# Energy-Efficient Cache Coherence Protocols in Chip-Multiprocessors for Server Consolidation

Antonio García-Guirado*, Ricardo Fernández-Pascual*, Alberto Ros*† and José M. García*

*Dept. de Ingeniería y Tecnología de Computadores
Facultad de Informática - Universidad de Murcia
Murcia, Spain
Email: {toni, rfernandez, a.ros, jmgarcia}@ditec.um.es

†Dept. de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia
Valencia, Spain
aros@gap.upv.es

***Abstract*—As the number of cores in a chip increases, power consumption is becoming a major constraint in the design of chip multiprocessors. At the same time, server consolidation is gaining importance to take advantage of such a number of cores. Our goal is to alleviate this constraint by reducing the power consumption of chip multiprocessors used for consolidated workloads by means of the cache coherence protocol. For this, we statically divide the chip in areas, which allows us to reduce the directory overhead needed to support coherence and to reduce the network traffic. This translates into less power consumption without performance degradation. Cache coherence is maintained per area and pointers are used to link the areas, thereby achieving isolation among virtual machines and savings in memory requirements. Additionally, the coherence protocol dynamically selects one node per area as responsible for providing the data on a cache miss, thus lessening the average cache miss latency and the traffic among areas. Compared to a highly-optimized directory implementation, the leakage power consumption is reduced by 54% and the dynamic power consumption of the caches and the network-on-chip by up to 38% for a 64-tile chip multiprocessor with 4 virtual machines, showing no performance degradation.**

## I. Introduction

Nowadays, the number of cores in chip multiprocessors (CMPs) is steadily increasing. CMPs with hundreds of cores will be a reality in the near future. For example, Intel's Tera-scale Computing Research Program [1] has already presented an 80-core processor, as well as a Single-chip Cloud Computer microprocessor [2] composed by 24 tiles containing 2 processor cores each. With the popularization of cloud computing, one of the main ways to make the most out of such parallel machines is by means of server consolidation, running many virtual machines (VMs) in a single CMP.

One of the main concerns in the development of CMPs is the so called power wall, which can prevent those chips from integrating the expected amount of cores due to their excessive power consumption. Therefore, the design of every element of the chip should aim at reducing power consumption. This includes networks-on-chip (NoCs) and caches, which currently account for up to 50% of the overall power consumption of the chip [3].

In a shared-memory architecture like that of a CMP, the coherence protocol is a key element in the performance and power consumption of the system. Hence, when the system is meant to be used in a consolidated environment, the coherence protocol should take advantage of the special characteristics of such an environment in order to improve performance and reduce power consumption.

Typically, there are two kinds of data in a VM: private data to the VM and data shared between VMs. The first one is only accessed by a single VM, and therefore, there is no use in keeping coherence information beyond the limits of that VM.

On the other hand, data shared between VMs is expected to be read-only data which are being shared because the hypervisor has applied memory deduplication [4, 5]. Deduplicated memory pages are read-only memory pages, with identical contents, that are present in the virtual memory of more than one single VM. The hypervisor detects these identical pages and a single physical page is allocated in physical memory. If a deduplicated memory page is written, a copy-on-write policy is used and a new page is allocated for being used by the writer of the page. Memory deduplication can provide important memory savings, which has made this technique very important for server consolidation. Linux has supported memory deduplication in the KVM hypervisor since version 2.6.32, and other hypervisors such as Xen [5] or VMware [4] already support it. Regarding the cache coherence protocol, if cache coherence is kept strictly per VM it could achieve significant savings in cache storage. However, deduplicated data would need to be reduplicated at the L2 level to give each VM its own copy, which would increase cache pressure and thus reduce performance. In [6] it is shown that the performance of a flat directory protocol improves by 6.6% in average when using deduplication thanks to the cache pressure reduction provided by storing a single copy of the deduplicated data in the shared last-level cache.

In this paper we present a new coherence scheme that addresses some of the challenges posed by server consolidation: it reduces power consumption, reduces the cache coherence storage overhead, keeps a single copy of deduplicated data in the shared cache, provides (partial) isolation among cores of different VMs and reduces the average latency of cache misses.

Our proposal is based upon Direct Coherence (DiCo) [7], whose characteristics make it a suitable baseline for the consolidated scenario in a CMP, especially its ability to resolve cache misses in just two hops without visiting the home node for the memory block.

We have derived two cache coherence protocols from DiCo. One of them, called DiCo-Providers, is also well suited for non-virtualized environments whereas the other, named DiCo-Arin trades some performance in the general case for increased simplicity and reduced power consumption in virtualized scenarios. In order to allow the presence of sharers for a block in any L1 cache of the chip, in DiCo-Providers one L1 cache in each area (the *provider*) tracks the sharers in its area, while DiCo-Arin does not keep exact information about sharers from more than one area and relies on broadcast to invalidate them when necessary.

An area is a subset of all the tiles of the chip. The basic idea is to isolate each VM in a different area by instructing the OS to schedule the threads of each VM to tiles in different areas, although any configuration of VMs is possible with a small cost in performance when a VM uses tiles from more than one area. Coherence information is kept per area, which significantly reduces the storage overhead of the proposed coherence protocols and therefore their power consumption.

Our protocols achieve a 59–64% reduction in directory information in cache for a 64-tile CMP with just 4 VMs. This reduces static power consumption by 45–54% and improves scalability. In Apache, our protocols reduce dynamic power consumption by up to 38% and achieve speedups up to 6% with respect to an optimized directory protocol.

The rest of this paper is organized as follows. Section II gives a view of current proposals aimed at reducing power consumption in consolidated servers and describes the basics of Direct Coherence (DiCo) protocols. In section III we present DiCo-Providers and DiCo-Arin. Section IV gives further details on the operation of our protocols. Then, section V evaluates the proposed protocols comparing them to a flat directory protocol and the original DiCo protocol. Finally, our conclusions are shown in section VI.

## II. Related Work and Background

Power-aware cache coherence has gained interest in recent years. The Tagless Coherence Directory [8] reduces the overhead compared to a flat directory by using bloom filters to store coherence information which leads to less power consumption in the directory. TurboTag [9] uses bloom filters to avoid unnecessary tag lookups and reduce power consumption. Coherence protocols can also take advantage of heterogeneous networks to reduce power consumption by transmitting critical and short messages through fast power-consuming wires and non-critical messages through slower low-power wires [10]. SARC Coherence [11] reduces traffic by using tear-off copies of the block in a weak-ordered memory system and by using prediction to avoid the indirection of the directory.

As for virtualization, some proposals address scalability or performance, but up to our knowledge ours is the first one to address power consumption. Coherency Domains (CDs) have been proposed to increase the scalability of cache coherence in such a scenario [12, 13, 14]. They isolate the coherence transactions of different VMs, each one executed in a different set of processors, preventing different VMs from interacting with each other. Moreover, the data in the L2 cache is stored closer to the cores that use them, since each coherency domain is given a private L2 cache. However, CDs do not allow the simultaneous use of all the resources of the core for a single task. Tasks are confined to the static coherency domains defined by the design on the chip. In addition, CDs do not enable deduplication of data in memory since each VM is given its own independent share of the physical memory and no coherence is kept for data between different domains.

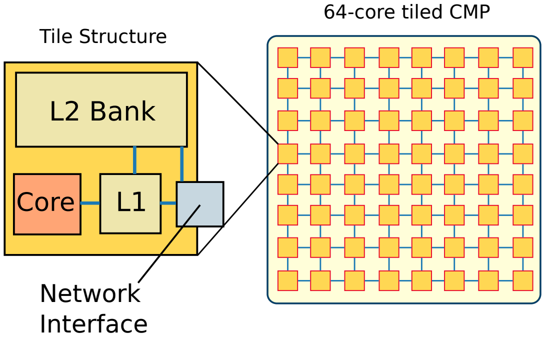


Figure 1. Structure of a 64-core tiled CMP.

