COMP 60621

Three hours

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

Designing for Parallelism and Future Multi-core Computing

Date: Tuesday 23rd January 2018

Time: 14:00 - 17:00

Please answer the single Question provided

Two academic papers are attached for use with the examination.
Otherwise this is a CLOSED book examination.

The use of electronic calculators is NOT permitted.

[PTO]

1. Provide an analysis of **one** of the following two papers**:**

   a) TLB-Based Temporality-Aware Classification in CMPs with Multilevel TLBs. TPDS 2017

   or

   b) An Algorithmic Approach to Communication Reduction in Parallel Graph Algorithms. PACT 2015

by answering the following questions:-

a)	What is the problem being addressed?	(10 marks)
b)	What is the proposed solution?	(12 marks)
c)	What are the assumptions?	(6 marks)
d)	How is it evaluated?	(12 marks)
e)	What are the limitations?	(6 marks)
f)	Overall assessment of paper and possible improvements?	(4 marks)

(Total 50)

END OF EXAMINATION

TLB-Based Temporality-Aware Classification in CMPs with Multilevel TLBs

Albert Esteve, Alberto Ros, María E. Gómez, Antonio Robles, and José Duato, *Member, IEEE*

Abstract—Recent proposals are based on classifying memory accesses into private or shared in order to process private accesses more efficiently and reduce coherence overhead. The classification mechanisms previously proposed are either not able to adapt to the dynamic sharing behavior of the applications or require frequent broadcast messages. Additionally, most of these classification approaches assume single-level translation lookaside buffers (TLBs). However, deeper and more efficient TLB hierarchies, such as the ones implemented in current commodity processors, have not been appropriately explored. This paper analyzes accurate classification mechanisms in multilevel TLB hierarchies. In particular, we propose an efficient data classification strategy for systems with distributed shared last-level TLBs. Our approach classifies data accounting for temporal private accesses and constrains TLB-related traffic by issuing unicast messages on first-level TLB misses. When our classification is employed to deactivate coherence for private data in directory-based protocols, it improves the directory efficiency and, consequently, reduces coherence traffic to merely 53.0 percent, on average. Additionally, it avoids some of the overheads of previous classification approaches for purely private TLBs, improving average execution time by nearly 9 percent for large-scale systems.

Index Terms—Distributed shared TLB, data classification, TLB usage predictor, coherence deactivation

1 Introduction

Over the last years high performance processors have evolved by doubling the core count every 18 months. This, coupled with the fact that most chip multiprocessors (CMPs) provide programmers with a shared-memory model, turns coherence maintenance in multilevel cache hierarchies into an increasingly critical issue. To meet these new scalability challenges, many proposals focus on directory-based coherence solutions as they tend to use less network bandwidth than their snooping-based counterparts, thus representing the most scalable alternative. However, as core count grows, directory-based protocols demand larger amounts of directory storage and energy. Due to physical and technological constraints, the storage area dedicated to the directory in the die must be limited. This fact definitively causes a larger amount of directory-induced invalidations as a result of replacements in the directory, which may severely degrade performance [1].

An effective way to improve directory efficiency and reduce replacements is based on classifying data as private or shared in order to handle blocks more efficiently according to their sharing status. Considering such a classification, Cuesta et al. propose deactivating coherence maintenance for private [1], or more generally non-coherent [2], blocks. The deactivation reduces the coherence overhead, ultimately enabling smaller or more efficient directory designs as it eludes the tracking of these blocks. Alternatively, Demetriades and Cho [3] avoid the invalidation of private blocks stored in the cache when evicting directory entries by delegating block-discovering responsibilities to the last-level cache (LLC) organization, and therefore reducing the number of required directory entries while barely affecting performance.

The aim of a classification mechanism is to detect as much private data as possible without degrading the overall system performance, minimizing its overheads. However, data classification usually requires a large amount of extra storage in order to track the data sharing status. Some mechanisms cannot dynamically reclassify data from shared to private, thus missing the detection of temporarily-private data and limiting the potential performance benefits of data classification. Recently, Ros et al. [4] propose a mechanism that deals with these drawbacks by (i) storing the sharing information in the system translation lookaside buffers (TLBs) and (ii) discovering the current sharers when a TLB miss takes place by broadcasting requests to other cores' TLBs and reclassifying the page if necessary. The evaluation of this proposal only assumes *private*, *single-level* TLBs.

Nevertheless, current multicore systems use multilevel TLB structures which help to address the increasing application memory footprints and constrain the performance loss that comes with them. Furthermore, shared last-level TLBs have been already explored by Bhattacharjee et al. [5], and Lustig et al. [6], exhibiting a reduction in TLB misses for parallel workloads when compared to private last-level TLBs.

Our Proposal. In this paper we propose and evaluate techniques to perform a TLB-based data classification for

- *A. Esteve, M.E. Gómez, A. Robles, and J. Duato are with the Department of Computer Engineering, Universitat Politècnica de València, València 46022, Spain. E-mail: alesgar@gap.upv.es, {megomez, arobles, jduato}@disca.upv.es.*
- *A. Ros is with the Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Murcia 30100, Spain. E-mail: aros@ditec.um.es.*

Manuscript received 20 Nov. 2015; revised 14 Oct. 2016; accepted 19 Nov. 2016. Date of publication 25 Jan. 2017; date of current version 14 July 2017. Recommended for acceptance by A. Gordon-Ross.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2658576

multilevel TLBs that employ either a private or a shared last-level TLB, and compare them with previous single-level TLB classification approaches. The main contributions of this work are:

- We propose the first adaptive TLB-based classification mechanism for shared TLB structures.
- We show the importance of usage prediction techniques for multilevel TLBs, which make the classification insensitive to TLB size.
- We propose a usage predictor for shared TLB structures (SUP), which naturally avoids overheads over previous predictors for private TLB structures.
- We perform an extensive evaluation considering single and multilevel private TLBs, and multilevel TLBs with a shared last-level TLB.
- We show how classification schemes improve system performance in contemporary architectures when coherence maintenance is deactivated for private data.

Results. Through full-system simulations, we show how execution time improves by 6.8 percent over a baseline configuration with private L2 TLBs classification scheme, and up to 8.2 percent with a distributed shared L2 TLB classification scheme. Furthermore, through precise traffic analysis, we reveal how TLB-based classification mechanisms benefit from shared TLB structures, reducing TLB and cache traffic to only 53.0 percent. Finally, *SUP* avoids traffic overhead for a 32-core system, ultimately reducing the traffic issued by 30.9 percent, compared to the baseline.

The rest of the paper is structured as follows. Section 2 reviews current classification mechanisms and TLB organizations in the literature. Section 3 describes the temporal-aware classification mechanism for multilevel TLBs with private last-level TLBs. Section 4 describes the proposed classification mechanism for multilevel TLBs with shared last-level TLBs, and discusses some key aspects for its design. Section 5 introduces the simulation environment and methodology used to obtain the results presented in Section 6. Finally, Section 7 draws some conclusions.

2 Background and Motivation

Data classification mechanisms are gaining importance as they allow treating blocks or accesses more efficiently depending on their sharing status. Specifically, Kim et al. [7] avoid broadcast messages in *snoopy* protocols when accessing private blocks, thus leading to a reduction of network traffic. Li et al. [8] introduce a small buffer structure close to the TLB, namely partial sharing buffer (PSB). When a page becomes shared it will feasibly be found on the PSB upon a TLB miss, obtaining the page translation with both lower latency and fewer storage resources. Hardavellas et al. [9] and Kim et al. [10], [11] keep private blocks on the local bank in distributed shared caches in order to reduce access latency. Ros and Kaxiras [12] propose an efficient and simple cache-coherence protocol by implementing a write-back policy for private blocks and a write-through policy for shared blocks. End-to-End SC [13] allows instruction reordering and out-of-order commits of private accesses from the write-buffers, since they do not affect the consistency model enforced by the system. Finally, Cuesta et al. [1] propose avoiding directory storage of private blocks, therefore deactivating coherence maintenance for those blocks and leading to smaller and faster directories. The evaluation of this work is built on top of this optimization, which is explained in Section 2.3.

2.1 Classification Techniques

All proposals described above use classification approaches that take advantage of existing OS structures (i.e., TLBs and page table) in order to perform the page classification scheme. An OS-based classification mechanism annotates the first requesting core in the corresponding page table entry (keeper [1] or FAC—first accessing core—[8] field) after the first TLB miss and classifies it as private. On subsequent page table accesses to the same page, the keeper field is compared to the current requester. If they do not match, then the entry is reclassified as shared. To this end, an extra-bit field is added to the page table and replicated in the TLB in order to fast access the sharing status. When a reclassification occurs, the sharing information in the keeper's TLB must be updated accordingly to avoid inconsistencies.

Other classification approaches have also been proposed. Directory-based mechanisms [14], [15], [16], [17], [18] suffer the important drawback that most of the data-optimization techniques for classification schemes are not applicable due to a late (post cache miss) discovery of the classification. Compiler-assisted approaches [10], [11] deal with the difficulty of knowing at compile time (i) whether a variable is going to be accessed or not, and (ii) in which cores the data will be scheduled and rescheduled. Finally, approaches based on the properties of programming languages [19], [20], despite being very accurate, are not applicable to most existing codes. Conversely, run-time approaches perform accurate classification for any code, thus avoiding these difficulties.

2.2 TLB-Based Classification

The sharing status of a page may evolve through different application phases from private to shared and back to private. An OS-based mechanism performs a non-adaptive classification, i.e., when a page transitions from private to shared it remains in that state for the rest of the execution time (or until it is evicted from the main memory). In applications that run for a long time, most pages may be considered shared at some point, thus neglecting the advantages of the classification. Allowing reclassification to private may significantly increase the number of private data detected when compared to an OS-based mechanism, and also alleviate miss-classification due to thread migration.

Temporality-Aware Classification. The goal of a *TLB-based classification mechanism* [4], [21], [22] is to achieve an adaptive classification that accounts for temporarily-private pages and tolerates thread migration. The TLB-based classification mechanism is based on inquiring the other cores' TLBs in the system (through TLB-to-TLB requests) on every TLB miss. The other TLBs reply to the requester indicating whether or not they are caching the page translation. This way, the TLB suffering the cache miss knows whether the page is shared or not.

Fast TLB-Miss Resolution. In TLB-based classification mechanisms, when a TLB miss takes place, a TLB-to-TLB request is broadcasted, in parallel with a page table walk. The TLBs that are caching the page translation include that information into the responses to that request as well. When the first page translation is received, it is stored in the requester TLB and the page walk is canceled. This way, the TLB miss latency can be considerably reduced [23], [24]. If no page translation is received from remote TLBs, the miss is completed when the page table walk process ends. A token-counting approach may be employed in order to avoid the requirement for all TLBs to reply upon every TLB-to-TLB request, which increases traffic requirements [22].

TLB-Cache Inclusion Policy. TLB-based classification mechanisms rely on a strict inclusion policy between the TLB and the local caches to the processor (e.g., the L1 cache in this work). Hence, the absence of a valid page translation in the TLB ensures the absence of any valid block belonging to that page in the L1 cache. This is the reason why probing TLBs is a sufficient condition to guarantee that a page is private (i.e., avoids *false privates*). The TLB-cache inclusion policy does not necessarily hurt system performance, since evicted TLB entries have probably not been recently accessed (LRU-like policy is applied), and thus it is not common to find blocks for the evicted page in cache.

Usage Predictor. TLB-based classification is determined by the presence of page entries in TLBs, despite the fact that they may have ceased to be accessed, thus making classification accuracy sensitive to the TLB size. A usage prediction mechanism for TLBs is essential to decouple classification from TLB size [4]. This predictor resembles the one employed in the Cache Decay approach [25] and determines whether or not a page is going to be accessed in the near future. In particular, the predictor uses one saturated counter per TLB entry that it is periodically increased according to an internal timeout and reset on every memory access to the page. When a TLB entry is predicted not to be used (its counter is saturated) and it is probed with a TLB-to-TLB request, the entry is invalidated. When a TLB entry is invalidated, all the blocks pertaining to the related page in the local cache hierarchy are invalidated, and the core is disqualified as a potential sharer of the page. This way, the usage predictor effectively increases the amount of private data detected for large or multilevel TLBs.

Forced Sharing. When the usage predictor is very aggressive, shared pages can be eagerly invalidated from the TLBs. Premature invalidations induce more TLB misses, which in turn induce further invalidations. It can be seen as a positive feedback process. To prevent this scenario, a *forced-sharing* request can be issued when there is a TLB miss for an entry that is still present in the TLB but has been eagerly invalidated [21]. The forced-sharing request bypasses the usage predictor and avoids invalidating disused entries in remote TLBs.

2.3 Coherence Deactivation

On current CMPs, the directory cache suffers from scalability issues. The directory area grows quadratically with the number of cores. Additionally, due to its limited associativity, it cannot simultaneously track all blocks stored in the processors' local caches. This causes directory entry evictions, which usually entail the invalidation of cached blocks. When a core accesses a block which has been invalidated due to directory coverage constraints, it causes a type of miss known as *Coverage* miss [26]. These misses may cause severe performance degradation.

Coherence Deactivation [1] is a technique that avoids the tracking of private (or non-coherent [2]) blocks by the directory. Consequently, directories exploit their limited storage capacity more efficiently, as long as the classification mechanism that detects private or non-coherent data is accurate.

Coherence Recovery. Since coherence deactivation bypasses the coherence protocol for non-coherent accesses, a *recovery* operation is required when a page becomes coherent again, in order to avoid inconsistencies. To this end, the blocks of a non-coherent page that becomes coherent must be either evicted from the cache (flushing-based recovery) or updated in the directory cache (update-based recovery). Once the recovery mechanism has finished, the directory cache is in a coherent state according to the new page classification. Both recovery strategies show negligible performance impact, given that the recovery is triggered less than 5 times per 1,000 cache misses, on average [2].

2.4 Multilevel TLBs

The organization of the TLB is becoming more and more crucial for performance in current CMPs. TLB misses are in the critical path of memory operations, and they may result in a long latency penalty. On a TLB miss, the page table is accessed in order to obtain the page address translation. "Walking" the page table often requires several memory accesses (e.g., up to twenty-four memory accesses on x86-64 virtual address space [27], or fifteen memory accesses for the recent 32-bit ARMv7 virtual address space [28], both supporting virtualized environments).

Since TLBs are usually accessed in parallel with L1 caches, the L1 TLB is commonly split into data and instruction TLBs. Moreover, the concept of multilevel cache hierarchies has been extended to TLBs, allowing private unified L2 TLBs to be present on many current architectures (e.g., Intel Xeon [29] or ARMv7 [28]). However, purely private TLB organizations exhibit some deficits, as they do not completely fulfill inter-core TLB sharing opportunities [30], leading to some avoidable and predictable TLB misses. Particularly on parallel applications, the same page translation is likely replicated on multiple TLBs when implementing private TLB structures, which wastes both storage resources and energy. Also, even on sequential applications, the system may experience TLB thrashing on some cores while other cores' TLBs exhibit small page footprints, provided that each individual TLB allocates a fixed set of resources.

To overcome this, Bhattacharjee et al. [31] use inter-core cooperative prefetchers to improve the TLB hit ratio, emulating a distributed shared TLB. They also propose a centralized shared L2 TLB [5], improving translation latency in a 4-core CMP. Both approaches have recently been explored further [6]. Nevertheless, centralized shared L2 TLBs are not relied upon when scaling to a large number of cores, since centralized organizations increase end-to-end latencies as the core count grows, which, as previously noted, will be added to the critical path. Furthermore, centralized

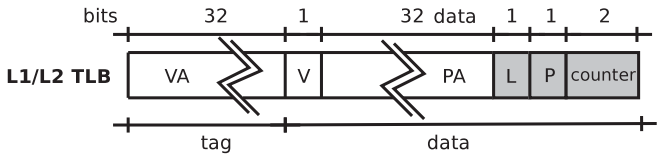


Fig. 1. L1/L2 TLB entry with the extra fields in gray.

shared TLBs may need a high-bandwidth interconnect to be able to connect to all the cores of a CMP.

TLB-based classification techniques have never been thoroughly evaluated for multilevel TLB hierarchies. Nonetheless, as previously noted, current architectures demand deeper and more efficient TLB structures. This work explores how TLB-based classification schemes perform comparatively under single and multilevel TLB hierarchies, including a comprehensive trade-off analysis. While assuming private L2 TLBs only requires straightforward extensions compared to single level TLBs, the use of a shared last-level TLB leads to a number of optimizations that decrease the overheads over previous classification mechanisms (see Section 5).

3 TLB-Based Classification in Systems with Private L2 TLBs

The TLB hierarchy of contemporary architectures include split data and instruction private L1 TLBs and unified private L2 TLBs. Among the most common architectures adopting this TLB hierarchy we find AMD's K7, K8, and K10, Intel's i7 and Xeon, ARMv7 and ARMv8, or the HAL SPARC64-III [28], [32], [33], [34], [35]. TLB-based classification schemes, however, have been mostly analyzed for single-level TLBs. This section explains the implications of implementing a TLB-based classification mechanism [21] for a private, two-level TLB organization.

