NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

A study of a dynamic placement policy in a NUCA cache

by

Alexandros I. Daglis

Supervisor: Nectarios Koziris
Associate Professor

Athens, July 2012

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

A study of a dynamic placement policy in a NUCA cache

by

Alexandros I. Daglis

Supervisor: Nectarios Koziris
Associate Professor

Approved by the three-man evaluation committee on the 18th of July 2012.

.....	.....	.....
Nectarios Koziris Associate Professor	Panayiotis Tsanakas Professor	Nikolaos Papaspyrou Assistant Professor

Athens, July 2012.

.....
Alexandros I. Daglis
Electrical and Computer Engineering Diploma holder

Περίληψη

Ένα σημαντικό πρόβλημα στα μοντέρνα υπολογιστικά συστήματα είναι η διάρκεια των προσβάσεων στην ιεραρχία μνήμης. Καθώς τα μεγέθη των μνημών μεγαλώνουν και μαζί τους και αυτές οι καθυστερήσεις, το να απαιτείται ένας ομοιόμορφος, μεγάλος χρόνος για κάθε πρόσβαση στη μνήμη, είναι απαγορευτικό. Για να αντιμετωπιστεί το πρόβλημα αυτό, η κλασσική μονολιθική αρχιτεκτονική της μνήμης εξελίσσεται σε μια κατανεμημένη μορφή, η οποία παρέχει τη δυνατότητα για μη ομοιόμορφους χρόνους προσπέλασης. Η νέα αυτή αρχιτεκτονική είναι γνωστή ως Non-Uniform Cache Architecture (NUCA). Μια μνήμη NUCA απαρτίζεται από πολλά μικρά κομμάτια, τα οποία κατανέμονται χωρικά στην επιφάνεια του chip. Ο χρόνος πρόσβασης σε καθένα από αυτά τα κομμάτια είναι μεταβλητός και εξαρτάται από την απόσταση μεταξύ του αιτούντος επεξεργαστή και του κομματιού της NUCA που εξυπηρετεί το αίτημα.

Σε μια στατική NUCA, τα δεδομένα τοποθετούνται στατικά ανάλογα με τη διεύθυνσή τους. Αυτή η τακτική είναι αρκετά περιοριστική, αφού δεν εξασφαλίζει ότι η πλειονότητα των αιτημάτων θα εξυπηρετηθεί από τα ταχύτερα κομμάτια της NUCA, αυτά δηλαδή που βρίσκονται πιο κοντά στον αιτούντα επεξεργαστή. Έτσι, η έρευνα έχει οδηγήσει τις μνήμες NUCA σε δυναμικές μορφές οργάνωσης, όπου τα δεδομένα τοποθετούνται και μετακινούνται ελεύθερα μέσα στη NUCA. Σε αυτή τη διπλωματική εργασία, αξιολογείται η χρήση δυναμικής NUCA σε ένα πολυεπεξεργαστικό σύστημα, εστιάζοντας κυρίως σε μια πολιτική δυναμικής τοποθέτησης δεδομένων που στοχεύει στη μεγιστοποίηση του αριθμού των προσπελάσεων στα ταχύτερα κομμάτια της. Η πολιτική αυτή συμπληρώνεται με κατάλληλες πολιτικές για την αντικατάσταση, μετακίνηση και αναζήτηση δεδομένων στη μνήμη. Η πολιτική αντικατάστασης επιτρέπει σε ένα κομμάτι της μνήμης να τοποθετεί δικά του δεδομένα σε γειτονικά κομμάτια, αντί αυτά να επιστρέφουν απ' ευθείας στην κύρια μνήμη, ενώ η πολιτική μετακίνησης δεδομένων αποσκοπεί στη μεταφορά τους προς τους επεξεργαστές που τα ζητούν συχνότερα. Για την αξιολόγηση της δυναμικής NUCA χρησιμοποιήθηκαν μετροπρογράμματα με διαφορετική συμπεριφορά, όπως επιστημονικά, server και multi-programmed workloads. Η δυναμική μας NUCA πετυχαίνει σημαντική βελτίωση για όλα τα workloads έναντι της στατικής: 7.7% κατά μέσο όρο και μέγιστη βελτίωση 15.6% για το multi-programmed workload.

