ReDirect: Reconfigurable Directories for Multicore Architectures

GEORGE PATSILARAS, QUALCOMM TECHNOLOGIES, INC.

JAMES TUCK, NORTH CAROLINA STATE UNIVERSITY

As we enter the dark silicon era, architects should not envision designs in which every transistor remains turned on permanently but rather ones in which portions of the chip are judiciously turned on/off depending on the characteristics of a workload. At the same time, due to the increasing cost per transistor, architects should also consider new ways to re-purpose transistors to increase their architectural value.

In this work, we consider the design of directory-based cache coherence in light of the dark silicon era and the need to re-purpose transistors. We point out that directories are not needed all of the time, and we argue that directories (and coherence) should be off unless it is actually needed for correctness. In our design, directories will be disabled and powered off for workloads with no sharing. Then only when parallel workloads need cache coherence will directories be enabled in proportion to the sharing that is present.

At the same time, we exploit the structural similarities of directories and cache. If a directory is idle, then we reconfigure it to be used as extra capacity in the last-level cache. Since our novel approach can keep most directories off, we are free to select sparse overprovisioned directory designs that are reconfigurable to large amounts of cache that can significantly boost performance when the directory is idle.

We call these combined features *Reconfigured Dark Directories*, since directories are usually dark (off) and can be reconfigured. Our results for Reconfigurable Dark Directories running SPEC 2006 applications show a performance benefit, on average, of 17% for an 8× overprovisioned fully mapped directory on a 64-tile system under low system concurrency (10% under heavy concurrency), or a 29% average speedup for a 2× overprovisioned directory on 256-tile system (10% under heavy concurrency) to systems with a conventional sparse directory design using the same overprovisioning factor.

CCS Concepts: • **Computer systems organization** → **Multicore architectures; Reconfigurable computing**;

Additional Key Words and Phrases: Multicore, reconfigurable

ACM Reference format:

George Patsilaras and James Tuck. 2017. ReDirect: Reconfigurable Directories for Multicore Architectures. *ACM Trans. Archit. Code Optim.* 14, 4, Article 50 (December 2017), 23 pages.
<https://doi.org/10.1145/3162015>

1 INTRODUCTION

The end of Dennard's Scaling poses significant problems for future multicore designs. Designers can no longer afford to add more transistors and keep them powered on at the same clock frequency

Authors' addresses: G. Patsilaras, 10105 Pacific Heights Blvd. San Diego, CA 92121; email: georgep@qti.qualcomm.com; J. Tuck, 890 Oval Drive, Campus Box 7911, Raleigh, NC 27695-7911; email: jtuck@ncsu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1544-3566/2017/12-ART50 \$15.00

<https://doi.org/10.1145/3162015>

while retaining a fixed power density. In response to this trend, some have suggested that a large portion of future chips will have to be powered off (Esmailzadeh et al. 2011) at any given time. Such inactive regions of the chip are called dark silicon. At the same time, increasing chip area is not preferred when considering the increase in cost due to low yields as nodes scale down. As a result, we believe that reconfigurable dark silicon is the solution to this problem: using transistors that can be turned on or off or used for more than one purpose.

To scale cache coherence protocols to a large numbers of cores, the trend is toward point-to-point protocols typically supported by directories.¹ To maintain coherence, a cache miss must find its home directory, determine the sharing status of the line, and perform the necessary coherence operations. This results in operations over the interconnection network and accesses to structures, including directories. Furthermore, directories remain powered up to service requests. Even though they are idle, they consume static power. As the number of sharers increases, this power can be significant when compared to a processor with a single tile composed of a core and cache hierarchy.

Directories are present in multicores to ensure correctness. The additional power cost of these operations is necessary only when shared data are present in multithreaded workloads. However, for single or lightly threaded workloads that do not have shared data, this additional power does not serve any effective purpose. Furthermore, as multicores scale to larger numbers of cores/tiles, the cost of performing coherence operations will increase if we stay the course with conventional designs. This cost could be in terms of area due to tracking increasing numbers of potential sharers or power due to the increasing complexity of structures performing coherence. Traditionally always on, coherence reduces the available power budget and decreases the effectiveness of techniques like TurboBoost (Turbo 2012) that aim to capitalize on lower-than-expected power utilization to boost frequency and performance.

Recent directory-based coherence designs do not significantly mitigate this problem. Even with aggressive assumptions from the latest research on directory design (Cuesta et al. 2011; Ferdman et al. 2011; Yang et al. 1992; Wallach 1992; Sanchez and Christos 2012; Zebchuk et al. 2009; Lotfi-Kamran et al. 2010; Zebchuk et al. 2009; Gupta et al. 1990), a directory's size relative to a core/tile will not reduce as the number of cores scale up. Hence, the chip real-estate devoted to directories will increase² and cost more static power as systems scale. At the same time, such costs are unnecessary much of the time, since coherence is only needed when sharing is frequent for correctness. Finally, the impact of serial work that cannot be parallelized will become an ever-greater bottleneck. Spending power needlessly will limit the peak performance that can be attained through techniques like DVFS or TurboBoost.

To illustrate our case, Table 1 shows the average power of each component as calculated using our methodology and data in Section 5. For the directories, we assume they are idle and only consume standby (static) power. Looking at a tile's average power the fraction consumed by one directory is small. Directories are always powered on, so in a 64- and 256-core multicore when all directories are aggregated ($\times 64$ or $\times 256$), they consume a large fraction of power compared to a single tile. Sixty-four directories consume the equivalent power of two active tiles, and 256 directories consume the equivalent power of eight tiles on a 256-core processor. These calculations assume that as technology scaling occurs, the relative cost of a directory remains constant. From this, we can see that given even the most aggressive techniques not yet implemented (Martin et al.

¹Even if clustering is initially applied among four or eight neighboring cores using snoopy coherence, some form of directory is ultimately required for scalability.

²Number of Tiles $\times$ Size of a Directory/Tile.

Table 1. Average Power of System Components at 22nm

Component	Power (W)	Power Norm. to 1 Tile
1 Core	0.259	0.46
1 Directory (8×)	0.018 <i>standby</i>	0.03
Cache Hierarchy	0.229	0.41
1 Tile	0.563 W	1.00
64 × 1 Directory	1.15 <i>standby</i>	2.04
256 × 1 Directory	4.6 <i>standby</i>	8.17

Directories are overprovisioned by 8×.

2012), there will soon be a new power wall tradeoff between core counts, due to directories, and operating frequency.