3.1 Main Considerations

Differently from a single-level TLB classification scheme, in a multilevel TLB hierarchy, a L1 TLB miss does not trigger a TLB-to-TLB request. Instead, the private L2 TLB is consulted. On an L2 TLB hit, the page translation is obtained and cached in the L1 TLB, thus resolving the L1 miss. On a L2 TLB miss, the TLB-to-TLB request is initiated along with the access to the page table.

TLB-to-TLB requests look up both the L1 and the L2 TLB looking for in-use TLB entries. If there is a hit in any of the TLBs, the response is positive, and the page translation is sent to the requester. This lookup can be performed in parallel. The usage predictor is implemented in both TLB levels. The 2-bit saturated counter is reset on a hit to its corresponding entry.

The forced-sharing optimization (see Section 2.2) also needs to be adapted for a multilevel TLB environment. In this case, the forced-sharing request takes place when accessing a present but invalid entry either in the L1 or the L2 TLB. In this case we consider that the TLB entry has been prematurely invalidated, as it has not been evicted yet from the TLB structure.

Fig. 1 shows how TLB entries are extended for both TLB levels. The *Lock* (L) bit allows blocking memory accesses to the corresponding page while the sharing status is uncertain (e.g., while a TLB miss request is ongoing). The *Private* (P) bit tracks the page sharing status. The 2-bit saturated *counter* is used for the usage prediction mechanism, as explained in Section 2.2.

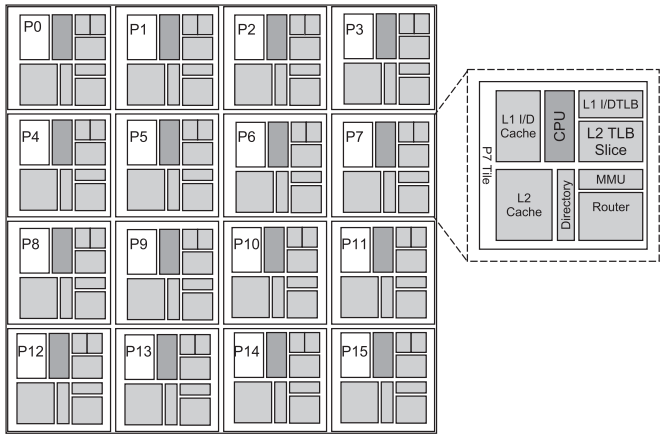


Fig. 2. Baseline architecture with a distributed shared L2 TLB organization.

3.2 Implementation Details

This section discusses the key implementation choices made for the proposed TLB-based classification scheme considering private two-level TLB hierarchies.

TLB Inclusion Policy. We employ an exclusive policy between the L1 and the L2 TLBs, since it maximizes the TLB capacity. Note that an increase in the number of TLB misses can dramatically degrade system performance. The downside of keeping exclusive TLBs is that, upon the reception of a TLB-to-TLB request, both TLB levels have to be accessed. We opt for performing this operation in parallel, thus incurring only the access latency of the L2 TLB, but at the cost of increasing the energy consumption in the case of a TLB hit. Since TLB-to-TLB requests are not frequent, we believe that this is the most appropriate design choice.

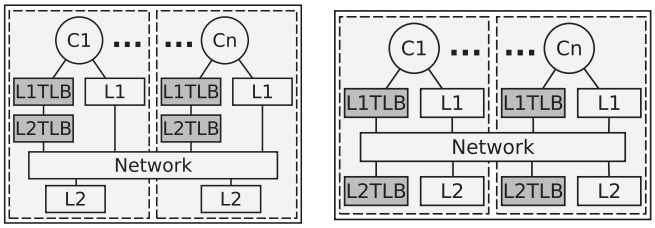
TLB Consistency. The TLB hierarchy is shootdown-aware. As both TLB levels implement an exclusive policy, they can be checked in parallel. Furthermore, this property avoids incurring wrong (outdated) page classification, by flushing both the TLB and the cache.

4 TLB-Based Classification in Systems with Distributed Shared L2 TLBs

Current application memory footprints demand deeper TLB hierarchies. Using a centralized shared last-level TLB, as an alternative to purely private TLB organization, provides high performance improvement mainly with parallel applications [5]. However, a centralized structure is not scalable.

In this work, we exploit the use of a distributed shared L2 TLB similar to the NUCA cache organization [36] (Fig. 2), with the aim of supporting data classification while avoiding some of the overheads associated to TLB-based mechanisms. Such a distributed organization has been previously suggested [8], however it has not been extensively explored. The baseline CMP considered in this work includes per-core L1/L2 TLBs, memory management unit (MMU), L1/L2 caches, and directory cache.

Fig. 3 shows how different TLB hierarchies are logically connected for both private (Fig. 3a) and distributed shared



(a) Private two level TLB structure.

(b) Two level TLB structure with a distributed shared second level TLB.

Fig. 3. System configurations in the evaluation.

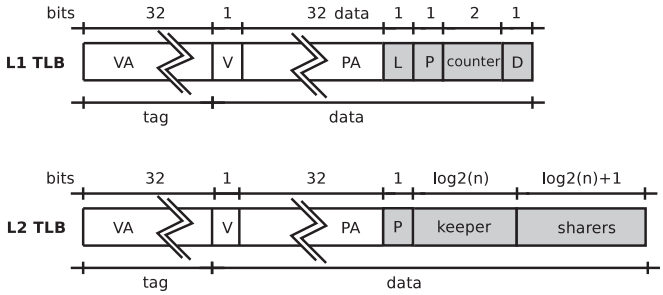


Fig. 5. L1 and L2 TLB entries, with the extra fields in gray for shared second level TLB structures.

(Fig. 3b) last-level (L2) TLBs. The interconnect network considered is a mesh topology. By employing a shared L2 TLB, the classification is naturally obtained through unicast messages to the corresponding L2 TLB bank, issued upon every L1 TLB miss. The sharing status is stored in the TLB structure and accessible for all requesting cores through the network.

4.1 Basic Protocol

The main idea behind this work is to leverage the shared L2 TLB in order to track the sharing information, thus naturally classifying memory accesses into private or shared at page granularity. To this end, a counter is associated with each second-level entry, recording up-to-date information about potential page sharers.

Fig. 4 outlines the actions required by the proposed classification scheme in order to resolve memory operations through the TLB hierarchy, including recovery and reclassification operations, explained in Sections 4.2 and 4.3, respectively.

Upon a memory access on a given core, prior to accessing the cache hierarchy, a TLB lookup is performed to look for the virtual-to-physical address translation. If it hits the L1 TLB, the sharing status information for that page is retrieved. Otherwise, on an L1 TLB miss, the L2 TLB is consequently accessed. Requesting the translation to the L2 TLB increases its sharers count. Therefore, whether there is a miss in the L2 TLB (and thus the page table in main memory has to be accessed), or there is a hit and no other L1 TLB currently holds the page (i.e., there is no other potential sharer), the page ends with a single sharer, and is thus marked as private. Otherwise, if the page had one or more

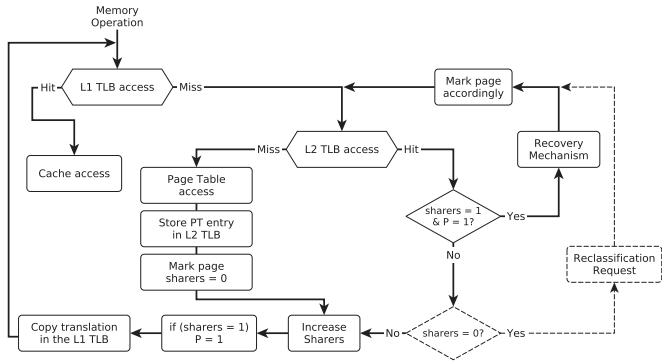


Fig. 4. Block diagram of the general working scheme under a memory operation for shared last-level TLBs. Dashed arrows and boxes operate only under the usage predictor.

sharers upon the reception of an L1 TLB miss request, it ends with more than one sharer and is marked as shared. In either case, the sharing status is specified alongside the response message containing the virtual-to-physical page translation to the upper TLB hierarchy level. The translation is finally stored in both TLB levels.

When the L2 TLB suffers an eviction, the sharing status is lost. In this case, in order to avoid classification inconsistencies, all L1 TLB entries holding the related page must be evicted (invalidating the corresponding L1 cache blocks). An inclusive policy between L1 and L2 TLBs is therefore recommended when assuming a shared L2 TLB. This policy brings also the advantage of exploiting inter-core sharing patterns in parallel applications, hitting on subsequent accesses from different cores to the L2 TLB.

As commented above, the L2 TLB keeps track of the sharing status for each page in the TLB hierarchy. Therefore, the L2 TLB entries require some extensions, as illustrated in Fig. 5. A *Private* (P) bit specifies whether the page is private (bit set) or shared (bit clear). A *sharers* field counts the number of current sharers of a page, allowing shared-to-private page reclassification. The *sharers* field is updated after every miss or eviction on L1 TLBs. This implies that L1 TLB evictions must be notified to the L2 TLB. This is essential to accurately unveil reclassification opportunities when the page ceases to have sharers. The page sharing status is updated in the L2 TLB according to the sharers count. Finally, a *keeper* field contains the identity of the holder of a private TLB entry. The *keeper* field helps to avoid broadcasts when updating the sharing state in the private TLBs when a transition to shared occurs. The *keeper* is updated every time the L2 TLB receives a request for a page and the current number of sharers is zero. In this case, the requester core's TLB becomes the new *keeper*.

The total storage resources required by this information is $1 + log_2(N)$ bits for the *sharers* field (counting from 0 to N sharers, both inclusive), $log_2(N)$ bits for the *keeper* field, and one bit for P, where N is the number of cores in the system. This adds up to $2 + log_2(N) * 2$ total bits. Since the TLB entry data field often contains some unused bits [37] that are reserved, hardware overhead may be avoided by taking advantage of them. Anyhow, the shared second level TLB entry format requires only 10 bits assuming a 16-core CMP, or 22 bits for a 1,024-core CMP. Therefore, for a TLB to support our proposed classification approach, the area overhead increases logarithmically with the system size, representing $\sim$14 or $\sim$23 percent of the L2 TLB area for a 16-core or a 1,024-core CMP respectively, according to CACTI [38].

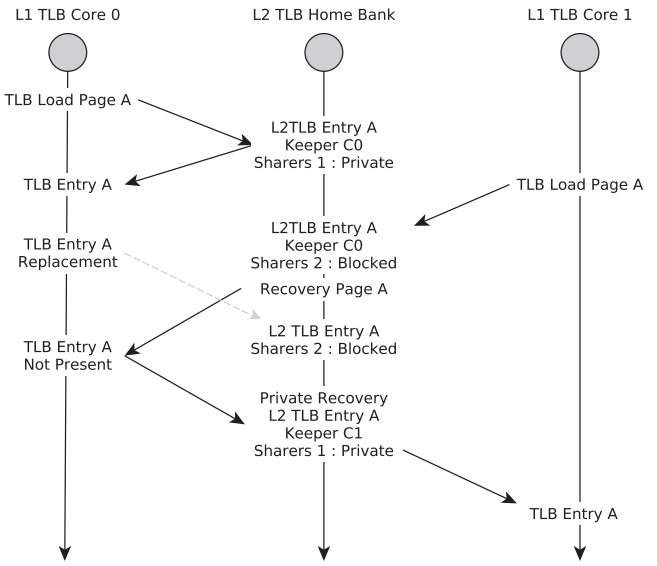


Fig. 6. Coherence recovery mechanism resolved to private. Page A in the keeper (*C0*) is evicted prior to receiving the recovery message and thus, the recovery is resolved to private and the keeper is updated.

4.2 Coherence Recovery Mechanism

This section reviews the coherence recovery mechanism for our proposed shared TLB classification scheme, which is required by the coherence deactivation technique that we employ as data optimization for our study case (see Section 2.3).

If a non-coherent page (i.e., private) transitions to a coherent state (i.e., shared), coherence status needs to be recovered in order to avoid the presence of untracked blocks (not cached in the directory). In our case, when an L1 TLB miss request reaches a private L2 TLB page entry, the sharing status may evolve to shared and thus, a coherence recovery is initiated (see Fig. 4). A special *recovery* request is issued to the current page *keeper*. If an L1 TLB receives a *recovery request*, all blocks pertaining to that page in the L1 cache are flushed to avoid inconsistencies. Then, after flushing all the page's blocks in the private cache, the sharing status in the L1 TLB is securely set to shared and a *recovery response* is sent to the corresponding L2 TLB bank. Upon the reception of the *recovery response*, page sharing status is updated in the L2 TLB, which becomes coherent (i.e., shared). Thus, directory cache can start tracking all blocks accessed for that page.

During a recovery process, a race may occur if the keeper evicts its TLB entry due to a conflict. Therefore, if the recovery request misses on the keeper's L1 TLB, a special *recovery* response is sent back with no further actions required (Fig. 6). Then, the requester becomes the new keeper and the page remains as private.

4.3 Shared TLB Usage Predictor

The described classification mechanism is so far dependent on the size or associativity of the L1 TLBs. In order to decouple TLB size from classification accuracy, a usage predictor is required. Unlike usage prediction for private TLB structures, which relies on broadcast TLB-to-TLB requests to discover the usage status of a TLB entry, shared TLB structures send a single request to the L2 TLB bank. As a consequence, the TLB usage predictor needs to be reworked and tuned for the new environment.

We propose a shared TLB usage predictor (SUP), employing a 2-bit saturated counter only for L1 TLB entries (see Fig. 5). SUP also adds a new *Disused* (*D*) bit, which is set when the corresponding counter saturates for the first time. Altogether, four additional fields are required per L1 TLB entry, for a total of 5 bits which are insensible to system size. Thus, the area overhead is only ~4 percent of the L1 TLB area.

Every time the *D* bit is set, a *disuse announcement* is sent to the L2 TLB, which decreases the *sharers* counter. Therefore, we operate under the assumption that the page is not going to be reaccessed soon from a core that has already fallen into disuse. If an access occurs, the counter is reset, but the *D* bit remains set. No more messages will be sent to the L2 TLB bank while it remains so, even if a disused TLB entry is evicted (i.e., disused translations evict silently), since the sharers count has already been appropriately decreased. Therefore, the *sharers* field tracks the number of pages currently in use, unveiling early reclassification opportunities. However, reclassification requires probing L1 TLBs as they might have been reaccessed from the time they fell into disuse.

Reclassification Process. In particular, whereas an L1 TLB miss hits in the L2 TLB bank and the *sharers* count is 0, a reclassification process is triggered (see Fig. 4). First, in case the *P* bit remains unset (i.e., the page has been shared), and a reclassification opportunity arises, it starts by sending a broadcast request, excluding the requester that initiated the reclassification. L1 TLBs reply according to the information of their predictor counters. On the one hand, if the counter is saturated (i.e., the page is currently not in use) the L1 TLB invalidates the page entry (flushing the blocks in the L1 cache) and responds with a NACK. On the other hand, a reaccessed L1 TLB unset its *D* bit and responds with an ACK. If not present, it still has to respond with a NACK. When all responses are collected, the *sharers* count is updated accordingly to the number of positive acknowledgments received. Finally, the miss that originated the reclassification is resolved (which increases the sharers count once more), setting *P* according to the resulting sharers count (i.e., private if there is 1 sharer and shared otherwise).

Conversely, when a reclassification process starts for a private page (the sharers count is 0 and the *P* bit is set), only the keeper needs to be probed with a unicast request. Therefore, a reclassification process might be used in order to keep the classification as private for longer. As in a broadcast reclassification, the *keeper* responds with a positive or negative acknowledgement depending on its usage prediction. Finally, the sharers count in the L2 TLB is either kept as 0 (and the page is brought as private for the requester, which becomes the new *keeper* and the only sharer that is accounted for), or restored to 1 (and the page subsequently transitions to shared).

In addition, the *forced-sharing* optimization can be also adapted in the context of a shared L2 TLB, although we expect that premature invalidations are not going to hurt system performance under this configuration. The reason is that *disused* entries are only invalidated when the L2 TLB considers that a reclassification probe should take place, naturally acting as a filter for premature invalidations. Nonetheless, L1 TLBs send a *forced-sharing* request to the L2

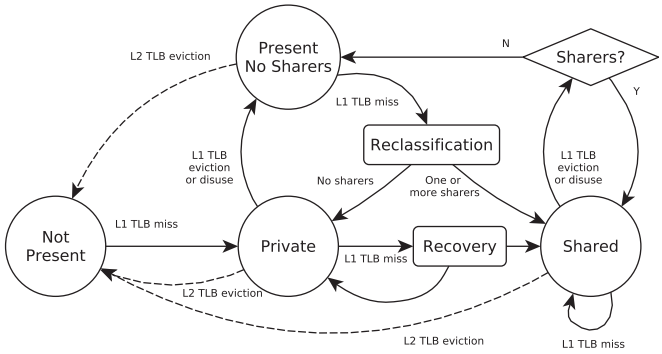


Fig. 7. L2 TLB classification state diagram with SUP.