Another proposal is Virtual Hierarchies (VHs) [15], which also achieves isolation among VMs, and additionally allows for the dynamic allocation of resources to VMs. However, contrary to our proposals, VHs increase the overhead and power consumption of the cache coherence protocol due to the second level of coherence information that is needed. Furthermore, VHs reduplicate previously deduplicated data in the shared levels of the cache hierarchy, which also results in an increase of the L2 miss rate [6].

Mechanisms at the level of the interconnection network have also been proposed to isolate the traffic of each VM [16].

### A. Base Architecture

We assume a tiled-CMP with an optimized directory-based coherence protocol. This kind of chip is built by replicating basic building blocks, namely tiles, each one containing a processor, an L1 cache, an L2 cache bank and a network interface for communication between tiles. The L2 cache, although physically distributed, is logically shared among all tiles. Some bits of the address of a memory block are used to map the block to its home L2 bank (the bank that caches the block and its coherence information). L1 and L2 caches are non-inclusive. To store the directory information for those blocks not present in the L2 cache we use the same approach as NCID [17], in which extra tags in the L2 cache are used to store a virtual directory cache.

This reduces network traffic and improves the performance of the directory. Additionally, if a block is evicted from the L2 cache, the directory information can remain, hence preventing invalidations of the block in the L1 caches. Only when a directory entry is evicted, the block is also evicted (if present), and every copy of the block is invalidated. When copies of the block are present in the chip, the home L2 for the block stores their directory information. Upon an L1 miss, a request is sent to the home L2 bank, where the directory information for the block can be found. This architecture is depicted in Figure 1.

It is important to notice that we use a full-map bit-vector instead of a coarse bit vector, limited pointers or other sharing code because the full-map provides the best performance and lowest traffic for the base architecture. Other sharing codes trade-off reduced directory overhead for extra network traffic and worse performance, whereas our proposals improve all of these metrics. Additionally, our protocols could be implemented using any of those alternative sharing codes to further reduce the directory overhead if desired.

### B. Direct Coherence

Our protocols are based upon a Direct Coherence (DiCo) scheme [7]. In DiCo, the coherence information and the ownership of the block are stored along with the data in the L1 caches. This makes it possible to resolve most misses in just two hops (i.e. without indirection) by predicting the destination of requests upon an L1 miss. Requests are sent straight to the owner. Additionally, upon a write miss the owner L1 itself can send invalidations to the sharers because it knows who they are. In this way, the distinctive indirection of directory protocols is avoided.

As for deduplicated data, direct coherence does not reduplicate it in the shared L2 cache. Only one copy of the data is needed for all the tiles, reducing the space needed for shared data in cache, just like in a flat directory.

We use DiCo as a baseline because of its ability to resolve misses in two hops without visiting the home node (just the owner node), and because its prediction technique has been proven pretty accurate [7]. Since the owner node will be located in the area where the application is running while the home node can be located anywhere in the system, DiCo becomes a very suitable protocol for the environment considered in this work.

## III. Energy-Efficient Coherence Protocols

One advantage of DiCo when used in a consolidated server is that it isolates the VMs, brings data closer to the requestors (they are found in the corresponding VM), and reduces the number of hops upon a cache miss (two instead of three as in directory based coherence protocols). This is achieved because in the common case the directory information for a block private to a VM can be found in an L1 cache belonging to that VM. This data can be accessed in just two hops, without needing to send any coherence message out of the VM.

With DiCo, deduplicated data is not reduplicated: the owner and directory information of a deduplicated block is present in only one of the VMs, and the other VMs must send their read requests to the owner in order to get the block. This results in higher latency and power consumption for these misses with respect to misses to VM private data and interferences among VMs.

Upon DiCo, we propose to statically divide the CMP in a fixed number of areas (subsets of the tiles of the chip). The use of a static division of the chip in areas enables a reduction of the directory information and therefore its power consumption, contrary to a dynamic division of the chip that would need to increase the directory size to support the different configurations of the chip, hence increasing the power consumption of the caches.

Our claim is that we can take advantage of the circumstance that data shared across VMs is expected to be read-only data in most cases and our new coherence scheme allows L1 misses for such data to be resolved inside the area, with less power consumption, and yet keep a single copy of deduplicated data in the shared level of the cache. At the same time, directory information gets noticeably reduced.

The division of the chip in areas is hard-wired. The OS or hypervisor could optionally be made aware of the different areas in the chip to better map processes to cores and provide isolation between VMs. The hardware needs no information about the VMs running on top of it.

The hard-wired division in areas is not an important issue when the VMs do not exactly match the areas. For instance, when an application uses all the cores of the chip the data shared by several areas can still be accessed without leaving the areas of the requestors, so we still have the benefits of shortened misses. In addition we have the power benefits of the smaller directory entries enabled by the static division of the chip. This makes our proposals attractive for other scenarios in addition to consolidated servers using virtualization.

Next, we propose two protocols that reduce the latency and power consumption of misses to deduplicated data while keeping a single copy in L2 cache.

### A. DiCo-Providers

In DiCo-Providers, coherence information is kept for the sharers of an area, instead of for the whole chip, thereby reducing space requirements. If no VM uses tiles from more than one area only deduplicated data will be shared between areas.

We introduce the *providership* concept and its associated state, *provider* state. With the addition of the provider state, every area can have its own provider for every block shared between areas, hence allowing the resolution of requests to these data in two hops without leaving the area. However, a single ordering point remains in the chip: the owner. This way, the protocol has only one level, like a flat-directory. To simplify, the term *supplier* is used to refer to a node that can be either an owner or a provider.

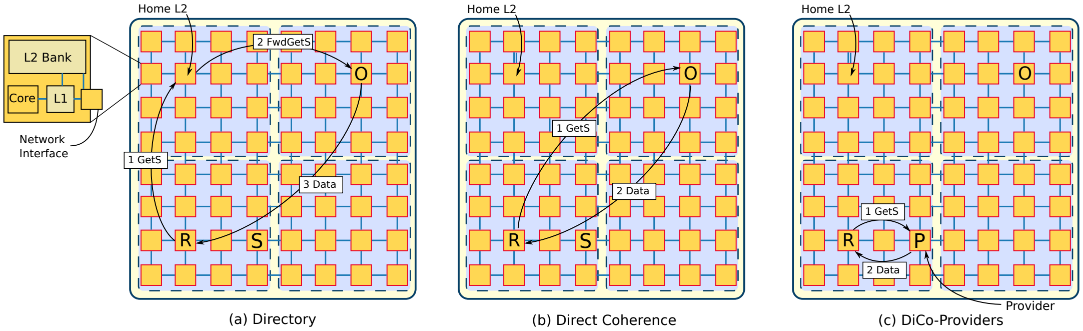


Figure 2. Read request to a deduplicated block. Four VMs running on the chip (dashed lines). One sharer exists in the requestor's area. (a) Directory indirection causes a long 3 hop miss. (b) DiCo avoids one hop by sending the request straight to the owner in L1. (c) DiCo-Providers additionally reduces the number of traversed links by sending the request to the sharer (Provider) in the area. (O = Owner; S = Sharer; P = Provider; R = Requestor).

The regular operation to resolve an L1 miss for a deduplicated block can be seen in Figure 2 for the directory protocol, DiCo and DiCo-Providers.

In DiCo-Providers, the directory information of each data block is distributed across the chip. Every block has a fixed home L2, which is determined by using several fixed bits of its address. The ownership of a block can be held by its home L2 or by any L1 cache. If the ownership is held by an L1 cache the home L2 keeps a pointer in a special structure (called L2C$, see section IV) to store the current location of the ownership. The owner (be it an L1 cache or the home L2) keeps the directory information about the providers (up to one provider per area). The directory information regarding the sharers of each area is kept by the corresponding provider. When an L1 cache holds the ownership it also keeps the coherence information about the sharers in its area. That is, the owner L1 behaves as the only provider for its local area. Notice that if the home L2 holds the ownership, the home L2 does not keep coherence information about sharers since that information is stored by the providers. Figure 3 shows the distribution of coherence information in a flat-directory, the original DiCo protocol and DiCo-Providers.