In this article, we argue for a different philosophy and make some headway in moving in that direction. Instead of designing coherence in a way that requires the full chip-wide coherence mechanism to be active all the time, we propose designing coherence in a simple but reconfigurable way that should be off unless it is needed. This way, the design is to function incrementally and proportionally to the number of cores and to the amount of sharing. When directories are needed, there should be no new overheads to avoid degrading the performance of highly threaded workloads (Pusukuri et al. 2011). Finally, to improve the value of transistors, we reconfigure directories that are not used to act as caches. We call this approach *Reconfigurable Dark Directories*, because we prefer for directories to be dark (disabled) by default and only enabled when needed for performance (single-threaded workloads) or correctness (parallel workloads).

Our approach offers three interesting tradeoffs over conventional directory-based coherence. First, if most of the directories are off, then we can run with lower average power, since costly leakage power can be shut off, and dynamic power from accesses to directories and coherence requests on the interconnect are eliminated. Second, we can trade off the peak/average power savings to run at a higher sustained frequency within the system's power envelope. Finally, we argue that directories should be re-usable for other purposes when not needed, which is a very novel tradeoff we have discovered. This increases the value of the transistors used as a directory. Given the similarity between directories and cache, it is reasonable to implement a reconfigurable directory that serves as L2 cache. Even though this may result in a larger directory per tile, most directories will be either disabled or used as cache, leading to better overall efficiency and higher performance in the common case.

The rest of this article proceeds as follows: Section 2 provides some related background. Section 3 discusses our motivation. Section 4 describes the architecture. Section 5 describes our experimental methodology, and Sections 6 and 7 provide the experimental setup and the full evaluation in the context of scheme. Section 8 discusses related work and Section 9 concludes.

2 BACKGROUND AND MOTIVATION

2.1 Base Design

We now describe the basic architecture we assume in this article, which is based on the Xeon Phi (formally known as Knights corner) processor from Intel (Goodwins 2010). Figure 1 shows a tile-based multicore design with private cache hierarchy. Each tile includes a core, an L1 cache tightly coupled to its core, a private L2 cache, a fraction of the distributed directory, and a router. The directory tracks state and sharer information for a set of cache blocks. All cache blocks are assumed to map to a given tile based on their physical page address. To scale this design, the tile

Fig. 1. Private cache hierarchy.

Table 2. Area Scaling as Number of Tiles Increases

Name	Area Eq.
Sparse	$OverP \times Lines \times (Sharers + extra)$
Coarse	$OverP \times Lines \times (4 \times \log_2(Sharers) + extra)$
Hier (Yang et al. 1992)	$Lines \times \sqrt{Sharers}$

may be replicated as many times as possible within an area budget in a given process technology. We assume this architecture due to its relatively simple scaling for the rest of the article.

2.2 Area Overheads

As we scale our tile-based design, the area of a single tile is not perfectly constant. The size of the directory will scale up based on the number of sharers and the number of lines being tracked by the directory. We can think of these two dimensions as the height and width of the directory. The width is influenced by the number of sharers, which is usually the same as the number of cores. The height scales in proportion to the number of cache lines that can be shared. Directories for private last level caches are often *overprovisioned* using factors between $2\times$ or $8\times$ (Sanchez and Christos 2012; Ferdman et al. 2011) the number of entries in the private cache, and hence an increase in cache capacity can significantly increase directory height.

Table 2 shows the scaling trends for three different directory designs. The three shown in the table are compatible with private caches in our tiled architecture. The Sparse uses a full bit-vector to fully encode sharers. The coarse-grained design represents a fixed number of sharers as an ID with $\log_2(sharers)$ bits. The last design in the table, *Hier* (a.k.a. hierarchical), is not compatible with a private last-level cache, but it is an interesting alternative for a shared cache architecture.

Whether we increase the size of the cache or the number of cores, we can be certain that the directory cost will grow in response (even if very slowly). Given that the design of directories remains an important research topic (Sanchez and Christos 2012; Zebchuk et al. 2009; Yang et al. 1992), the rate at which it will grow in terms of height and width is still undecided.

Fig. 2. Total chip-wide directory static power normalized to a single tile as the number of tiles increases.

2.3 Directory Static Power

We consider the impact of directory design compared to an application with fixed resource needs. To simplify the comparison, we use a single tile's power and area as the comparison point, since this represents a single-threaded program.

Figure 2 shows a scaling curve for total directory static power for directory designs that occupy fixed fractions of the L2 cache size, namely 5%, 10%, 20%, 30%, 40%, and 50%. Note that a directory that is a fixed fraction of the L2 data array even as the number of tiles increase is highly optimistic, and we do not know of any design that has this property. But making this assumption makes the graph easier to understand and provides valuable insights.

We see that the relative static power cost of maintaining coherence will grow rapidly as we scale the number of tiles even for relatively efficient implementations when we consider lightly threaded and single-threaded applications. We drew a horizontal line to represent the static power cost of a 2 $\times$ overprovisioned Sparse directory for a 64-tile chip.

Note that directories with smaller fixed areas eventually reach the same overhead; it just takes a few more generations to get there. For that reason, we choose to focus on sparse directories. Sparse directories have been successfully designed and deployed by industry, they are easy to scale up, and they are the simplest to reconfigure and have the lowest power per access (Ferdman et al. 2011).

2.3.1 Directory Static Power over 1 Tile. Now we consider static power overhead when compared to 1 tile. Figure 3 shows the percentage of static power for each component per tile as we scale the number of tiles assuming only one core is active. We assume a directory overprovisioned at a rate of 2 $\times$. Note that for each tile count on the x-axis, the total static power is normalized to 100% of the static power for a tile in that processor. Overall, since the size of the directory increases as we scale the number of tiles, the percentage of static power devoted to the directory increases—and quite rapidly. At 16 tiles, the overhead is roughly 20%; at 64 cores, the overhead is just shy of 30%. This is a large fraction of the peak static power. Analogous to the plot in the previous section, we see that the static power of the directory is greater than the static power of the L2 with 256 tiles or more. Clearly, this is already a big concern for thread counts equivalent to current multicore processors and will grow quickly to be a major chip-level overhead. On top of that, as we scale down to smaller nodes, the static power of directories will become more significant compared to the dynamic power (Kim et al. 2003).

Fig. 3. Static power component breakdown per tiles as a function of the number of tiles increases.

2.4 Dark Silicon's Impact on Directories

Due to the end of Dennard's Scaling, we can no longer afford to increase the area of chips and keep everything powered on. As the directory size and area increase in chips, the power budget available for cores (and performance) will decrease. This could result in cores running at lower frequencies to reduce their power consumption or not all cores being active at a time. The second option is what is known today as dark silicon (Esmailzadeh et al. 2011). Dark Silicon, however, results in transistors that are underutilized. As technology nodes are decreasing, chip-to-chip variation and low yields are increasing the cost of transistors.

