requests: broadcasting direct requests improve runtime (22% for oltp, 19% for apache, and by 14%, on average), at the cost of increasing interconnect traffic by 145%, on average, and an increase in the cache controller traffic proportional to the number of cores. With jbb, apache, and ocean, almost all sharing misses are resolved with direct requests (represented by the bar component labeled "Direct"). In some cases, direct requests decrease the cycles spent not waiting on second-level cache misses (signified by the reduction in the "Processor" component), which is largely due to an increase in spinning. For this bandwidth-rich baseline system configuration, PATCH's additional traffic due to broadcast results in little actual queuing in the interconnect. Thus, direct requests provide a significant performance improvement.

8.4 PATCH's Variants of Token Tenure

A system can implement token tenure in multiple ways. The bars in Figure 9 labeled PATCH-TIMEOUT:ALL, PATCH-NOTIFY:ALL, and PATCH-SPLIT:ALL correspond to three always-broadcast variants of PATCH representing different implementation tradeoffs described in Section 5.3.3. For these benchmarks, all four implementations of token tenure generally perform similarly. For barnes and ocean, PATCH-CHAIN's activation chaining mechanism reduces the performance penalty of racing requests, providing a small performance improvement. The interconnect traffic results in Figure 10 show that PATCH-NOTIFY's use of notifications increases traffic somewhat (as indicated by larger "Forward" component of each bar). The increase is noticeable for those benchmarks with the most races (i.e., oltp, barnes, and ocean). PATCH-SPLIT generates additional traffic because it issues twice the number of indirect requests (as indicated by the approximate doubling in size of the "Ind. Req" component of each bar). The per-miss directory controller traffic data (Figure 12) clearly illustrates the extra nonqueued requests sent by PATCH-SPLIT.

8.5 Comparison to TOKENB

When broadcasting direct requests for all misses, PATCH essentially behaves similarly to token coherence's TOKENB protocol (which also broadcasts all requests). The primary difference between TOKENB and PATCH:ALL is the mechanism used for forward progress (persistent requests versus token tenure, respectively). Whereas persistent requests can limit the scalability of token-counting protocols due to their reliance on broadcast and

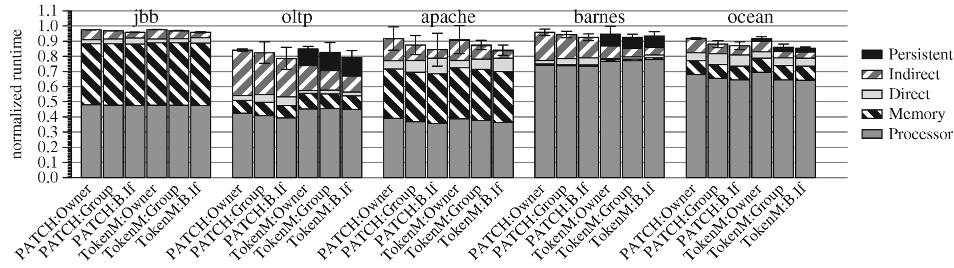


Fig. 13. Destination Set Prediction: runtime.

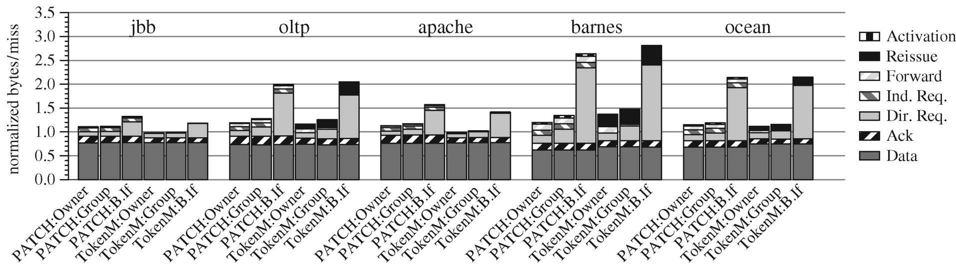


Fig. 14. Destination Set Prediction: traffic.

