The Impact of Dynamic Directories on Multicore Interconnects

MATT SCHUCHHARDT, Northwestern University
ABHISHEK DAS, Intel Corporation
NIKOS HARDAVELLAS, Northwestern University
GOKHAN MEMIK, Northwestern University
ALOK CHOUDHARY, Northwestern University

On-chip interconnects consume a significant portion of processor power. We observe that a large fraction of on-chip traffic originates not from actual data transfers, but from communication between the cores to maintain data coherence. Motivated by this, we evaluate one method of co-locating directories near their shared data, which utilizes existing virtual-to-physical address translation mechanisms to place directories near the data sharers. This eliminates a large fraction of on-chip interconnect traversals, thus reducing interconnect power and energy consumption by up to 37.3% (22.9% on average for scientific workloads, and 8.0% for Map-Reduce).

1. INTRODUCTION

To combat the increasing on-chip wire delays as the core counts and cache sizes grow, multicore architectures have become increasingly distributed: the last-level on-chip cache is divided into multiple cache slices, which are distributed across the die area along with the cores (e.g., Intel Xeon Phi, Tilera Tile-Gx) [1]. To facilitate data transfers and communication among the cores, such processors employ elaborate on-chip interconnection networks which consume 10-28% [2,3] of the overall chip power, stressing an already limited resource. As core counts continue to scale, the power consumption of the on-chip interconnect is expected to grow even higher.

To minimize the power consumption of on-chip interconnects, recent research proposes circuit-level techniques to improve the power efficiency of the link circuitry and the router microarchitecture [5], dynamic voltage scaling and power management [4], and thermal-aware routing [6]. However, prior works miss one crucial observation: a significant fraction of the on-chip interconnect traffic stems from packets that facilitate data coherence, rather than from packets that transfer data.

The coherence requirement is a consequence of performance optimizations for on-chip data. To allow faster data accesses, the distributed cache slices are typically treated as private caches to the nearby cores, forming tiles with a core and a cache slice in each tile [1]. These caches are kept coherent through a directory-based coherence protocol, where a distributed directory is address-interleaved among the tiles [1]. However, address interleaving is oblivious to the data access and sharing patterns; it is often the case that a cache block maps to a directory in a tile physically located far away from the accessing cores. To share a cache block, the sharing cores need to traverse the on-chip interconnect multiple times to communicate with the directory, instead of communicating directly between them. These unnecessary network traversals increase traffic and consume power and energy.

Tilera's TilePro64 [12], in recognition of this effect, implements a mechanism to reduce directory coherence traffic by allowing the software to designate a page's home node. This technique is similar to Dynamic Directories [11], which cooperate with the operating system to eliminate the need to place directory entries on a predetermined tile. While TilePro64 provides the placement mechanism, it does not advocate a specific placement policy and no evaluation of this technique appears in the literature.

In this paper we evaluate the impact of Dynamic Directories on the performance, power, and energy consumption of a multicore processor. We show its effectiveness under a simple directory placement policy, which places directory entries close to the most active sharers of the corresponding cache blocks, and compare it to Virtual Hierarchies [10], a previously proposed technique that also helps with directory placement, and the work by Cuesta *et al*. [13], which for simplicity we refer to as *Private Coherence Deactivation (PCD)* in the remainder of this paper. PCD deactivates coherence tracking for cache blocks within a private page.

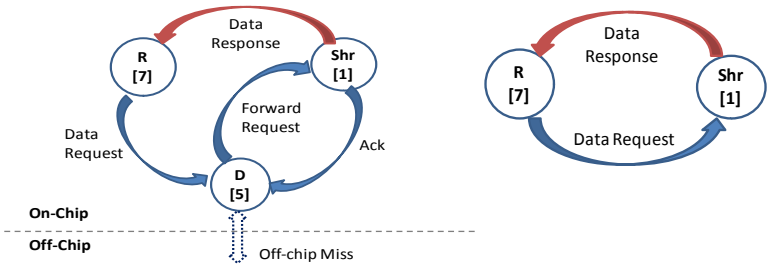
Compared to the baseline architecture, Dynamic Directories reduce the interconnect power and energy by up to 37.3% (22.9% on average for scientific workloads, and 8.0% for Map-Reduce),

with negligible performance impact and hardware overhead, and save 4x more interconnect energy than Virtual Hierarchies. Dynamic Directories exhibit the same gains as PCD on private pages.