TLB when the page is accessed and their corresponding entry is found present but invalid. If the miss triggers a reclassification, the probes issued to L1 TLBs just unset the D bits rather than invalidating the translations (independently from the status of the saturated counter). Thus, the page is kept as shared and can be securely reaccessed without incurring extra L1 TLB misses due to predictor-induced invalidations.

4.4 Classification Status

The sharing status of a page is managed by the L2 TLB through the P bit and the *sharers* count. This status is updated as a consequence of L1 TLB miss requests, evictions, or disuse announcements, and L2 TLB evictions. Fig. 7 depicts the state-transition diagram.

Pages in the L2 TLB can be in four states: (i) not present; (ii) present but not in use in any L1 TLB ($sharers = 0$); (iii) present and private with only one L1 TLB holding the entry ($P = 1$ and $sharers = 1$); and (iv) present and shared with one or several L1 TLBs currently holding the entry ($P = 0$ and $sharers > 0$).

If the page translation is *Not Present* in the L2 TLB and a miss request is received, the page transitions to *Private* after "walking" the page table. Then, if a different L1 TLB misses while the page is in *Private* state, a recovery mechanism is initiated and the page transitions to *Shared*. Note, thought, that classification might transition back to *Private* if a race condition occurs, as explained in Section 4.2. On the other hand, receiving an eviction or a disuse announcement for a *Private* page causes a transition to *Present No Sharers*. In this state, the L2 TLB acts as a victim TLB for the next missing L1 TLB. When the miss occurs, disused entries might have silently reaccessed the page translation and the classification coherence must be assured by means of a *Reclassification* request. If the reclassification ends up with no sharers, the miss is resolved to *Private*. Otherwise, the miss is resolved to *Shared* and the sharers count is updated. Finally, further L1 TLB miss requests for a *Shared* page leave the page in the same state, just increasing the sharers count. Receiving evictions or disuse announcements for a *Shared* page decreases the sharers count. Finally, the page transitions to *Present No Sharers* only if the count reaches 0.

5 Simulation Environment

We evaluate the classification schemes described in this work through full-system simulations using Virtutech Simics [39], along with the Wisconsin GEMS toolset [40], which enables detailed simulation of multiprocessor systems. The interconnection network has been modeled using the GARNET simulator [41]. We simulate a 16-tile CMP architecture that implements directory-based cache coherence and employs the parameters shown in Table 1. The TLB sizes are the same whether it is a purely private two-levels TLB hierarchy or it implements a shared second-level TLB. The L2 TLB miss latency considers four memory references to walk the page table, as in the 48-bit x86-64 virtual address space. Cache and TLB latencies have been calculated using the CACTI tool [38] assuming a 32 nm process technology. Predictor timeout values in the evaluation are based in the inter-access study in [21]. Every predictor value evaluated corresponds to the number of cycles required to increase the predictor counter field. Thus, four timeouts are required for the field to saturate and the translation entry to fall into disuse.

TABLE 1
System Parameters for the Baseline System

Memory Parameters	
Processor frequency	2.8 GHz
Cache hierarchy	Non-inclusive
Cache block size	64 bytes
Split instr & data L1 caches	64 KB, 4-way (256 sets)
L1 cache hit time	1 (tag) and 2 (tag+data) cycles
Shared unified L2 cache	1 MB/tile, 8-way (2,048 sets)
L2 cache hit time	2 (tag) and 6 (tag+data) cycles
Directory cache	256 sets, 4 ways (same as L1)
Directory cache hit time	1 cycle
Memory access time	160 cycles
Split instr & data L1 TLBs	32 sets, 16-way (512 entries)
L1 TLB hit time	1 cycle
Unified L2 TLB	128 sets, 16-way (2,048 entries)
L2 TLB hit time	3 cycles
Prediction timeouts	250 K, 50 K, 10 K, and 2 K cycles
Page size	4 KB (64 blocks)
Network Parameters	
Topology	2-dimensional mesh (4x4)
Routing technique	Deterministic X-Y
Flit size	16 bytes
Data and control message size	5 flits and 1 flit
Routing, switch, and link time	2, 2, and 2 cycles

The evaluation is performed with a wide variety of parallel workloads from several benchmarks suites, covering different sharing patterns and sharing degrees. *Barnes* (8,192 bodies, 4 time steps), *Cholesky* (tk15.O), *FFT* (64 K complex doubles), *Ocean* (258 $\times$ 258 ocean), *Radiosity* (room, -ae 5,000.0 -en 0.050 -bf 0.10), *Raytrace* (teapot), *Volrend* (head), and *Water-NSQ* (512 molecules, 4 time steps) are from the SPLASH-2 benchmark suite [42]. *Tomcatv* (256 points, 5 time steps) and *Unstructured* (Mesh.2K, 5 time steps) are two scientific benchmarks. *FaceRec* (script), *MPGdec* (525 tens 040.m2v), *MPGenc* (output of MPGdec), and *SpeechRec* (script) belong to the ALPBenchs suite [43]. *Blackscholes* (simmedium), *Swaptions* (simmedium), and *x264* (simsmall) come from PARSEC [44]. Finally, *Apache* (1,000 HTTP transactions), and *SPEC-JBB* (1,600 transactions) are two commercial workloads [45]. All reported experimental results correspond to the parallel phase of the benchmarks.

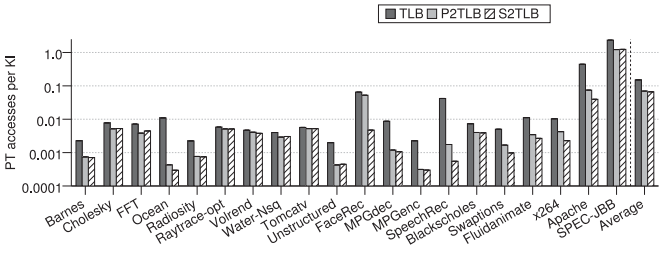


Fig. 8. TLB misses ending up as page table accesses per 1,000 instructions. No classification.

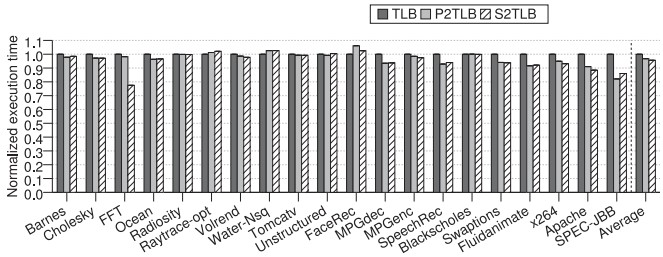


Fig. 9. Normalized execution time. No classification.

6 Evaluation Results

In this section, we first analyze how the aforementioned TLB hierarchies behave prior to applying a classification mechanism. Next, we evaluate how TLB-based classification schemes behave on multilevel TLB hierarchies, assuming both a private L2 TLB, which is a common design nowadays, and a distributed shared L2 TLB of the same size. Specifically, we evaluate our shared TLB usage predictor (SUP) and compare it to previous approaches in order to highlight how the classification is improved, and most of the flaws of the private predictor design are avoided. As a consequence, the scalability of the classification scheme is substantially improved.

6.1 TLB Architecture Analysis

This section shows how different TLB configurations behave prior to the application of any classification mechanism and the coherence deactivation technique that benefits from it. Specifically, we compare: (i) a system with a single-level TLB with TLB-to-TLB transfers used to accelerate TLB misses but without classification purposes (*TLB*); (ii) a system with per-core private L2 TLBs and, again, TLB-to-TLB transfers to accelerate L2 TLB misses (*P2TLB*); and (iii) a system with private L1 TLBs and distributed shared L2 TLBs (*S2TLB*).

Fig. 8 shows the number of accesses to the page table per kilo instructions, i.e., TLB misses that are resolved in the page table (TLB-MPKI). When the fast TLB-to-TLB transfer miss resolution mechanism is implemented, TLB misses are only resolved by accessing the page table if no other TLB is currently holding the translation. Notice that the y axis is plotted on a logarithmic scale in order to discern the different magnitudes of each application. As can be expected, including a second level TLB (*P2TLB* or *S2TLB*) reduces the average number of accesses to the page table, compared to a single-level TLB structure (*TLB*) with fast TLB miss resolution through broadcast TLB transfers. However, in some cases, *S2TLB* can be observed effectively exploiting the size of the L2 TLB over the private TLBs approach. For instance, SpeechRec or Apache, among others, reduce the number of accesses to the page table to a greater extent, showing how a shared TLB configuration helps preventing redundant page translation copies. Finally, the number of TLB-MPKI is remarkably low, less than 0.07 misses on average using a two-level TLB structure. Differently, using a single-level TLB retains TLB-MPKI slightly over 0.15 misses on average, despite implementing TLB-to-TLB transfers.

Accessing the page table in the main memory after a TLB miss implies performing an expensive page walk operation. Therefore, when the page table access is avoided, execution time is improved. Fig. 9 shows how *S2TLB* slightly improves execution time by only 1 percent on average compared to *P2TLB* with the same total size, and up to 4.4 percent compared to the *TLB* scheme. Despite the fact that the shared L2 TLB reduces the number of accesses to the page table, the improvement is comparatively low as a result of the small absolute amount of TLB misses reported in Fig. 8. In the case of SPEC-JBB or Apache, where the MPKI reported is greater, the improvement over the *TLB* scheme is more noticeable. Differently, in some benchmarks, as Barnes or SPEC-JBB, a private L2 TLB slightly outperforms a distributed shared L2 TLB scheme. The performance shrinkage occurs on account of the additional latency of accessing a shared L2 TLB, which needs to traverse the network in order to reach the home TLB slice. Therefore, on benchmarks with high L2 TLB hit ratio or accessing a great amount of private data, a greater access latency may hurt system performance, overmatching the potential benefits of inter-core sharing patterns exploitation. On the contrary, as FFT has more accesses to shared pages, *S2TLB* execution time improved.

Finally, TLB-to-TLB transfers significantly increase TLB traffic, since every TLB miss induces a broadcast message and many responses, potentially including many replicated translations. Fig. 10 shows all network flits (the flow control unit in which network packets are divided—see Table 1) transmitted across the network, normalized to *TLB*. Essentially, *P2TLB* reduces the TLB traffic by 48.0 percent compared to *TLB*, as it first relies in the L2 TLB to resolve L1 TLB misses. Therefore, the broadcast TLB miss resolution mechanism is only invoked after an L2 TLB miss. In contrast, L1 TLB misses with a shared L2 TLB are resolved through unicast messages. As a consequence, TLB network traffic with *S2TLB* is reduced to barely 6.5 percent compared to *TLB*.

6.2 TLB-Based Classification Mechanisms

So far, different TLB hierarchies have been analyzed, without any classification scheme whatsoever. This section analyzes how temporal-aware TLB-based classification techniques work assuming systems with two-level TLBs,

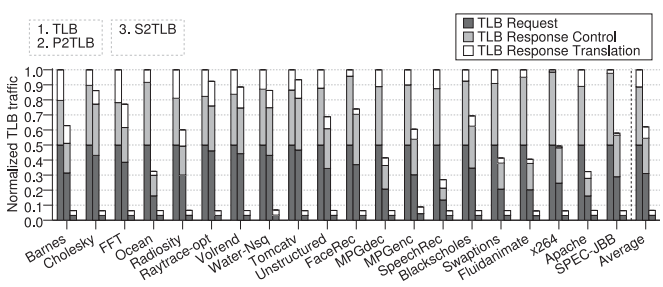


Fig. 10. Normalized traffic attributable to the TLB. No classification.

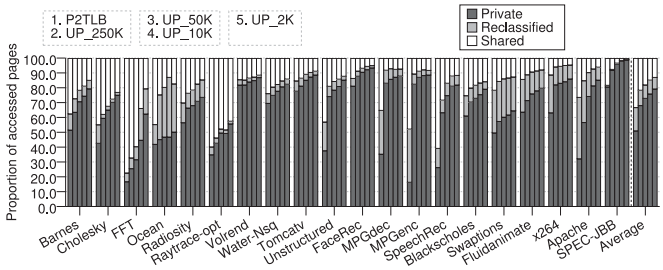


Fig. 11. Private/shared page classification with private TLBs.

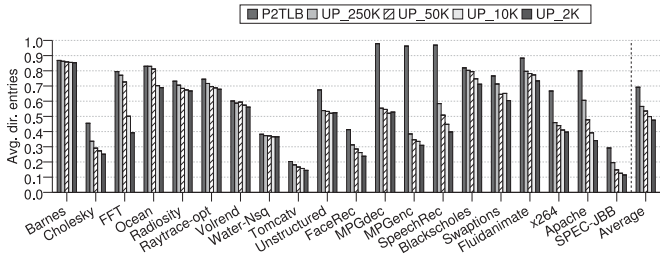


Fig. 13. Average amount of directory entries under coherence deactivation. Classification with private TLBs.

and evaluating their potential when applied to coherence deactivation for private data. First, using a private L2 TLB; then the classification scheme proposed in this paper for distributed shared L2 TLBs.

6.2.1 Private Second Level TLB

This section analyzes how a system with a purely private two-level TLB hierarchy behaves alongside the TLB-based classification scheme, as explained in Section 3. The quality of the classification scheme is tested by applying it to deactivate coherence maintenance.

Classification Accuracy. A good, first, general metric to determine the effectiveness of TLB-based classification is the amount of private data detected, provided they do not allow *false private* classification (see Section 2.2). Fig. 11 shows the amount of pages classified as private or shared, both with and without a TLB usage predictor (*UP*), including several predictor timeout values ranging from 250,000 to 2,000 cycles. The characterization is extended to discern the amount of shared pages that are reclassified to private at least once (*Reclassified*). By differentiating reclassified data we offer more insight into the potential benefits of a temporal-aware classification. Note that for a page to be considered private in the figure, it must remain so for the entire execution time.

Particularly, *P2TLB* classifies slightly above 50 percent of all accessed pages as *Private*, and 15.7 percent of shared pages are *Reclassified*. However, when a usage predictor is employed, classification accuracy depends on the page access patterns, decoupling it from the size and associativity of the TLB. As a consequence, the amount of *Private* pages is increased with *UP_2K* up to 76.6 percent on average. Private data detection with UP is improved even over *Private* and *Reclassified* pages combined in *P2TLB*. Moreover, *UP_2K* still reclassifies nearly 10 percent of pages.

Coherence Deactivation. Fig. 12 shows the total normalized amount of network flits transmitted through the interconnection network, classified into cache- and TLB-related traffic. Applying the classification to deactivate the coherence maintenance reduces cache traffic directly proportional to the amount of private data detected. The baseline system has the same overall configuration but without employing a classification scheme. Specifically, *P2TLB* reduces cache traffic to just 68.46 percent, on average, when compared to *Base*. Moreover, applying TLB usage predictor further follows this progression, requiring on average just 58.0 percent of *Base* cache traffic when employing the lowest predictor value. On the contrary, since TLB miss rate is increased due to some additional predictor-induced TLB invalidations, TLB traffic is increased as the predictor value decreases. Particularly, TLB traffic represents 10.4 percent of the total network traffic for *UP_2K*. Furthermore, TLB traffic overhead for TLB-based classification will not presumably scale horizontally with the system as broadcast cost greatly increases with the number of cores.

As the classification becomes more accurate, the pressure in the cache directory is alleviated, as blocks that pertain to private pages are not stored for non-coherent data (i.e., private) under coherence deactivation. Fig. 13 shows how, as expected, *P2TLB* reduces the average number of directory entries required per cycle by 30.8 percent compared to the baseline system. Moreover, when employing a usage predictor for TLBs, directory usage is further reduced, by up to 52.5 percent on average for the lowest predictor timeout.

Coherence deactivation also entails an improvement in execution time, since reducing the directory storage requirements contributes to a significant reduction of coverage misses (induced by directory evictions) in the cache structure. Additionally, *P2TLB* effectively reduces L2 TLB miss penalty by means of TLB-to-TLB transfers. Fig. 14 shows how *P2TLB* improves execution time by 6.8 percent compared to the baseline. TLB usage predictor does not contribute into a system performance reduction (even damaging it to some extent). This is due to the fact that directory ceases being a performance bottleneck (i.e., almost all coverage misses are prevented) with plain *P2TLB*, whereas

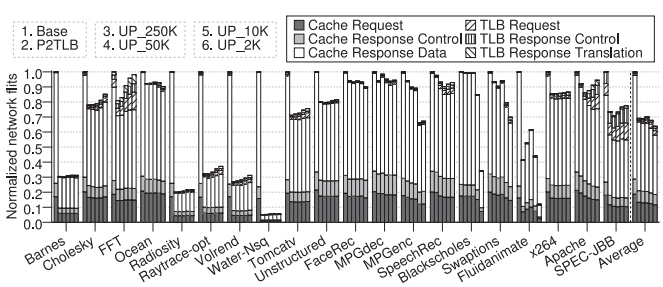


Fig. 12. Normalized network traffic under coherence deactivation. Classification with private TLBs.