dedicated persistent request tables, token tenure avoids such scalability bottlenecks and facilitates PATCH's use of deprioritized best-effort requests. The rightmost bars in each group of Figure 9 show that PATCH:ALL and TOKENB generally perform similarly. TOKENB has a small performance advantage over PATCH-TIMEOUT:ALL and PATCH-NOTIFY:ALL in `barnes` and `ocean` due to frequent races. However, PATCH-CHAIN closes this gap to provide equivalent performance to TOKENB for all of our benchmarks. The overall traffic (Figure 10) of TOKENB and PATCH are similar because of two largely offsetting effects: TOKENB's reissued and persistent requests increase its traffic and PATCH's indirect requests, forwarded requests, and owner-upgrade activations increase its traffic.

8.6 PATCH with Predictive Direct Requests

To explore different latency/bandwidth trade-offs, Figures 13 and 14 show the effects of using three previously published destination-set predictors [Martin et al. 2003] with PATCH-TIMEOUT (results for PATCH-SPLIT and PATCH-NOTIFY are qualitatively similar) and TOKENM. The OWNER predictor sends a direct request to a single processor (i.e., the predicted owner) in addition to the indirect request to the directory. The GROUP predictor sends a direct request to a predicted subset (but not all) of the processors based on which processors recently cached the block. The BROADCASTIFSHARED predictor (labeled "B. If") sends direct requests to all cores if the block was recently shared. The predictors have 8,192 entries and use 1,024-byte macroblock indexing.

These results show that predictive direct requests allow PATCH to increase performance over PATCH without direct requests. For example, sending a direct request to just one processor (PATCH:OWNER) achieves about half of the runtime reduction as broadcasting direct requests to all processors (PATCH:ALL), while PATCH:OWNER's additional

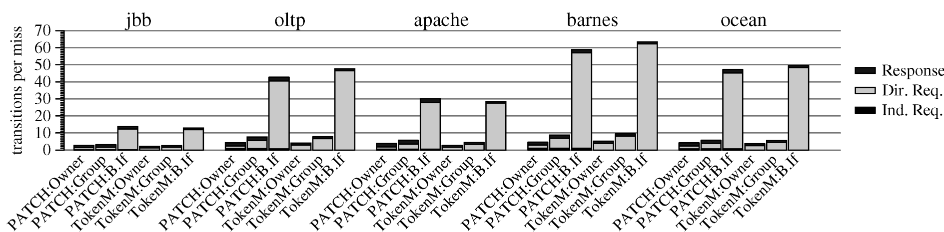


Fig. 15. Destination Set Prediction: cache-controller throughput.

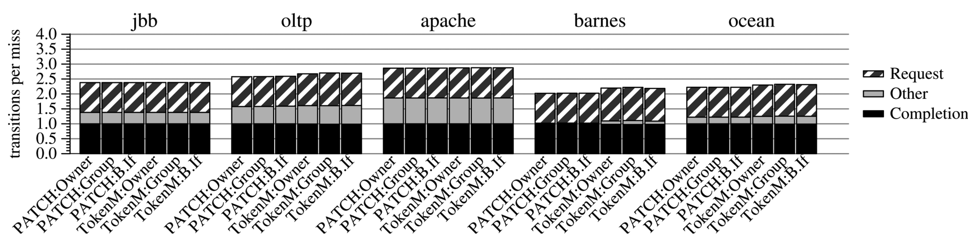


Fig. 16. Destination Set Prediction: directory-controller throughput.

traffic (just 20%, on average) is significantly lower than PATCH:ALL's 145% average increase in traffic. PATCH:GROUP uses 23% more traffic and is 10% faster, on average, than PATCH:NONE. PATCH:BROADCASTIFSHARED uses 22% less traffic, on average, than PATCH:ALL while achieving a runtime within 4% of PATCH:ALL. The cache controller traffic results (Figure 15) closely track the increase in direct request interconnect traffic.