As the cost of transistors increases, some might say that dark silicon is too expensive to justify. In response, reconfigurable designs that increase the value of a transistor may become favored. To reuse transistors for multiple purposes, each use of the transistor needs to be important but complementary in terms of when it is powered on. The design should allow both to excel but leverage combined principles to keep reconfiguration simple.

One possibility is to reconfigure directories as caches. Directories are similar to caches in that they both use tags and have a data array (to track actual data or sharers). Furthermore, we can choose directory designs that make their dimensions (height and width) closely match the design of the cache. For example, on small tile counts, this may mean overprovisioning by a larger-than-normal factor. For large tile counts, use smaller overprovisioning ratios or encoded entries or novel designs that compress sharing information (i.e. move from sparse to coarse directories). In the next section, we present our design and how it significantly mitigates this trend.

3 IDEA: RECONFIGURABLE DARK DIRECTORIES

To prevent the wasteful scaling trend demonstrated in Section 2, we propose a new way of designing directories called Reconfigurable Dark Directories (ReDirect). Our design is based on two key objectives: (1) Keep directories off when they are not needed, because sharing is not present, and (2) re-use directories as cache when they are idle. We consider each objective in turn.

3.1 Keep Directories Off (Dark)

In conventional designs, directories are always enabled. If we assume there is a way to track which directories are unused, then they can be powered off. The static and dynamic power saved by

Fig. 4. Active core effects on frequency on 256-tile design with 125W peak power.

disabling them can be used to either reduce system power or boost frequency. Boosting frequency may seem like an odd choice, but, in some cases, we might be more interested in performance when applications are single threaded with high Instruction Level Parallelism (IPC). Our goal is to have the ability to spend resources most effectively, whether that translates into saving power or gaining performance. Certainly, boosting the frequency of serial or lightly threaded applications is more effective than wasting the power on directories.

Figure 4 evaluates the benefits of disabling directories and re-using their power to boost chip-wide frequency. It considers what would happen on a 256-tile design with a fixed 125W power budget if inactive directories are completely disabled. Along the x -axis, we consider varying degrees of threaded-ness from 1 active thread to 256 active threads. The *No Directory* experiment computes peak frequency under the assumption that no directory is powered on and the number of powered-on tiles is the same as the number of active threads; as a result, disabled tiles contribute to frequency scaling, too. The *256-Tile Sparse 2x* experiment assumes that all directories are always enabled for a 256-tile-based system. Tiles are powered off in the same way as *No Directory*. We use a $2\times$ overprovisioned directory in this experiment, which is consistent with industry implementations. All the static power savings from disabling the directories or tiles are devoted to frequency scaling. The y -axis shows the max frequency possible for a workload. Naturally, the advantage of frequency scaling increases under a given power budget (125W) when applied to fewer threads, since more of the chip is inactive.

For single or lightly threaded workloads, it is clear that shutting off all directories and scaling frequency can significant benefits. For a single active thread, the *No Directory* design can reach a frequency of 1.66GHz compared to a design with enabled directories, which can run at 1.3GHz, a frequency increase of 28% which is a significant performance boost for high-IPC applications. As the active cores increase, we have a reduction in boosted frequency.

Disabling directories can lead to a large frequency boost even when a directory is only a small fraction of the tile (Table 1). After all, we are turning off all 256 directories, and the cost quickly adds up. Furthermore, it is worth noting that this frequency boost can be used for all single-threaded applications, whereas a directory cannot.

3.1.1 Architectural Approach to Keep Directories Off. In our approach, we wish to keep directories off by default rather than powered on. This change in approach has several implications.

I. Keep as few directories on as possible. Much of the data in a system are private and do not need coherence for program correctness. Tracking private data in the directory would undercut the benefits of our approach. Hence, we adopt as a baseline a design that is able to detect private data and avoid maintaining any directory state for it.

To make this work effectively, we need a design that recognizes the difference between private data accessed by one thread and shared data accessed by multiple threads. Cuesta et al. (2011) proposed to track private and shared data using information in the Page Table. If a page is only accessed by one thread, then the TLB entry informs the cache that the block does not need to be kept coherent, and no requests will be sent to the directory on that cache block's behalf. If sharing is detected, then the page is upgraded and all of its data will be managed by the directory.

II. Since directories are usually off, we need a way to turn them on. The approach used by Cuesta et al. (2011) provides a keen opportunity for enabling directories. At the moment in time when sharing is detected for a page, a directory must be assigned to the page. At this moment, if the directory is powered off, then it should be powered up. We add new architectural support to turn a directory on precisely when it is needed.

III. Directories should be turned off when not in use. Turning off directories is challenging. One option is to do it lazily when all pages mapped to that directory are reclaimed by the OS, but this may lead to inefficient use, because they are often shared by multiple applications and the OS, and only when all pages mapped to the directory are discarded can they be disabled. Alternatively, they could be disabled eagerly by flushing pages from the cache and/or re-mapping them into a fewer number of directories.

We choose a lazy approach in this work, not because we know it is better, but because a thorough evaluation of eager versus lazy techniques will not fit in this article along with the other discussion.³ The lazy approach uses a flow-in/flow-out approach in the OS to determine when a directory can be disabled. Ideally, when an application finishes, all the pages at one or more directories will be reclaimed and the directory will become idle again. However, if other applications are also using the directory, then such use will be detected, and it will remain active.

IV. Operating system carve-out. The operating system's memory has to be treated differently than application memory. Modern operating systems leverage many shared data structures, and assuming everything is private, or can be made private, is not realistic. As such, we understand that some directories will always be on. To limit which directories are on, we assume a memory carve-out that the operating system would use for shared data structures.

3.2 Reuse Directories as Cache

To keep the design principle simple, we will adopt a directory size with the same raw data capacity (in number of bits) as the L2 cache. In essence, this allows us to double the size of the L2 when the directory is re-used as cache. This may seem excessive for small tile counts, but by making these transistors dark, while at the same time working as directory or cache (multipurpose), we have increased its value to the system. Note that we can still have designs with smaller cache area increments, such as only one way.

Even though the raw bit size for the data array is the same, the relationship between directory capacity and cache capacity is different, because a block of directory holds sharers that varies with tile count and directory design. We can calculate the resulting difference in capacity as the

³Careful comparison of both approaches is needed in future work; however, our infrastructure limits our ability to perform the necessary simulations.