Four are the events that may cause an L1 cache to become the owner of a block: (1) the L1's tile's core performs a write to that block; (2) the L1's tile's core performs a read to a block that is not present in the chip; (3) an L1's tile's core belonging to an area with no supplier performs a read to a block whose ownership is held by the home L2; (4) an ownership transference between L1s takes place due to a block replacement, as discussed in section IV-A1.

The home L2 can become the owner of the block only due to the replacement of the ownership in an L1 cache belonging to an area with no sharers, as discussed in section IV-A1.

As for providership, only the L1 caches can be providers. There are two ways for an L1 cache to become the provider of a block: (1) an L1 cache performs a read request in an area with no supplier while another L1 cache holds the ownership; (2) because of a providership transference due to a block replacement.

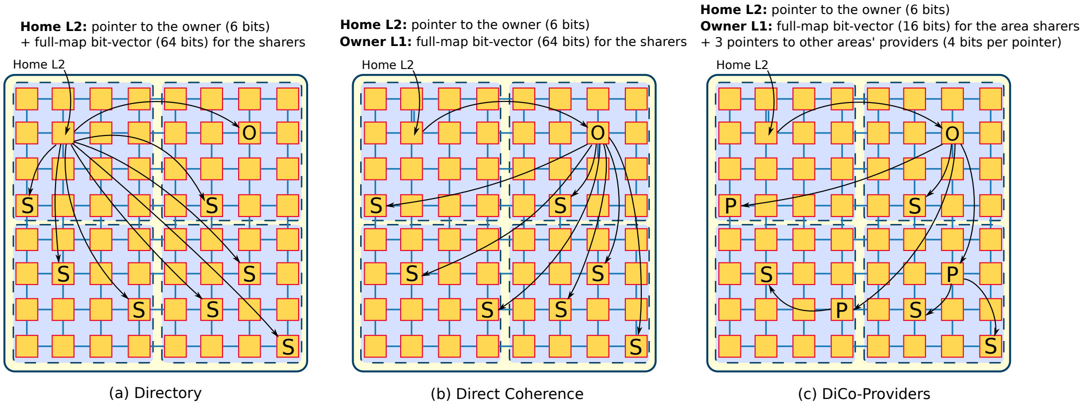
### B. *DiCo-Arin*

Unfortunately, DiCo-Providers shows great complexity if compared to the original DiCo and other directory-based protocols. The providership and ownership transferences present in DiCo-Providers cause a number of coherence races, and therefore the implementation of the protocol is pretty complex.

For this reason, we have polished and simplified our proposal by optimizing it for the virtualized scenario, taking advantage of the fact that deduplicated pages are expected to be read-only pages and that the data shared between the areas in the chip is expected to be deduplicated data. The result is DiCo-Arin: a protocol with similar complexity as the original DiCo in which no precise information about sharers is kept for data shared between areas. Instead, a simple broadcast mechanism is used to invalidate all the copies of these blocks when needed, which should be very infrequently. Our broadcast mechanism is never used to locate data to answer a read request since at least one copy of the data can always be found by using the available directory information. We make sure that our broadcast approach is safe by adding some constraints, as discussed in section IV-B1.

The main problem in DiCo-Providers we have found is that for some cache misses, the critical path to resolve the miss is five hops. This happens when a read miss for a block shared between areas takes place, and the resulting request reaches the home L2 due to an owner misprediction (fortunately, not very common). The request is forwarded to the L1 cache that holds the ownership (if any), and then it is forwarded to the provider in the requester's local area (if any) who finally responds with the requested data and updates the directory information of the block (by adding the requestor). To avoid this, we decided for DiCo-Arin that a block shared between areas will be always present in the home L2, so that requests to such blocks will be answered directly avoiding two of the hops incurred by DiCo-Providers to locate the owner and the provider.

Two different ways to keep the coherence for blocks shared between areas are possible with this design. The


Home L2: pointer to the owner (6 bits)
+ full-map bit-vector (64 bits) for the sharers
Home L2
O
S
Home L2: pointer to the owner (6 bits)
Owner L1: full-map bit-vector (64 bits) for the sharers
Home L2
O
S
Home L2: pointer to the owner (6 bits)
Owner L1: full-map bit-vector (16 bits) for the area sharers
+ 3 pointers to other areas' providers (4 bits per pointer)
Home L2
O
P
S

(a) Directory

(b) Direct Coherence

(c) DiCo-Providers

Figure 3. Coherence information (valid bits not shown). DiCo-Providers needs noticeably less storage for coherence information than the directory and DiCo. (O = Owner; S = Sharer; P = Provider).

first one consists of keeping precise directory information, which would require some mechanisms to coalesce all the directory information, including that about the new sharers that received their data straight from the home L2 instead of from the owner L1. This, however, increases the complexity of the protocol, and as previously stated, one of the main purposes of DiCo-Arin is to simplify it. So we chose the second way, which consists of keeping inexact directory information for data shared between areas and relying on a broadcast mechanism in order to find all the sharers of a block when needed.

As long as the copies of a block are confined to one area of the chip, DiCo-Arin behaves the same as the original DiCo protocol for that block. However, as soon as a read request coming from a remote area reaches the owner L1, the ownership disappears and its former holder becomes a provider for the block. Blocks shared between areas have no owner with precise directory information, hence these blocks rely on broadcast for invalidation (the ordering point is the home L2 in that case) and every new sharer of the block can act as a provider. In addition, the former owner sends the data to L2, which also becomes a provider (and therefore it can serve the block to read requests as stated previously). Notice that in the case that the L2 is already the owner, the last step (sending the data to L2) is not necessary and the L2 becomes a provider immediately upon the reception of the request.

## IV. Detailed operation of the protocols

In order to better understand the proposed protocols, it is first necessary to introduce a few concepts.

Regarding the areas, we must differentiate the *local area* from the *remote areas*. When talking about an L1 cache, its local area is the area that the L1 cache belongs to. Any other area is a remote area for the L1 cache.

We use two kinds of pointers that hold sharing information in our protocols to point to L1 caches: the general pointer, named *GenPo*, and the pointer to provider, named *ProPo*. The size of a GenPo is $log(ntc)$, where $ntc$ is the number of tiles in the chip. Thereby, a GenPo can point to any L1 cache of the chip. The size of a ProPo is $log(nta)$, where $nta$ is the number of tiles in the area. Hence, given one area, a ProPo can point to any tile in that area. Notice that a GenPo is larger than a ProPo. Since a GenPo can point to any tile, it can be used to point to an owner or to a provider, whereas a ProPo can only be used to point to a provider.

Two structures of a DiCo protocol that the reader might not be familiar with are the *L1 Coherence Cache* (L1C$) and the *L2 Coherence Cache* (L2C$). The L1C$ is indexed by the block address and each entry contains a tag and a GenPo. The GenPo holds a prediction of where the supplier of the block is. Upon an L1 miss this prediction (if present) is used as the destination for the request, otherwise the request is sent to the home L2 of the block. The mechanism to update the L1C$ is detailed in section IV-A2. In general, when a block is evicted from the L1 cache, the identity of the supplier is retained in the L1C$ to resolve future cache misses in two hops. The reuse of blocks provides the L1C$ with a good hit ratio [7].

As for the L2C$, it is a cache at the L2 level indexed by the block address that contains tags and GenPos. The information in the L2C$ is not a prediction but the precise identity of the L1 cache that holds the ownership for the block.

### A. *DiCo-Providers*

First we discuss the operation of the protocol upon an L1 miss. The L1 first checks the L1C$ for a supplier prediction. If there is a hit in the L1C$, the request is sent to the predicted cache. Otherwise, the request is sent to the home L2. The objective of using the L1C$ is to resolve the request without indirection (two hops instead of three) by sending it straight to the supplier in the local area. If a *misprediction* occurs, the request reaches an L1 cache that cannot provide the data, which forwards the request to the home L2.