These results mirror the results found in prior work on destination-set prediction [Martin et al. 2003] and that of TOKENM [Martin 2003]. As a point of comparison, Figures 13 and 14 also include results for TOKENM. The TOKENM and PATCH have similar runtime and traffic profiles for all the predictor types used, with any differences stemming from the same differences discussed earlier when comparing TOKENB and PATCH. Overall, these results indicate that PATCH's use of destination-set prediction can be valuable in cases of constrained bandwidth or where the extra power consumed by direct requests is a concern.

8.7 Bandwidth Adaptivity and Scalability

We next study the impact of PATCH's bandwidth adaptivity via best-effort requests. Figures 17 and 18 show the impact of limiting the available interconnect bandwidth by varying the link bandwidth for `ocean` and `jbb` (the other three benchmarks are qualitatively similar). These graphs show the runtime of PATCH-TIMEOUT configured to send direct requests to all processors using best-effort delivery (labeled PATCH:ALL), and a broadcast variant of PATCH-TIMEOUT that uses guaranteed delivery for all messages (labeled PATCH-NA:ALL). The runtimes are normalized to the runtime of DIRECTORY with the same link bandwidth, thus the DIRECTORY line is always at 1.0. When bandwidth is plentiful, both variants of PATCH identically outperform DIRECTORY. As bandwidth becomes scarce, however, the runtime of PATCH's nonadaptive variant quickly deteriorates as compared to DIRECTORY. In contrast, the runtime of the adaptive variant of PATCH always

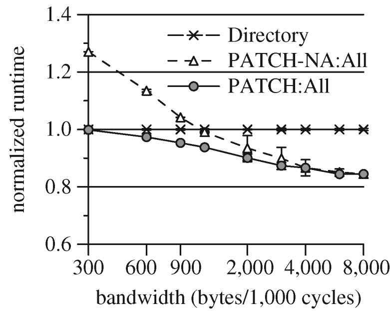


Fig. 17. Bandwidth adaptivity: `ocean`.

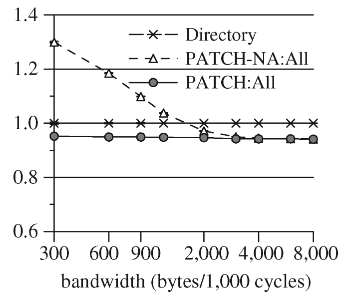


Fig. 18. Bandwidth adaptivity: `jbb`.

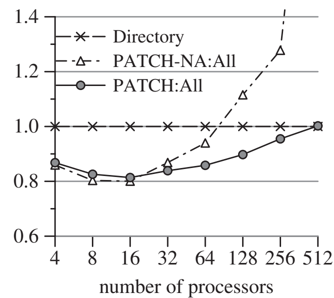


Fig. 19. Scalability: `microbench`.

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, the adaptive variant is faster than both other configurations (by as much as 6.3% for `ocean` and 5.2% for `jbb`).

We next show that PATCH's best-effort requests enable it to match the scalability of DIRECTORY. Figure 19 shows the microbenchmark's runtime of PATCH:ALL and PATCH-NA:ALL (as defined earlier) with a link bandwidth of two bytes per cycle on 4 cores to 512 cores. The nonadaptive protocol performs significantly better than DIRECTORY up to 64 cores but sharply worse from 128 cores onward. The adaptive protocol, by contrast, matches both the performance of PATCH-NA:ALL on low numbers of cores and the scalability of DIRECTORY on a large number of cores. PATCH outperforms directory up to 256

