Appears in the Proceedings of the 41st International Symposium on Microarchitecture (MICRO-31), November 2008

Token Tenure: PATCHing Token Counting Using Directory-Based Cache Coherence

Arun Raghavan Colin Blundell Milo M. K. Martin
Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA USA
{*arraghav, blundell, milom*}*@cis.upenn.edu*

Abstract

Traditional coherence protocols present a set of difficult tradeoffs: 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 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 *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.*

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 broadcast requests directly to all other processors using a totally ordered interconnect or a physical or logical ring [24, 27, 28]. 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 [5, 19] or bandwidth adaptivity [22] 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.*, [1, 2, 6, 15]), but these proposals target only specific sharing patterns or introduce significant implementation complexities for handling races.

This paper 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 contention. Furthermore, PATCH requires only true sharers to acknowledge requests. This property allows 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 [20] 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 non-scalable interconnect [18].

Although adding token counting to a directory protocol enables indirection-free sharing misses, PATCH also inherits the challenge of ensuring forward progress. 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 paper 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 within a certain amount of time. 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 time out and eventually flow to the winning processor, allowing it to complete its request.

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 switches 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 paper makes three main contributions. First, token tenure provides broadcast-free forward progress for token counting protocols by using local timeouts to avoid a need for 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, allowing 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 [20]. 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* that 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 1 [18]. Table 2 shows the correspondence between token counts and MOESI coherence states [29]. 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.

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).

Table 1. **Token Counting Rules**

State	M	O	E	F [13]	S	I
Tokens	All	Some	All	Some	Some	None
Owner?	Dirty	Dirty	Clean	Clean	No	No

Table 2. **Mapping of MOESI states to token counts.**

Races may still introduce protocol forward progress issues. Several mechanisms for guaranteeing forward progress in token counting protocols have been proposed [7, 20, 23, 24]. Token coherence uses *persistent requests* [20, 23] 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 [20] or distributed arbitration [23] 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 [7] are similar to persistent requests in that they are broadcast-based and use per-processor tables. Ring-Order [24] 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 multi-socket multi-core systems [23], virtual hierarchical coherence [25], fault-tolerant coherence [9, 26], and multicast interconnection networks [14].

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.

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 same flexibility to use destination-set prediction to send direct requests to zero, some, or all other processors in the system [18]. 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 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 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, token counting can also be used to avoid unnecessary acknowledgments for indirect requests [25]. 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.

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 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 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.

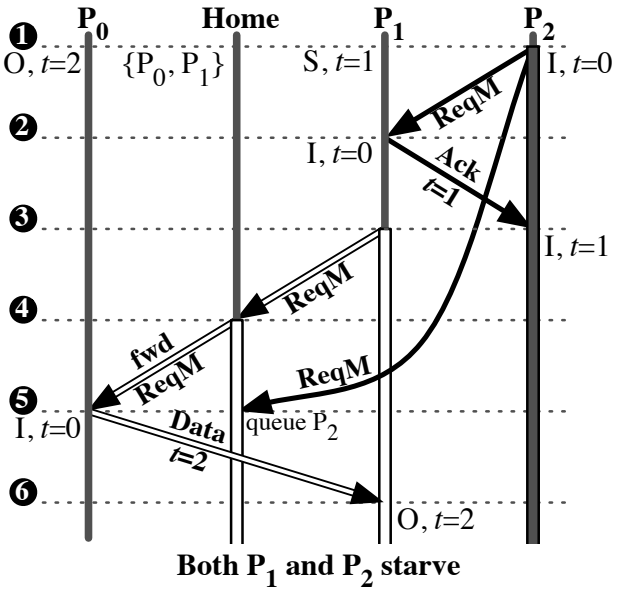

Figure 1. **Protocol race that can occur when direct requests and token counting are naively added to a directory protocol.** Initially, P_0 has the owner token and one other token (placing it in the O state) and P_1 has one token (placing it in the S state). At time ❶, P_2 (solid lines) initiates a "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.