Λέξεις-κλειδιά: δυναμική μνήμη μη ομοιόμορφου χρόνου προσπέλασης, πολιτική αντικατάστασης, πολιτική δυναμικής τοποθέτηση, πολιτική μετακίνησης δεδομένων, Flexus

Abstract

An important problem in modern systems is long memory access times, which are a major bottleneck to performance. More specifically, as cache capacities grow, suffering a long, uniform access latency is intolerable. To mitigate this problem, the classic monolithic cache evolves into a distributed cache design, that provides non uniform access, known as Non-Uniform Cache Architecture (NUCA). A NUCA cache is split into smaller slices, which are distributed across the die. Accesses to data that resides in one of these tiles display a variable latency, depending on the physical distance between the requesting core and the cache slice servicing the request.

In a static NUCA, data is statically placed in the cache, according to its address. However, this design imposes limitations, since accesses to local, faster cache tiles are not maximized. Therefore, research has led to dynamic NUCA designs, where data can be freely placed and moved in the cache.

This diploma thesis investigates dynamic NUCA policies in a tiled Chip Multiprocessor (CMP) system, focusing on a dynamic placement policy that aims to maximize accesses to the fastest cache slices. The dynamic placement policy is complemented with an appropriate replacement policy, a migration policy and a lookup mechanism. The replacement policy allows a cache bank to spill data to its neighbors, if there are additional capacity needs, while the migration policy gradually moves data towards the cores that are most frequently accessing it. For our design's evaluation, a diverse workload set comprising of server, scientific and multi-programmed workloads was used. Our dynamic NUCA scheme has shown an average performance improvement of 7.7% over the static NUCA and a maximum performance improvement of 15.6%.

Keywords: dynamic NUCA, placement policy, replacement policy, migration policy, Flexus, tiled architecture, CMP

Acknowledgements

This diploma thesis was conducted in the Computing Systems Laboratory of the School of Electrical and Computer Engineering of the National Technical University of Athens, under the supervision of Associate Professor Nectarios Koziris.

I would like to thank my supervisor, Dr. Nectarios Koziris, for his guidance during the process of this thesis and throughout my undergraduate studies.

I want to express my gratitude to the Post-Doctoral Researcher Dr. Konstantinos Nikas, for his continuing support and encouragement in the course of the months of work required for the completion of this thesis and to the PhD student Stavros Volos, for his help with the numerous technical issues of the toolchain that was used.

Finally, I especially thank my parents and brothers for standing by my side throughout the duration of my undergraduate studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chip Multiprocessors (CMPs) are dominating modern systems. With integration of transistors on a single chip doubling almost every 18 months, CMPs seem to be a one-way road. The successors of the classic uniprocessor design manage to provide solutions to a wide range of problems: frequency scaling, design complexity, power dissipation. Furthermore, new horizons are opened: parallel execution on a single chip, which reveals new opportunities as well as architectural challenges.

1.1 Memory hierarchy

A system's memory hierarchy is of crucial importance for overall performance. In general, the greater the computing power of the system, the greater the access intensity to the memory. With memory access latency being orders of magnitude slower than processors' operation speed, the memory hierarchy can easily turn out to be a major bottleneck to performance. Therefore, memory structures that provide fast access to data are needed. These structures are no other than caches, i.e. small and fast memories that are placed as close to the processors as possible. To balance the tradeoff between size and access latency, multi-level memory hierarchies have emerged as a common architectural trend. Figure 1.1 shows the memory architecture of a system with two levels of caches. The closer to the processor a cache level is located, the smaller its capacity and the faster its access.