Table I describes the actions performed by a cache upon the reception of a request. The request is forwarded as many times as necessary until it reaches a supplier. The

Table I
ACTIONS PERFORMED UPON THE RECEPTION OF A REQUEST.

| Request Type | Receiver | State | Request coming from local area | Provider Exists | Owner in L1 | Actions taken |
|---|---|---|---|---|---|---|
| read | L1 | owner | yes | | | Send data. Store coherence info in bit vector (requestor becomes sharer) |
| | | | no | yes | | Forward request to provider |
| | | | | no | | Send data. Store coherence info in ProPo (requestor becomes provider) |
| | | provider | yes | | | Send data. Store coherence info in bit vector (requestor becomes sharer) |
| | | | no | | | Forward request to home L2 |
| | | other | | | | Forward request to home L2 |
| | L2 | owner | | yes | | Forward request to provider |
| | | | | no | | Send data. Store coherence info in the L2C$ (requestor becomes owner) |
| | | other | | | yes | Forward request to owner |
| | | | | | no | Send request to memory controller to fetch data from memory. Store coherence info in the L2C$ (requestor will become owner in exclusive state) |
| write | L1 | owner | | | | Start invalidation. Send data. Send Change_Owner message to home L2 to store coherence info in the L2C$ (requestor becomes owner in modified state) |
| | | other | | | | Forward request to home L2 |
| | L2 | owner | | | | Start invalidation. Send data. Store coherence info in the L2C$ (requestor becomes owner in modified state) |
| | | other | | | yes | Forward request to owner |
| | | | | | no | Send request to memory controller to fetch data from memory. Store coherence info in the L2C$ (requestor will become owner in modified state) |

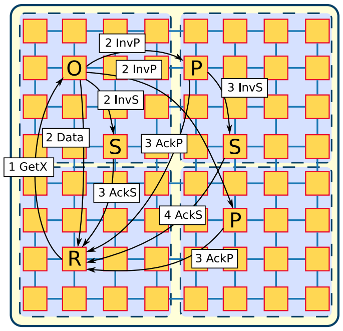


Figure 4. Write request and invalidation process. The supplier prediction succeeds.

original deadlock-avoidance mechanism of DiCo applies to our protocol to prevent a message from being forwarded indefinitely.

The invalidation process on a write miss can be seen in Figure 4. The owner (ordering point) invalidates the sharers in its area and the providers, which in turn invalidate the sharers in their areas. The owner also sends a Change_Owner message to the home L2 (not shown in the image) to let it know the identity of the new owner (the requestor). The ownership cannot be transferred again until an acknowledgement from the home L2 to the Change_Owner message is received by the new owner. This prevents a former owner to be stored in the L2C$ of the home L2 due to the unordered arrival of consecutive Change_Owner messages.

Two counters are needed in the MSHR of the requestor, one to track the number of pending acknowledgement messages from the providers and another to track the number of pending acknowledgement messages from the sharers. The latter counter is incremented every time an acknowledgement message from a provider containing the number of sharers in its area is received. The invalidation is complete once both counters are zero. We need separate counters to prevent protocol races while enabling the concurrent invalidation of all the copies of the block.

One special case is that in which the requestor of a write request is a provider. When this happens, the requestor must invalidate the sharers in its area. However, the invalidations cannot be sent until the requestor receives the ownership or an invalidation message. The latter case happens when a write request from another L1 cache is being served before the request issued by the provider.

The mechanism to evict a block from its home L2 and invalidate every sharer in the chip is the same as the one used to resolve a write request. In the case of a replacement, the L2 cache acts as both the owner (by sending the invalidations) and the requestor (by receiving the acknowledgements).

*1) Block and L2C$ Information Replacements:* Table II describes the replacement of L1 cache blocks depending on their state.

Like it happened in write requests, to prevent races the Change_Owner message (see Table II) requires the reception of an acknowledgement message from the home L2 before the ownership can be transferred again. The same applies to the Change_Provider message.

In addition, when a cache is no longer a sharer due to a previous replacement, it cannot accept neither the ownership nor the providership. In this case the ownership is forwarded to another sharer or the home L2 if no more sharers exist.

Another type of replacement is that of the L2C$ information. The L2C$ has a limited number GenPos, and therefore they may need to be evicted. When this happens, a message is sent to the owner to make it relinquish the ownership and send back the identity of the providers and the data (if dirty). When the ownership is transferred to the home L2, the former owner L1 becomes the provider for its area.

*2) L1C$ update mechanism:* The prediction of the destination of requests is done by searching the L1C$, that stores pointers to L1 caches. The information stored in the L1C$ should be as precise as possible in order to achieve a high ratio of correct predictions, although incorrect information affects only the performance of the system, not its correctness.

L1 cache entries can store one GenPo at no additional cost with respect to DiCo since it can be stored in the space needed by DiCo for the directory information when the L1 holds the ownership. This way, cached blocks do not use a pointer in the dedicated array of the L1C$. The pointers in

Table II
ACTIONS TAKEN UPON A BLOCK REPLACEMENT.

| Block state | Sharers exist in the area | Actions taken |
|---|---|---|
| shared | | Silent eviction |
| provider | yes | Send providership and sharing code to a sharer (the sharer will send a Change_Provider message to the owner) |
| | no | Send No_Provider to the owner |
| owner (including any exclusive state) | yes | Send ownership and sharing code to a sharer (the sharer will send a Change_Owner message to the home L2) |
| | no | Send ownership (and data if dirty) to the home L2 |

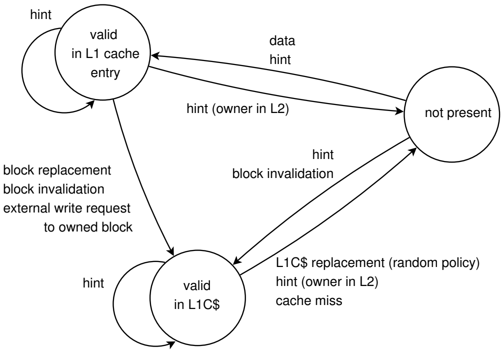


Figure 5. State diagram for the prediction of the supplier of a block.

L1 cache entries are considered part of L1C$ and looked up too when making a prediction.

Figure 5 shows the three possible states for the prediction of a block. The objective is to store the identity of a potential supplier, hence those messages sent by a possible supplier (data messages, invalidations and write requests) update the predictions of the block. We send some hint messages that also update the predictions, for instance when the ownership or providership moves, to let the sharers know the identity of the new owner or provider.

### B. *DiCo-Arin*

Like DiCo-Providers, Dico-Arin is a provider based protocol, which means that a number of nodes, called providers, can serve data to read requests in addition to the owner. However, contrary to DiCo-Providers, where the directory information regarding the location of the providers was located in the node holding the ownership, in DiCo-Arin this information is always located in the home L2 along with the data shared between areas. This way, when a read request to data shared between areas reaches the home L2, the information about the provider, if present, is sent along with the data to the requestor so it can store the identity of the provider in the L1C$. Subsequent misses from that L1 cache will be sent to that provider. If there was no provider in the area, the home L2 stores the requestor as the provider for the area.

One optimization is that every time a copy of such a block is sent to an L1 cache, that L1 cache becomes a provider instead of a sharer. Therefore, read requests are more likely to find a provider. Notice that sharers cannot provide the data because they assume that there is an owner that tracks all the copies of the block. Since no directory information about the sharers for blocks shared between areas is kept, we can use this optimization. Nevertheless, the home L2 stores the identity of a single provider per area along with the data to keep the storage requirements low.

In order to keep the information about providers updated in the home L2, when a request to data shared between areas is forwarded by an L1 cache and reaches the home L2, it checks whether the provider stored in the home L2 for the area matches the identity of the L1 cache that forwarded the request. If so, it means that the cache that forwarded the request is no longer a provider. In that case, the requestor is stored in the home L2 as the new provider for the area. For doing this, the identity of the forwarder of a request is included in the forwarded message.