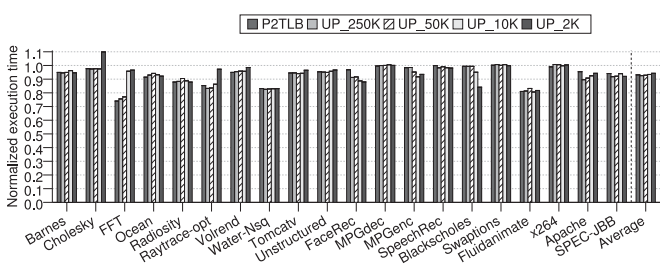


Fig. 14. Normalized execution time under coherence deactivation. Classification with private TLBs.

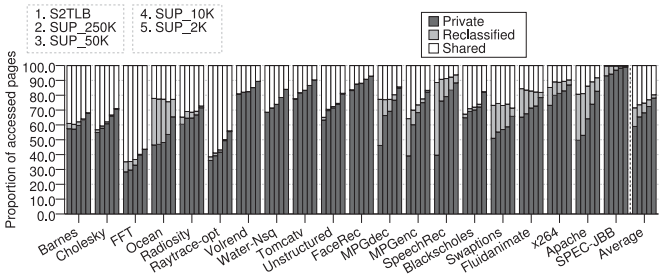


Fig. 15. Private/shared page classification with distributed shared last-level TLB.

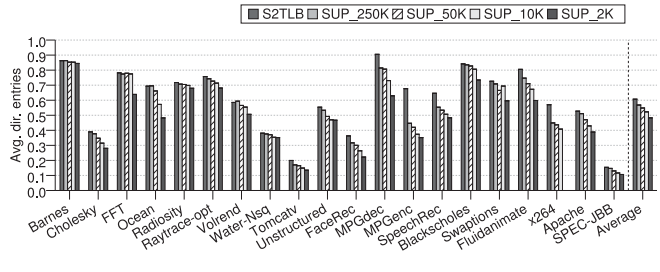


Fig. 17. Average directory usage under coherence deactivation. Classification with distributed shared L2 TLB.

prediction implies some extra overheads. Nonetheless, this factor is mostly on account of the optimization chosen to test the different classification approaches (i.e., coherence deactivation). Of course, alternative (or additional) data optimization could be applied, ultimately offsetting prediction overheads with the potential benefits of a more accurate classification.

Conclusion. To sum up, employing the classification mechanism for private two-level TLBs reveals how a TLB usage predictor is required to perform an accurate private classification. The TLB usage predictor timeouts considered in the analysis increase classification accuracy as its value decreases, and consequently, directory usage and cache traffic are improved when the classification is applied to coherence deactivation. However, TLB-based classification comes with some overheads, as it increases TLB traffic due to the TLB-to-TLB miss resolution mechanism, and incurs some performance degradation. Despite the fact that this does not completely discourage the use of a low predictor timeout over its potential benefits, it evidences the flaws of a TLB-based classification scheme for private TLB structures. Finally, TLB-to-TLB transfers are not supposed to scale with the number of cores in the system.

6.2.2 Distributed Shared Second Level TLB

This section evaluates a TLB-based classification for distributed shared L2 TLBs using the shared TLB usage predictor presented in Section 4.3.

Classification Accuracy. Fig. 15 depicts how pages are classified into private and shared for different classification mechanisms in a scenario with a distributed shared L2 TLB. The evaluation is focused in our classification mechanism detailed in Section 4 (*S2TLB*), and different predictor timeouts for a shared TLB usage predictor (*SUP*). On the one hand, *S2TLB* classifies 59.4 percent of pages as *Private*, and 14.6 percent as *Reclassified*, showing how it performs a precise and adaptive classification at page level. On the other hand, *SUP* increases the proportion of *Private* pages up to a 78.1 percent, but reduces *Reclassified* pages to merely a 2 percent of all accessed pages for a 2,000 cycles timeout. However, *SUP* accuracy for private data detection beats even *UP* for private TLB structures, as can be seen comparing with Fig. 11. Note that *SUP* relies on the shared L2 TLB as a filter for short-term reclassifications. Our proposal for shared TLB structures initiates a reclassification process only when the home L2 TLB tile is accessed and found in a *Present No Sharers* state (see Section 4.4). Even so, a reclassification process may still fail to transition to private again if a core reaccesses a disused page. Conversely, a TLB predictor for purely private TLB structures is more dynamic and forceful, which favors short-term reclassifications, albeit possibly hurting system performance.

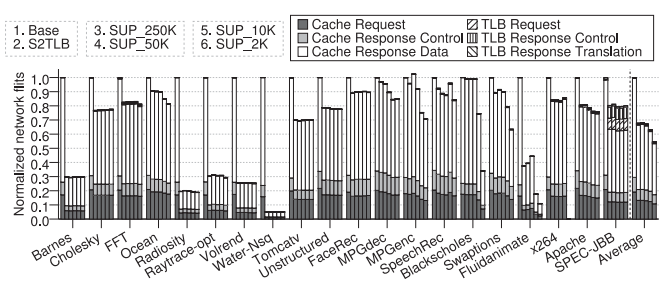


Fig. 16. Total flits injected under coherence deactivation. Classification with distributed shared L2 TLB.

Coherence Deactivation. Fig. 16 shows the total normalized network usage and its classification into cache or TLB messages when the classification is applied to coherence deactivation. *Base* is our baseline system with the same overall configuration, including a distributed shared second level TLB, but without data classification nor coherence deactivation. Classification mechanisms for shared TLB structures reduce the network traffic since they avoid the costly TLB-to-TLB broadcast transfers. *S2TLB* TLB traffic represents only 1.8 percent of the total. Furthermore, *SUP* prevents the TLB traffic increase attributed to lower predictor timeouts, keeping it to as much as 2.3 percent of the total traffic for the lowest considered timeout, while the cache traffic is halved. All in all, *SUP* improves classification accuracy while significantly reducing traffic overhead by avoiding unnecessary TLB invalidations. Therefore, fewer TLB misses are induced by the predictor, which represents a far more scalable approach in terms of traffic.

Fig. 17 shows the average number of required directory entries, in this case normalized to the baseline system with a distributed shared last-level TLB. Particularly, *S2TLB* prevents the storage of 39.3 percent of directory entries per cycle on average. Furthermore, when applying our shared TLB usage predictor, directory storage requirements are reduced to merely 51.7 percent with a low predictor timeout value, obtaining similar figures compared to those of a purely private TLB structure, as seen in Fig. 13 (roughly 1 percent difference).

Finally, Fig. 18 shows the execution time of different applications employing our TLB-based classification for shared TLB structures under coherence deactivation. *S2TLB* reduces execution time by 6.8 percent compared to a baseline system without coherence deactivation. Additionally, *SUP* further contributes to better system performance,

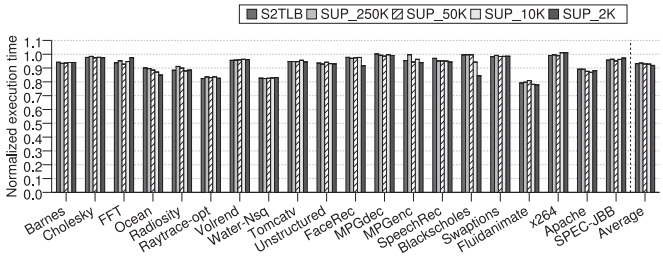
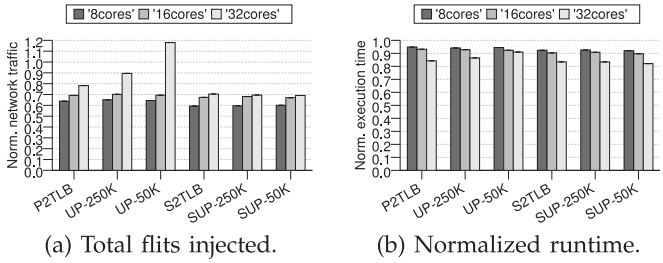


Fig. 18. Execution time under coherence deactivation. Classification with distributed shared L2 TLB.



(a) Total flits injected. (b) Normalized runtime.

Fig. 20. Scalability analysis.

reducing execution time up to 8.2 percent, even using a 2,000 cycles predictor timeout.

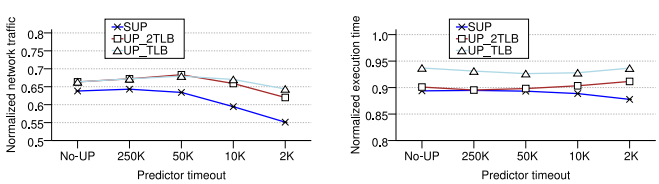
Conclusion. The classification scheme introduced in this paper for shared TLB structures benefits system performance as expected when applied to coherence deactivation. Moreover, our shared TLB usage predictor (SUP) enhances private detection without performance degradation.

6.3 Private Against Shared TLB Classification

This section offers a comparative analysis of how the classification mechanisms behave for both private and shared TLB structures when applied to coherence deactivation, focusing on the usage predictors herein detailed.

Overhead Analysis. We show results for the TLB-based classification mechanisms with a single level and two private level TLB structures (*UP_TLB* and *UP_2TLB*, respectively), and the shared TLB classification mechanism (*SUP*). All results presented in Fig. 19 are normalized to a baseline with a single TLB level without coherence deactivation, which employs broadcast TLB-to-TLB transfers for TLB miss resolution purposes.

Fig. 19a shows the total traffic for the different TLB structures. We observe how, in this case, both purely private TLB structures behave similarly, reducing traffic over 30 percent compared to the baseline. As TLB usage predictor decouples the page lifetimes from TLB size or associativity, including a private second TLB level hardly contributes to reduce network traffic. If a TLB entry is invalidated, either in the first or the second TLB level, accessing that page would result in a miss in the TLB hierarchy, thus invoking the broadcast TLB-to-TLB resolution mechanism that is causing the traffic overhead. Conversely, *SUP* reduces both TLB and cache traffic over the approach for private TLB structures. *SUP* efficiently avoids premature cache flushes and reduces TLB invalidation frequency even with low predictor values, provided that the distributed shared second TLB level acts as a filter for premature invalidations. Moreover, TLB misses are resolved through unicast messages, which represent a far more scalable solution. Particularly, *SUP* reduces total traffic to just 55.1 percent with a 2,000 cycles predictor timeout.



(a) Total flits injected. (b) Normalized runtime.

Fig. 19. Classification overhead analysis.

Similarly, Fig. 19b shows the average execution time evolution for the different predictor timeout values considered. We observe how applying the shared TLB usage predictor (*SUP*) reduces execution time as the predictor timeout decreases, a reduction of up to 12.23 percent over baseline with a 2,000 cycles timeout. Differently, even though *UP_2TLB* predictor also improves the classification accuracy, it moderately increases execution time compared to not employing a predictor, dropping its performance gainings to just 8.8 percent with a 2K predictor timeout. In other words, the performance divergence with *SUP* reaches 3.5 percent. *UP_TLB* follows a similar trend as *UP_2TLB*, but with lower performance gainings over baseline. This is due to the predictor induced invalidations, which result in higher TLB miss rates and, ultimately, into performance shrinkage.

Scalability Analysis. This section shows how the different classification schemes scale when deactivating coherence maintenance. Due to the slow pace of the simulation tools, this study is only performed using SPLASH 2 benchmarks and scientific application. All the results in Fig. 20 are normalized to a system of the same core count, with a purely private TLB structure that does not perform data classification; therefore, coherence maintenance is not deactivated. Lower predictor timeouts have not been included in this study to favor better figure readability, although they follow a similar trend.

The page classification mechanisms based on purely private TLB structures evaluated in this paper aim for small- or medium-scale systems. As can be seen in Fig. 20a, broadcast messages issued after every TLB miss do not scale for larger systems. Particularly, when employing a usage predictor for TLBs, which induces extra TLB misses, up to 50.7 percent more traffic is issued compared to *P2TLB* for a 50,000 cycles predictor timeout, ultimately offsetting the benefits of deactivating coherence for a 32-core CMP. Conversely, our classification mechanism for shared TLB hierarchies completely avoids broadcast requests, relying solely on unicast messages after missing on the first TLB level. As a consequence, *SUP* not only avoids traffic overhead for larger systems, it even reduces traffic by 30.9 percent compared to the baseline for *SUP-50K*.

Consequently, leveraging a shared last-level TLB for page classification improves global system performance over its competitors, specially with *SUP*, which improves private data detection, avoids the overheads of the prediction scheme for private TLB structures, and exploits intercore sharing patterns. Therefore, *SUP* contributes to reduce execution time by 17.8 percent for a 50,000 cycles timeout with a 32-core CMP (Fig. 20b), nearly 9 percent better performance than *UP* with the same predictor timeout.

Conclusion. Employing SUP on a system with shared last-level TLBs has proved to squeeze coherence deactivation potential to its maximum, providing a better classification while avoiding the majority of its overheads, specially for low predictor timeout values, ultimately representing the best-suited classification scheme for large-scale CMPs.

7 Conclusions

In this paper we have evaluated different approaches to the classification of data into private or shared at page level in CMPs with different TLB hierarchies. We revisit TLB-based classification for purely private multi-level TLB structures, which get similar figures to previous works. Furthermore, we propose a TLB-based classification approach which leverages the use of a distributed shared last-level TLB. A shared last-level TLB structure allows to naturally discover the sharing access pattern through the L1 TLB misses received from different cores to the home L2 TLB tile. Additionally, we show the interest of employing a usage predictor for TLBs in order to achieve an accurate classification independent from TLB size, and propose a predictor for systems with distributed shared last-level TLBs (SUP).

Our proposal improves classification accuracy while avoiding some of its overheads. Specifically, SUP improves the classification of private pages up to 78.1 percent on average. As a consequence, execution time is improved by 12.2 percent and traffic is reduced to only 55.1 percent over a single level baseline system when SUP is applied to coherence deactivation. Conversely, a TLB-based classification for a private two-level TLB structure slightly hurts execution time as long as the predictor timeout value is reduced, progressively hurting system performance up to 3.5 percent over SUP.

Acknowledgments

This work has been jointly supported by the MINECO and European Commission (FEDER funds) under the project TIN2015-66972-C5-1-R and TIN2015-66972-C5-3-R and the *Fundación Seneca-Agencia de Ciencia y Tecnología de la Región de Murcia* under the project *Jóvenes Líderes en Investigación* 18956/JLI/13.

References

[1] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proc. 38th Int. Symp. Comput. Archit.*, Jun. 2011, pp. 93–103.

[2] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by avoiding the tracking of non-coherent memory blocks," *IEEE Trans. Comput.*, vol. 62, no. 3, pp. 482–494, Mar. 2013.

[3] S. Demetriades and S. Cho, "Stash directory: A scalable directory for many-core coherence," in *Proc. 20th Int. Symp. High-Performance Comput. Archit.*, Feb. 2014, pp. 177–188.

[4] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-aware mechanism to detect private data in chip multiprocessor," in *Proc. 42th Int. Conf. Parallel Process.*, Oct. 2013, pp. 562–571.

[5] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *Proc. 17th Int. Symp. High-Performance Comput. Archit.*, Feb. 2011, pp. 62–73.

[6] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs," *ACM Trans. Archit. Code Optimization*, vol. 10, no. 1, Apr. 2013, Art. no. 2.

[7] D. Kim, J. A. J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Tech.*, Sep. 2010, pp. 111–122.

[8] Y. Li, R. Melhem, and A. Jones, "PS-TLB: Leveraging page classification information for fast, scalable and efficient translation for future CMPs," *8th Int. Conf. High-Performance Embedded Archit. Compilers*, vol. 9, no. 4, Jan. 2013, Art. no. 28.

[9] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *Proc. 36th Int. Symp. Comput. Archit.*, Jun. 2009, pp. 184–195.

[10] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Tech.*, Sep. 2010, pp. 501–512.

[11] Y. Li, R. G. Melhem, and A. K. Jones, "Practically private: Enabling high performance CMPs through compiler-assisted data classification," in *Proc. 21st Int. Conf. Parallel Archit. Compilation Tech.*, Sep. 2012, pp. 231–240.

[12] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *Proc. 21st Int. Conf. Parallel Archit. Compilation Tech.*, Sep. 2012, pp. 241–252.

[13] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *Proc. 39th Int. Symp. Comput. Archit.*, Jun. 2012, pp. 524–535.

[14] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "SWEL: Hardware cache coherence protocols to map shared data onto shared caches," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Tech.*, Sep. 2010, pp. 465–476.

[15] H. Hossain, S. Dwarkadas, and M. C. Huang, "POPS: Coherence protocol optimization for both private and shared data," in *Proc. 20th Int. Conf. Parallel Archit. Compilation Tech.*, Oct. 2011, pp. 45–55.