Fig. 5. Directory overprovisioning as tile count scales up.

overprovisioning factor for a sparse directory as follows:

$$Overprovision = \frac{CacheBlockSize}{(NumberOfSharers/8)}. \quad (1)$$

Figure 5 shows overprovisioning factors versus number of sharers (x-axis) with a cache block size of 64 bytes. This calculation assumes a sparse (fully mapped) directory. Worth noting is that few sharers result in larger overprovisions, but with many sharers (≥ 64) the overprovision is $8\times$ or less. While the directory for 64 cores is somewhat oversized at a factor of 8, it offers other benefits. A larger directory could mean fewer directory conflicts. Therefore, fewer directories are needed to be on at a time to support a given amount of shared data, thereby allowing more to be re-used as cache. Congestion is not a problem at the directory level as we have seen from our own experiments, not included for space, primarily due to the cores being in-order.

Even though the 64-tile directory seems oversized, the $0.25\times$ provisioning at 1,024 cores is quite small and would require innovative designs that optimize for directory size (Ferdman et al. 2011; Yang et al. 1992).

In summary, our organization is a directory that is reconfigurable in such a way to double cache capacity per node. For small sharer (tile) counts, sparse directories will work well. We implement a reconfigurable cache, in the same vein as prior work (Albonesi 1999; Ranganathan et al. 2000). We discuss the details of the implementation in Section 4.4.

3.3 Summary

These two techniques integrate naturally and allow re-allocation of directory resources to boost single thread performance. They also avoid the wasteful allocation of power to resources that do not help performance. Disabled directories can either be inactive to boost frequency or re-purposed as additional cache. Furthermore, these policies apply to individual threads whether they are in single-threaded or multithreaded programs. However, the performance benefits from frequency scaling and additional cache will likely benefit single-threaded programs the most, since more directories will be idle.

4 IMPLEMENTATION

4.1 Base System

We adopt as our baseline the architecture proposed by Cuesta et al. (2011) for deactivating coherence on private lines. Their mechanism determines, at runtime, which pages are shared by

Fig. 6. System support for Reconfigurable Dark Directories.

multiple cores and which ones are private. We will briefly describe how the baseline mechanism works. First, they extend the page table to include extra information, shown in Figure 6(a), that identifies if the page is private (P bit) and which core is the keeper of the private page (keeper), and whether the entry is currently cached in the TLB. The TLB inherits the P bit from the page table and add an additional L-bit that allows the TLB entry to become locked in certain situations.

4.2 Memory Operation Flow for Reconfigurable Dark Directories

Figure 6(b) shows a flow chart for the overall workings of the architecture. On each memory reference, the TLB is referenced. If the TLB entry is a hit and the P-bit is set, then no coherence action is needed. If the action misses in the L1 and L2 cache, then the L2 miss can be sent directly to the memory controller for reduced latency to DDR, bypassing the directory. If the P-bit is not set, then standard coherence actions apply. On a TLB miss, the page table is accessed. If there is no such page table entry, then a new one is allocated using conventional OS mechanisms and the P-bit is set and the keeper field is initialized to the requested core's ID. Otherwise, if an entry exists, then the entry is copied to the TLB. If the P-bit is clear, then no actions are needed and the TLB miss has completed. If the P-bit is set in the new entry and the keeper is the same as the requesting core's ID, then no further action is needed.

If the P-bit is set but the keeper is different from the requesting core, then another core may have the data cached remotely. At this point in time, a coherence recovery action occurs. First, the page is locked so that no further actions can happen on it on remote cores. Next, a message is sent to the keeper asking it to flush all lines that are cached for the page in question. When the keeper receives this request, it locks the corresponding TLB entry (if it is still cached there) by setting the L-bit, waits for all pending requests in the MSHRs to finish, and flushes all dirty data and invalidates all clean lines within the page. When all operations are complete, the page is marked shared (clear P-bit in page table and TLB), and the L-bit is cleared in the TLB of the keeper. At this point, the TLB access is retried and will hit.

4.3 Shared State Tracking with Per-Tile Directory Power Management

We extend the baseline with support for per-tile directory power management so that the OS or hardware can shut off the directory when it is not being used. This requires support for independently disabling and enabling power to the directory. We believe such support is already available for other structures, like cores and cache, so we assume extending it to the directory will be straightforward.

By default, our architecture assumes that a directory should be powered off. The directory is fully clock gated and powered off to save both static and dynamic power. Only when a shared page is actively used by multiple tiles will the directory (or directories) that support it be powered on. To support a default-off strategy, we add logic in hardware that assists in determining if a directory needs to be powered up.

4.3.1 Turning a Directory On. We leverage an on-chip power management unit (PMU) to power up a directory on demand. Most chips produced today have a centralized power unit controlling voltage, frequency, and other clock gating options. We add a register to the PMU that tracks the status of each directory, shown in Figure 6(c) as DirStatus. When we identify a page transitioning from private to shared status, we send a message to the PMU requesting that the directory be powered on. If the directory is already powered up, an acknowledgement is immediately returned. Otherwise, the PMU initiates power-up and initializes the directory. When the directory is ready, an acknowledgement is sent back to the requesting core and the corresponding bit in DirStatus. Figure 6(b) shows the modified flow chart with our changes to power up the directory.

4.3.2 Turning a Directory Off. After turning a directory on, we will eventually want to power it down again when it is no longer needed. We choose to shut down a directory lazily when it is provably idle. We adopt a simple algorithm that allows the OS to accurately measure when the state in a directory is dead so that it can turn idle directories off.

Detecting idle directories can be formulated as a flow-in and flow-out problem. Some number of shared pages are activated during a program's execution, and only when all of them are reclaimed by the system is the directory idle. We can easily account for reclaimed pages (part of the flow-out calculation). Pages that are moved to disk can be accounted for when they are copied out of memory and their page table entry invalidated. Also, when a process terminates, the OS can traverse its active pages and account for all shared pages and which directories they use. This takes care of shared pages flowing-out of use.

However, it is a bit harder to know when pages become shared. In the baseline system, when a page transitions to shared state, the OS is not notified. Therefore, we add a table to the PMU that tracks the number of pages that transition to shared state for each directory. A 64-bit register per directory is more than sufficient to track these pages even for a long period of execution. The OS will periodically read values out of the table to determine if a given directory has no active pages. Once a directory's flow-in balances its flow-out, it is idle and can be powered off.

We use a bit vector that tracks which cores have accessed a directory. As a request reaches a directory, the cores bit is set. As the OS clears the pages at program end, we reset that bit by sending a clear transaction to the directory. Once a directory is eligible to be powered off (has a clear bit vector), the OS should turn it off as soon as possible to save power, but it need not interrupt a working program to do so. Hence, the performance overhead should be low or balanced against other power saving policies.