Broadcast-free forward progress via token tenure. The above attributes provide a framework for a fast, scalable, and adaptive protocol. However, races may prevent forward progress. Figure 1 provides an example of the problems caused by racing requests. To ensure forward progress without impeding scalability, PATCH introduces token tenure, a

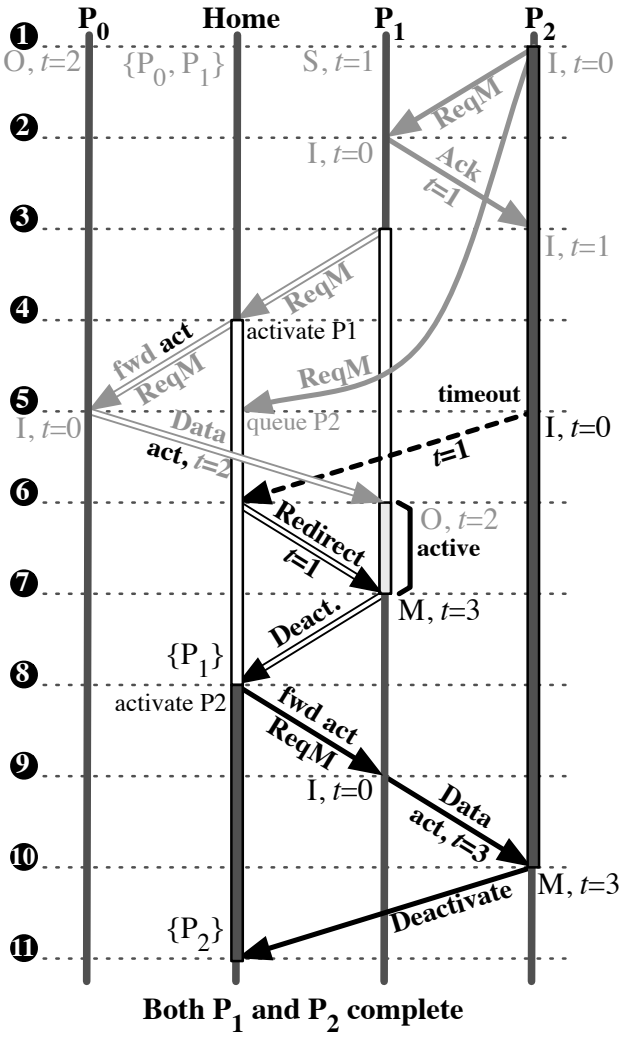

Figure 2. **PATCH's token tenure resolves the race from Figure 1.** The operation of PATCH in this scenario proceeds initially as in Figure 1. However, when the home activates the request of P_1 at time ❹, it augments the forward to P_0 with the "activated bit". At time ❺ P_2 times out because it has not been activated and sends its token to the home. At time ❻, P_1 becomes active when it receives P_0's acknowledgement, which includes the activated bit. Also at time ❻, the home receives P_2's token. The home redirects the token to P_1. At time ❼, 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 ❽ it activates P_2, sending a forward to P_1 that includes the activation bit destined for P_2. At time ❾ P_1 receives this message and sends data, tokens, and the activation bit to P_2, which receives them at time ❿. At this time, P_2 completes its miss and sends a deactivation message to the home.

forward progress mechanism for token counting protocols that neither relies on broadcast nor requires any specific interconnect properties. 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 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 2 illustrates token tenure resolving the race from Figure 1.

4. Token Tenure

The rules of token counting enforce coherence safety, but they say nothing about ensuring forward progress. This section introduces analogous rules for ensuring 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 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 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 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 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 3 contains the complete set of rules for token tenure.

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 3). 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.

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.

Tokens that are not at the home or tenured at the active requester may be either in-flight, untenured, or tenured at one or more non-active processors. 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*).

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.
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 Redirect Rule): The home node redirects 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.

Table 3. **Token Tenure Rules**

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 (rules #1*b* and #6*b*). The forwarded request will likely send the tokens to the active requester, but it could be a lingering (stale) forwarded request from a prior activation. However, the number of such lingering requests is bounded (rule #1*b*), 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. Once the active requester has been satisfied, it will

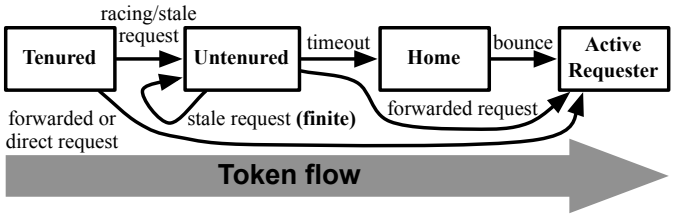

Figure 3. **Illustration of the flow of tokens to the active requester.**

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 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 4), token tenure does not require broadcast [7, 20, 23, 24], per-processor tables [7, 20, 23], or a ring-based interconnect [24].