[16] M. Alisafaee, "Spatiotemporal coherence tracking," in *Proc. 45th IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2012, pp. 341–350.

[17] J. Zebchuck, B. Falsafi, and A. Moshovos, "Multi-grain coherence directory," in *Proc. 46th IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2013, pp. 359–370.

[18] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "An efficient, self-contained, on-chip, directory: DIR_1-SISD," in *Proc. 24th Int. Conf. Parallel Archit. Compilation Tech.*, Oct. 2015, pp. 317–330.

[19] A. Ros and A. Jimborean, "A dual-consistency cache coherence protocol," in *Proc. 29th Int. Parallel Distrib. Process. Symp.*, May 2015, pp. 1119–1128.

[20] A. Ros and A. Jimborean, "A hybrid static-dynamic classification for dual-consistency cache coherence," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3101–3115, Nov. 2016.

[21] A. Esteve, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Efficient TLB-based detection of private pages in chip multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 3, pp. 748–761, Mar. 2016.

[22] A. Esteve, A. Ros, A. Robles, M. E. Gómez, and J. Duato, "TokenTLB: A token-based page classification approach," in *Proc. Int. Conf. Supercomputing*, Jun. 2016, Art. no. 26.

[23] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "UNified instruction/translation/data (UNITD) coherence: One protocol to rule them all," in *Proc. 16th Int. Symp. High-Performance Comput. Archit.*, Feb. 2010, pp. 1–12.

[24] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for high performance address translation in chip multiprocessors," in *Proc. 43rd IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2010, pp. 313–324.

[25] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proc. 28th Int. Symp. Comput. Archit.*, Jun. 2001, pp. 240–251.

[26] A. Ros, et al., "EMC^2: Extending magny-cours coherence for large-scale servers," in *Proc. 17th Int. Conf. High Performance Comput.*, Dec. 2010, pp. 1–10. [Online]. Available: http://ditec.um.es/aros/papers/aros-hipc10.pdf

[27] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A mechanism for speculative address translation," in *Proc. 38th Int. Symp. Comput. Archit.*, Jun. 2011, pp. 307–318.

[28] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R*, ARM, Cambridge, U.K., 2012.

[29] Technical resources: Intel xeon processors. (2012, Nov.). [Online]. Available: http://goo.gl/ZlM8I2, Accessed on: Nov. 2015.

[30] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors," in *Proc. 18th Int. Conf. Parallel Archit. Compilation Tech.*, Feb. 2009, pp. 29–40.

[31] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative TLB for chip multiprocessors," in *Proc. 15th Int. Conf. Archit. Support Program. Language Operating Syst.*, Mar. 2010, pp. 359–370.
[32] Advanced micro devices. (2005, Mar.). [Online]. Available: http://www.amd.com, Accessed on: Nov. 2015.
[33] Intel corporation. (2008, Jun.). [Online]. Available: http://www.intel.com, Accessed on: Nov. 2015.
[34] *ARM Architecture Reference Manual ARMv8-A*, ARM, Cambridge, U.K., 2015.
[35] Sun microsistems. (2011, Oct.). [Online]. Available: http://www.sun.com, Accessed on: Nov. 2015.
[36] C. Kim, D. Burger, and S. W. Keckler, "Nonuniform cache architectures forwire-delay dominated on-chip caches," in *Proc. 23rd IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2003, pp. 99–108.
[37] AMD, "AMD64 architecture programmer's manual volume 2: System programming," *whitepaper*, Jun. 2010.
[38] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," HP Labs, Palo Alto, CA, USA, Tech. Rep. HPL-2008–20, Apr. 2008.
[39] P. S. Magnusson, et al., "Simics: A full system simulation platform," *IEEE Comput.*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
[40] M. M. Martin, et al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.
[41] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Proc. IEEE Int. Symp. Performance Anal. Syst. Softw.*, Apr. 2009, pp. 33–42.
[42] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Int. Symp. Comput. Archit.*, Jun. 1995, pp. 24–36.
[43] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, "The ALPBench benchmark suite for complex multimedia applications," in *Proc. Int. Symp. Workload Characterization*, Oct. 2005, pp. 34–45.
[44] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Tech.*, Oct. 2008, pp. 72–81.
[45] A. R. Alameldeen, et al., "Evaluating non-deterministic multithreaded commercial workloads," in *Proc. 5th Workshop Comput. Archit. Eval. Using Commercial Workloads*, Feb. 2002, pp. 30–38.

Albert Esteve received the MS degree in computer science from the Universitat Politècnica de València, Spain, in 2012. He is currently working toward the PhD degree in the Parallel Architecture Group (GAP), Universitat Politècnica de València with a fellowship from the Spanish Government. His research interests include cache coherence protocols, and chip multiprocessor architectures.

Alberto Ros received the MS and PhD degrees in computer science from the University of Murcia, Spain, in 2004 and 2009, respectively. In 2005, he joined the Computer Engineering Department at the same university as working toward the PhD degree with a fellowship from the Spanish government. He has been working as a postdoctoral researcher with the Universitat Politècnica de València and at Uppsala University. Currently, he is an associate professor with the University of Murcia. His research interests include cache coherence protocols memory hierarchy designs, and memory consistency for manycore architectures.

María E. Gómez received the MS and PhD degrees in computer science from the Universitat Politècnica de València, Spain, in 1996 and 2000, respectively. She joined the Department of Computer Engineering (DISCA), Universitat Politècnica de València, in 1996, where she is currently an associate professor of computer architecture and technology. She has published more than 50 conference and journal papers. She has served on program committees for several major conferences. Her research interests include the field of interconnection networks, network-on-chips, and cache coherence protocols.

Antonio Robles received the MS degree in physics (electricity and electronics) from the Universitat de València, Spain, in 1984 and the PhD degree in computer engineering from the Universitat Politècnica de València, in 1995. He is currently a full professor in the Department of Computer Engineering, Universitat Politècnica de València. He has taught several courses on computer organization and architecture. His research interests include high-performance interconnection networks for multiprocessor systems and clusters and scalable cache coherence protocols for SMP and CMP. He has published more than 70 refereed conference and journal papers. He has served on program committees for several major conferences. He is a member of the IEEE Computer Society.

José Duato received the MS and PhD degrees in electrical engineering from the Universitat Politècnica de València, Spain, in 1981 and 1985, respectively. He is currently a professor in the Department of Computer Engineering, Universitat Politècnica de València. He was an adjunct professor in the Department of Computer and Information Science, Ohio State University, Columbus. His research interests include interconnection networks and multiprocessor architectures. He has published more than 380 referred papers. He proposed a powerful theory of deadlock-free adaptive routing for wormhole networks. He was a member of the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, and the *IEEE Computer Architecture Letters*. He was cochair, member of the steering committee, vice chair, or member of the program committee in more than 55 conferences, including the most prestigious conferences in his area of interest: HPCA, ISCA, IPPS/SPDP, IPDPS, ICPP, ICDCS, EuroPar, and HiPC. He has been awarded with the National Research Prize *Julio Rey Pastor 2009*, in the area of Mathematics and Information and Communications Technology and the *Rei Jaume I Award on New Technologies 2006*. He is a member of the IEEE.

An Algorithmic Approach to Communication Reduction in Parallel Graph Algorithms

Harshvardhan[†], Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger
Parasol Laboratory
Dept. of Computer Science and Engineering
Texas A&M University
{ananvay, fidel, amato, rwerger}@cse.tamu.edu

***Abstract*—Graph algorithms on distributed-memory systems typically perform heavy communication, often limiting their scalability and performance. This work presents an approach to transparently (without programmer intervention) allow fine-grained graph algorithms to utilize algorithmic communication reduction optimizations. In many graph algorithms, the same information is communicated by a vertex to its neighbors, which we coin algorithmic redundancy. Our approach exploits algorithmic redundancy to reduce communication between vertices located on different processing elements. We employ algorithm-aware coarsening of messages sent during vertex visitation, reducing both the number of messages and the absolute amount of communication in the system. To achieve this, the system structure is represented by a hierarchical graph, facilitating communication optimizations that can take into consideration the machine's memory hierarchy. We also present an optimization for small-world scale-free graphs wherein *hub vertices* (i.e., vertices of very large degree) are represented in a similar hierarchical manner, which is exploited to increase parallelism and reduce communication. Finally, we present a framework that transparently allows fine-grained graph algorithms to utilize our hierarchical approach without programmer intervention, while improving scalability and performance. Experimental results of our proposed approach on $131,000+$ cores show improvements of up to a factor of 8 times over the non-hierarchical version for various graph mining and graph analytics algorithms.**

***Keywords*—parallel graph processing; graph analytics; big data;**

I. Introduction

Graph algorithms are used in many real-world problems, from mining social networks and big-data analytics to scientific computing where meshes are used to model physical domains. With the advent of big data, there has been a significant interest in improving the scalability of graph algorithms. Moreover, certain graphs of interest – such as web-graphs and social networks – exhibit small-world scale-free characteristics with a power-law degree distribution and small diameters. While regular and irregular meshes can be partitioned to decrease the edge-cut between partitions, partitioning scale-free graphs is difficult due to the presence of *hub vertices* with very high out-degrees. As communication in many algorithms is directly proportional to the number of cut-edges, these types of graphs experience heavy communication, severely limiting scalability.

A key pattern that arises in many parallel graph algorithms is the need for a vertex to propagate the same information to all vertices in its neighborhood. For example, consider the traditional fine-grained expression of the breadth-first search algorithm in Figure 1. First, the *vertex-operator* (Figure 1(a)) checks to see if a vertex is active (grey), and if so, it propagates its distance from the source to its neighbors using the *neighbor-operator* (Figure 1(b)). The neighbor-operator then updates the distance of the neighbor if needed, and marks the neighbor as active (grey) in the next iteration. A crucial observation is that the vertex-operator visits all of its neighbors with semantically identical information, which may lead to redundant communication in distributed-memory architectures. This communication pattern is present in a large class of important algorithms such as breadth-first search, betweenness centrality, connected components, community detection, PageRank, k-core decomposition, triangle counting, etc. These algorithms are used in graph mining and big-data applications where performance is critical.

```
bool bfs_vertex_op(vertex v)
  if (v.color == GREY)          // Active if GREY
    v.color = BLACK;
    VisitAllNeighbors(bfs_neighbor_op(_1, v.dist+1), v);
    return true;                // vertex was Active
  else  return false;           // vertex was Inactive
```

(a) vertex-operator

```
bool bfs_neighbor_op(vertex u, int new_distance)
  if (u.dist > new_distance)
    u.dist = new_distance;  // update distance
    u.color = GREY;         // mark to be processed
    return true;            // vertex was updated
  else  return false;
```

(b) neighbor-operator

Fig. 1. The vertex- and neighbor-operators for breadth-first search.

The cost of memory accesses on large-scale distributed-memory systems is highly non-uniform. The communication patterns exhibited in graph algorithms executed on these systems are well-known to limit scalability. For this situation, optimizations to alleviate communication pressure can occur in several forms:

- Messages destined to the same processing element can be aggregated and combined into a single message.
- Creation of redundant messages that are a result of algorithmic redundancy (such as in breadth-first search) can be eliminated completely, and a single copy can instead be sent between processing elements.
- Bottlenecks caused by the presence of hub vertices can be mitigated by reducing incoming communication to hubs locally, and a single contribution can then be forwarded off-processor.

[†]Now at Google Inc., Mountain View, CA, USA.

1089-795X/15 $31.00 © 2015 IEEE
DOI 10.1109/PACT.2015.34

However, as algorithms are best expressed in a fine-grained manner that makes them oblivious to the graph structure, and as graphs themselves are represented as flat data-structures, only the first of these three optimizations is typically applied, as the knowledge of machine hierarchy is unavailable.

Our approach is to allow algorithms to be expressed in the natural vertex-centric manner while transparently applying communication optimizations without programmer intervention. The mechanism by which this is achieved is through the construction of a hierarchical representation of the input graph that is aligned with the machine hierarchy to identify local and non-local elements. By using this hierarchical representation and algorithm-level knowledge, we are then able to identify and transparently apply these communication optimizations for fine-grained algorithms.

Our contributions include:

- A software framework to transparently (i.e., without user intervention) allow fine-grained graph algorithms to execute in a manner that is machine-hierarchy aware, enabling important communication optimizations to be applied.
- A novel algorithmic approach to reduce communication, both in number of messages sent and *total amount of data* sent by exploiting algorithmic redundancy observed in parallel graph algorithms.
- Experimental evaluation on two large-scale systems at $12,000+$ and $130,000+$ cores, with several important graph mining and graph analytics algorithms, such as breadth-first search, connected components, k-core and PageRank showing improvements of $2.5\times$ to $8\times$ over traditional approaches.

II. Our Approach

Our goal is to improve scalability and performance of graph algorithms by reducing the number and total volume of messages sent during execution. We first discuss locality-based communication optimizations for graphs in Section II-A and then follow with an additional optimization suited specifically for reducing bottlenecks due to hubs in Section II-B.

A. Locality-Based Communication Optimization

Communication in graph algorithms occurs between vertices through edges, which represent a source-target pair (s, t). For edges with the source s and with targets $t_0, t_1, \ldots$ stored on the same destination location, it is possible to aggregate messages for all such equivalent edges. Further, if all messages represent the same information, a single message can be sent to the destination location and then applied to $t_0, t_1, \ldots$. This concept can be applied recursively by considering a grouping of vertices and identifying same destination location pairs amongst these groupings. The same communication reduction techniques apply to all levels of this recursive process.

To enable communication reduction, we need a model of the system that captures the locality of the graph. To achieve this, we overlay the input graph on top of the machine hierarchy, thereby creating a hierarchically coarsened graph. This is described in Section II-A1. Once the graph is coarsened, we use a translation layer, called the hierarchical paradigm (Algorithm 2), to execute the fine-grained algorithm on the coarsened hierarchy. This is described in Section II-A2. Combining the fine-grained specification with the hierarchy results in a coarsening of the algorithm itself, and allows for the use of algorithm-driven communication optimizations.

1) Hierarchy Construction: The transformation of the flat input graph to a hierarchical graph based on the machine-locality information is a mutating process that groups the graph's vertices into partitions based on the locality of each vertex provided by the machine hierarchy. This replaces multiple vertices in a partition, along with intra-partition edges, with a single super-vertex representing the underlying sub-graph. All edges between two partitions are replaced with a super-edge (composite edge) representing the coarsened communication between two sub-graphs. The super- vertices and super-edges form a super-graph. This process is shown in Figure 2 and detailed in this section.

A hierarchical graph consists of two or more levels of graphs, where the graph at level i is a super-graph of the graph at level $i-1$. The lowest level is the base-level. For building our hierarchy, we first partition the input graph into sub-graphs, where each sub-graph has a similar number of vertices, and assign each sub-graph to a processor. This partitioning can be computed with an external partitioner to reduce the edge-cut of the graph, which may improve the overall performance, while the hierarchical approach will take care of the remaining cross-edges. This forms the base-level of our hierarchical representation (G_0). Next, we create a hierarchy of M levels on this, matching the machine hierarchy. For graph G_i at every level ($0 \leq i \leq M$), we create a super-graph G_{i+1}, such that each vertex in G_{i+1} represents a sub-graph partition of G_i. Super-edges are added between two super-vertices of G_{i+1} if there exist inter-partition edges of their corresponding sub-graphs in G_i. The inter-partition edges of lower-level graph G_i are then removed and their information is stored on the corresponding target's super-edge. This is done to preserve the locality of edges, as all edges that point to the same target are stored at that target vertex's processor.

Our hierarchical graph consists of the base-level graph, along with one or more levels of super-graphs of the base-level graph, with super-vertices representing each vertex-partition and super-edges representing inter-partition edges (communication). The hierarchical representations obtained thus naturally expresses the machine topology. We note that creating the hierarchy is a one-time event that happens immediately following graph construction. Once the hierarchy is created, multiple algorithms can take advantage of it.