4.4 Reconfiguring Directory as Cache

The directory is extended to function as a cache when it is not needed for coherence and the tile is actively running a thread. As discussed in the prior section, we assume the directory is sized to provide double cache capacity.

Here we would like to point out all the similarities of a directory structure to a cache and why they can look alike. Both caches and directories have tag arrays and both have a form of data. In the case of cache, it is data from memory; in the case of a directory, it is the sharers. It is easy to assume many big vectors can compose the data line of a cache.

Fig. 7. Logic to unify L2 and Directory as a single cache.

We could allow the block size of the reconfigured directory to be smaller than the L2 cache's block size. However, standard coherence protocols assume fixed sized blocks, and allowing anything other than that would require alterations to cache coherence. As a result, we do not consider this possibility further and require that all cache blocks are uniform in size. As a result of our assumption of cache block size, there are two implementations that can be used to have a directory data array serve as a cache data array.

4.4.1 Cache Reused as Directory. This implementation is area inefficient but easy in terms of complexity. We replace a directory structure with a cache. When in directory mode, we index the tag array, and if there is a hit, then we select the correct data array. Once the data array is selected if we are in cache mode, we forward the entire line. If we are in directory mode, then we only select a smaller granule that would fit the core count. For 64-tile configurations, we would select the first 64 bits and use that part of the line for the directory information.

4.4.2 Directory Reused as Cache. This implementation has less area overhead but is complex in terms of the data array. We use a directory as we know it today but use the overprovisioning as a means to increase the associativity of the directory. In this case, a directory when used as a cache will have to use half of the tag array, and once there is a tag hit, it selects multiple entries in the data array. This is because when core counts are <512 tiles, a sharer vector is less than a cache line. As such in a 64-tile configuration (which we overprovision by 8) you would use the overprovisioned data array to form a 64B line.

The number of ways we can expand the cache is given by Equation (2). As we can see here the overprovisioning parameter and the number of tiles in the system can be tuned to enable reconfiguring the directory as a cache. For 64 tiles, to enable cache reconfiguration, it would require 8 ways of a Sparse directory to create a full 64B cache line. As a result, an 8× overprovisioning doubles the size and associativity of the cache. In early stages overprovisioning 8× might seem excessive; however, current designs already do overprovision by this much to prevent directory-induced invalidations (Sanchez and Christos 2012; Ferdman et al. 2011). A mere 2× overprovisioning of the directory would double the size of the cache at 256 tile configuration,

$$ReuseWays = \frac{CacheLineSize}{\#ofSharerBits \times DirAssoc \times overprovisioning}. \quad (2)$$

Figure 7 provides a logical view of how the directory state extends the cache state. In part (a), we show the two structures together. When the Directory is operating normally (as a directory), the index selects a bank and the selector logic picks output from the correct bank. In cache mode,

the index is modified and sent to each Way. Then the Selector selects the data from the matching way based on the results of the comparator logic.

In part (b), we show a zoomed-in view of one Directory Bank or Cache Way, depending on the required configuration. In Directory Mode, the Tag Array and Data Array implement associative ways. The Comparator and Selector logic select the sharing vector from the matching way. However, in Cache Mode, the ways are merged together to form a single data cache line (we only need to use one tag field from the directory). We assume that the MSHRs and other miss logic is retained in the L2 and that the Directory has its own state tracking for its replacement policy, so they are simply reconfigured in cache mode.

Since most of the area for a cache is in the tag and data array, we expect the overall area and timing impact to be under 5% (in keeping with Ranganathan et al. (2000)). Given that we are reconfiguring at the L2 level, such an increase in access time can easily be accommodated by the overall chip design.

4.4.3 Restoring the Directory to Normal Operating Mode. If a page is transitioned to shared status and is mapped to a directory currently operating as a cache, then the architecture takes actions to restore the directory to its normal operating mode. The PMU keeps track of the current status of all directories. Note that part of the DirStatus state for a directory indicates if it is powered on but used for cache. If this is the case, then the PMU sends a message to the directory, resetting its Reconfigure_Cache_Tile register, asking it to return to normal status. At this point, the directory must write back any dirty lines and evict all shared lines. Only after all of these operations are complete can it notify the PMU that it is ready to serve as a directory again.

If a lot of dirty data have been cached in the directory, then it must be detected by walking the cache, and it must be written back in accordance with the cache coherence protocol.

4.5 Operating System Challenges

4.5.1 OS Level Thread Migration. In current multicore processors, threads can be migrated to other cores. Due to the fact that we keep track of sharers in the page table, thread migration is interpreted as sharing on a page for the first time. In this case, all dirty lines must be written back before the page transitions to shared state. Clearly, this is bad for our approach. Current systems still prefer to limit the amount of migration after a context switch by selecting the same core; however, this is not guaranteed. We believe future work could explore limiting thread migration or preserve the private status of pages when migrated across cores.

Due to the fact that thread migration caused by the OS can have a significant impact on performance, we expect that future processors that use our technique will limit thread migration. After all, current systems already assign I/O device interrupts and critical OS kernel code to specific cores, so it is reasonable to have a specialized pool of cores available with limited thread migration. Also, affinity scheduling is already commonly used in industry to bind specific threads. An example use of this would be to schedule first to cores closer to memory controllers. Such support can be added to the OS scheduler or managed using core binding utilities such as numactl (Kleen 2004).

4.5.2 OS Level Object Sharing. OS level object sharing can undermine some of our benefits, because shared objects in the OS may be spread across many tiles requiring additional directories to be powered on. Recent studies have made progress reducing such sharing (Baumann et al. 2009), but they do not totally eliminate it. To address the OS level sharing, we propose to have a memory carve out only for OS shared data. This carve out would be mapped to specific tiles and so we would not be able to reconfigure or turn off certain tiles. This way OS processes can be statically assigned closer to the OS mapped directories for reduced coherence lookups.

4.6 Complexity and Design Alternatives

Since detection of shared and private data is easily done with help from the OS, our proposal integrates well with the prior work. Furthermore, managing directory state using the OS provides opportunity to tune policies in software. For example, using a page coloring algorithm, pages can be mapped to suitable directories to reduce the number of active directories. We evaluate this possibility in Section 7.

5 EXPERIMENTAL METHODOLOGY

Modeled System. We evaluate our proposal with a full-system simulation using Virtutech Simics (Magnusson et al. 2002). For modeling the power contributions of each tile’s components we used Fabscalar (Choudhary et al. 2011), extending it to measure the power of the directory and interconnect. The Fabscalar framework synthesizes Verilog models of arbitrary superscalar processors and, in our case, in-order processors. It uses the FreePDK OpenAccess 45nm Standard Cell Library. These models assume power-efficient LSTP transistors. Since Fabscalar is in 45nm, we scale using the ITRS (2011) roadmap to 22nm.