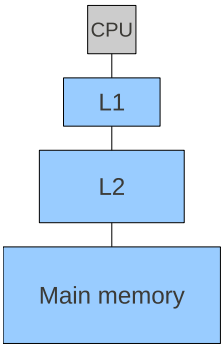


Figure 1.1: A memory hierarchy with two cache levels

Due to their importance, multi-level memory hierarchies have been extensively studied in the context of uniprocessors. However, CMPs give birth to further challenges. As the number of cores on the chip is increased, the pressure to the memory hierarchy heightens. Each running thread could have its own data workset but there could also be sharing between them. Typically, the memory hierarchy becomes more complex, with each processor having one or two private levels of caches and varying degrees of sharing for lower levels. Figure 1.2 illustrates a memory hierarchy with three cache levels for a quad core system, where the L1 caches are private for their processors, L2 caches have a sharing degree of two and the L3 cache has a sharing degree of four, thus it is shared by all processors.

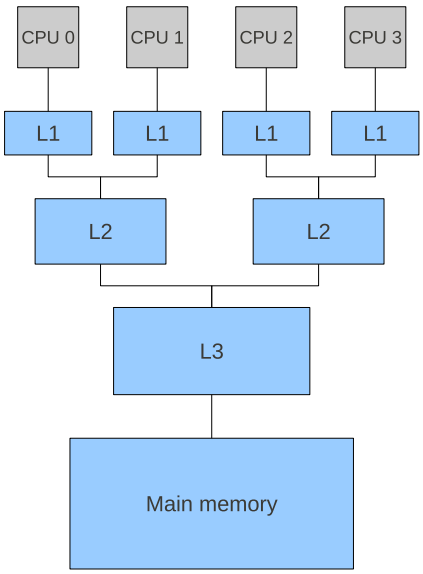


Figure 1.2: A memory hierarchy for a quad core CMP with three cache levels

1.2 Last level cache

The Last Level Cache (LLC) is a component of the memory hierarchy with a particular significance:

- Being the last on-chip memory component, finding data in it is the last chance to avoid a long off-chip request.
- It is usually shared by all of the chip's processors, thus being used and manipulated by all possible request sources.

Thus,the LLC is designed to be as big as practically possible. However, with the ongoing size shrinkage of transistors and increase of integration density, LLCs become so big that new problems arise. A major consequence is that access latency grows prohibitively, resulting in an important performance bottleneck.

1.3 The limitations of the UCA design

Increasing a cache's size is not a straightforward design choice, since it bears a significant tradeoff. Larger capacity comes at the cost of increased access latency, as wire delays grow along with the physical size of the memory structure. For that reason, large on-chip caches with a single, large and uniform latency are undesirable. Especially for an LLC, being the largest on-chip cache, the latency lies in the tens of cycles, making accesses to the LLC unbearably slow. In other words, increasing cache sizes only makes the existing gap between processor and memory access speeds grow even wider.

Tech. (nm)	L2 Size	Num. Sub-banks	Unloaded Latency	Loaded Latency	IPC	Miss Rate
130	2MB	16	13	67.7	0.41	0.23
100	4MB	16	18	91.1	0.39	0.20
70	8MB	32	26	144.2	0.34	0.17
50	16MB	32	41	255.1	0.26	0.13

Table 1.1: Performance of UCA organizations

The limitations of the classic design where the memory exhibits a Uniform Cache Access time (UCA), are very well presented by Kim et al. [17], who used Cacti 3.0 [21] to estimate the access times for various cache designs. The parameters and achieved Instructions Per Cycle (IPC) of the UCA organization are shown in Table 1.1. Unloaded latency is the average time in cycles, assuming uniform bank access distribution and no contention, while loaded latency was obtained through an experimental evaluation and is the actual L2 cache access time -including contention- across all of the used benchmarks. Contention includes both *bank* and *channel* contention: bank contention occurs when a request must stall because a precious request is being serviced by the same bank, while channel contention occurs when the bank is free but the routing path to it is busy.