2) Using the Hierarchy: The hierarchical graph paradigm is presented in Algorithm 2. In order to execute algorithms, the paradigm proceeds in *supersteps* similar to the bulk-synchronous parallel (BSP) model [22]. However, the supersteps are executed in a manner that is aware of the machine hierarchy. Within each hierarchical superstep, the paradigm processes the active vertices in each level of the graph hierarchy iteratively. For each active vertex in the base-level graph, the results from processing it are sent to its neighbors in the same partition (lines 4-8). Any cross-partition edges for that level of hierarchy are ignored, as they will be processed by the upper-level of the hierarchy. The results from processing

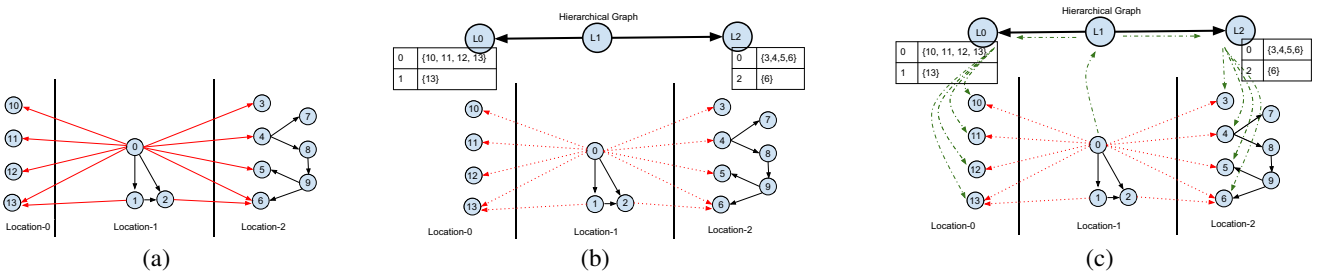


Fig. 2. Creating a hierarchy: (a) A flat graph, with remote, inter-partition edges (red lines) and (b) its hierarchical representation with metadata. Using the hierarchy: (c) Remote communication during algorithm execution (shown in green). Hierarchical graph replaces inter-partition edges in original graph (red dashed-lines represent deleted edges) with a single super-edge between each partition (solid black lines on top-level graph). Generalizable to multiple levels of machine hierarchy.

Algorithm 1 Hierarchical Graph Creation

Input: Graph G_i, int N

1: **if** $i = N$ **then**
2: return
3: **end if**
4: Graph G_{i+1} = Partition(G_i, MachineHierarchy)
5: **for all** $(u, v) \in Edges(G_i)$ **do**
6: **if** $Super_{i+1}(u) \neq Super_{i+1}(v)$ **then**
7: $E_i := E_i \backslash (u, v)$
8: $E_{i+1} := E_{i+1} \bigcup (Super_{i+1}(u), Super_{i+1}(v))$
9: **end if**
10: **end for**
11: Hierarchy.add(G_{i+1})
12: Construct(G_{i+1}, N)

Algorithm 2 Hierarchical Graph Paradigm

Input: Graph G_0, Locality-Hierarchy H, Hubs H_{hubs}

1: **while** active vertices $\neq \emptyset$ **do**
2: **for all** active vertices $v_i \in G_0$ **par do**
3: $W_i \leftarrow$ process(v_i)
4: **for all** neighbors $u_i \in G_0$.adjacents(v_i) **do**
5: **if** H.parent(u_i) = H.parent(v_i) **then**
6: visit-neighbor(u_i, W_i)
7: **end if**
8: **end for**
9: **for all** neighbors $p_i \in H$.adjacents(H.parent(v_i)) **do**
10: async(visit-neighbor(p_i, W_i))
11: // apply visit-neighbor recursively on children of p_i who are neighbors of v_i
12: **end for**
13: **for all** hubs $h_i \in H_{hubs}$.adjacents(v_i) **do**
14: visit-neighbor(h_i, W_i)
15: **end for**
16: **end for**
17: **end while**

the active vertex are then forwarded to the super-vertex of the current partition (lines 9-12), which then transmits them via super-edges to the target super-vertices. If the *process* function in line 3 sends the same update to multiple neighbors (by calling VisitAllNeighbors, for example, as in the case of BFS, PageRank, connected components, k-core, betweenness centrality, etc.), the paradigm only creates a single copy of the update to send via the super-edges. We also provide a specialization for high-degree vertices (described later in Section II-B) that uses a similar communication reduction mechanism (lines 13-15). Finally, the updates are then recursively pushed to the respective target vertices in the lower-level graphs at the target partition, whose edges were replaced by the super-edge.

The graph algorithms themselves do not change and are hierarchy-oblivious, allowing us to reuse fine-grained vertex-centric algorithms on coarsened graphs. This decouples users from the details of machine and locality exploitation.

B. Distributed Hubs Optimization

Hub vertices, or hubs, are vertices with a high in- or out- degree. Many small-world scale-free graphs, such as web-graphs and social-networks, exhibit a power-law degree distribution, with most vertices connected to a few other vertices, but very few ($< 1\%$) vertices connected to an extremely large number of vertices. While our locality-based hierarchy reduces outgoing communication caused by hub vertices, communication can be further reduced by specialized optimizations for high in-degree vertices.

We introduce a new approach to reduce all messages to a hub from the same processor to a single value which can then be applied to the hub. We assume that the applied operator is both associative and commutative. When a vertex tries to update (send a message to) a hub vertex, the operator is applied to a local representative for that hub. At the end of each superstep of the algorithm, the results of the local updates are flushed and applied to the original hubs. This in effect distributes the work for hubs across the system.

Construction of the distributed hubs occurs by first identifying the hub vertices using a simple scan through the graph, and then creating a super-graph where each super-vertex contains metadata about the hub vertices, and all edges to the hub are replaced by the super-edge. This metadata can only be written to and not read from, which does not require the value to be kept coherent, and works with our asynchronous update model (Section IV-B).

We handle the identification of hubs and creation of the hierarchy automatically using the degree (number of edges) of each vertex. The user may choose to provide a cutoff for this size, beyond which vertices will be treated as hubs, or they may choose to use the top k% vertices as hubs. In either case, the framework identifies the hubs so algorithm-developers do not need to account for this.

This approach is similar to recent work presented in [20], which replicates the hub vertices on other processors (called hub-representatives), allowing local vertices to read from and write to the hub vertices. However, their approach replicates data for each hub vertex, which needs to be kept synchronized, leading to extra communication and affecting scalability. Their approach also requires algorithms to be aware of the existence of these hub representatives and need to be modified to specify the synchronizing and reduction behaviors of the representatives. In contrast, our approach is framework-level, and keeps the algorithm itself agnostic to hubs. Further, our approach does not replicate vertex data.

III. Modeling

In this section, we describe how we exploit both the redundant nature of many parallel graph algorithms and the power-law characteristics of many input graphs, resulting in a reduction of total communication in the system.

A. Communication Reduction

The hierarchical approach can reduce communication in the system by reducing the number of bytes required to update neighboring vertices. Without loss of generality, we assume a non-multi-edged graph where two vertices are connected by at most one edge, and that the graph algorithm visits each vertex (and consequently every edge, due to the non-multi-edged property). We also assume a BSP/level-synchronous model [22], [17] where an active vertex performs some computation and updates its neighboring vertices with the result of this computation. The BSP model gives the most conservative estimate for this analysis. Any algorithm that communicates more, such as in an asynchronous model with redundant work, will observe a greater communication reduction using our approach. Let us assume the size (in bytes) of this result is r $bytes$, and that it is uniform for all vertices. In the traditional BSP case, used in existing graph libraries such as Pregel [17], the Parallel Boost Graph Library [10], [7] and the Graph500 benchmark reference implementation [1], this will imply potentially $O(|adj(v)|)$ $bytes$ being communicated, where $adj(v)$ gives adjacencies of vertex v. In fact, the amount of communication is:

$$Comm(v) = \sum_{i \in P} m_i(v) \cdot r \quad bytes \tag{1}$$

Where P is the number of processors, $m_i(v)$ gives the number of adjacents of vertex v that are stored on processor i, and r is the size of the message being sent. Traditional BSP libraries employ aggregation to reduce the number of messages. However, the total number of bytes stays the same.

In many cases, for a large class of graph algorithms, such as breadth-first search, PageRank, k-core decomposition, connected components, strongly-connected components, topological sort, betweenness centrality, triangle counting, etc., the vertex sends the same information to all its neighbors. In such cases, the amount of data sent (in bytes) can be reduced using our approach. The following reduction is applicable to such algorithms. For other cases, we may not observe any reduction in the amount of data, but we will still reduce the number of messages sent. The amount of data communicated in the first case is the lowest possible given the algorithm.

Using a hierarchical approach, the vertex may only need to send the result of its computation once for every *super-edge* instead of once for every edge. Therefore, the amount of communication for a hierarchical graph can be given by:

$$Comm^H(v) = \sum_{i \in P} \begin{cases} 1 & : m_i(v) \geq 1 \\ 0 & : m_i(v) = 0 \end{cases} \cdot r \quad bytes \tag{2}$$

This effectively means that for any vertex the hierarchical approach performs the least amount of communication possible (both in number of messages and total size of communication) for a given partitioning strategy, as it sends only a single result to each processor that stores any of the vertex's neighbors. Any lower communication can not guarantee the correctness of a general graph algorithm, without changing it.

The total bytes communicated across the system in the hierarchical approach for a graph G can then be given by:

$$\begin{aligned} Comm^H(G) &= \sum_{v \in V} Comm^H(v) \\ &= \sum_{v \in V} \sum_{i \in P} \begin{cases} 1 & : m_i(v) \geq 1 \\ 0 & : m_i(v) = 0 \end{cases} \cdot r \quad bytes \end{aligned} \tag{3}$$

This gives the upper bound of $O(min(V \cdot P, E))$. However, we note that the worst-case upper bound is equivalent to the case of a dense graph, where every vertex is connected to every other vertex and the communication for the traditional level-synchronous approach would have otherwise been $O(V^2)$. For sparser graphs, Equation 3 gives a more accurate estimate. We show empirical evaluation of this communication reduction in Section V-C.

B. Space Overhead

Creating a hierarchical graph adds space overhead compared to the traditional flat graph approach. The hierarchical graph G_1 uses a single vertex corresponding to each partition of the base-graph G_0, which requires $O(p)$ space overall, assuming p partitions in G_0, one per processor. Further, the number of edges in G_1 correspond to the number of partitions in G_0 that have edges between them, with one super-edge per pair of neighboring partitions in G_0. In the worst-case, there can be $O(p^2)$ super-edges in G_1 corresponding to a complete graph in G_0. Therefore the bound on the size of G_1 is $O(p^2)$ for the entire graph, or $O(p)$ overhead per-processor for using the hierarchy. Note that the metadata on super-edges does not contribute to overhead, as it replaces the deleted inter-partition edges from G_0, keeping the total size constant in this regard.

For the hub-hierarchy, every processor stores metadata for any hub vertices that have an edge to a vertex on that processor. This implies $O(|hubs|)$ storage per processor, only if there is an edge to that hub from that processor. However, since the number of such hubs is generally small, the space overhead is

small in practice. Further, our algorithm for creating the hub-hierarchy is parameterized based on a user-specified lower-limit on the size of hub vertices, so the number of vertices treated as hubs can be varied to suit any space constraints on the system.

IV. Implementation

While the hierarchical approach is generally applicable to any distributed memory graph library, for this work, we implemented our approach in the STAPL Graph Library (SGL) [11] to evaluate performance, due to SGL's ease of use and modification, and scalable performance [11], [12] (Section V-A). In this section, we give an overview of SGL and describe the relevant features and extensions needed to support the hierarchical approach. We also show how users can express important graph mining and graph analytics algorithms using our hierarchical paradigm, and thus benefit from our approach.

A. The STAPL Graph Library

SGL is a generic parallel graph library that provides a high-level framework which allows the user to concentrate on parallel graph algorithm development and decouples them from details of the underlying distributed environment. It consists of a parallel graph container (`pGraph`), a collection of parallel graph algorithms to allow users to easily process graphs at scale, and a graph engine that supports level-synchronous and asynchronous execution of algorithms.

The `pGraph` container is a distributed data storage built using the `pContainer` framework (PCF) [21] provided by the Standard Template Adaptive Parallel Library (STAPL) [6]. It provides a shared-object view of graph elements across a distributed-memory machine. The STAPL Runtime System (RTS) and its communication library ARMI(Adaptive Remote Method Invocation) is decoupled from the underlying platform, providing portable performance, thus eliminating the need to modify STAPL applications. The RTS abstracts the physical parallel processing elements into *location*s, components of a parallel machine where each one has a contiguous memory address space and associated execution capabilities (e.g threads). ARMI uses the remote method invocation (RMI) abstraction to allow asynchronous communication on shared objects while hiding the underlying communication layer (e.g MPI, OpenMP).

B. Expressing Graph Algorithms

Graph algorithms in SGL are expressed in a vertex-centric fine-grained manner, and decoupled from parallelism and communication details, as well as from the processing of the graph (e.g. flat or hierarchical). In this section, we show how an example algorithm, breadth-first search (BFS) (Figures 1 and 3) can be expressed in SGL's graph paradigm. Other algorithms such as connected components, k-core, PageRank, community detection, graph coloring, betweenness centrality, pseudo-diameter, etc. can also be expressed in a similar manner. We evaluate these algorithms in Section V-C.

To express an algorithm, the user provides two operators – a *vertex-operator* (Figure 1(a)) which performs the computation of the algorithm on a single vertex and a *neighbor-operator* (Figure 1(b)) that updates the neighbor-vertices of the source vertex with the results of the computation. These two operators are provided to the graph paradigm along with the input graph (Figure 3(a)). The graph paradigm (Figure 4) executes the provided operators on active vertices of the input graph and handles communication, termination-detection of the algorithm, current active vertices, and the execution strategy (level-synchronous or asynchronous). The algorithm terminates when all vertex-operators return false. The user's vertex operators are therefore decoupled from these details and can focus on expression of the algorithm.

```
void BFS(Graph graph, vertex source)
  source.color = GREY;
  graph_paradigm(bfs_vertex_op(), bfs_neighbor_op(), graph);
```

(a) Fine-grained BFS

```
void Hierarchical_BFS(Graph graph, HierarchyGraph H,
                      vertex source)
  source.color = GREY;
  hierarchical_paradigm(bfs_vertex_op(), bfs_neighbor_op(), graph, H);
```

(b) Hierarchical BFS

Fig. 3. The (b) traditional fine-grained BFS algorithm. The hierarchical version of BFS is shown in (d), where only the paradigm has changed.

```
void graph_paradigm(Graph g, VertexOp wf, NeighborOp uf)
 bool active = true;
 while(active)
   pre_compute(g);
   // apply vertex-operator to each vertex, reduce to
   // find #active vertices. wf returns true (active),
   // or false (otherwise), spawns neighbor-operators.
   active =
    reduce(map(vertex_wf(wf, visitor(uf)), g), logical_or());
   global_fence();
   post_compute(g);
```

Fig. 4. Pseudocode for the graph paradigm.

Breadth-first search (BFS) is an important algorithm due to its widespread direct uses and indirect uses as a part of numerous other algorithms (e.g., betweenness centrality, pseudo-diameter). The overall BFS algorithm results from invoking the *graph_paradigm* (Figure 3(a)) and providing it the operators presented earlier in Figure 1 to obtain the flat (non-hierarchical) BFS. Alternatively, invoking our *hierarchical_paradigm* (Figure 3(b)) with the same operators results in a hierarchical BFS.

Hierarchical Paradigm in SGL. The hierarchical graph paradigm allows the execution of vertex-centric algorithms on hierarchical graphs, as it is a drop-in replacement for the standard graph paradigm, as seen in Figure 3(b), such that existing vertex and neighbor operators need not be modified to take advantage of the hierarchy, or even be aware of it. Different graph algorithms can take advantage of the hierarchical paradigm simply by swapping the call to *graph_paradigm* with *hierarchical_paradigm* and providing it the hierarchical graph.

V. Experimental Evaluation

In this section, we evaluate the performance of our hierarchical approach as compared with the traditional (flat) approach for a set of important graph mining and graph analytics algorithms. We also compare our base-line performance with other graph libraries.

Our experiments were run on three platforms – a Cray XE6 machine with 153,216 cores at the National Energy Research Scientific Computing Center (Hopper), an IBM Blue Gene/Q with 393,216 cores and a smaller Cray XE6m machine with 576 cores available to us. Experiments were run on the Graph 500 benchmark inputs [1], a benchmark for data-intensive and graph applications, as well as other real-world graphs available to us. Results reported are averaged over 32 runs with a high confidence interval. We used the default partitions specified in the input graph files, since partitioners such as ParMETIS are ineffective in partitioning graphs such as Twitter or Kronecker (Graph500 input). Our code was compiled using the GCC 4.8.3 compiler.

A. Comparisons: SGL and Other Libraries

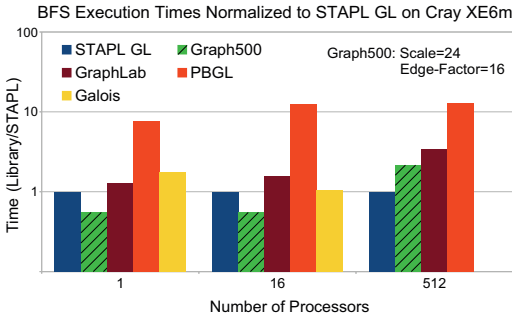


Fig. 5. Execution times of Graph500 on various graph libraries normalized to SGL on CRAY XE6m. Shared-memory libraries shown to 16 cores.

We compare SGL's base (non-hierarchical) implementation, with existing graph libraries for the Graph500 benchmark to establish a base-line for our performance results to follow. These include various popular distributed and shared-memory graph libraries, including the Graph500 benchmark implementation in MPI, Parallel Boost Graph Library (PBGL), Galois, and GraphLab/PowerGraph.