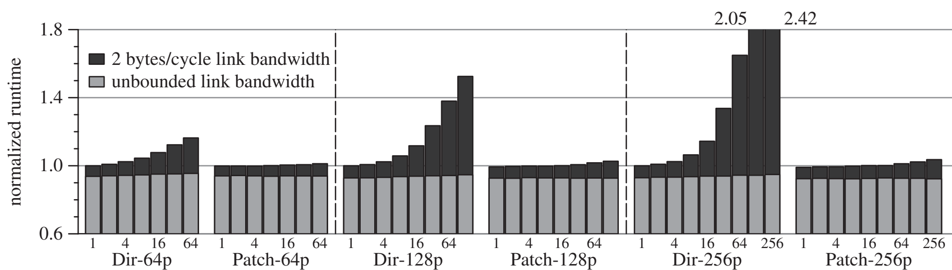


Fig. 20. PATCH-TIMEOUT:NONE vs DIRECTORY runtime for inexact directory encodings.

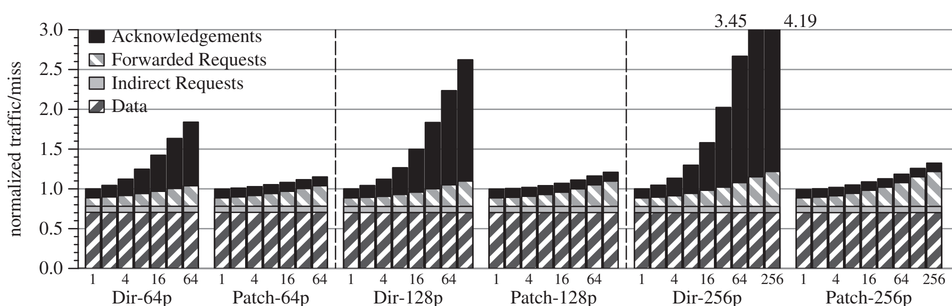


Fig. 21. PATCH-TIMEOUT:NONE vs. DIRECTORY traffic for inexact directory encodings.

cores, which shows that even for reasonably large systems, direct requests provide some benefit (without sacrificing scalability).

8.8 Scalability with Inexact Directory Encodings

To show that PATCH can exhibit better scalability properties than DIRECTORY when the directory encoding is inexact, Figure 20 compares PATCH:NONE and DIRECTORY on the microbenchmark using varying degrees of inexactness of directory encoding. In the inexact encoding used in this experiment, 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 20 show the runtime for 64, 128, and 256 cores for varying levels of coarseness normalized to a full-map bit vector. 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 shows substantial degradation (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 21 shows that the traffic of DIRECTORY is dominated by acknowledgement messages 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).

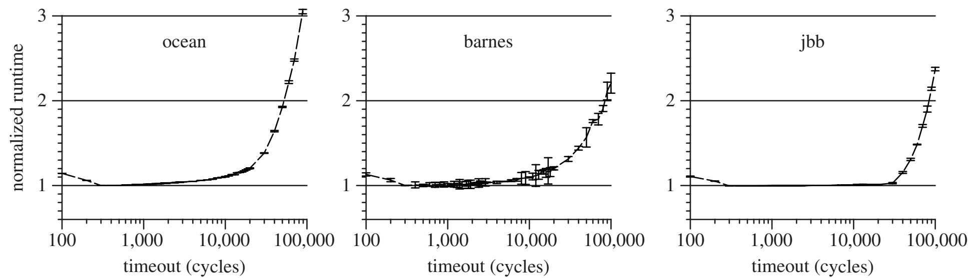


Fig. 22. Timeout sensitivity: runtime.

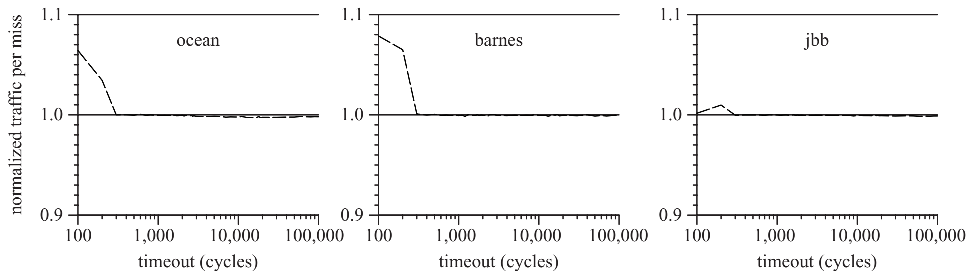


Fig. 23. Timeout sensitivity: traffic.