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 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. The next section (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 [21]. 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 [16], it is reflective of more recent protocols such as Sun's WildFire [12]. In this approach, the arrival order at the home unambiguously determines the order in which racing requests are serviced. DIRECTORY does not depend upon 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

Mechanism	Broadcast-free?	Interconnect	Reissues?	State at processor	State at home
Persistent/priority requests [7, 20, 23]	no	any	yes	tokens & P.R. table	tokens
RingOrder [24]	no	ring	no	tokens	1 bit
Token tenure	yes	any	no	tokens	tokens & sharers set

Table 4. **Comparison of forward progress mechanisms proposed for token counting protocols.**

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 [29] and F [13] states plus a migratory sharing optimization. To reduce memory accesses, DIRECTORY uses the dirty-owner (O) state and a clean-owner state (F) [13]. 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 non-shared data, but it does not support silent eviction of blocks in the E state.

5.2. PATCH Implementation

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: (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 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 data-less message with a token count back to the home.

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.

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.

Token tenure mechanism. Token tenure requires three 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 upon 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 DIRECTORY. To support the third mechanism PATCH adds a timer per outstanding request at the processors.

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).

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 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 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. Second, because many racing direct requests to the same block may cause tokens to inefficiently flow to processors other than the next active requester, PATCH reuses the timer when a processor sends its deactivation message. During this timeout interval, processors continue to ignore direct requests (but not forwarded requests). This optimization mitigates the impact of racing requests by providing a window for the home to direct the movement of tokens to the next active requester without interference from direct requests.

6. Prediction and Bandwidth Adaptation

Inspired by a multicast variant of token coherence [18], 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 [5, 19] 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 [19].

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 [19, 22]. BASH used 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 [22]. 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 [22].

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 gives 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 [10]. 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 [8, 14]. 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 8.4). 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.*, [11, 16]). 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, *e.g.*, fan-out multicast (which reduces forwarded request traffic), acknowledgement combining (which reduces acknowledgment traffic), or cruise missile invalidations [4] (which reduces both).

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 above example).

The Virtual Hierarchies protocol exploited the same properties of token counting to avoid unnecessary acknowledgements in the context of a directory protocol [25]. 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 (1) achieve higher performance than DIRECTORY without sacrificing scalability, (2) adapt to varying amounts of available interconnect bandwidth, and (3) out-scale DIRECTORY for inexact directory encodings.

8.1. Methods

We use the Simics full-system multiprocessor simulator [17] and GEMS [21]. 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. The interconnect deprioritizes direct requests and drops them if they have been queued for more than 100 cycles. Cache controller throughput was set to be sufficiently large as not to be a performance bottleneck.

We use two scientific workloads from the SPLASH2 suite [30] (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 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. Comparison to DIRECTORY and TokenB

The first two bars of each group in Figure 4 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 5) 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 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 4 show that PATCH configured to send direct requests to all other cores on each miss (PATCH-ALL) generally performs the same as token coherence's broadcast-based TokenB [20]. Overall traffic (Figure 5) 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. Direct Requests and Destination-Set Prediction

Comparing PATCH-NONE and PATCH-ALL (in Figure 4 and Figure 5) 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 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 bars of these figures show the effects of using PATCH with two previously-published destination-set predictors [19]. 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. The predictors have 8192 entries and use 1024-byte macroblock indexing.

PATCH-OWNER achieves speedups over PATCH-NONE that are about half those of PATCH-ALL (7% on average), while