2. ARCHITECTURE OVERVIEW

The baseline architecture is a tiled multicore modeled after [1], where each tile consists of a processing core, private split I/D first-level caches, a private second-level cache (L2), a slice of the distributed directory, and a router. The tiles are connected through a 2D-folded torus. Without loss of generality, and similarly to most relevant works, we consider a full-map distributed directory that is address-interleaved among the tiles.

Address interleaving does not require a lookup to extract the directory location; all nodes can independently calculate it by the cache block address. However, address-interleaved placement statically distributes the directories without regards to the location of the accessing cores, leading to unnecessary on-chip interconnect traversals. Figure 1-a shows a typical data sharing traffic pattern, where the directory is placed at an arbitrary node, oblivious to the location of the sharing cores. Ideally, the directory would be co-located with the sharer at Tile 1 (Figure 1-b), which would eliminate two unnecessary network messages and allow the sharers to communicate directly, rather than through an intermediate node. Such placement is the goal of Dynamic Directories. Note that even if the data are core-private, L2 misses are still routed to memory through the directory, thereby unnecessarily involving an intermediate node.



(a) (b)

Figure 1. Data sharing traffic patterns between cores. Diagram **(a)** shows the packet flow when data is requested by tile 7 for a block owned by tile 1, with its directory at tile 5. Diagram **(b)** shows the data sharing traffic when the directory is co-located with the data.

3. DYNAMIC DIRECTORIES

Dynamic Directories, similarly to Tilera's TilePro64, reduce unnecessary on-chip interconnect traffic by placing directory entries on tiles with cores that share the corresponding data. To achieve this, for every page, Dynamic Directories designate an owner tile of the directory entries for the blocks in that page, and store the owner ID in the page table. By utilizing the already existing virtual-to-physical address translation mechanism, Dynamic Directories propagate the directory location to all cores touching the page through the Translation Lookaside Buffer (TLB). There are two important aspects of this scheme: the classification of pages by the operating system (OS), and the directory placement and distribution among the cores.

3.1 OS Support and Directory Placement

To categorize pages and communicate their directory location to the cores, Dynamic Directories utilize the existing virtual-to-physical address translation mechanism. To access L2, a core translates the virtual address of the data to a physical address through the TLB. Upon a TLB miss (e.g., the first time a core accesses a page, or if the TLB entry has been evicted) the system locates the corresponding OS page table entry and loads the address translation into the TLB.

Dynamic Directories modify this process slightly. The first time a page is accessed by any of the cores, the page is declared private to that core (we refer to that core as the *first accessor*). This

information is stored in the page table. No directory entries need to be allocated for a private page, as there is no need to maintain coherence without sharers.

If another core accesses the page, that core will also miss in its TLB as it has no valid entry. Upon the TLB miss, the OS (or the hardware page walk mechanism) discovers that this page is already accessed by a core, and reclassifies the page as shared. At the same time, the first accessor core becomes the owner of the page's directory entries and the necessary directory entries are allocated in its tile. The directory location is recorded in the page table, and communicated to the core through the TLB fill. Thus, any subsequent accessor of the page is also notified of the directory location for the blocks in the page. This mechanism guarantees that the directory is co-located with one of the sharers of the page, and at the same time provides a simple mechanism to locate the directory entries.

When a page is reclassified from private to shared, the system needs to allocate directory entries for the cached blocks of that page. However, directory entries for private pages are not allocated, so when a page becomes shared, the system does not know which blocks of that page are cached, and what state the blocks are in. The only known fact is that any blocks cached thus far can only be at the first accessor's cache. The least complex solution is simply to flush all the blocks of that page from the first accessor's cache, similar to [1], eliminating the need to allocate directory entries for previously cached blocks. From that point on, directory entries can be allocated on demand at the first accessor's directory, as the first accessor remains the directory owner. Although simple, this solution increases the cache miss rate and increases the number of cache accesses to perform the cache flush, both of which come with potentially detrimental effects on performance, power, and energy consumption.