8.9 Timeout Sensitivity

One potential concern about timeout-based forward progress mechanisms is that their behavior may be too sensitive to the specific timeout value. In this experiment, we vary the timeout value of PATCH-TIMEOUT between 100 and 100,000 cycles for `jbb`, `barnes`, and `ocean` (the other two workloads are qualitatively similar). Figure 22 shows PATCH-TIMEOUT's runtime is relatively insensitive to timeout over a wide range of timeout values, showing little variation between timeout values of 300 and 4,000 cycles.

When the timeout is set too low (200 cycles or less), the performance degrades because requesters timeout prematurely and give up tokens too early. This unnecessary bouncing of tokens and/or data also results in additional traffic as show in Figure 23. This lower bound on the timeout is largely determined by the system configuration, that is, memory and cache look-up latencies and interconnect delay, so the performance of all three workloads start to degrade once the timeout value is less than the typical nonrace miss latency.

In contrast, the upper bound on the timeout value depends on the workload characteristics. As the value of the timeout increases, requesters hoard untenured tokens for longer, unduly delaying the active requester from completing. The `jbb` benchmark exhibits little contention and hence tolerates large timeout values of up to 30,000 cycles. The performance of `barnes` and `ocean`—which have more frequent races than `jbb`—degrade significantly when the timeout is 4,000 cycles or more. Traffic overhead does not increase for larger timeout values because delaying the discarding of tokens does not lead to additional messages.

The significant tolerance to timeout variation may reduce the cost of implementing per-request timers. For example, an array of 2-bit counters (one per outstanding request) that

are incremented whenever a 10-bit counter (per-core) overflows would suffice to provide a timeouts in the range of a few thousand cycles.

8.10 Sensitivity to Races

In our final experiment, we explore the performance robustness of PATCH in the situation when many processors are making simultaneous (i.e., racing) requests to the same block. To systematically study the impact of races, we use a microbenchmark with 64 processors each repeatedly loading or storing (70% load, 30% store) to a randomly chosen location. We vary the number of block-aligned locations between 2 and 16,384 to vary the prevalence of races (i.e., fewer locations result in more frequent races). As the total amount of data is less than the capacity of the primary caches, almost all of the cache misses are sharing misses.

Figure 24 shows the runtimes of DIRECTORY, three broadcast variants of PATCH (PATCH-TIMEOUT, PATCH-NOTIFY, and PATCH-CHAIN), and TOKENB all normalized to performance of DIRECTORY with the same number of locations, thus the DIRECTORY line is always at 1.0. When the number of locations is large (the rightmost part of the graph), all the broadcast-based protocols (PATCH variants and TOKENB) perform similarly; DIRECTORY has higher runtime due to the prevalence of sharing misses.

As the number of locations is reduced, the absolute performance of all protocols degrades as the protocols serialize racing requests, but PATCH and TOKENB degrade more quickly than DIRECTORY (as illustrated by the shrinking gap in normalized runtime when moving from the right to middle of Figure 24). PATCH-TIMEOUT degrades most rapidly (and becomes slower than DIRECTORY) due to its passive timeout-based recovery from races. The performance of PATCH-NOTIFY degrades less than PATCH-TIMEOUT, because PATCH-NOTIFY's active notification mechanism quickly corrects any errant transfers of tokens (PATCH-NOTIFY includes the direct forward optimization described in Section 5.4.2). In fact, when races dominate PATCH-NOTIFY's performance is only slightly faster than that of DIRECTORY. PATCH-CHAIN (which was described in Section 5.4.3) is more robust than PATCH-NOTIFY; instead of its runtime converging to close to that of DIRECTORY when races dominate, PATCH-CHAIN substantially outperforms DIRECTORY because of its quick hand-off of tokens to the next requester.