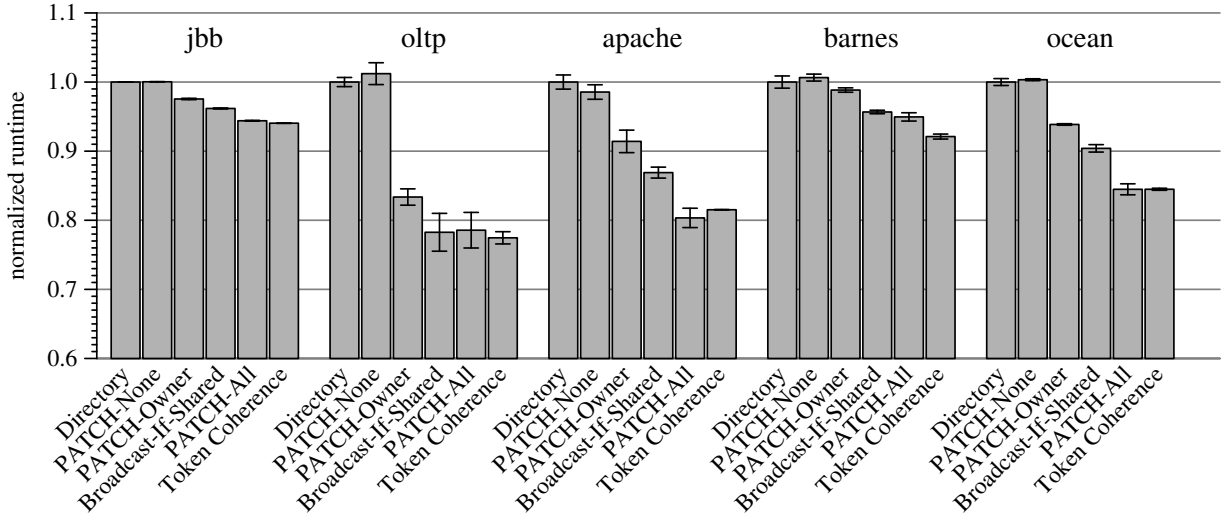

Figure 4. PATCH runtime

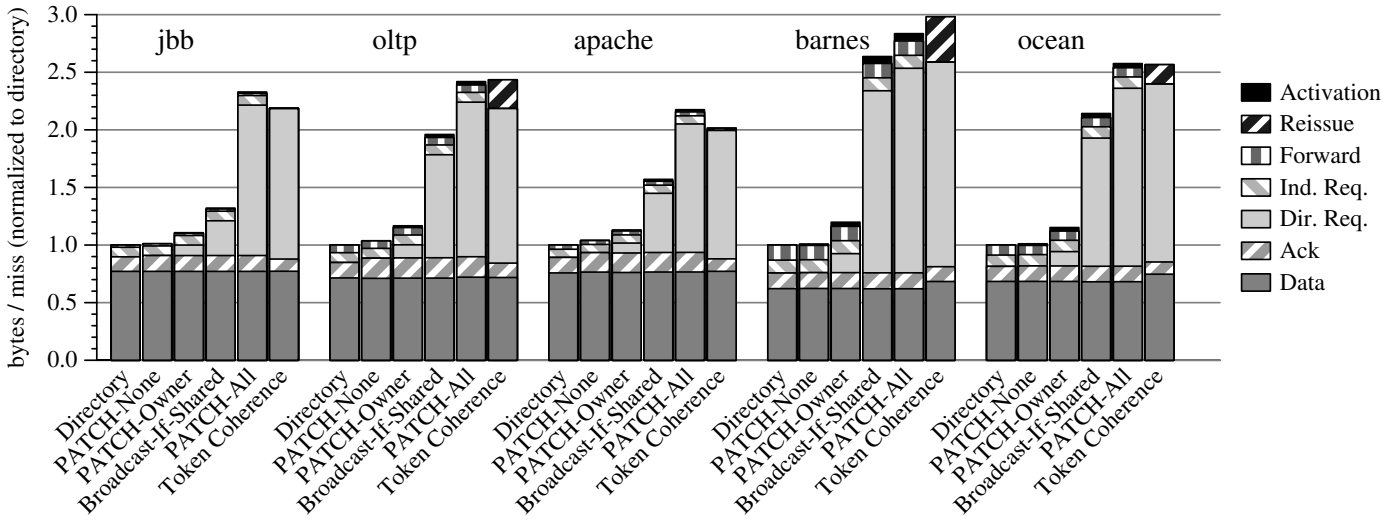

Figure 5. PATCH traffic

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 mirror the results found in prior work on destination-set prediction [19] and that of TokenM [18]. Our 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.