Alternatively, the system can conservatively allocate directory entries for all the blocks in the page, and declare the first accessor's cache as the owner of the blocks. This way, all requests for this page will be forwarded to the first accessor, who can easily resolve the state of the requested blocks locally (both the cached blocks and the directory entries reside in the same tile). In either method, the first accessor remains the owner of the directory entries, and the corresponding TLB entry at the first accessor core is invalidated to remove the stale private page state (the page is now shared). In this paper we implement the latter option: we conservatively allocate directory entries for all the blocks in the page, as this method has negligible power and performance impact, and the only downside is that it may allocate more directory entries than absolutely necessary during the relatively rare reclassification events.

The Dynamic Directories mechanism allows cache coherence at the granularity of individual cache blocks; it only places the corresponding directory entries at some tile. The placement is performed at the granularity of pages, i.e., the directory entries for all blocks within a page are placed at the same tile. Finer-grain placement is possible but incurs significant overhead. To support placement at granularities smaller than a page, the TLB and page table entries would need to store multiple directory owners (one per placement-grain), and each sub-section of the page would generate a separate TLB trap to extract the directory location for it. However, complex fine-grain techniques are not justified, as granularities smaller than a page provide negligible energy benefits (Figure 3). Nonetheless, for completeness, we provide the energy savings of such a hypothetical approach.

3.2 Thread Migration

When threads migrate, the corresponding directory entries could either (a) stay in the original tile and be accessed remotely by the migrating thread, (b) move along with the migrating thread, or (c) we could simply turn off Dynamic Directories.

The first option (leave the directory entries in place and let the migrating thread access them remotely) is the simplest choice. It does not require any new mechanisms or structures, and does not change any existing ones either. Having directories at a remote node and accessing them via the interconnect is similar to what the baseline system already does. The downside is that the directory placement may no longer be optimal, and the interconnect may suffer from hot spots as all the directory entries frequently accessed by this thread are concentrated to the same remote node. Thus, under this scheme thread migration becomes even more expensive than it already is, thereby raising the importance of affinity scheduling. This would be the best option if threads

migrate away from their core only for short periods of time and relatively infrequently, and quickly return back due to affinity scheduling.

The second choice (migrate the directory entries together with the thread) would preserve the affinity of the directory entries and the thread, at the cost of identifying the entries to migrate, performing the actual migration, and resolving any races occurring during the directory migration. Identifying the directory entries to migrate is easy when a core executes only a single thread: all entries owned by that tile should migrate. However, a core that executes multiple threads through multithreading or time-sharing needs to tag the directory entries with the process/thread IDs, so the system knows exactly which entries to move. Performing the migration would require TLB shootdowns, and bandwidth and energy consumption on the interconnect for the actual transfer. Races could easily appear, as requests could arrive while the directory migration is in flight, and the system would have to handle them successfully. Overall this is an expensive proposition, akin to traditional directory migration. It is unlikely that it would provide enough additional benefit to justify its implementation, unless threads migrate to a new core and stay there for a very long time, before migrating again.

It is important to note here that it is simple to turn off Dynamic Directories in pathological cases: one bit per page could indicate whether its directory entries are managed by Dynamic Directories or by traditional address interleaving. Thus, the third option is easy to implement. However, the moment Dynamic Directories for a page is turned off, the page would need to go through a process similar to reclassification: the cached blocks should be flushed from the caches, and the corresponding TLB entries shot down so they can be updated. Thus, while easy to implement, this solution also comes with a potentially significant overhead. To avoid excessive overhead, Dynamic Directories should remain switched off for the entire duration of high thread migration rates. In that case the overhead would be a one-time event amortized over billions of accesses, and the scheme would perform almost identically to the baseline. While this would prevent Dynamic Directories from providing any benefit over the baseline, at least it would guarantee it does no harm.

We did not observe any significant migration rates in our workloads, hence we did not separately evaluate these options. However, each solution is best suited for different conditions of migration frequency and length of stay. A sophisticated system could monitor these metrics and employ the best option each time.

3.3 Overhead

Dynamic Directories extend each TLB entry by $\log_2 N+1$ bits, where N is the number of tiles. The current Intel Core i7 (Sandy Bridge) has two-level TLBs, with 64 L1-TLB entries and 512 L2-TLB entries for a 4KB page size. For 16-core chip-multiprocessors (CMPs), Dynamic Directories add 4 bits for the directory owner and 1 bit for the shared/private state. This amounts to only 5.63 KB additional SRAM. We modeled the TLB with CACTI 6.5 [7] and found that the energy overhead is negligible (0.7% of the TLB read energy).