*1) Ensuring safety of broadcast invalidations:* As we have explained in section III-B, DiCo-Arin uses a broadcast mechanism to invalidate the copies of a block shared between areas upon the occurrence of a write request or an L2 replacement.

In order to use a broadcast mechanism, we must ensure that it keeps the correctness of the protocol, that is: the broadcasts cannot interfere with other requests for the block causing unexpected results (like deadlocks), and coherence cannot be violated for the block (i.e. no copies of the block must remain in the chip after the invalidation).

To ensure these two conditions, we use a three-way invalidation mechanism. First, the home L2 of the block broadcasts the invalidation message. When this message is received, the L1 caches block the block and will not respond to other requests to the same block. Second, every L1 cache acknowledges the invalidation to the requestor or the home L2, depending on whether the invalidation was caused by a write request or an L2 replacement, respectively. Finally, the receiver of the acknowledgements broadcasts another message to let the L1 caches unblock the block and issue responses to requests regarding that block again. This third step prevents L1 caches from obtaining copies of the block from other L1 caches by means of prediction in the middle of the invalidation process.

## V. EVALUATION

### A. *Methodology*

We use Virtual-GEMS [18] to simulate a server running a number of consolidated workloads. To model the network we use a version of Garnet [19] to which we have added broadcast support [20]. Memory access latency is modelled as a fixed number of cycles (plus a small random delay) although we have performed simulations with a more detailed DDR memory controller model and we have found that this does not affect the results. We simulate a 64-core

Table III
SYSTEM CONFIGURATION.

| | |
|---|---|
| Processors | 64 UltraSPARC-III+ 3 GHz. 2-ways, in-order. |
| L1 Cache | Split I&D. Size: 128KB. Associativity: 4-ways. 64 bytes/block.<br>Access latency: 1 (tag) + 2 (data) cycles. |
| L2 Cache | Size: 1MB each bank. 64MB total. Associativity: 8-ways. 64 bytes/block.<br>Access latency: 2 (tag) + 3 (data) cycles. |
| RAM | 4 GB DRAM. 8 memory controllers along the borders of the chip.<br>Memory latency 300 cycles + on-chip delay. Page Size: 4 KB. |
| Interconnection | Bidimensional mesh 8x8. 16 byte links.<br>Latency: 2 cycles/link + 2 cycles/switch +<br>1 cycle/router (in absence of contention)<br>Flit Size: 16 bytes. Control packet size: 1 flit. Data packet size: 5 flits. |

Table IV
BENCHMARK CONFIGURATIONS.

| **Workload** | **Description** | **Size** | **Simulation** | **Performance Metric** | **Memory saved by deduplication** |
|---|---|---|---|---|---|
| apache4x16p | Web server with static contents | 500 clients per VM, 10ms between requests | 4 16-core Apache VMs | No. of transactions in 500 million cycles | 21.72% |
| jbb4x16p | Java server | 1.5 warehouses per tile | 4 16-core JBB VMs | No. of transactions in 500 million cycles | 23.88% |
| radix4x16p | Sorting of integers | 1M integers | 4 16-core Radix VMs | Average execution time of all the VMs | 24.18% |
| lu4x16p | Factorization of a dense matrix | 512x512 matrix | 4 16-core lu VMs | Average execution time of all the VMs | 32.71% |
| volrend4x16p | Ray-casting rendering | Head | 4 16-core volrend VMs | Average execution time of all the VMs | 19.77% |
| tomcatv4x16p | Vectorized mesh generation | 256 | 4 16-core tomcatv VMs | Average execution time of all the VMs | 36.82% |
| mixed-com | Commercial benchmarks: Apache, JBB | See the size of the corresponding benchmarks | 2 16-core Apache VMs and 2 16-core JBB VMs | Weighted no. of transactions in 500 million cycles | 15.74% |
| mixed-sci | Scientific benchmarks: Radix, Lu, Volrend, Tomcatv | See the size of the corresponding benchmarks | 4 16-core VM: Radix, Lu, Volrend and Tomcatv | Average execution time of all the VMs | 15.21% |

tiled CMP and run simulations with 4 VMs running in a single server. Each VM executes its own operating system (Solaris 10) and runs in 16 tiles. Memory deduplication is activated in every simulation. As explained before, the chip is statically divided in four square areas of 16 tiles for DiCo-Providers and DiCo-Arin. In our default configuration we assume that the OS or Hypervisor has been instructed to schedule the threads so that each VM executes in tiles from a different area to take as much advantage as possible from our protocols. We also show an alternative configuration in which the threads of each VM have not been carefully scheduled and each VM uses tiles from more than one area, as shown in Figure 6. We expect that this second case would represent the worst case for DiCo-Arin. This alternative configuration is shown in Figure 6 and is denoted by the suffix "-alt". Tables III and IV show the system configuration and benchmarks used, respectively. Table IV also shows the average memory savings provided by memory deduplication in our benchmarks.

We use CACTI 6.5 [21] to calculate the power consumption (static and dynamic) due to the cache structures (tags, data and directory) with their different sizes in the various protocols, assuming a 32nm process. We consider every event of the cache coherence protocols in the calculation of power consumption, including invalidations, block replacements, directory information updates, etc. As for the network, we calculate the power consumed by message routing and flit transmissions. For this, we use the model proposed in [22] because of its simplicity, in which routing a message consumes as much power as reading an L1 block, and four times as much power as transmitting a flit.

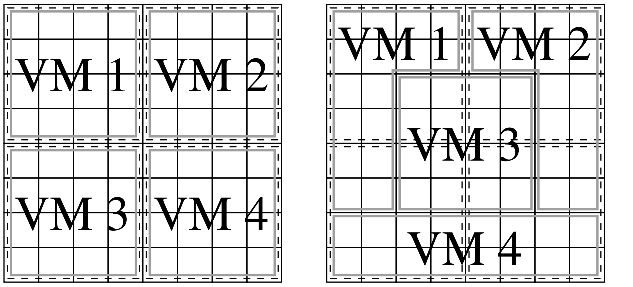


Figure 6. Configuration in which VMs fit the areas on the left. Alternative configuration on the right. Areas shown in dashed lines. VMs shown in grey lines.

### B. Static Power Consumption

DiCo-Providers and DiCo-Arin provide significant savings in cache storage for directory information compared to DiCo or to a flat directory. This translates into static power savings.

We assume an 8x8 tiled CMP divided in four areas. All the areas in the chip are composed by sixteen tiles. Table III shows the size of the caches. We assume physical addresses of 40 bits. There are five different types of tags in a tile: L1Tag (25 bits), L2Tag (17 bits), DirTag (17 bits), L1CTag (23 bits) and L2CTag (17 bits). A GenPo has a size of 6 bits to point to any of the 64 tiles of the chip. A ProPo has a size of 4 bits to point to any of the 16 tiles of an area. We also consider for some of the structures that the validity of an entry can be determined by the state of the block. However, a valid bit is needed for some other structures.

In the case of a flat directory, each L2 entry contains a full-map bit-vector to track the sharers of the block. In addition, a directory cache is needed to track the blocks in exclusive state in the L1 caches. Each entry of this cache contains a full-map bit-vector, a GenPo to store the owner in L1 and a DirTag.

As for DiCo, a full-map bit-vector is needed in each entry of both L1 cache and L2 cache. The L2C$ requires one GenPo (for the case in which the owner is held by an L1) and an L2CTag. An L1C$ entry in any protocol needs a GenPo to store the supplier prediction and an L1CTag.