When races occur occasionally (the middle of the graph), TOKENB's reliance on timeout causes it to perform worse than PATCH-NOTIFY and PATCH-CHAIN (but still faster than PATCH-TIMEOUT). When races dominate, TOKENB's performance is more robust than any of the other protocols (including PATCH-CHAIN), primarily due to its broadcast-based distributed persistent requests [Marty et al. 2005; Martin 2003] in which processors directly broadcast persistent request deactivations and use the global consensus encoded in the persistent request tables to immediately forward tokens to the next queued persistent request. Although PATCH-CHAIN's runtime is somewhat higher than TOKENB when races are frequent, PATCH-CHAIN still substantially outperforms DIRECTORY and the other PATCH variants. We next explore the traffic implications of TOKENB's broadcast-based persistent requests as compared to PATCH-CHAIN's broadcast-free forward progress mechanisms.

Figure 25 plots the normalized interconnection network traffic per miss for the same configurations as Figure 24. This graph shows that the traffic per miss for DIRECTORY and the PATCH variants are largely independent of the frequency of races. PATCH-TIMEOUT

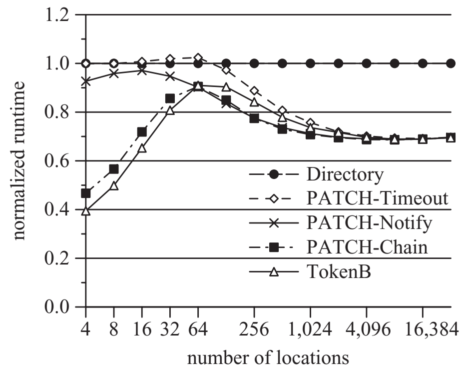


Fig. 24. PATCH sensitivity to races: runtime.

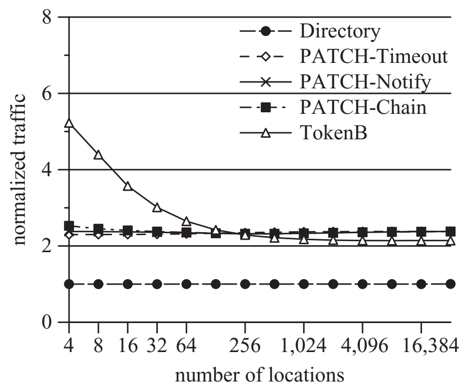


Fig. 25. PATCH sensitivity to races: traffic.

traffic increases only due to redirection of tokens. PATCH-NOTIFY introduces the notify messages, and PATCH-CHAIN further adds an extra message for chaining the requests. However, these nondata point-to-point messages contribute only a small percentage of overall traffic. In contrast, TOKENB's request retry mechanism and broadcast-based persistent requests result in a dramatic increase in traffic when races are frequent.

Altogether, the data in Figures 24 and 25 indicate that PATCH-TIMEOUT exhibits performance robustness similar to DIRECTORY, that PATCH-NOTIFY's measured performance always exceeded DIRECTORY, and that PATCH-CHAIN achieves most of TOKENB's performance robustness benefits without the traffic overhead associated with TOKENB's broadcast-based persistent requests.

9. CONCLUSION

This article introduced PATCH (Predictive Adaptive Token Coherence Hybrid), a protocol that obtains high performance without sacrificing scalability by augmenting a standard directory protocol with token counting and a broadcast-free forward progress mechanism called token tenure. The combination of token counting and token tenure allows PATCH to support direct requests over an unordered interconnect without relying on broadcast

messages for any part of correctness. PATCH employs best-effort direct requests, a form of bandwidth adaptation, wherein the system deprioritizes direct requests. When bandwidth is plentiful, PATCH matches the performance of broadcast-based protocols such as TOKENB. When bandwidth is scarce, PATCH retains the scalability of a directory protocol. In fact, microbenchmarks indicate PATCH can out-scale a standard directory protocol when both are using inexact directory encodings. The combination of token counting and best-effort direct requests allows PATCH to adapt smoothly between the extremes of broadcast and directory protocols for different system configurations. These properties result in a protocol that is both fast and scalable.