On the software side, Linux on Intel Core i7 typically uses 64-bit page table entries, in which bits 9-11 and 48-62 are unused. These extra bits can accommodate up to 128K cores and hence Dynamic Directories do not add any overhead in the page table. The only software overhead incurred is a few extra lines of kernel code to initialize the directory owner and state bits, and to orchestrate page reclassifications when necessary.

3.4 Mechanism Justification

Instead of utilizing unused page table entry bits to encode the owner ID for the directory entries of a page, Dynamic Directories could directly utilize the physical address bits. This could be achieved by designating $\log_2 N$ bits of the physical address to encode the owner tile ID. The selection of a physical frame for a virtual page would then be limited only to addresses that assign to these bits the correct tile ID.

However, this would couple the memory allocation with the directory placement. Limiting physical addresses to specific address ranges for each tile would cause memory fragmentation, degrade performance, and complicate other optimizations that pose conflicting address translation requests (e.g., page coloring for L1). Overloading the address bits within the cache index for directory placement would result in these bits being all the same for pages assigned to the same

tile, thereby utilizing only a subset of the available L2 cache sets. Similarly, utilizing higher address bits for directory placement may conflict with the bits used for DRAM bank, channel, or column selection, leading to the underutilization of the DRAM banks, the memory channels, or the row buffer. Moreover, if address bits are used for directory placement, each core in a N-core CMP will be able to allocate locally only 1/N of the overall physical pages, limiting the performance potential of Dynamic Directories if more pages are private to (or accessed mostly by) that core.

Dynamic Directories avoid all these problems by fully decoupling page allocation from directory placement.

4. EXPERIMENTAL RESULTS

We evaluate Dynamic Directories by simulating a 16-core tiled CMP running scientific workloads (a mixture of compute-intensive applications and computational kernels) and Map-Reduce workloads (Phoenix). We simulate the CMP on Flexus [8], and follow the SimFlex multiprocessor sampling methodology [9]. The full details of our simulation methodology, the power model, and details on the simulated architecture and workloads appear at [11]. We simulate the entire execution of the Map phase for Phoenix applications (which constitutes the majority of execution time) and three complete iterations for the scientific applications.

4.1 Directory Placement Policy

Ideally, in the absence of directory migration, the directory entries for a page would be placed at the tile that issues the most accesses to them. Identifying this tile, however, requires complex techniques. Fortunately, the first accessor of a page issues on average only 6% fewer accesses than the top accessor, and it is trivial to identify. Thereby, Dynamic Directories choose to allocate the directory entries of a page at its first accessor.

4.2 Distribution of Directory Entries Across Tiles

Dynamic Directories may skew the distribution of directory entries to tiles, in contrast to the uniform distribution of traditional address interleaving. If some tiles allocate vastly more directory entries than others, they may require a disproportionately large area for the directory, or cause traffic hotspots that degrade performance.

Figure 2 presents the distribution of directory entries across tiles for private and shared pages. The figure shows a band of 16 bars for each workload, with the height of each bar designating how many pages have their directories allocated at that tile. The red line indicates a hypothetical uniform distribution.

While the distribution is sometimes skewed, the imbalance could be minimized. First, private data are accessed by only one core, obviating the need for a directory. Thus, Dynamic Directories allocate directory entries only for shared pages, reducing the on-chip directory capacity requirements by 48% on average.

Second, while the directory entries for shared pages are mostly evenly distributed, oversubscribed tiles still exist (excluding Kmeans and Fmm, a tile gets at most 18% of the directories). Dynamic Directories have the flexibility to allocate the directories to nearby tiles or to another sharer when the first accessor is overloaded, thus spreading the load while still guaranteeing directory proximity to the accessing cores.

Finally, the uneven distribution of directory entries is not a direct indicator of increased traffic hotspots. The baseline may already exhibit imbalanced traffic, some pages are colder than others, and Dynamic Directories reduce the number of messages traversing the interconnect by 22.7% on average, easing traffic congestion. While Dynamic Directories on Kmeans, Fmm, and Dsmc exacerbate already existing traffic imbalances, these hotspots have a negligible performance impact, as the applications spend only a small fraction of execution time on the distributed L2 cache. In the remaining applications, a tile may receive on average 8% more directory accesses from remote cores compared to the baseline, but these imbalances are relatively small and do not impact the performance (Figure 5). It is important to note here that in pathological cases it is simple to turn off Dynamic Directories as explained in Section 3.2.