Figure 5 shows their execution times normalized to SGL's base (non-hierarchical) implementation for different processor-counts on a Cray XE6m with 576 cores. SGL's base implementation, using standard techniques such as combiners and aggregators, performs similarly to existing graph libraries, though it is more scalable than comparable distributed-memory libraries such as PBGL and GraphLab, and comparable to shared-memory ones such as Galois [13].

The Graph500 benchmark implementation is initially 1.9x faster than SGL on 16 cores due to low overhead, as the implementation uses arrays of integers to represent their graphs, whereas SGL has a generic graph container. However, SGL scales better in distributed-memory, where at 512 cores SGL is 2.2x faster than the benchmark. There exists an implementation of the benchmark [5] that is 2-3x faster than our baseline, however, this too is not a general-purpose library, but a benchmark-specific implementation that can tradeoff genericity and programmability for targeted performance. GraphLab/PowerGraph [16], [9] is a popular graph processing framework that allows asynchronous and bulk-synchronous computations, similar to SGL. However, it exposes

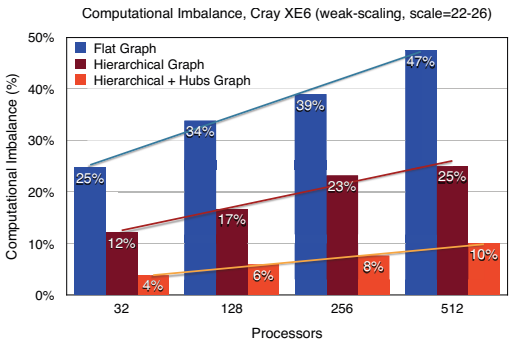


Fig. 6. Comparison of work-imbalance (number of edges) for flat and hierarchical graphs for Graph500 inputs (weak-scaling) on a CRAY XE6m, along with trendlines showing growth rates.

users to low-level details of parallelism such as memory consistency and concurrency, as they have to choose a consistency model for their application and understand its implications. PowerGraph includes optimizations for processing small-world scale-free graphs such as web-graphs. PBGL [10], [7] is a distributed graph library that uses ghost-vertices for communication, which may limit performance at scale. A detailed comparison of SGL with these libraries and others was shown in [11].

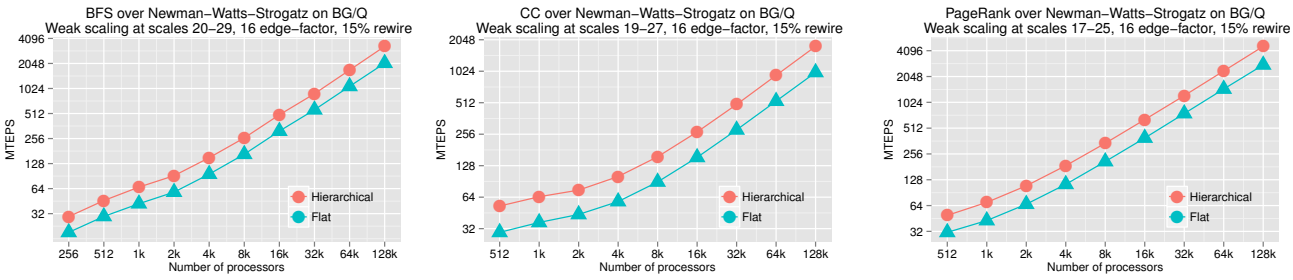
B. Improving Work Imbalance

As the number of processors increases, the graph partition can become imbalanced in the amount of edges each processor has to process. This is further aggravated by scale-free graphs where the hub-size also increases drastically with the size of the graph. This results in the processor storing a high-degree vertex performing more work processing its outgoing edges, which negatively affects scalability and performance. Our hierarchical approach alleviates this by reducing the number of inter-partition edges for each vertex to a single super-edge, and distributing the work of applying updates more evenly as the updates are now applied on the target location instead of the high-degree source.

We show the effect of our hierarchical approach on the work imbalance across processors for different input sizes of the Graph500 input graph at varying processor-counts in Figure 6. The flat partitioning has severe work imbalance which gets worse as the problem and machine size is scaled. The locality-based hierarchical approach is able to reduce this imbalance significantly, while also reducing the rate at which the imbalance grows (demonstrated by the diverging trendlines). The locality-based hierarchical approach in combination with the hubs-based hierarchy further reduces the imbalance significantly and slows down the growth even more. This, as we will observe in the next section, improves scalability and performance through a better load-balanced execution, even while being decoupled from the algorithm.

C. Applications

In this section, we first demonstrate the effectiveness of our approach at scale ($12,000+$ and $130,000+$ cores on Figures 7, 8) on two different large-scale machines, and then dive down to lower core-counts to observe trends and explain



(a) (b) (c)

Fig. 7. Throughput of (a) BFS, (b) connected components and (c) PageRank on Watts-Strogatz input on BGQ, **131,072 cores**. At scale, the hierarchical approach has an improvement of 1.59x, 1.78x and 1.64x over the flat approach, respectively.

our results. The performance benefit of our approach depends on the computation/communication cost ratio. For instances where the communication becomes a bottleneck, our approach benefits greatly. This is dependent on three things: the input graph, the algorithm and the system. A denser graph is harder to partition effectively, and therefore will result in cross-edges with a higher probability, resulting in higher communication costs. On the other hand, algorithms such as betweenness centrality perform heavier computation than algorithms such as breadth-first search, and can therefore hide the communication overhead better, while some systems, such as the IBM Blue Gene/Q, have slower processors and faster networks to achieve the same effect. Figure 7 shows the performance of various algorithms on a Watts-Strogatz small-world network on up to **131,072 cores**. At scale, the hierarchical approach is able to see improvements of $1.59\times$ to $1.78\times$ over the traditional flat algorithm. Our experiments are designed to evaluate our approach on a wide range of important graph algorithms that are representative in their class, as well as different systems.

Performance at Scale. We ran breadth-first search (BFS), connected components (CC), PageRank (PR) and k-core decomposition (KC) algorithms at scale on **12,288 cores** on a Cray XE6 machine to demonstrate the performance benefits of the hierarchical approach at scale. Our results (Figure 8) show a $2.35\times$ performance improvement for breadth-first search, a $7.26\times$ to $8.54\times$ improvement for connected components, a $3.6\times$ to $3.95\times$ improvement for PageRank, and a $4.43\times$ to $5.84\times$ improvement for k-core decomposition. We attribute these improvements to lower communication and better load-balance provided by the hierarchical approach, which improves scalability, as we will demonstrate in this section. Compared to the IBM Blue Gene/Q system, the performance benefits on the Cray XE6 are more substantial using our approach. This is due to the Blue Gene system having slower cores and a faster network, which lowers the computation/communication ratio compared to the Cray XE6. When the ratio is higher (due to a slower interconnect, for example), our approach yields better benefits.

1) Fundamental Algorithms: **Graph500 BFS.** We evaluated the Graph500 benchmark application that simulates data-intensive HPC workloads. The benchmark performs breadth-first traversals of the input graph from multiple source vertices. The Graph 500 input graph simulates social networks and web-graphs and exhibits small-world scale-free behaviour, i.e.,

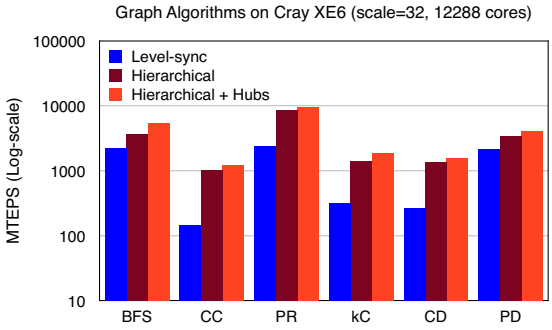


Fig. 8. Scalability (Throughput, log-scale) of various algorithms using hierarchical approach. Graph500 input graph with 4 billion vertices and 64 billion edges at **12,288 cores**.

it has a short diameter ($<$ 16 hops) and vertices with very high out-degrees. These high out-degree vertices (hubs) cause scalability bottlenecks due to communication.

Figure 9(a) compares the number of bytes communicated in the base-line (flat) approach with that in our hierarchical approach. As can be observed, the hierarchical approach is able to significantly reduce the number of bytes sent over the network by $2.7\times$ to $3.3\times$. This directly translates to the performance of the algorithm (Figure 9(b)), where we observe a $1.8\times$ to $2.1\times$ improvement over the base algorithm. For example, at 512 cores, BFS on the Graph 500 input graph communicated (sent/received) 33.87 GB of data across the system in the base (flat) case. This was reduced to 12.56 GB for our hierarchical approach and then further to 10.3 GB for the hierarchical with hubs approach. With the network sending only a third of the data, performance improved.

Connected Components. Connected components [15] (CC) has a heavier communication pattern than breadth-first traversal, and therefore, we expect our hierarchical strategy to provide a higher speedup versus the traditional paradigm. Figure 10 shows this to be the case. As we increase in scale, the benefits of our approach become more evident. At 512 cores, the total bytes communicated was reduced from 111.9 GB in the base-case to 68.91 GB for our hierarchical approach, to 44.1 GB for the hierarchical with hubs approach, a reduction of $1.6\times$ to $2.6\times$ respectively (Figure 10(a)), providing a $2.54\times$ to $3.38\times$ speedup over the base-case (Figure 10(b)).

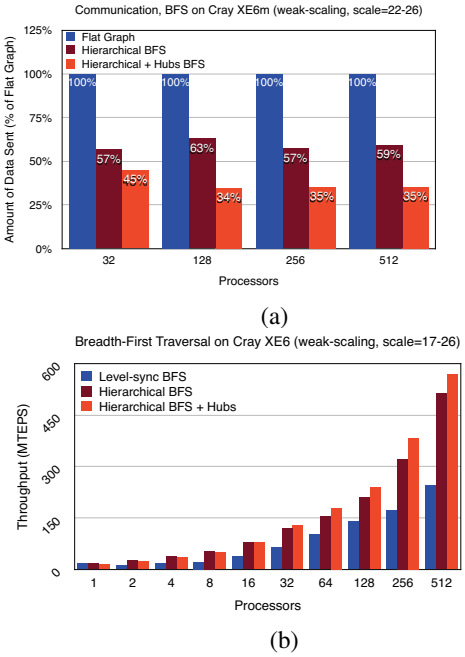


Fig. 9. Communication reduction using hierarchical approach and (b) Scalability (Throughput) of BFS on Graph500 benchmark.

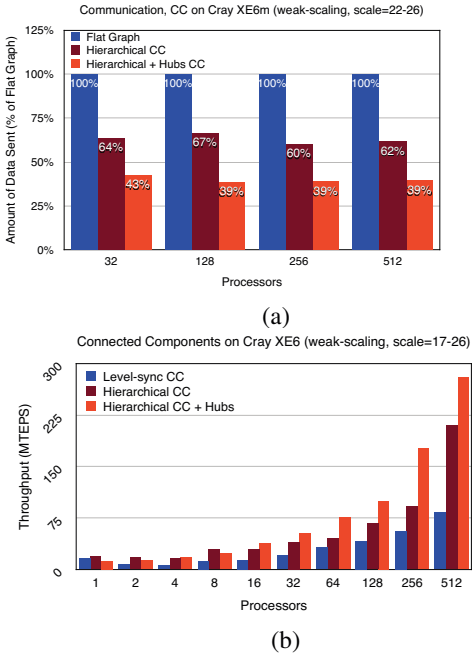


Fig. 10. Communication reduction using hierarchical approach and (b) Scalability (Throughput) of Connected Components on Graph500 input.

2) Graph Mining Algorithms: **PageRank.** PageRank [4], [18] is an important algorithm used to rank web-pages on the internet in order of relative importance. As an example of an iterative random walk algorithm, each vertex calculates its rank in iteration i based on the ranks of its neighbors in iteration $i-1$ and then sends out new ranks to its neighbors for the next iteration. The algorithm terminates when either a certain number of iterations have been reached or the ranks have converged.

PageRank exhibits even heavier communication than connected components, due to all vertices being active (and communicating over all edges) in every iteration of the PageRank algorithm, making it a worst-case scenario for communication. Our evaluation of this algorithm in Figure 11 shows a communication reduction of $1.36\times$ to $2.18\times$ (from 711 GB for flat to 521 GB for hierarchical and 326 GB for hierarchical with hubs), corresponding to a $2.54\times$ to $2.73\times$ speedup over the base-case at 512 cores, which shows the worst-case communication is substantially improved.

k-core Decomposition. A k-core of a graph G is a maximal connected sub-graph of G in which all vertices have degree at least k. The k-core algorithm is widely used to study clustering and evolution of social networks [2]. It is also used to reduce input graphs to more manageable sizes while maintaining their core-structure. The typical parallel algorithm iteratively deletes vertices with degree less than k until only vertices with degree greater than or equal to k exist. k-core has a different communication pattern than either of BFS, CC or PageRank. Here too, the hierarchical approach is able to provide benefits over the base case (Figure 12). At 512 cores, our approach is able to reduce the total bytes communicated across the system from 48.8 GB for the base-case to 21.4 GB for the locality-based hierarchy to 11.2 GB for the hierarchy with hubs, a reduction of $2.3\times$ to $4.3\times$, leading to a speedup of $1.2\times$ to $1.9\times$.

Community Detection. Community Detection is an important application that is widely used to detect groups or clusters in social networks. We use a modularity maximization algorithm to label vertices to their assigned communities. The application iteratively assigns the most frequently occuring label in the local neighborhood of each vertex, until the global quality of the labeling can no longer be improved. Figure 13 shows the improvement in performance obtained by using our hierarchical approach. We note that while the locality-based hierarchy provides a significant improvement in performance, the addition of the hub-based hierarchy does not further improve the performance appreciably. This is due to the fact that, as described in Section II-B, the hub-based hierarchy relies on reducing updates on hubs to a single value on each location. However, for community detection the labels of the entire neighborhood is needed, leading to a minimal reduction for the hub-hierarchy.

3) Application of Traversals: **Betweenness Centrality.** Betweenness Centrality is an important graph-mining algorithm that is used to identify vertices with large influence in a network. For example, it is used to understand the social influence of users on Facebook and Twitter. Our implementation uses Brandes' algorithm [3], which performs forward and backward traversals of the graph to compute the number of shortest paths passing through each vertex. This application directly benefits from our hierarchical approach, as seen in Figure 13. Due to the large number of traversals involved, the actual saving in time is significant. For example, on 512 cores, the execution time of the application was reduced from $1,635.93$ seconds to $1,094.63$ seconds, an improvement of 50%.

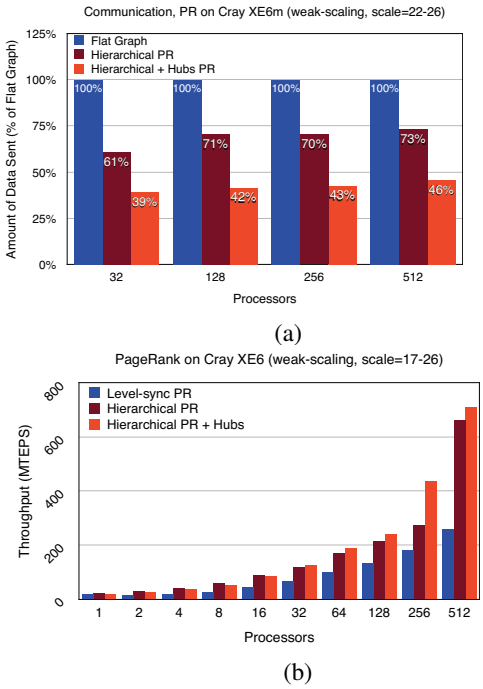


Fig. 11. Communication reduction using hierarchical approach and (b) Scalability (Throughput) of PageRank on the Graph500 input graph.

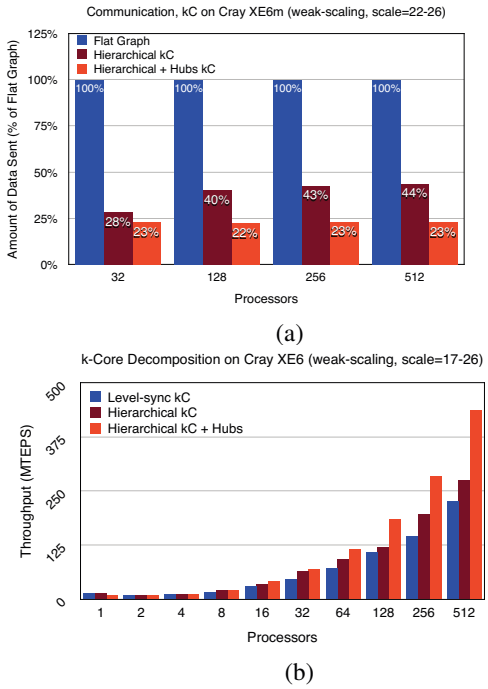


Fig. 12. Communication reduction using hierarchical approach and (b) Scalability (Throughput) of k-core on Graph500 input.