8.4. Bandwidth Adaptivity and Scalability

We next study the impact of bandwidth adaptivity via best-effort requests in PATCH. Figure 6 and Figure 7 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). The graphs show the runtime of PATCH-ALL and PATCH-ALL-NONADAPTIVE, a variant of PATCH-ALL that uses guaranteed delivery for all messages. 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, PATCH-ALL-NONADAPTIVE and PATCH-ALL identically outperform DIRECTORY. As bandwidth becomes scarce, however, PATCH-ALL-NONADAPTIVE's runtime quickly deteriorates compared to DIRECTORY. PATCH-ALL's runtime, in contrast, always stays at or better than DIRECTORY. Furthermore, in the middle of the graph, where there is

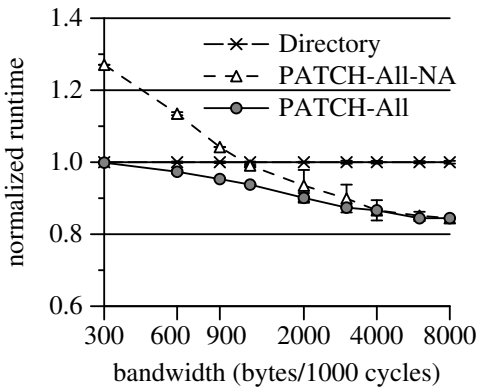

Figure 6. Bandwidth adaptivity: ocean

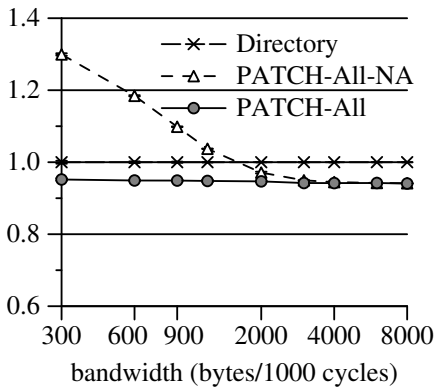

Figure 7. Bandwidth adaptivity: jbb

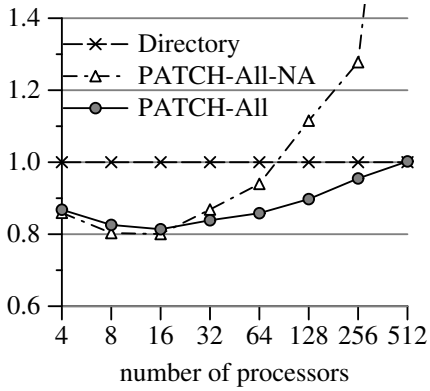

Figure 8. Scalability: microbench

enough bandwidth for some but not many direct requests, PATCH-ALL 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 8 shows the microbenchmark's runtime of PATCH-ALL and PATCH-ALL-NONADAPTIVE (as defined above) with a link bandwidth of two bytes per cycle on 4 cores to 512 cores. PATCH-ALL-NONADAPTIVE performs significantly better than DIRECTORY up to 64 cores but sharply worse from 128 cores onward. PATCH-ALL, by contrast, matches both the performance of PATCH-ALL-NONADAPTIVE 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).

8.5. Scalability with Inexact Directory Encodings

To show that PATCH can exhibit better scalability properties than DIRECTORY when the directory encoding is inexact, Figure 9 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 9 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 two 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 10 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).

9. Conclusion

This paper 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, PATCH out-scales 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.

Acknowledgements

The authors thank Mark Hill, Mike Marty and Dan Sorin for comments on this work. Special thanks to Mike Marty for his suggestion to extend deprioritization of direct requests to coherence controllers in addition to interconnect switches. This work is supported in part by NSF CAREER award CCF-0644197 and donations from Intel Corporation.