ACKNOWLEDGMENTS

The authors thank Mark Hill, Mike Marty, Dan Sorin, and David Wood for comments on this work and for their roles in the original work on token coherence. Special thanks to Mike Marty for his suggestion to extend deprioritization of direct requests to coherence controllers in addition to interconnect switches.

REFERENCES

ACACIO, M. E., GONZÁLEZ, J., GARCÍA, J. M., AND DUATO, J. 2002a. Owner prediction for accelerating cache-to-cache transfers in a cc-NUMA architecture. In *Proceedings of ACM/IEEE Conference on Super-Computing (SC'02)*. ACM, New York.

ACACIO, M. E., GONZÁLEZ, J., GARCÍA, J. M., AND DUATO, J. 2002b. The use of prediction for accelerating upgrade misses in cc-NUMA multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE, Los Alamitos, CA, 155–164.

AGARWAL, N., PEH, L.-S., AND JHA, N. K. 2009. In-network snoop ordering (INSO): Snoopy coherence on unordered interconnects. In *Proceedings of the 15th Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 67–78.

ALAMELDEEN, A. R., MARTIN, M. M. K., MAUER, C. J., MOORE, K. E., XU, M., SORIN, D. J., HILL, M. D., AND WOOD, D. A. 2003. Simulating a $2M Commercial Server on a $2K PC. *IEEE Comput. 36*, 2, 50–57.

BARROSO, L. A., GHARACHORLOO, K., MCNAMARA, R., NOWATZYK, A., QADEER, S., SANO, B., SMITH, S., STETS, R., AND VERGHESE, B. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. IEEE, Los Alamitos, CA, 282–293.

BILIR, E. E., DICKSON, R. M., HU, Y., PLAKAL, M., SORIN, D. J., HILL, M. D., AND WOOD, D. A. 1999. Multicast snooping: A new coherence method using a multicast address network. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*. IEEE, Los Alamitos, CA, 294–304.

CHARLESWORTH, A. 1998. Starfire: Extending the SMP envelope. *IEEE Micro 18*, 1, 39–49.

CHENG, L., CARTER, J. B., AND DAI, D. 2007. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 328–339.

CUESTA, B., ROBLES, A., AND DUATO, J. 2007. 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)*. IEEE, Los Alamitos, CA.

DALLY, W. J. AND TOWLES, B. 2004. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, San Francisco, CA.

FERNANDEZ-PASCUAL, R., GARCÍA, J. M., ACACIO, M. E., AND DUATO, J. 2007. A low overhead fault tolerant coherence protocol for CMP architectures. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA.

GALLES, M. 1997. Spider: A high-speed network interconnect. *IEEE Micro 17,* 1, 34–39.

GHARACHORLOO, K., SHARMA, M., STEELY, S., AND DOREN, S. V. 2000. Architecture and design of AlphaServer GS320. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 13–24.

GUPTA, A., WEBER, W.-D., AND MOWRY, T. 1990. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. IEEE, Los Alamitos, CA, 312–321.

HAGERSTEN, E. AND KOSTER, M. 1999. WildFire: A scalable path for SMPs. In *Proceedings of the 5th Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 172–181.

HUM, H. H. J. AND GOODMAN, J. R. 2005. Forward state for use in cache coherency in a multiprocessor system. http://www.patentstorm.us/patents/6922756/fulltext.html

JERGER, N. E., PEH, L.-S., AND LIPASTI, M. 2008a. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ACM, New York.