TurboBoost. For all of our designs, we assume that reductions in *static power* can be leveraged either for power savings or for frequency scaling. Using an approach much like Intel’s TurboBoost (Turbo 2012), we assume we can scale up frequency to the chip’s peak power envelope. However, since we do not have a thermal model in our infrastructure, we do not exploit lower than expected dynamic power to further scale up frequency (as is possible in TurboBoost). Also, all of our results already assume we have applied TurboBoost on the static and dynamic peak power savings from shutting off cores.

Simulated System. Table 3 shows our tile details and configuration parameters. For most experiments, we used a 64-tile configuration unless otherwise noted. One key observation is that the frequency of all cores active and the frequency when all cores are active but directories are off.

Table 4 describes the keys for each system we evaluate. Filtered is the system we compare against; Filtered is a conventional cache coherent processor with a sparse directory implementation. Filtered also detects and removes directory accesses on private data. Dark is our basic approach that leaves a directory off until it is needed. Dark+R adds the cache re-use component. Note, for this design, we configure the directory for use as a cache during the entire execution. Overall, a configuration will specify four components: the architecture, the overprovision factor of the directory, the number of tiles (default is 64), and the frequency the design can operate at due to frequency scaling.

Directory Implementation. We use in our evaluation a sparse organization for our directory with a full-bit map vector. However, we will extrapolate the behavior and possible benefits of our technique on less-area-expensive designs as we scale the number of cores (Section 6.2).

Workloads. We evaluated our proposal using single-threaded workloads and parallel benchmarks. For single-threaded workloads, we used SPEC 2006 (Henning 2006), and for parallel workloads we ran a variety of benchmarks from two suites, SPLASH-2 (Woo et al. 1995) and PARSEC (Bienia et al. 2008), using ref inputs or simmedium for parsec. We ran single-threaded applications by skipping initialization code (5 billion instructions) and then executing for a significant amount of time (2 billion instructions). In our results, we show benchmarks that had a L2 hit rate <90%. This is because benchmarks with >90% hit rate would not be affected by changes in caches; they would, however, experience bigger frequency scaling increases. For parallel workloads, we marked the parallel regions and executed between them using breakpoints. For the 256-tile projections, we use the performance of the 64-tile system and apply dynamic voltage-frequency scaling (DVFS) to estimate the effects.

Table 3. System Configurations

In-order Tile count	64-tiles	256-tiles
All Cores and Directories On Frequency	1.54GHz	1.3GHz
All Cores On, Single Core Boost, No Directories Frequency	1.7GHz	1.66GHz
System Peak Power	125W	
Per tile configurations		
Cache Block Size	64Byte	
L1 Caches	32KB 4-way 2-cycle latency	
L2/LLC Caches	512KB 8-way, 10-cycle latency	
Directory cache	Overprovision * (16K entries)	
Overprovision	8×	2×
Memory Bus Latency	83ns	
Page Size	4KB	
Network Parameters		
Network Topology	8×8 mesh	
Control Message Size	8Bytes	
Data Message Size	72Bytes	
Link Bandwidth	1 Flit/cycle	
Flit Size	64bits	

Table 4. System Configuration Key

Label	Meaning
Filtered	Traditional system + Private Data Filtering
Dark	Power off directory
Dark+R	Dark + Cache reconfiguration
Overall Key	Label+OverProvisionFactor+#ofTiles+GHz
Example	Filtered+2x+64+1.2

6 PERFORMANCE EVALUATION

6.1 Performance for Single-Threaded Applications

First, we evaluate the performance benefits of our design normalized to the Filtered architecture. Dark directories are more beneficial for underutilized systems, and so we look at a system running a single thread of a SPEC 2006 benchmark.

Figure 8 shows the speedup from frequency scaling by turning off directories and reconfigurability effects, normalized to Filtered+8x+64+1.54. Filtered is for one active core with all directories on. Dark+R+8x+64+1.70GHz shows the benefits of reconfiguring the directory and scaling frequency at the same time for 1-tile having all chip directories dark (off). Our technique achieves a geometric mean speedup of 1.17×. Looking deeper, we can observe that all benchmarks show an improvement. The origin of the benefits can be broken down from Dark+8x+64+1.70 for benefits from frequency and Dark+R+8x+64+1.54GHz for benefits from reconfigurability.

In the Dark-only design examining the frequency scaling effect. The geometric mean across all benchmarks improves by 5.6%. However, looking at the frequencies of the configurations (1.54GHz

Fig. 8. Performance normalized to filtered with 8 $\times$ overprovisioning.

vs. 1.70GHz), we see that our technique increases the frequency by 10%. This lower-than-expected speedup is primarily due to memory component of applications. Frequency benefits do not affect memory latency, so the time to access memory remains a bottleneck, and a couple of examples of these are mcf and omnetpp

Dark+R+8x+64+1.54GHz is the effect of reconfiguring the directory to extend cache capacity. Overall the benefits from reconfiguring the directory as a cache are 10%. Unlike the effects of frequency, we see that memory-intensive applications with large working sets have a big gain. Mcf, omnetpp, and xalan exploit the added LLC capacity and improve by 1.32 $\times$, 2.06 $\times$, and 1.63 $\times$, respectively. This shows the potential of the added, reconfigurable directory/cache area. Another interesting fact is that some memory-intensive applications such as lbm and milc do not experience any speedups due to their working set being larger than the Dark+R cache. For CPU intensive applications that do not benefit from the extra cache, turning off the directory could result in leakage savings.

As the number of tiles scale up, even schemes with very little overhead per tile eventually consume a large amount of power across the whole chip. The relative gains provided at the 8 $\times$ provisioning via frequency scaling are eventually available for fewer overprovisioned directories at larger core counts (see Figure 2).

Static Power. In Figure 9, we plot the static power as a function of number of active tiles for a 64-tile configuration. An active tile's static power is composed of the Core, L1, and L2 static power. For all configurations, all routers on a chip are contributing to the static power. However, the directories are only included when the design has them powered on. Note that directories are always powered on in all tiles in the Filtered+8x designs; they are always powered off in the Dark+8x designs, and they are powered on only in the active cores for the Dark+R designs. In this figure, we show the Filtered and Dark architectures with a 8x overprovisioned directory. The y-axis shows the total chip static power normalized to Filtered+8x with 1 active tile. Note that this normalized value already includes all directories and routers plus 1 active tile. For reference, we also show the static power for Filtered which is only 2x overprovisioned which is most like the implementation of current directories.