As observed in Table 1.1, unloaded access latency grows significantly with the cache size, which, in turn, has an even greater impact on the loaded latency. A small increase in the cache's unloaded latency greatly increases the probability of overlying requests, which makes contention much worse; thus, it results in a great increase of the loaded latency. This observation suggests that even multiported cells are a poor solution for overlapping accesses in large caches, as increases in area expand loaded access times significantly. To illustrate this further, for a 2-ported, 16 MB L2 cache at 50nm, Cacti

reports a significant increase in the unloaded latency, which makes the 2-ported cache perform worse than a single-ported cache of the same size [17].

Kim et al. presented these results in 2002, when one of the latest processors at the time, IBM's POWER 4 [5], featured a 1.41 MB L2 cache as its LLC. Nowadays, a 16 MB or even bigger on-chip LLC is common, as can beseen in Table 1.2. It is obvious, therefore, that the inefficiencies of the UCA design regarding access latency and contention need to be addressed.

Processor codename	Bloomfield	Beckton	Beckton	Interlagos
Branding & Model	Intel Core-i7 940	Intel Xeon E7540	Intel Xeon X7560	AMD Opteron 6284SE
Technology	45 nm	45 nm	45 nm	32 nm
Number of Cores	4	6	8	16
CPU Clock Rate	2.93 GHz	2.00 GHz	2.26 GHz	2.70 GHz
Year of Release	2008	2010	2010	2012
LLC size (L3 cache)	8 MB	18 MB	24 MB	2x8 MB

Table 1.2: Last level cache sizes on modern processors

Chapter 2

NUCA

In order to adapt to the ever-growing needs of modern memory-hungry workloads, on-chip caches keep growing bigger. Unfortunately, expanding the cache size alone is not sufficient to increase modern systems' efficiency, since the traditional UCA design exhibits serious limitations, as was briefly explained in Chapter 1. The solution lies in a distributed cache design, that manages to provide varying access times and increased bandwidth.

2.1 Introducing the NUCA design

What makes big uniform caches inefficient is the fact that all accesses require the same large amount of time to be serviced, regardless of the data's physical location on the cache array. Ideally, we would like data that resides in the part of the cache that is physically located close to the processor to be accessed faster than data that resides physically farther from the processor. In order to achieve this goal, a complete shift in the cache architecture design paradigm was required. The previously single, monolithic chunk of cache is transformed to a finer-grained structure, as shown in Figure 2.1. More specifically, the last-level cache is composed by physically independent *banks*, which are evenly distributed across the die area. This design provides varying access latencies between the cores and the cache banks, depending on the physical distance between the requesting core and the cache bank where the requested data resides. Thus, we are led to a Non-Uniform Cache Access (NUCA) organization of the cache.

So, NUCA provides faster access to cache blocks in the banks that reside closer to the processor. For example, as suggested by Kim et al. [17] and

(a) UCA
Number of banks: 1 bank
Avg.loaded access time: 255 cycles

(b) NUCA
32 banks
24 cycles

Figure 2.1: Level-2 Cache Architectures

illustrated in Figure 2.1b, the closest bank in a 16 MB, on-chip L2 cache built in a 50 nm process technology can be accessed in 4 cycles, while an access to the farthest bank might take up to 47 cycles. On the other hand, every access to a UCA of the same size would require a constant latency of 41 cycles. As access time is directly related to the block's placement, the placement is an important decision.

NUCA can be classified into two great categories:

- *Static NUCA*, which only places a block in a specific location in the cache .
- *Dynamic NUCA*, which supports a more flexible placement scheme for blocks by not limiting a block's placement to a single location.