JERGER, N. E., PEH, L.-S., AND LIPASTI, M. H. 2008b. Circuit-switched coherence. In *Proceedings of the IEEE International Symposium on Networks-on-Chip*. IEEE, Los Alamitos, CA.

LAUDON, J. AND LENOSKI, D. 1997. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ACM, New York, 241–251.

MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HÅLLBERG, G., HÖGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *IEEE Comput. 35*, 2, 50–58.

MARTIN, M. M. K. 2003. Token coherence. Ph.D. thesis, University of Wisconsin.

MARTIN, M. M. K., HARPER, P. J., SORIN, D. J., HILL, M. D., AND WOOD, D. A. 2003. 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*. ACM, New York, 206–217.

MARTIN, M. M. K., HILL, M. D., AND WOOD, D. A. 2003a. Token coherence: A new framework for shared-memory multiprocessors. *IEEE Micro. 23*, 6, 108–116.

MARTIN, M. M. K., HILL, M. D., AND WOOD, D. A. 2003b. Token coherence: Decoupling performance and correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ACM, New York, 182–193.

MARTIN, M. M. K., SORIN, D. J., AILAMAKI, A., ALAMELDEEN, A. R., DICKSON, R. M., MAUER, C. J., MOORE, K. E., PLAKAL, M., HILL, M. D., AND WOOD, D. A. 2000. Timestamp snooping: An approach for extending SMPs. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 25–36.

MARTIN, M. M. K., SORIN, D. J., BECKMANN, B. M., MARTY, M. R., XU, M., ALAMELDEEN, A. R., MOORE, K. E., HILL, M. D., AND WOOD, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Comput. Archit. News*. 33, 4, 92–99.

MARTIN, M. M. K., SORIN, D. J., HILL, M. D., AND WOOD, D. A. 2002. Bandwidth adaptive snooping. In *Proceedings of the 8th Symposium on High-Performance Computer Architecture*. ACM, New York, 251–262.

MARTY, M. R., BINGHAM, J. D., HILL, M. D., HU, A. J., MARTIN, M. M. K., AND WOOD, D. A. 2005. Improving multiple-CMP systems using token coherence. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*. ACM, New York.

MARTY, M. R. AND HILL, M. D. 2006. Coherence ordering for ring-based chip multiprocessors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York.

MARTY, M. R. AND HILL, M. D. 2007. Virtual hierarchies to support server consolidation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, New York.

MEIXNER, A. AND SORIN, D. J. 2007. Error detection via online checking of cache coherence with token coherence signatures. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*. ACM, New York.

RAGHAVAN, A., BLUNDELL, C., AND MARTIN, M. M. K. 2008. Token tenure: PATCHing token counting using directory-based cache coherence. In *Proceedings of the 41st Annual International Symposium on Microarchitecture*. ACM, New York.

ROS, A., ACACIO, M. E., AND GARCÍA, J. M. 2008. DiCo-CMP: Efficient cache coherency in tiled CMP architectures. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE, Los Alamitos, CA, 1–11.

STRAUSS, K., CHEN, X., AND TORRELLAS, J. 2006. Flexible Snooping: Adaptive forwarding and filtering of snoops, in embedded-Ring Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. ACM, New York.

STRAUSS, K., CHEN, X., AND TORRELLAS, J. 2007. Uncorq: Unconstrained snoop request delivery in embedded-ring multiprocessors. In *Proceedings of the 40th Annual International Symposium on Microarchitecture*. ACM, New York.

SWEAZEY, P. AND SMITH, A. J. 1986. 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*. ACM, New York, 414–423.

TENDLER, J. M., DODSON, S., FIELDS, S., LE, H., AND SINHAROY, B. 2002. POWER4 system microarchitecture. *IBM J. Res. Dev. 46*, 1.

WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The SPLASH-2 Programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ACM, New York, 24–37.

Received August 2009; revised October 2009; accepted December 2009