Now we compare the Filtered and Dark+R designs. The Filtered design has all directories powered on but the Dark+R design only has directories powered on within active tiles. Here we see the power benefits of turning off directories and reconfiguring the active tile's directory as cache.

Fig. 9. Total static power normalized to Filtered+2x on 1 tile.

Fig. 10. Energy delay² including static and dynamic power.

The savings are 88% for 1 active tile. What is also extremely interesting is that Dark directories are more power efficient than even a 2 $\times$ overprovisioned directory while Dark-R directories are more power efficient up to 16 cores active despite requiring 4 $\times$ the area per directory. As we increase the number of active cores, Dark+R consumes the same static power as the Filtered architecture, since Dark+R reconfigures the transistors as cache.

Dark+*x shows the difference of having directories completely powered off even when all tiles are active. Such a design is beneficial for a multiprogrammed workload or a multithreaded workload with no sharing. Looking at the case of 64 active tiles, we see that the power for the Filtered+8x+64 configuration is 3.2 $\times$ the static power of one active tile. This is due to the routers and directories of those tiles being powered on for both configurations.

Energy-Delay² Reduction. In Figure 10, we show the effects of $Energy \times Delay^2$ (ED^2) on our system including static and dynamic power. Overall, we reduce the ED^2 by 25% for 8 $\times$ overprovisioning. This is due to the high static power reduction for unused directory tiles. We further characterize coherence level behavior in Figure 11. We show the effects of filtering private data accesses to the directory on coherence message traffic. Overall, we are able to save about 8% of the interconnect flit traffic. This is on par with the results from Cuesta et al. (2011).

Fig. 11. Traffic Reduction by avoiding accessing directories on private data.

Fig. 12. Performance when Scaled to a 256-tile processor with 2X overprovisioning.

6.2 Future Scaling of Directories

We have argued that the overhead from directories will grow ever larger when compared to a single tile, and that means we can extrapolate from our results to future generations with more tiles and smaller directories to estimate future benefits. To justify this in part, we increase the number of tiles to 256 and assume a 2X overprovisioning. As we can see in Figure 12, the performance trends are similar to the 8X overprovisioned 64-tile configuration. This is because the normalized area and static power per tile is similar for Dark+8x+64 and Dark+2x+256. However, the number of tiles has increased providing a larger frequency boost.

An important conclusion here is that regardless of the actual directory design, as the total directory storage increases with tile count, the frequency scaling benefits will improve for cases when most directories are disabled. Even with a smaller directory, the benefits will just be delayed a few processor generations. Also, once the directory is a significant fraction of the L2 cache size, the benefits will also grow, since a larger cache will reduce long latencies through the interconnect and to main memory. This means that cheaper schemes (less overprovisioning) than the ones we evaluated will eventually benefit from Dark Directories in a big way.

Fig. 13. Mapping of shared pages to tiles for cholesky.

Fig. 14. Mapping of shared pages to tiles for radix.

7 FUTURE DIRECTION

In Figure 13 and Figure 14, we plot a surface diagram showing the number of shared data pages (z-axis) used by each tile (x-axis going across the page) per directory (y-axis going into the page). For cholesky, each tile touches many shared pages mapped to many different directories. It is unlikely any directories could be turned off or that pages could be colored to map to fewer directories. On the other hand, in radix, each tile only sparsely uses pages per directory, with a lower degree of sharing.

Parallel programs tend to have a large number of shared pages. Page-to-directory mapping strategies in hardware try to avoid hot spots by either interleaving based on cache block (spreading each block in a page to a different directory) or by mapping whole pages to a specific directory as in Cuesta et al. (2011). Since dark directories can be turned off, new optimizations may try to keep

Fig. 15. Directory usage for shared data on Dark+PC+8x+64.

as many directories off as possible. A page coloring algorithm in the OS that tends to cluster shared pages into fewer directories while still preventing hot spots would be advantageous. Prior work has discussed page coloring algorithms for managing data placement on multicores (Awasthi et al. 2009). We perform a limit study to see the potential of a coloring algorithm that aims to reduce the directory usage. Our algorithm groups pages into the fewest number of directories possible and examines the resulting impact on performance and directory provisioning.

While the directories are under-utilized, the OS places pages into a directory such that all can fit without leading to conflict misses. Once the directories are fully utilized, the OS then overprovisions pages to directories arbitrarily. Figure 15 show the effects of our page coloring technique compared to a default coloring algorithm that is agnostic of Dark Directories. On the y -axis, we show the total number of directories needed for Dark+PC+8x+64⁴ using a varying number of threads per application.

We see that most applications need only a handful of directories to keep all of their shared pages. For the Filtered design, all applications used all 64 directories except for barnes (60), blackscholes (40), and raytrace (40). The reduction in directory use is quite significant for 16 threads, since most applications go from using the full 64 directories to only a handful. We do not show the effect of congestion on performance due to space; however, when a few threads are running this translates to a performance advantage.

Figure 16 shows the effect of Dark Directories on frequency scaling for a 64-tile system assuming our page coloring mechanism can tightly pack the pages into one directory. Each application uses 16 threads. Overall, the performance benefits would be around 6% for Dark+PC+8x+64 due to frequency scaling per benchmark. This proves that managing directory space effectively is good for both parallel and single-threaded programs that under-utilize the full chip resources.

Effective Chip-Level Directory Usage. We also measure the effective directory size as a function of the total area devoted to the directory. Interestingly, on average, Dark Directories reduce the effective directory size by 95% with the optimal page coloring. Note that this reduction is much greater than what is cited in previous work (Cuesta et al. 2011) by scaling down the width and height. However, our approach does not penalize applications that need a large directory; directories are simply turned on as needed.

Page Walking Overhead. Figure 17 shows the percentage overhead to walk the page entries to clean up directories that are no more in use once an applications ends. The overhead is calculated by assuming a page-table traversal for the application in which all page entries miss in the cache (a

⁴PC stands for Page Coloring and is added to the key to indicate inclusion of our ideal page coloring algorithm.

Fig. 16. Parallel application performance using ideal page coloring and frequency scaling.

Fig. 17. Overhead to turn off a directory once an application finishes.

pessimistic assumption). Also, we compare it only to the simulated region, not the full application run. We see that the percentage overhead is quite low for all the applications except *fft*, which has an overhead of 1%. Without our pessimistic assumptions, this overhead may be reduced.

8 RELATED WORK

Recent works have described the problem of dark silicon as it relates to multicore scaling (Esmailzadeh et al. 2011) and cache design (Hardavellas et al. 2011). However, no work has considered powering off directories to save power.