2.1.1 Static NUCA

Figure 2.1b shows a banked NUCA cache, as opposed to the classic UCA shown in Figure 2.1a. This static NUCA design uses a two-dimensional switched network, permitting a large number of small, fast banks to be interconnected. Table 2.1 shows the performance metrics for this NUCA cache [17]. Unloaded latency values are estimated by Cacti and loaded latency values are the average of the experimental evaluation on a range of different workloads. Cacti provides three different values for the unloaded latency: min, max and average. This illustrates the nature of the NUCA: it allows accessing each bank at different speeds, proportional to the distance of the bank from the cache controller. Thus, the closest bank can be accessed in the minimum time, while an access to the farthest is the slowest. As shown in Table 2.1, not only does latency (both unloaded and loaded) scale much

better with cache size, but there is also a boost in IPC.

Tech (nm)	L2 Size	Num. Banks	Unloaded Latency				Loaded Latency	IPC
			bank	min	max	avg		
130	2MB	16	3	4	11	8	9.7	0.55
100	4MB	32	3	4	15	10	11.9	0.58
70	8MB	32	5	6	29	18	20.6	0.62
50	16MB	32	8	9	32	21	24.2	0.65

Table 2.1: Performance of NUCA organizations

Apart from the varying access times, there are no functional differences between the UCA and the static NUCA. Block placement treats the cache as logically unified: a block will be placed in a location statically determined by its address. A block can only be placed in a single location during its lifetime. This, of course, imposes serious limitations: a frequently accessed block may be placed in a bank located far from the cache controller, thus suffering the overhead of a high access time everytime it is accessed. The block cannot be placed to any other bank, closer to its requester, in order to improve its access time, since its location in the cache is statically defined by its address. This limitation of the static NUCA gave birth to NUCA's next generation designs, the dynamic NUCA, which address the problems that rise from static placement.

2.1.2 Dynamic NUCA

Modern workloads already spend most of their execution time on on-chip cache accesses. Enabling the latency of a cache access to be possibly smaller than that of accessing a large unified cache is not enough. Even for the multi-banked design presented in the previous section, performance can still be improved by exploiting the fact that accessing closer banks is faster than accessing farther banks. Various strategies can be applied in order to maximize the number of accesses to local banks by allowing the cache to dynamically manage its contents. Ideally, frequently used data can be placed in the closest banks, or moved into them, using the interconnection of the cache banks. Of course, for a transition from a completely static configuration to a dynamic one, certain mechanisms have to be designed and implemented to enforce the new functionality. Thus, we move towards a new design of a dynamic NUCA. The physical layout described in Section 2.1.1 and shown in

Figure 2.1b remains the same; what changes is the logic behind data placement in the cache array.

2.1.3 Policies of a NUCA design

A NUCA design can be characterized based on four policies which define its behavior:

- *Bank placement,* which determines the first location of data in the cache.
- *Bank lookup,* which defines the searching algorithm across the banks.
- *Bank migration,* which decides data movements between the NUCA's banks.
- *Bank replacement,* which deals with the evicted data and any actions required upon its eviction.

Static NUCA implements static placement of data (standard placement depending on its address), which also allows a simple static lookup mechanism, using the same static function that is used for placement. It also implements a classic replacement policy, e.g. LRU, and no migration of data at all. A data block is placed in a predefined, statically determined by its address, position and never moves until evicted.

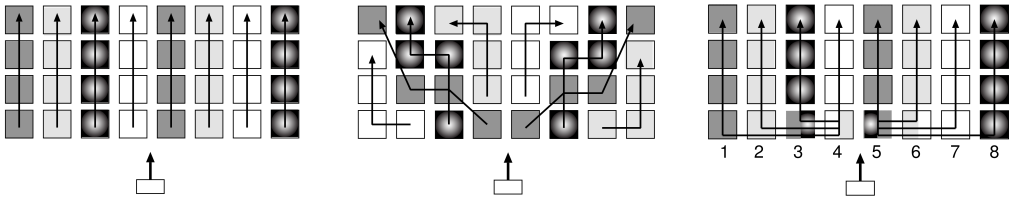
At the other extreme, in a dynamic NUCA, a data block can be placed in *any* bank of the cache. This approach provides the greatest flexibility and unlocks the possibility for greater performance gains. However, such an extremely dynamic placement strategy comes at a cost. The overhead of locating a data block in the cache when it could be found *anywhere*, can be too large. Locating data blocks with no limitation on their possible location, requires a broadcast to all the banks for each access. That would be prohibitive in terms of both latency and energy. Therefore, placement is strongly paired with the lookup mechanism and the greatest challenge is developing hybrid solutions that lay somewhere between the static and the extremely dynamic policies, which would deliver high performance at an affordable cost.