In DiCo-Providers, the only directory information that must be stored along with a block in the home L2 is one ProPo per area (for the case in which the owner is in the home L2). No information about sharers is necessary in the home L2 thanks to the replacement mechanism described in section IV-A1 that only replaces the ownership to L2 when no sharers exist in the area. The directory information required in L1 is a full-map bit-vector with a bit for each node in the area (to store the sharers when the L1 is the provider or the owner) and one ProPo per area (to store the

Table V
MEMORY OVERHEAD INTRODUCED BY COHERENCE INFORMATION (PER TILE) IN OUR 8X8 TILED CMP.

| | Structure | Entry size | Entries | Total size (KB) | Overhead |
|---|---|---|---|---|---|
| Data | L1 cache | L1Tag (25 bits) + 64 bytes | 2048 | 134.25 | |
| | L2 cache | L2Tag (17 bits) + 64 bytes | 16384 | 1058 | |
| Directory | L2 dir. inf. | 8 bytes | 16384 | 128 | 12.56% |
| | Dir. cache | DirTag (17 bits) + 8 bytes + GenPo (6 bits) | 2048 | 21.75 | |
| DiCo | L1 dir. inf. | 8 bytes | 2048 | 16 | 13.21% |
| | L2 dir. inf. | 8 bytes | 16384 | 128 | |
| | L1C$ | L1CTag (23 bits) + GenPo (6 bits) + Valid Bit | 2048 | 7.5 | |
| | L2C$ | L2CTag (17 bits) + GenPo (6 bits) + Valid Bit | 2048 | 6 | |
| DiCo-Providers | L1 dir. inf. | 2 bytes + 3 ProPos (3×4 bits) + 3 Valid Bits | 2048 | 7.75 | 5.14% |
| | L2 dir. inf. | 4 ProPos (4×4 bits) + 4 Valid Bits | 16384 | 40 | |
| | L1C$ | L1CTag (23 bits) + GenPo (6 bits) + Valid Bit | 2048 | 7.5 | |
| | L2C$ | L2CTag (17 bits) + GenPo (6 bits) + Valid Bit | 2048 | 6 | |
| DiCo-Arin | L1 dir. inf. | 2 bytes | 2048 | 4 | 4.49% |
| | L2 dir. inf. | 2 bytes + 2 bits (area number) | 16384 | 36 | |
| | L1C$ | L1CTag (23 bits) + GenPo (6 bits) + Valid Bit | 2048 | 7.5 | |
| | L2C$ | L2CTag (17 bits) + GenPo (6 bits) + Valid Bit | 2048 | 6 | |

Table VI
LEAKAGE POWER OF THE CACHES PER TILE.

| **Protocol** | **Total Leakage Power (mW)** | **Difference with respect to directory** | **Tag Leakage Power (mW)** | **Difference with respect to directory** |
|---|---|---|---|---|
| Directory | 239 | | 37 | |
| DiCo | 241 | +1% | 39 | +5% |
| DiCo-Providers | 222 | -7% | 20 | -45% |
| DiCo-Arin | 219 | -8% | 17 | -54% |

providers when the L1 is the owner). The L1C$ and L2C$ have the same size as in DiCo.

As for DiCo-Arin, when the L2 is the owner for a block, the directory information needed is a full-map bit-vector of $nta$ bits to store the sharers in the area and $log(na)$ bits to store the number of the area (where $na$ is the number of areas). However, if the block is shared between areas, the L2 only needs one ProPo per area. Hence, since the full-map bit-vector and the ProPos are never needed at the same time, only the space for the largest of them is actually needed. In the L1 cache only a full-map bit-vector of $nta$ bits is needed to store the sharers in the area. The L1C$ and L2C$ have the same size as in DiCo.

Table V summarizes the amount of coherence information needed by each protocol. Contrary to the original DiCo, which needs even more coherence information than an ordinary directory protocol, DiCo-Providers reduces the overhead due to coherence information by 59% with respect to the flat directory and DiCo-Arin reduces it by 64%.

As a result, our proposals reduce leakage power noticeably with respect to the flat directory, as can be seen in Table VI. The total leakage power of the caches is reduced by 8% in DiCo-Arin, due to the reduction in the storage of the directory information which is included in the tag structures of the tile. Overall, tags consume 54% less in DiCo-Arin than in the flat directory. As the number of cores grows, the effect of tag leakage power would become bigger.

Table VII shows the storage overhead of these four cache coherence protocols for a range of number of processors and number of areas in the chip. The overhead of DiCo-Arin depends on the number of cores in each area, while the overhead of DiCo-Providers is also proportional to the number of areas (it needs one ProPo per area). As a result, as the number of areas increases, DiCo-Arin introduces less overhead while the overhead of DiCo-Providers increases.

Table VII
STORAGE OVERHEAD OF THE PROTOCOLS DEPENDING ON THE NUMBER OF CORES AND NUMBER OF AREAS OF THE CHIP.

| 64 cores | 2 areas | 4 areas | 8 areas | 16 areas | 32 areas | 64 areas | | | |
|---|---|---|---|---|---|---|---|---|---|
| Directory | 12.6% | 12.6% | 12.6% | 12.6% | 12.6% | 12.6% | | | |
| DiCo | 13.2% | 13.2% | 13.2% | 13.2% | 13.2% | 13.2% | | | |
| DiCo-Providers | 4% | 5.1% | 7.2% | 10% | 12.6% | 12% | | | |
| DiCo-Arin | 7.3% | 4.5% | 5.3% | 6.6% | 6.5% | 2.3% | | | |
| 128 cores | 2 areas | 4 areas | 8 areas | 16 areas | 32 areas | 64 areas | 128 areas | | |
| Directory | 24.7% | 24.7% | 24.7% | 24.7% | 24.7% | 24.7% | 24.7% | | |
| DiCo | 25.3% | 25.3% | 25.3% | 25.3% | 25.3% | 25.3% | 25.3% | | |
| DiCo-Providers | 5% | 6.2% | 8.8% | 13% | 18.7% | 24% | 22.7% | | |
| DiCo-Arin | 13.4% | 7.5% | 6.8% | 9.3% | 12% | 11.9% | 2.5% | | |
| 256 cores | 2 areas | 4 areas | 8 areas | 16 areas | 32 areas | 64 areas | 128 areas | 256 areas | |
| Directory | 48.9% | 48.9% | 48.9% | 48.9% | 48.9% | 48.9% | 48.9% | 48.9% | |
| DiCo | 49.6% | 49.6% | 49.6% | 49.6% | 49.6% | 49.6% | 49.6% | 49.6% | |
| DiCo-Providers | 6.7% | 7.6% | 10.6% | 16.2% | 24.8% | 36.2% | 47% | 44.3% | |
| DiCo-Arin | 25.5% | 13.5% | 8.5% | 12.2% | 17.4% | 22.7% | 22.7% | 2.6% | |
| 512 cores | 2 areas | 4 areas | 8 areas | 16 areas | 32 areas | 64 areas | 128 areas | 256 areas | 512 areas |
| Directory | 97.5% | 97.5% | 97.5% | 97.5% | 97.5% | 97.5% | 97.5% | 97.5% | 97.5% |
| DiCo | 98.2% | 98.2% | 98.2% | 98.2% | 98.2% | 98.2% | 98.2% | 98.2% | 98.2% |
| DiCo-Providers | 9.7% | 9.7% | 12.8% | 19.6% | 31.1% | 48.5% | 71.3% | 92.9% | 87.5% |
| DiCo-Arin | 49.8% | 25.7% | 13.7% | 15.2% | 23% | 33.6% | 44.3% | 44.3% | 2.8% |
| 1024 cores | 2 areas | 4 areas | 8 areas | 16 areas | 32 areas | 64 areas | 128 areas | 256 areas | 512 areas |
| Directory | 195% | 195% | 195% | 195% | 195% | 195% | 195% | 195% | 195% |
| DiCo | 195.6% | 195.6% | 195.6% | 195.6% | 195.6% | 195.6% | 195.6% | 195.6% | 195.6% |
| DiCo-Providers | 15.5% | 13.1% | 15.7% | 23.3% | 37.5% | 60.8% | 95.8% | 141.7% | 184.9% |
| DiCo-Arin | 98.5% | 50% | 25.9% | 18.6% | 28.8% | 44.6% | 66.1% | 87.6% | 87.6% |