Pseudo-Diameter. Pseudo-Diameter is a graph metric used in network analysis to understand the structure of networks. It iteratively traverses the graph from a source, and selects the farthest vertex as the source for the next traversal, until the distance from a source to its farthest vertex can not be increased. Being a direct application of multiple traversals, improvements (Figure 13) mirror those in BFS.

4) Other Graphs: We also evaluate our approach on the extreme cases of Erdos-Renyi graphs, toroidal meshes and the real-world Twitter social-network graph. A 2D toroidal mesh is the worst-case scenario for our hierarchical approach, as the maximum out-degree of any vertex in the mesh is 4, and when partitioned correctly, the number of cut-edges is minimized. We evaluated our approach on this graph to show that even though we do not expect to improve performance, we add no overhead for such cases either. Figure 14 shows the performance of a BFS traversal of a toroidal mesh with 16 million vertices and 64 million edges, where the overhead of using hierarchies is less than 3%.

On the other hand, an Erdos-Renyi random graph [8] is the best-case for our approach. An Erdos-Renyi network tries to connect each vertex of a graph with every other vertex with a probability p ($p = 1$ leads to a complete graph). We generate an Erdos-Renyi network of 1 million vertices and 5.5 billion edges (a probability $p = .5\%$) to show the benefit of our hierarchical approach when the connectivity of the input graph is high. This is shown in Figure 14, where the hierarchical approach is $42.5\times$ faster.

To evaluate how the performance improvement varies with graph connectivity, we measured the performance improvement of our approach on a Watts-Strogatz random network for

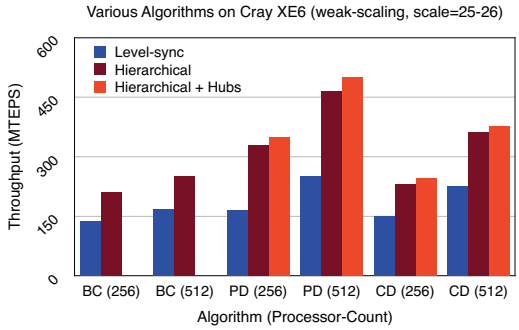


Fig. 13. Throughput of Betweenness Centrality (BC), Pseudo-Diameter (PD), and Community Detection (CD) on Graph500.

varying rewire probabilities. The Watts-Strogatz model [23] produces random networks with the small-world properties such as short average path lengths and a high degree of clustering. The model starts with a regular ring lattice with every vertex connected to its k nearest neighbors on either side, forming a ring-like structure. Thereafter, for each vertex, each of its edges is 'rewired' with a probability β, such that the target of the edge is selected with uniform probability from the remaining vertices, avoiding self-loops and duplicate edges. By varying β, one can generate graphs that are ring-like and do not exhibit small-world properties (for small values of β), to small-world networks with short path-lengths and high degree of clustering (as β approaches 1).

Figure 15 plots the speedup of our approach over the baseline as the rewire-probability is varied from 0 to 1 for various processor counts. While our approach has a 25% overhead

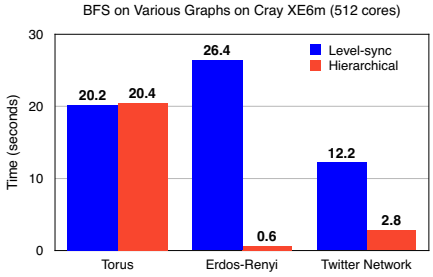


Fig. 14. Running times of BFS on various input graphs using flat and hierarchical approaches.

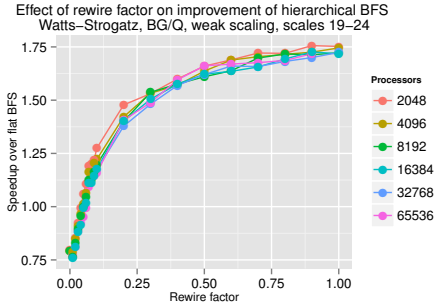


Fig. 15. Improvement of the hierarchical approach on BFS for varying rewiring-probability of a Watts-Strogatz graph.

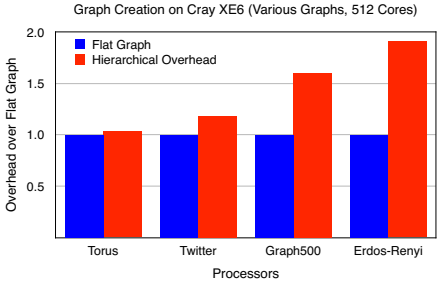


Fig. 16. Comparison of total construction times of flat and hierarchical graphs for various inputs.

on the BG/Q machine for rewire-factor $\leq 3\%$, it shows an improvement over the base-line for all other rewire-factors ($3\% - 100\%$). The initial overhead is due to the graph being well-partitioned with extremely few cut-edges. In this case, there is not much communication to reduce, and we just measure the overhead of going through the hierarchy. It is higher for the BG/Q machine than the Cray XE6 (3% for torus in Figure 14, and we measured 1.5% for Watts-Strogatz with 0% rewiring) as the processors are much slower and the network is extremely fast, so the effect is more pronounced. This can be eliminated by profiling the system using this example to compute the number of cross-edges beyond which the hierarchical approach should be used. As the algorithms remain the same, this can be done cheaply at runtime.

We can observe that the improvement over the base-line increases as the small-world behavior increases. Small-world graphs are harder to partition well and produce a large number of cross-partition edges, which result in heavy communication, thus limiting performance. Our approach alleviates this, improving performance. This is evident even in machines where the computation/communication ratio is low (i.e. better network, slower processors), such as the BG/Q. For machines with (comparatively) slower networks and faster processors, the improvement is even better, as observed in experiments on the Cray XE6 (Figure 8 and others).

Finally, we also evaluate our approach on the Twitter social network from 2010. This network has 65 million vertices and 1.2 billion edges, and has large hub vertices corresponding to popular users who have many followers. The presence of large hubs leads to poor scalability, which can be effectively addressed by using the hierarchical approach. As a result of communication reduction from 21 GB to 8.2 GB and a better load-balance, our approach provides a speedup of $4.36\times$ (Figure 14) over the flat approach.

D. Overhead of Hierarchy Creation

In order to run hierarchical algorithms, the input graph needs to be converted to its hierarchical representation on the machine. This is a one-time process following graph construction, after which multiple algorithms can take advantage of the hierarchy. Figure 16 compares the time to construct the flat graph to the time to construct the hierarchical graph. The time to generate the hierarchical graph includes the time to create the flat graph. Hierarchical graph creation overhead is proportional to the number of cross-processor edges in the graph. Therefore, for graphs with few cross-processor edges, such as a toroidal mesh, the hierarchy construction adds negligible overhead, while for graphs with a large number of cross-processor edges, such as a dense Erdos-Renyi graph, the hierarchy creation overhead is larger (Figure 16). Correspondigly, the benefit of the hierarchy is also significantly larger for the Erdos-Renyi graph vs. the torus (Figure 14). As an example, on 512 cores, the graph construction time for 67 million vertices and 2.1 billion edges was 8.5 seconds, while creating the hierarchy added 5.2 seconds to that time. A comparison showing the running time of a *single run* of various algorithms on hierarchical and flat graphs is shown in Figure 17, along with the time required to create the hierarchies, to show the overhead. For example, a PageRank algorithm using the traditional approach takes 68.7 seconds, while a PageRank using the hierarchical approach uses 39.7 seconds. To amortize the cost of hierarchy creation, BFS would need to be run 4 times, however, in many use-cases, such as betweenness centrality and pseudo-diameter, BFS is usually executed multiple times from different sources, allowing the hierarchy creation to be ammortized, even for a single execution of the algorithm (Figure 17). Other algorithms, such as community detection, PageRank, and k-core decomposition also observe significant overall benefits even in a *single* execution, including the cost of hierarchy creation, as shown in Figure 17. We note that the hierarchy is algorithm-agnostic and only depends on graph structure, and not the metadata (vertex/edge properties), and can therefore be used with different algorithms once created.

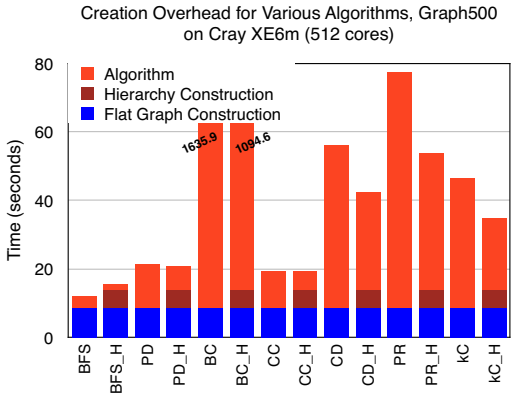


Fig. 17. Comparison of graph construction and algorithm execution times of flat and hierarchical graphs for Graph500 input.

VI. Related Work

To enable graph algorithms to scale, various different approaches have been adopted. Many implementations [1], [17], including our baseline implementation, aggregate messages being sent to a processor in order to reduce the number of messages sent. However, the *total number of bytes sent is not reduced*, but merely concatenated to form larger messages. Pregel [17] also allows users to specify a *combiner* that may be used to reduce multiple incoming messages to a given vertex to a single value per processor. However, this is not applicable to all algorithms and also not guaranteed to be executed in Pregel. Moreover, combining can only address reduction of fan-in, not high out-degree/fan-out vertices.

Another approach is to use ghost vertices to facilitate communication, which cache values of neighboring vertices stored on other processors. However, such vertices need to be coherent with their original vertices, and do not scale well in practice, due to storage and communication overhead of maintaining the ghosts, as shown in [9]. This approach is used by the Parallel Boost Graph Library [10], which we compare against in our experiments.

There have also been methods that propose a 2-dimensional partitioning of the graph and its edges to achieve better scalability by reducing communication cost [5]. To use such an approach, the algorithms need to be rewritten to account for a distributed edge-list and made aware of the underlying data-distribution, making this method difficult to use in practice. This method also does not reduce the message size, but does distribute the computation more evenly across partitions than 1-D partitioning. However, 2-D partitioning does not consider the locality of the target vertices of the edges being partitioned, and thus may in fact lead to more hops for the message to reach its destination, for example, one hop to get to the processor where the edge is stored, and a second hop to get to where the target-vertex of that edge is stored (as edges are not co-located in 2-D partitioning), generating extra communication. Our approach produces co-located edges in addition to lowering the computational imbalance, without user intervention.

Some approaches [9], [14], [19], [20], [24] have also proposed splitting hubs across multiple processors. PowerGraph [9], a version of GraphLab [16] designed for processing scale-free graphs uses the concept of *vertex-cuts/vertex mirroring* to split hub vertices across partitions. However, as mentioned in [9], due to maintaining the vertex-splits, their ghost vertices need to be kept synchronized across all machines it spans. This is addressed in PowerGraph by minimizing the number of processors across which the hubs are split to lower the synchronization costs. However, this limits the available parallelism for processing the hubs. Our approach is not limited by this, since we do not have ghost vertices and do not need to maintain state information across partitions. We compare our base-line with the latest available version of PowerGraph in Section V-A. Pregel+ [24] also uses mirroring, which reduces communication by partitioning the edges of high-degree (hub) vertices. However, this does not have much benefit in their published results, and performance actually degrades when the number of hubs is large. Our hierarchical technique, on the other hand, is applied to all vertices and consistently shows benefits, even when used for low-degree vertices.

On the other hand, [19], [20] create copies of the hubs on all processors, which may be wasteful if the hubs do not have adjacent vertices on all processors, and increases storage and communication overhead. In these cases, the hubs need to be coherent across the processors, which leads to extra communication, and a computational imbalance may still remain. This approach of splitting hubs also does not address non-hub vertices due to the overheads involved in splitting vertices and the storage requirements for replicating data. The algorithm-writers too need to be aware of the presence of such hub-representatives, and the communication between them, requiring the user to re-write their algorithm. We improve upon this work with our hubs approach which allows for local reduction of updates for hub vertices on processors where they are needed, while alleviating the need to keep them coherent. Further, it is transparent to the algorithm. This is in addition to our locality based hierarchy, and is explained in Section II-B.

The authors in [24] propose a request-response paradigm where a vertex can request data from another random vertex and it will be available in the next superstep. For algorithms that need this pattern, it can reduce the amount of back traffic by not sending duplicate values to the same processor. However, this is only applicable for a restricted set of algorithms explored in their paper, and can not be applied to widely used algorithms such as traversals, PageRank, etc. It also requires the algorithm to be modified to incorporate this paradigm for the cases where it is applicable. Their results show a modest improvement in performance for such cases.

Our hierarchical approach operates at a semantic level, allowing us to reduce the information sent over the network for both outgoing and incoming edges, thereby improving scalability. This is done for all vertices that have cross-processor edges, not only for hub vertices. Since information is not replicated, and due to using an asynchronous push-model, we do not incur significant storage or communication overheads endemic to previous approaches. Crucially, due to operating at a semantic level, the algorithm itself remains unaware of the hierarchy, allowing the reuse of existing fine-grained graph algorithms as-is with the hierarchy.

VII. Conclusion

We presented a technique for hierarchical algorithmic coarsening that reduces the number of bytes communicated in a distributed system, improving scalability and performance of graph algorithms, while allowing the reuse of existing fine-grained graph algorithms. We implemented this technique in the SGL framework, and evaluated it on two large-scale systems ($12,000+$ cores and $130,000+$ cores, Figures 7, 8) on various important graph analytics and graph mining algorithms with benchmark and real-world graphs, observing significant speedups of $2.5\times$ to $8\times$ at scale, without significant penalty in cases where performance was not improved.

VIII. Acknowledgments

We would like to thank Ioannis Papadopoulos for helping with our initial design and optimizations in our runtime-system. We would also like to thank our anonymous reviewers. This research is supported in part by NSF awards CCF 0702765, CNS-0551685, CCF-0833199, CCF-1439145, CCF-1423111, CCF-0830753, IIS-0917266, by DOE awards DE-AC02-06CH11357, DE-NA0002376, B575363, by Samsung, IBM, Intel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Dept. of Energy under Contract No. DE-AC02-05CH11231.

References

[1] The graph 500 list. http://www.graph500.org, 2013.

[2] J. I. Alvarez-hamelin, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Adv. in Neural Inf. Proc. Syst. 18*, pp. 41–50. MIT Press, 2006.

[3] U. Brandes. A faster algorithm for betweenness centrality. *J. of Math. Sociology*, pp. 163–177, 2001.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Sys.*, pp. 107–117, 1998.

[5] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. of Intl. Conf. High Perf. Comp., Networking, Storage and Anal.*, SC '11, pp. 1–12, 2011.

[6] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL: Standard Template Adaptive Parallel Library. In *Proc. Haifa Exp. Sys. Conf. (SYSTOR)*, pp. 1–10, 2010.

[7] N. Edmonds, J. Willcock, and A. Lumsdaine. Expressing graph algorithms using generalized active messages. In *Proc. Symp. on Princ. and Prac. of Par. Prog.*, PPoPP '13, pp. 289–290.

[8] P. Erdos, and A. Renyi. On Random Graphs. I In *Publ. Mathematicae*, pp. 290–297, 1959.

[9] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. *Proc. OSDI*, pp. 17–30, 2012.

[10] D. Gregor, and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Par. Object-Oriented Sci. Comp.*, 2005.

[11] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Graph Library. In *Lang. and Compilers for Par. Comp.*, LNCS, pp. 46–60. Springer, 2012.

[12] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. KLA: A new algorithmic paradigm for parallel graph computations. In *Proc. Intl. Conf. on Par. Arch. and Comp. Techniques*, PACT '14, pp. 27–38, Canada, 2014. ACM.

[13] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered and unordered algorithms for parallel breadth first search. In *Proc. Intl. Conf. on Par. Arch. and Comp. Techniques*, PACT '10, pp. 539–540, 2010.

[14] I. Hoque, and I. Gupta. LFGraph: Simple and Fast Distributed Graph Analytics. In *Proc. Conf. on Timely Results in O.S.*, pp. 1–17, 2013.

[15] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proc. IEEE International Conference on Data Mining*, ICDM '09, pp. 229–238. 2009

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed Graphlab: A framework for machine learning and data mining in the cloud. *VLDB*, pp. 716–727, 2012.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. Intl. Conf. on Mgmt. of data*, SIGMOD, pp. 135–146, 2010.

[18] L. Page, S. Brin, R. Motwani and T. Winograd. The PageRank Citation Ranking. 1998.

[19] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In SC '10, pp. 1–11, 2010.

[20] R. Pearce, M. Gokhale, and N. M. Amato. Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates. In *Proc. Intl. Conf. High Perf. Comp., Networking, Storage and Anal.*, SC '14.

[21] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. In *Proc. Symp. on Princ. and Prac. of Par. Prog.*, PPoPP, pp. 235–246, 2011.

[22] L. Valiant. Bridging model for parallel computation. *Comm. ACM*, pp. 103–111, 1990.

[23] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, pp. 440–442, 1998.

[24] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation *WWW* 2015.