2.2 Background - Related Work

2.2.1 NUCA for uniprocessors

Kim et al . [17] proposed organizing the NUCA banks into *banksets*. In this organization, the multibanked cache is treated as a set-associative structure, where each set is spread across multiple banks and each bank holds one "way" of the set. The collection of banks used to implement this associativity is called a *bankset* and the number of banks in the set corresponds to the associativity. The primary distinction between this organization and a classic set-associative cache is that each way has a different access time.

Three different topological mappings of banksets were evaluated: simple, fair and shared mapping, which are also displayed in Figure 2.2. Each of the mappings has its own advantages and disadvantages. Simple mapping is the simplest but latencies to all bank sets are not the same, since some rows are closer to the cache controller than others. Fair mapping eliminates that problem at the cost of a more complex routing path by a better grouping of banks in each set, so that the average access time across all bank sets is equalized. Finally, the shared mapping, which is proved to be the best, attempts to provide fastest-bank access to all banksets by sharing the closest banks among multiple banksets.



(a) Simple Mapping (b) Fair Mapping (c) Shared Mapping

Figure 2.2: Mapping bank sets to banks

Lookup policies for finding a data block in the cache were also investigated. There are two distinct search policies: the *incremental search*, in which all banks in the bankset are searched sequentially, and the *multicast search*, in which the requested address is multicast to all banks of the bankset. While the latter is significantly faster, it imposes great energy and network contention costs. However, the policy that was evaluated as the best one is a smart search mechanism that exploits the idea of *partial tag comparison*, proposed by Kessler et al . [16], which can be used to reduce both the number

of bank lookups and the miss resolution time. *Smart search* allowed nearly all cache misses to be detected without searching the entire bankset and displayed the best results among the search policies evaluated by Kim et al.

The migration mechanism proposed in that work is fairly simple, since it is tightly related to the organization of the banks in sets. When a hit occurs to a data block in one of the cache's banks, it is swapped with the corresponding block of another bank that belongs in the same bankset and is one step closer to the cache controller.

Finally, different approaches concerning the placement and replacement policies were investigated. A new block may be loaded close to the processor, displacing an important block, or in a distant bank, which would require several accesses before it is eventually migrated to the fastest banks. The replacement policy's decision involve what to do with the victim upon a replacement; two possible approaches are to simply evict the data from the cache (*zero-copy*) or to move the victim to a lower-priority bank (*one-copy*), replacing a less important line. After the evaluation, it was determined that the preferred solution is to place the incoming block in the bank close to the processor and apply the one-copy policy.

Chishti et al . [7] noticed that previously proposed NUCA designs did not fully exploit the non-uniformity in memory accesses, because of keeping data and tag placement coupled. This coupling results in having to tolerate multi-hop latencies to check for a miss, which is determined by accessing the tags. They also pointed out that NUCA can only place a few blocks within the closest banks with the lowest access times and must employ a high-bandwidth switched network to swap blocks within the cache for high performance. To tackle these problems, they proposed the Non-uniform access with Replacement and Placement using Distance associativity (NuRAPID), which leverages sequential tag-data access to decouple data placement from tag placement.