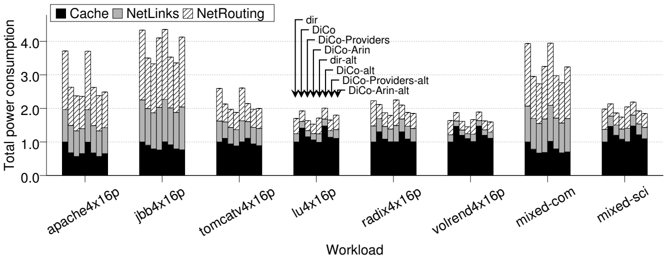
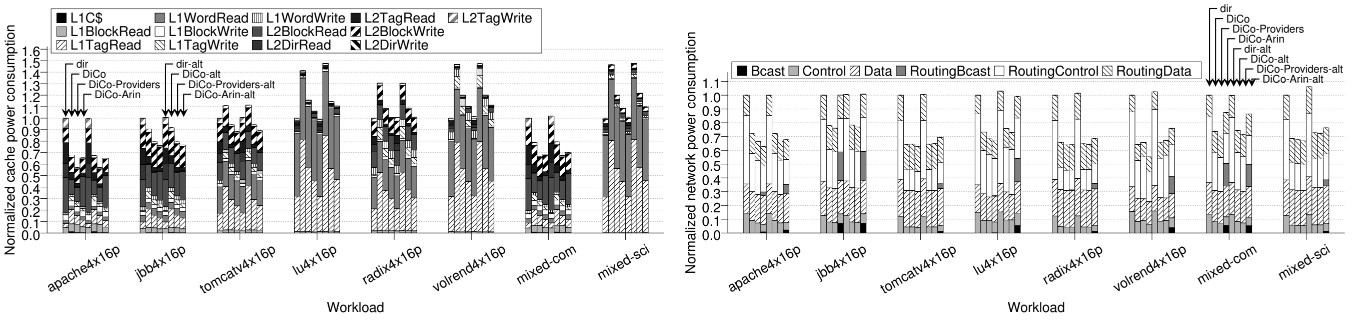


Figure 7. Total dynamic power consumption by protocol. Results normalized to the cache dynamic power consumption of the directory. Breakdown in cache, network links and network routing consumptions.

An appropriate number of areas should be chosen for a given number of cores, and higher storage savings can be achieved as the number of cores increases. A trade-off exists in which using smaller areas implies that providers will be closer to the requestors but also that finding a provider in the area is less likely.

## C. Dynamic Power Consumption

Figure 7 shows the total dynamic power consumption of our protocols. Two kinds of workloads can be observed: those in which power consumption is dominated by L1 caches, like Tomcatv, Lu, Radix and Volrend, and those in which it is dominated by L2 caches and network traffic, like Apache and JBB. L1-power-dominated workloads are those whose working set fits in the L1 cache, and therefore very little traffic and directory accesses are observed in them. On the other hand, L2-power-dominated workloads have working sets that are significantly larger than the L1 caches, which causes many L1 cache misses resulting in higher network usage and directory accesses. L2-power-dominated workloads are the norm in real scenarios since real applications have bigger working sets that those of ordinary benchmarks [23], but we show both kinds of workloads for completion in our analysis. We can see that



(a) Normalized cache dynamic power consumption by protocol. Breakdown in cache events that cause the consumption.

(b) Normalized network dynamic power consumption by protocol. Breakdown in link usage and routing consumption.

Figure 8. Power consumption.

our proposals reduce dynamic power consumption in every benchmark compared to the directory, but this reduction is more noticeable in L2-power-dominated workloads. We find Apache the most representative benchmark due to its large working set and because the other benchmark with a large working set, JBB, has a huge L2 cache miss rate over 40%.

In general, our protocols reduce L2 cache and network power consumption but increase L1 cache power consumption. DiCo noticeably reduces network usage with respect to the directory thanks to solving many requests in just two hops. DiCo-Providers and DiCo-Arin can reduce network usage even further thanks to the use of providers in the area for deduplicated data, shortening the average distance travelled by the messages.

Figure 8a depicts how, due to the directory information stored in the L1 caches, tag accesses are more power consuming in DiCo-based protocols than in the flat directory. This causes DiCo-based protocols to use more power in the caches in some L1-power-dominated workloads (lu, volrend, and radix with DiCo-Providers). Since network usage is pretty small in these workloads, compared to cache usage, our protocols can only improve the overall power consumption by a small margin, thanks to our savings in network traffic (Figure 8b). Nevertheless, both DiCo-Providers and DiCo-Arin improve the original DiCo total power consumption by at least 10% in every L1-power-dominated workload.

The power consumption of the L1C$ of DiCo-based protocols is not a significant share of the overall power consumption. This is thanks to its small size and because it is accessed only after a cache miss takes place and when its contents are updated by events such as invalidations, replacements, etc.

Regarding L2-power-dominated workloads, DiCo-Providers and DiCo-Arin reduce power consumption in Apache by 38% with respect to the directory (Figure 7). This reduction comes both from reductions in network power and in cache power (Figures 8b and 8a). L2 tags are smaller in DiCo-Providers and even smaller in DiCo-Arin. In addition, L2 block reads, which are more power consuming than L1 block reads, are more frequent in the directory since DiCo protocols try to use an L1 cache as the provider to resolve misses in two hops.

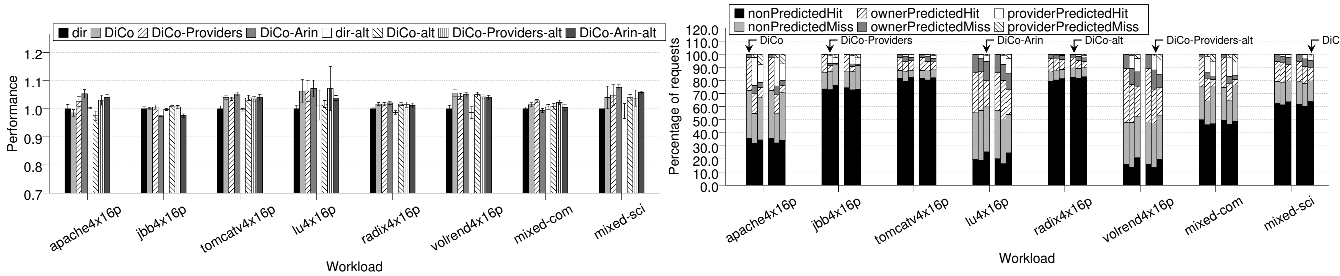
JBB represents the case in which pressure is highest in the L2 cache due to a huge working set. The L2 miss rate of JBB is over 40% for every protocol. We use it as the worst scenario for DiCo-Arin since this protocol uses more L2 space to store deduplicated data and issues broadcasts to invalidate their L1 copies upon an L2 replacement. We can see in Figure 8b that broadcasts make DiCo-Arin network consumption approach that of the directory. However, even in that worst case, DiCo-Arin shows 4% less power consumption than the directory (Figure 7) thanks to the smaller use of L2 caches in general due to the operation of DiCo that resolves many misses in a remote L1 cache. DiCo-Providers proves the most reliable protocol in terms of power consumption and also reduces total power consumption in JBB by 22% with respect to the directory.

Regarding the alternative configuration in which the VMs do not match the areas, no significant differences are observed with respect the optimal configuration in which the VMs fit the areas beyond the logical increment in broadcast traffic in DiCo-Arin due to the extra invalidations of read/write blocks that now are shared between areas. Nevertheless, despite this traffic increment the power consumption of DiCo-Provider keeps being smaller than that of the directory.

### D. Performance Results

The performance of the protocols can be found in Figure 9a. The main conclusion is that DiCo-Providers and DiCo-Arin show no significant degradation regarding the original DiCo. In the most representative benchmark (Apache), DiCo-Providers and DiCo-Arin outperform the directory by 3% and 6% respectively. Only in JBB does DiCo-Arin perform 2% worse than the highly optimized flat directory, and it is due to the particular characteristics of JBB pointed out early that this is the worst scenario for DiCo-Arin due to the heavy pressure over the L2 cache.