For completeness, we evaluated Kmeans and Fmm with a modified Dynamic Directories scheme, where the directories for the shared pages are address-interleaved at block granularity, similar to Reactive NUCA [1]. This directory placement removes the hotspots that Kmeans and Fmm experience.

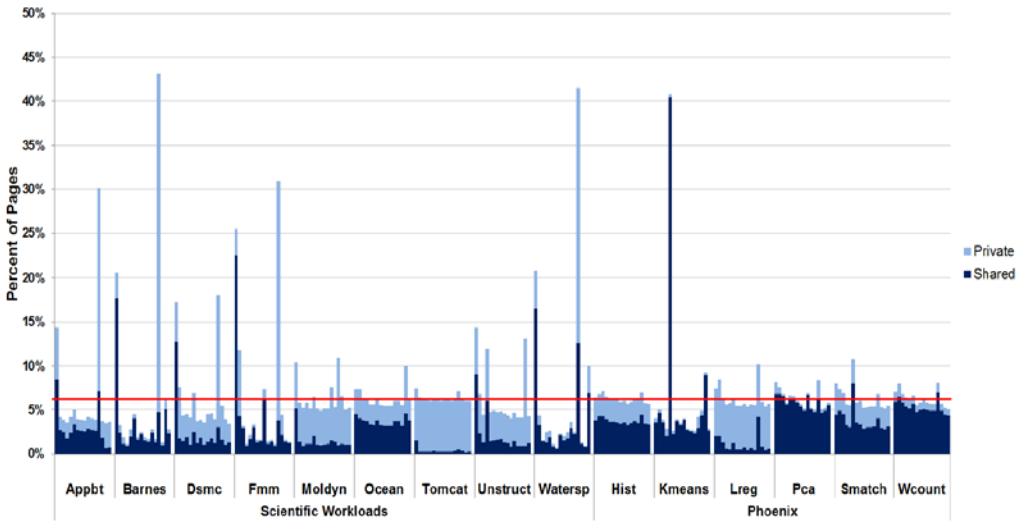


Figure 2. Distribution of directory entries across tiles. Flatter benchmarks indicate that the directory entries are relatively evenly distributed across the 16 tiles.

4.3 Energy Savings

We evaluate the impact of Dynamic Directories on energy and compare against three schemes: the baseline architecture, Virtual Hierarchies (VH) [10], and Private Coherence Deactivation (PCD) by Cuesta *et al.* [13]. Virtual Hierarchies implement a coherence protocol where cache blocks are dynamically assigned home tiles. An access that misses on the local tile finds the corresponding directory location by indexing a table using the bits of the physical address above the block offset. Thus, all pages with the same $\log_2 N$ bits in the physical address have their directory entries at the same tile. We implement a "perfect" VH where the tile with the most accesses to a memory region becomes the home tile for the entire region. PCD deactivates coherence tracking for cache blocks within a private page, while the directories for the shared pages are placed according to the baseline coherence scheme (in our case, at the address-interleaved location at block granularity).

Figure 3 presents the on-chip interconnect energy consumption of Dynamic Directories for cache-block and page granularity (DynDir-BLK and DynDir-8K, respectively), Virtual Hierarchies, and PCD, normalized to the energy consumption of the baseline architecture. Dynamic Directories reduce the network energy on average by 21.2% (block granularity) and 16.9% (page granularity). VH saves only 3.9% energy on average, as it allocates directory entries for an entire memory region, rather than individual pages.

PCD shows the same gains as DynDir-8K for private pages, as both coherence deactivation and directory homing eliminate interconnect traversals to the directory for private data. For shared data, PCD follows the baseline coherence and address-interleaves the directories at block granularity, placing the directories at arbitrary nodes. DynDir-8K places the directories together with the sharers, but placement at the page granularity increases false sharing. Overall, DynDir-8K attains higher energy savings for shared data in some applications (e.g., dsmc), while for others block-interleaving the directories of shared data is better (e.g., ocean).

To isolate the impact of directory homing from the adverse effects of page-grain directory placement, we evaluated a scheme that places the directories of a shared page together at the address-interleave page-grain location (not shown due to space constraints). We found that homing provides significant benefits (e.g., 35% energy savings in ocean, 29% in appbt, 27% in moldyn, and 9% in fmm). These benefits, however, are balanced out by the false sharing introduced by page-grain directory placement. As a result, the block-level interleaving of PCD and the directory-page-homing of DynDir-8K balance each other out across applications, resulting in average energy savings within 4% of each other.