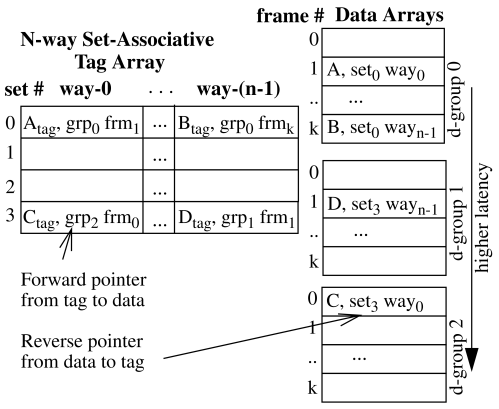


Figure 2.3: NuRAPID Cache

NuRAPID makes a distinction between tag and data placement: distance associativity is the placement of data at a certain distance (and latency) from the cache controller, while set associativity is the placement of tags within a set. Tags are placed in a centralized tag array close to the processor, which is smaller than even one data way. The cache's banks are grouped into distance groups (d-groups), which are used as *buckets*, and data is placed in them, decoupled from tags. What make this decoupling possible are a forward pointer, from the tag in the tag array to the data in the d-group, and a reverse pointer, which establishes the connection the other way around. Blocks can be promoted and demoted between faster and slower d-groups, while the tag array remains unchanged. Chishti et al. suggested that NuRAPID's data-tag decoupling enables flexible placement of the vast majority of frequently-accessed data in the fastest cache banks, with fewer swaps than the standard D-NUCA. As a result, NuRAPID provides both better performance and energy-efficiency.

2.2.2 NUCA for CMPs

NUCA designs described in section 2.2.1 assume a single cache controller as the only entry point to the cache. However, CMPs feature multiple cache controllers which represent individual entry points located all over the cache. Migration causes blocks that are requested by multiple processors at the same time to be pulled in multiple directions which can result in the block ending up in a non-optimal position. Therefore, simple migration mechanisms as the ones previously proposed, only work well in uniprocessor systems and are less effective in CMPs. Furthermore, the range of flexibility for the migration mechanism is more dependent on smart lookup techniques than its uniprocessor counterpart, since such searches are harder to implement in a CMP environment. Doubtlessly, CMPs present additional challenges to be addressed.

2.2.2.1 Migration

Beckmann and Wood [4] investigated NUCA policies for CMPs using the CMP-NUCA layout shown in Figure 2.4. They used this architecture to evaluate both a static NUCA (S-NUCA) and a dynamic NUCA (D-NUCA). In order to employ more flexible placement and migration policies, the banks were grouped in banksets, similar to the work by Kim et al. [17]. In addition, the banksets were also grouped in bankclusters, according to their proxim-

ity to the processors. The proposed placement policy is simple and static: blocks are initially allocated in the requesting core's corresponding bankset, according to their low-order bits. However, their investigation on migration policies provided some interesting results. While direct migration increases the number of local bankcluster hits for the requesting core, it also increases the proportion of costly remote hits by distant processors. Therefore, D-NUCA implements a *gradual migration* policy that moves blocks among the six bankcluster chain:

$$\begin{matrix} other \\ local \end{matrix} \Rightarrow \begin{matrix} other \\ inter \end{matrix} \Rightarrow \begin{matrix} other \\ center \end{matrix} \Rightarrow \begin{matrix} my \\ center \end{matrix} \Rightarrow \begin{matrix} my \\ inter \end{matrix} \Rightarrow \begin{matrix} my \\ local \end{matrix}$$

Consequently, in order for a block that is placed in processor's A local bankcluster to migrate to processor's B local bankcluster, five consecutive requests for the block have to be made by processor B.

This gradual migration allows blocks frequently accessed by one processor to congregate near that particular processor, while blocks accessed by many processors tend to move within the center banks. Based on this migration policy, hits are more likely to occur in one of the six bankclusters: on the requesting processor's local or inter bankclusters, or the four center bankclusters. Therefore, the first phase of the lookup policy is sending requests to these six bankclusters and if all six requests miss, the request gets broadcast to the rest of the bankclusters.