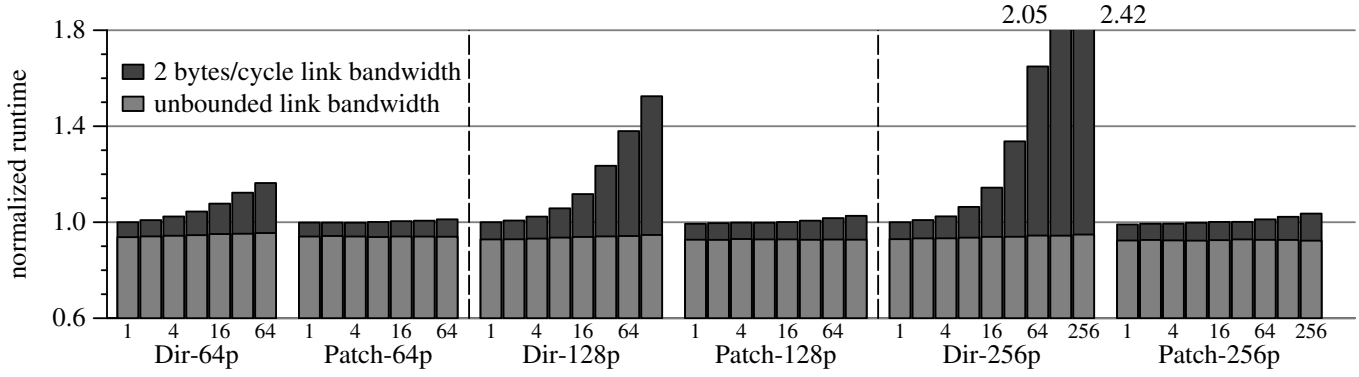

Figure 9. PATCH vs DIRECTORY runtime for inexact directory encodings

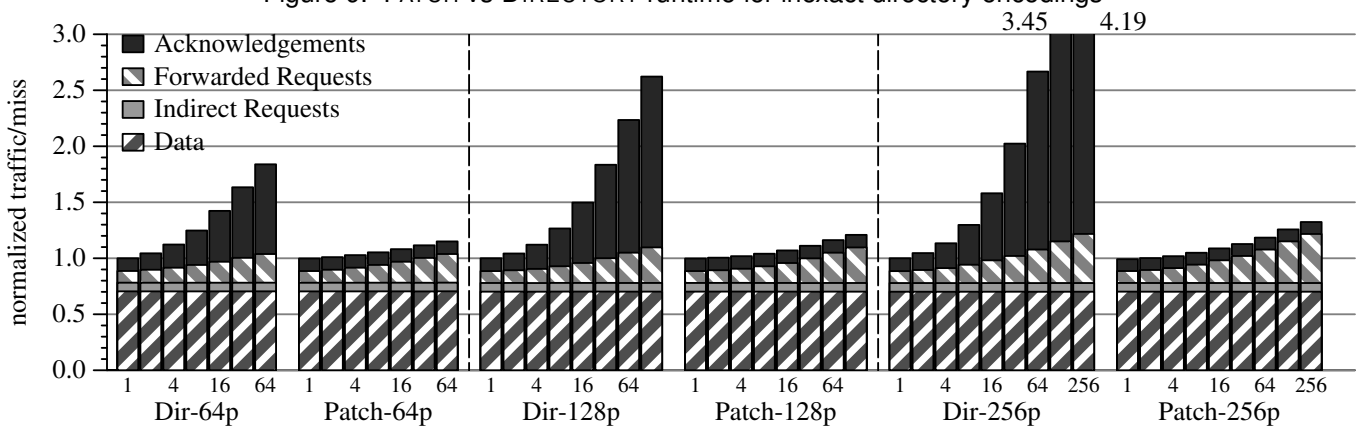

Figure 10. PATCH vs DIRECTORY traffic for inexact directory encodings

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*, Sep. 2002, pp. 155–164.

[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*, vol. 36, no. 2, pp. 50–57, Feb. 2003.

[4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Jun. 2000, pp. 282–293.

[5] 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*, May 1999, pp. 294–304.

[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*, Feb. 2007, pp. 328–339.

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

[9] 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.

[10] M. Galles, "Spider: A High-Speed Network Interconnect," *IEEE Micro*, vol. 17, no. 1, pp. 34–39, 1997.

[11] 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)*, vol. I, 1990, pp. 312–321.

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

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

[14] 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*, Jun. 2008.

[15] 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.

[16] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, Jun. 1997, pp. 241–251.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[18] M. M. K. Martin, "Token Coherence," Ph.D. dissertation, University of Wisconsin, 2003.

[19] 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*, Jun. 2003, pp. 206–217.

[20] 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*, Jun. 2003, pp. 182–193.

[21] 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.

[22] 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*, Feb. 2002, pp. 251–262.

[23] 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.

[24] 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.

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

[26] 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.

[27] K. Strauss, X. Chen, and J. Torrellas, "Flexible Snooping: Adaptive Forwarding and Filtering of Snoops, in Embedded-Ring Multiprocessors," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, Jun. 2006.

[28] K. Strauss, X. Chen, and J. Torrellas, "Uncorq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007.

[29] 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*, Jun. 1986, pp. 414–423.

[30] 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*, Jun. 1995, pp. 24–37.