In figure 9b we can see that in some benchmarks a significant percentage of the requests can be resolved by predicting the provider. In the case of Apache, 21% of the requests are resolved in this way in DiCo-Providers. Taking



(a) Performance (bigger is better).

(b) Breakdown of L1 misses in six categories depending on whether they were predicted or not, on whether the destination was an owner or a provider in the area, and on whether the prediction succeeded or not.

Figure 9. Performance and prediction accuracy.

into account that the theoretical average distance in a 2D mesh is $\frac{2}{3}\sqrt{ntc}$, where $ntc$ is the number of tiles in the chip, a two hop miss in our 64-tile CMP, with arbitrary origin and destination, would traverse 10.6 links on average. The misses that hit in the provider only take two hops inside a 16-tile area. This results in a theoretical average of 5.4 links traversed to resolve such misses, instead of the 10.6 links needed in DiCo, which matches our experimental results. In the end this means a reduction in number of links traversed with respect to DiCo of 38% and 40% in DiCo-Providers and DiCo-Arin, respectively. We call these misses resolved inside the area *shortened misses*. Overall, shortened misses cause a noticeable reduction in the average miss latency and power consumption. As the number of tiles and VMs increases, this potential benefit should grow. For example, in a densely virtualized 256-tile CMP with 4-tile areas (that is, 64 VMs), indirect misses would take an average of 32 links, normal misses would take 21.3 links, and shortened misses would take just 2.6 links.

The alternative configuration does not produce significant changes in the performance of the VMs in any of the protocols. This is the expected behaviour for the directory and DiCo. For our proposal a noticeable performance reduction could be expected, but it does not actually take place. Two reasons explain why DiCo-Providers and DiCo-Arin keep performing well. First, when a VM executes in cores of more than one area, the L1 owners are still located within the VM and are accessed in two hops. Second, now providers are also used for data private to the VM. These data can be supplied by a provider in the area of the requestor, which is closer to the requestor than the owner is, since the owner might be in another area. This way, the performance and prediction accuracy of our proposals remain almost the same even if the VMs span in several areas.

## VI. Conclusions

Server consolidation is increasingly gaining importance as the number of cores provided in a single chip increases. The number of virtual machines per server is also likely to grow to take advantage of such a number of cores. We have proposed a new scheme with the chip statically divided in areas in which deduplicated data is stored only once in the shared level of cache and yet the data is brought closer to the requestors thanks to the use of *providers*. We have proposed two different protocols based on this scheme: DiCo-Providers and DiCo-Arin. DiCo-Arin is simpler than DiCo-Providers and it requires less hardware for storing sharing information. However, DiCo-Arin relies on broadcast to invalidate data shared between areas (i.e. deduplicated data).

We have shown that our protocols achieve a 59–64% reduction in directory information in cache for a 64-tile CMP with just 4 VMs, which reduces static power consumption by 45–54% and improves scalability. They reduce dynamic power consumption up to 38% for the most representative workload (Apache). When the weak points of our protocols are tested with non-realistic scenarios in which little network traffic is generated and few L1 cache misses take place, the power consumption of our protocols is still lower than that of the optimized directory.

Additionally, speedups up to 6% with respect to the optimized directory protocol have been achieved in Apache. Our protocols do not show any significant degradation in performance with respect to the directory, not even if the placement of the VMs does not exactly match the static areas. On the contrary, they noticeably outperform the directory in most workloads thanks to the use of providers. We also expect that as virtualization density increases, with tens of virtual machines running in a single server, the advantages of our proposals will become even more noticeable.

## Acknowledgments

This work has been jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, by the Generalitat Valenciana under Grant PROMETEO/2008/060, and also by the Spanish MEC and European Commission FEDER funds under grants “Consolider Ingenio-2010 CSD2006-00046” and “TIN2009-14475-C04-02”. Antonio García-Guirado is also supported by a research grant from the Spanish MEC under the FPU National Plan (AP2008-04387).

# References

[1] J. Held, J. Bautista, and S. Koehl, "From a Few Cores to Many: A Tera-scale Computing Research Overview," Intel White Paper, 2006.

[2] J. Held and S. Koehl, "Introducing the Single-Chip Cloud Computer," Intel White Paper, 2010.

[3] N. Magen and A. Kolodny, "Interconnect-Power Dissipation in a Microprocessor," in *Proceedings of the International Workshop on System-Level Interconnect Prediction*, 2004, pp. 7–13.

[4] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *5th Symposium on Operating System Design and Implementation (OSDI)*, 2002, pp. 181–194.

[5] M. Jeon, E. Seo, J. Kim, and J. Lee, "Domain Level Page Sharing in Xen Virtual Machine Systems," in *The 7th International Symposium on Advanced Parallel Processing Technologies (APPT)*, 2007, pp. 590–599.

[6] A. García-Guirado, R. Fernández-Pascual, and J. M. García, "Analizing Cache Coherence Protocols for Server Consolidation," in *Proceedings of the 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2010, pp. 191–198.

[7] A. Ros, M. E. Acacio, and J. M. García, "A Direct Coherence Protocol for Many-Core Chip Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 21, no. 12, pp. 1779–1792, 2010.

[8] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 423–434.

[9] P. Lotfi-Kamran, M. Ferdman, D. Crisan, and B. Falsafi, "TurboTag: Lookup Filtering to Reduce Coherence Directory Power," in *Proceedings of the 16th International Symposium on Low Power Electronics and Design (ISLPED)*, 2010, pp. 377–382.

[10] A. Flores, J. L. Aragón, and M. E. Acacio, "Heterogeneous Interconnects for Energy-Efficient Message Management in CMPs," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 16–28, 2010.

[11] G. Keramidas and S. Kaxiras, "SARC Coherence: Scaling Directory Cache Coherence in Performance and Power (preprint)," *IEEE Micro*, 2010.

[12] R. C. Kinter, "Support for multiple coherence domains," Patent No. WO 2009/039417 A1, 2009, MIPS Technologies, Inc.

[13] Z. Offen, A. Berkovits, and P. Thomas, "Technique to share information among different cache coherency domains," Patent No. WO 2009/120997 A2, 2009, Intel Corporation.

[14] M. A. Blumrich and V. Salapura, "Programmable partitioning for high-performance coherence domains in a multiprocessor system," United States Patent No. US 2009/0006769 A1, 2009, International Business Machines Corporation.

[15] M. R. Marty and M. D. Hill, "Virtual Hierarchies to Support Server Consolidation," in *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, 2007, pp. 46–56.

[16] J. Flich, J. Duato, T. Sødring, Å. G. Solheim, T. Skeie, O. Lysne, and S. Rodrigo, "On the Potential of NoC Virtualization for Multicore Chips," in *International Workshop on Multi-Core Computing Systems (MuCoCoS)*, 2008, pp. 801–807.

[17] L. Zhao, R. Iyer, S. Makineni, D. Newell, and L. Cheng, "Ncid: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies," in *Proceedings of the 7th ACM international conference on Computing frontiers*, 2010, pp. 121–130.

[18] A. García-Guirado, R. Fernández-Pascual, and J. M. García, "Virtual-GEMS: An Infrastructure To Simulate Virtual Machines," in *Proc. of the 5th Int. Workshop on Modeling, Benchmarking and Simulation (in conjunction with ISCA)*, 2009, pp. 53–62.

[19] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 33–42.

[20] J. Duato, S. Yalamanchili, and N. Lionel, *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann, 2002.

[21] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," Hewlett Packard, Tech. Rep., 2009.

[22] N. Barrow-Williams, C. Fensch, and S. Moore, "Proximity coherence for chip multiprocessors," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*, 2010, pp. 123–134.

[23] R. C. Murphy and P. M. Kogge, "On the memory access patterns of supercomputer applications: Benchmark selection and its implications," *IEEE Transactions on Computers*, vol. 56, no. 7, pp. 937–945, 2007.

[24] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.