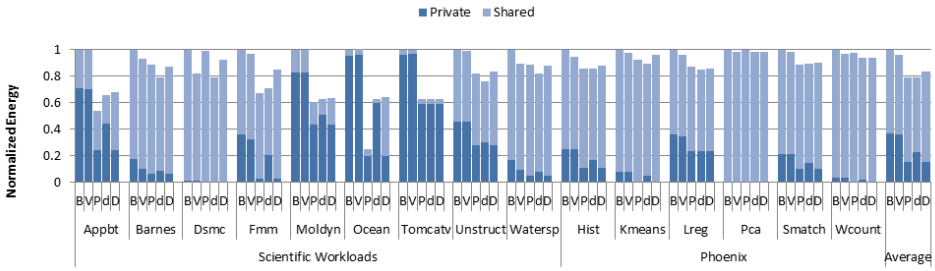


Figure 3. Network energy consumption normalized to the baseline architecture (B). Data for Dynamic Directories is analyzed at 64-byte cache-block (d) and 8K-page (D) granularities, and compared to Virtual Hierarchies (V), and Private Coherence Decoupling (P). The energy consumption of requests for private and shared data (classified at page granularity) is shown separately.

Comparing the two classes of workloads, we observe that the scientific applications attain higher energy savings (22.9% on average, and 37.3% maximum) compared to Phoenix (8.0% on average). Phoenix applications exhibit a higher fraction of shared data accesses, rendering Dynamic Directories less useful. The energy savings for all schemes are largely achieved through a reduction of control messages. On average, Dynamic Directories eliminate 22.7% of the control messages on the on-chip interconnect (Figure 4), while PCD eliminates 25.7% and VH eliminates 3.1%.

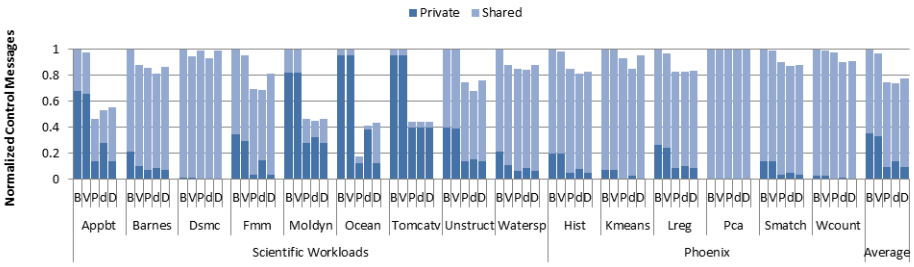


Figure 4. Number of interconnect control messages (i.e., all messages except data replies) normalized to the baseline architecture (B). Data for Dynamic Directories is analyzed at 64-byte cache-block (d) and 8K-page (D) granularities, and compared to Virtual Hierarchies (V), and Private Coherence Decoupling (P). The messages for private and shared data (classified at page granularity) are shown separately. The non-global sharing characteristics of the scientific workloads lead to generally higher savings.

4.4 Performance Impact

Figure 5 shows the overall speedup of Dynamic Directories compared to the baseline architecture. Dynamic Directories slightly increase performance in 7 out of 15 applications, and decrease performance in 2. Dynamic Directories improve performance by up to 7% (Ocean), and by 1.4% on average, while the maximum performance slowdown is 1.3% (PCA). Performance improves because (a) Dynamic Directories reduce the number of network packets, which eliminates congestion and hence reduces the effective interconnect latency, and (b) data transfers (on-chip and off-chip) are faster because many accesses to remote directories are eliminated. The performance improvement is limited because the working set of most applications is large, thus Dynamic Directories' savings are realized mostly by off-chip memory accesses. As the off-chip

memory access latency is already large, saving a few cycles does not improve the performance considerably.

We attribute the slowdown exhibited by PCA and Wcount to the page granularity in which Dynamic Directories assign directories, as this assignment can cause contention and hotspots. Especially for universally-shared pages, it is likely that blocks are accessed by different cores in nearly-consecutive cycles, causing contention in the directory tile, and increasing the directory's response time. On average, we observe that the positive and negative forces cancel each other out, leading to only a small performance improvement (1.4% on average).