A lot of recent proposals also focus on reducing the overhead of directories (Cuesta et al. 2011; Ferdman et al. 2011; Yang et al. 1992; Wallach 1992; Sanchez and Christos 2012; Zebchuk et al. 2009; Lotfi-Kamran et al. 2010; Zebchuk et al. 2009; Gupta et al. 1990) by trading off area for complexity. These proposals are interesting and either reduce the height and width components of the directory, or they try to reduce the number of accesses (Lotfi-Kamran et al. 2010). However, we take a different approach by looking at the overall chip resources dedicated to directories and pointing out that the chipwide costs do not scale efficiently with respect to applications with fixed resource needs. Our approach of turning off directories allows us to keep a large fraction of the directory powered off except when it is needed. We also suggest designing multipurpose

directories to be reconfigured as cache. As a high-level design strategy, we believe these are critical components in addition to making smaller directories.

We build on prior work that disables coherence for private pages (Cuesta et al. 2011), and we believe that effective page coloring will be important to extract the full potential of our technique. Dynamic Directories (Das et al. 2012) also allocates directory entries only for shared pages and places directories at sharers to avoid three-hop messages. Prior work has considered the design of efficient page coloring algorithms (Awasthi et al. 2009). For future work, we need to better understand the interaction of practical page coloring approaches in our architecture, as well as mapping dynamically directories to tiles to increase the possibility to power down directories.

Our re-usable directory builds on a large base of literature in reconfigurable caches, like in Ranganathan et al. (2000), Zhou et al. (2001), and Albonesi (1999). We are not the first to recommend changing the way a cache structure works for some performance/power savings. However, we do not know of other designs that reconfigure a directory to extend cache capacity.

9 CONCLUSION

Reconfigurable Dark Directories are an important step in the design of cache coherence mechanisms for future multicores. It heads in a new direction of freeing up power and memory resources for other purposes to boost performance. Dark Directories provide the means to trade off unused directory space for power savings or frequency scaling advantages. Furthermore, because directories are unused much of the time, we propose to re-use them as cache and size them to provide a significant performance boost for applications that benefit from more cache. Interestingly, since the size of one directory is not a big concern when most directories are disabled, we find that sparse directory designs are compelling due to their re-usability up to at least 256 tiles.

We have demonstrated that Reconfigurable Dark Directories benefit both single-threaded and parallel programs. Our combined results for Dark Directories and Directory Reconfiguration running SPEC 2006 applications show a performance benefit, on average, of 10% in a fully loaded system and up to 17% on a lightly loaded system with 64 tiles or a 10–29% average speedup (depending on the load) on a 256-tile system. We also project a 6% performance benefit for SPLASH assuming pages can be mapped into few directories. Our results also indicate that Reconfigurable Dark Directories works in tandem to page coloring, and they argue for research in efficient page coloring and OS object sharing. While we have focused on one particular tile-based architecture, we believe the lessons learned from this study can be extended to designs with shared caches.

REFERENCES

- David H. Albonesi. 1999. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 32)*. IEEE Computer Society, Washington, DC, 248–259.
- Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John B. Carter. 2009. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *Proceedings of the 15th International Conference on High-Performance Computer Architecture (HPCA-15'09)*. 250–261.
- Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'09)*. 29–44.
- Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques (PACT'08)*. 72–81.
- Niket Kumar Choudhary, Salil V. Wadhavkar, Tanmay A. Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H. Dwiell, Sandeep Navada, Hashem Hashemi Najaf-abadi, and Eric Rotenberg. 2011. FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA'11)*. 11–22.

- Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. 2011. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*.
- Abhishek Das, Matt Schuchhardt, Nikos Hardavellas, Gokhan Memik, and Alok Choudhary. 2012. Dynamic directories: A mechanism for reducing on-chip interconnect power in multicores. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 479–484.
- Hadi Esmailzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the ACM International Symposium on Computer Architecture*.
- Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. 2011. Cuckoo directory: A scalable directory for many-core systems. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*.
- Rupert Goodwins. 2010. Intel unveils many-core Knights platform for HPC. Retrieved from <http://www.zdnet.com/intel-unveils-many-core-knights-platform-for-hpc-3040089093>.
- Anoop Gupta, Wolf Dietrich Weber, and Todd Mowry. 1990. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of the International Conference on Parallel Processing*.
- Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward dark silicon in servers. *IEEE Micro* 31, 4 (2011), 6–15.
- John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* (2006), 1–17. DOI: <http://dx.doi.org/10.1145/1186736.1186737>
- ITRS. 2011. International Technology Roadmap for Semiconductors 2011 Edition Executive Summary.
- Nam Sung Kim, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. 2003. Leakage current: Moore's law meets static power. *Computer* 36, 12 (2003), 68–75.
- Andi Kleen. 2004. numactl(8). Linux man page. Retrieved from <http://linux.die.net/man/8/numactl>.
- Pejman Lotfi-Kamran, Michael Ferdman, Daniel Crisan, and Babak Falsafi. 2010. TurboTag: Lookup filtering to reduce coherence directory power. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'10)*.
- Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högborg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (Feb. 2002), 50–58. <http://dx.doi.org/10.1109/2.982916>.
- Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. 2012. Why on-chip cache coherence is here to stay. *Commun. ACM* 55, 7 (July 2012), 78–89. DOI: <http://dx.doi.org/10.1145/2209249.2209269>
- K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. 2011. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *Proceedings of the Workload Characterization (IISWC'11)*. 116–125.
- Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. 2000. Reconfigurable caches and their application to media processing. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'00)*.
- Daniel Sanchez and Kozyrakis Christos. 2012. SCD: A scalable coherence directory with flexible sharer set encoding. In *Proceedings of the 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA'12)*.
- Turbo. 2012. Intel Turbo Boost Technology- On Demand Processor Performance. Retrieved from <http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html>.
- Deborah A. Wallach. 1992. PHD: A Hierarchical Cache Coherent Protocol. Technical Report. Massachusetts Institute of Technology, Cambridge, MA, USA.
- Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*.
- Q. Yang, G. Thangadurai, and L. M. Bhuyan. 1992. Design of an adaptive cache coherence protocol for large scale multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 3, 3 (May 1992), 281–293. DOI: <http://dx.doi.org/10.1109/71.1392>.
- Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. 2009. A tagless coherence directory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*.
- Huiyang Zhou, Mark C. Toburen, Eric Rotenberg, and Thomas M. Conte. 2001. Adaptive mode control: A static-power-efficient cache design. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*.

Received June 2016; revised October 2016; accepted November 2016