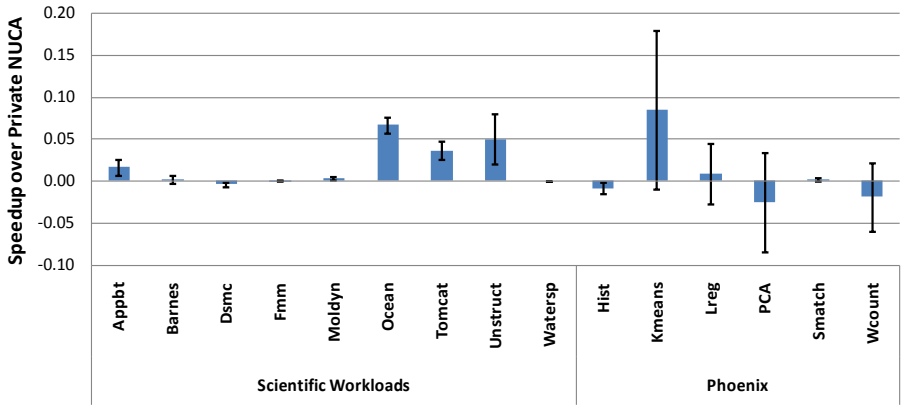


Figure 5. Speedup of Dynamic Directories over the baseline architecture. Performance is slightly improved on average, with a 1.4% average speedup.

5. CONCLUSION

A large fraction of the on-chip interconnect traffic stems from placing directory entries on chip without regards to the data access and sharing patterns. Future architectures should consider the implications of allocating directory entries close to the cores accessing the corresponding data. The primary impact of this is a reduction of the on-chip interconnect power and energy consumption; in this evaluation's case, the power and energy consumption is reduced by up to 37.3% (22.9% on average for the scientific workloads, and 8.0% for Map-Reduce) with negligible hardware overhead (5.63KB additional SRAM for a 16-core chip, and 0.7% increase in the TLB energy) and without any adverse performance impact. As the importance of on-chip interconnects is expected to rise with future process technologies, Dynamic Directories will scale and provide even higher power savings.

6. REFERENCES

[1] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Near-optimal cache block placement with reactive nonuniform cache architectures," *IEEE MICRO*, 30(1), 2010.

[2] E. Totoni, B. Behzad, S. Ghike, and J. Torrellas, "Comparing the power and performance of Intel's SCC to state-of-the-art CPUs and GPUs," *IEEE International Symposium on Performance Analysis of Systems & Software*, 2012.

[3] S. Borkar, "The exascale challenge," *20th International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[4] L. Shang, L.-S. Peh, and N.K. Jha, "Dynamic voltage scaling with links for power optimization of interconnection networks," *9th International Symposium on* High Performance Computer Architecture, 2003.

[5] H. Wang, L.-S. Peh, and S. Malik, "Power-driven design of router microarchitectures in on-chip networks," *36th Annual International Symposium on Microarchitecture*, 2003.

[6] L. Shang, L.-S. Peh, A. Kumar, and N.K. Jha, "Thermal modeling, characterization and management of on-chip networks," *37th Annual International Symposium on Microarchitecture, 2004*.

[7] N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi, "CACTI 6.0: A tool to model large caches," *Research Report hpl-2009-85, HP Laboratories*, 2009.

[8] N. Hardavellas, S. Somogyi, T.F. Wenisch, R.E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J.C. Hoe, and A. Nowatzyk, "Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *ACM SIGMETRICS Performance Evaluation Review*, 31(4), 2004.

[9] T.F. Wenisch, R.E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J.C. Hoe, "SimFlex: statistical sampling of computer system simulation," *IEEE MICRO*, 26(4), 2006.

[10] M.R. Marty and M.D. Hill, "Virtual hierarchies to support server consolidation," *34th Annual International Symposium on Computer Architecture*, 2007.

[11] A. Das, M. Schuchhardt, N. Hardavellas, G. Memik, and A. Choudhary, "Dynamic directories: a mechanism for reducing on-chip interconnect power in multicores," *Design, Automation, and Test in Europe*, 2012.

[12] Tilera, Tile Processor User Architecture Manual,
http://www.tilera.com/scm/docs/UG101-User-Architecture-Reference.pdf (accessed June 7, 2013).

[13] 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," *38th International Symposium on Computer Architecture*, 2011.