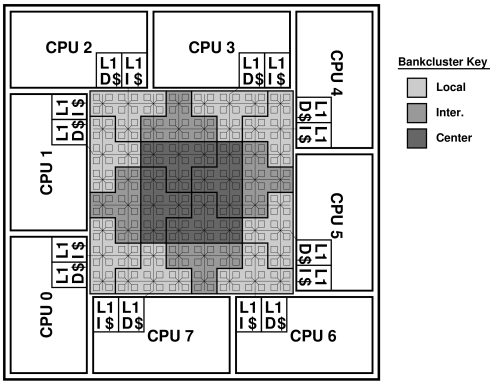


Figure 2.4: CMP-NUCA layout with bankcluster regions

An interesting characteristic of the migration policy proposed in that work is the *lazy migration* mechanism. Every pending migration gets delayed for a thousand cycles and cancelled if the block is accessed during that period by another processor. That way, more than 99% of *false misses*, i.e. where L2 requests fail to find a cache block because it is in transit from one bank to another, are avoided. Finally, Beckmann and Wood conclude that while block migration effectively reduces wire delay in uniprocessor caches, its capability to improve CMP performance relies on a - difficult to implement - smart lookup mechanism. Furthermore, the large amount of inter-processor

sharing that exists in many workloads, fundamentally limits the profit of block migration.

A few months later, Huh et al. presented their own NUCA design for CMPs [14], which was merely an extension of the first NUCA work for uniprocessor systems [17]. The proposed architecture included 16 processors and a very fine-grained L2 NUCA and introduced the concept of the *sharing degree*, which is defined as the number of processors sharing a given pool of cache. The NUCA can be utilized with a sharing degree of 1 as *unshared*, in which each processor has a private portion of the cache, with a sharing degree of 16 as *completely shared*, in which every processor shares the entire cache, and with any sharing other degree in between, as shown in Figure 2.5.

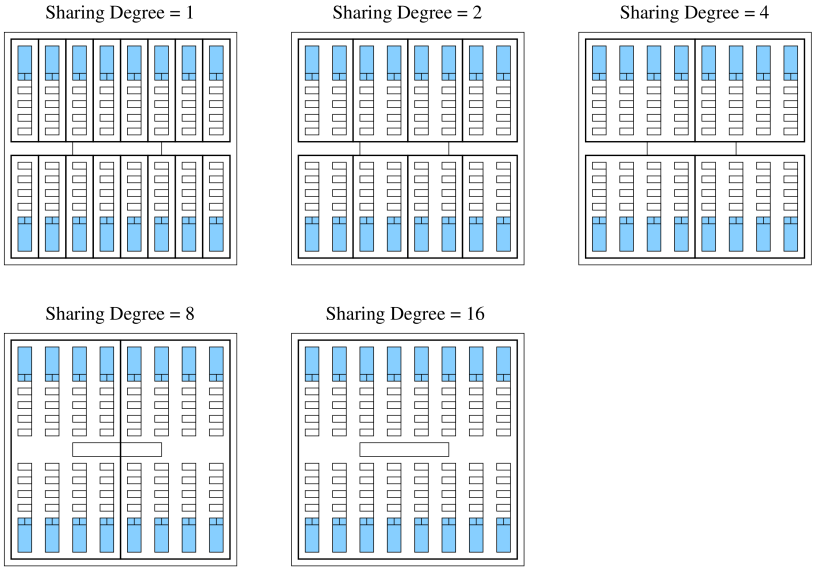


Figure 2.5: Sharing degree: 1, 2, 4, 8 and 16

The sharing degree is an important property of the NUCA design. Higher sharing degrees reduce cache misses, by providing greater effective cache capacity, since there are less data copies on the L2 cache. On the other hand, they lead to longer cache latencies, as the shared cache is larger than the individual private partitions. In addition, the data copies that occur on the different partitions require a L2 coherence mechanism. Thus, choosing the ideal sharing degree is a tradeoff between hit latency, hit rate, inter-processor communication and coherence maintenance overhead. The gap between the two extremes is significant: Huh et al. reported a 54% latency reduction with

private L2 caches and 33% reduction of external memory accesses with a fully shared cache [14].

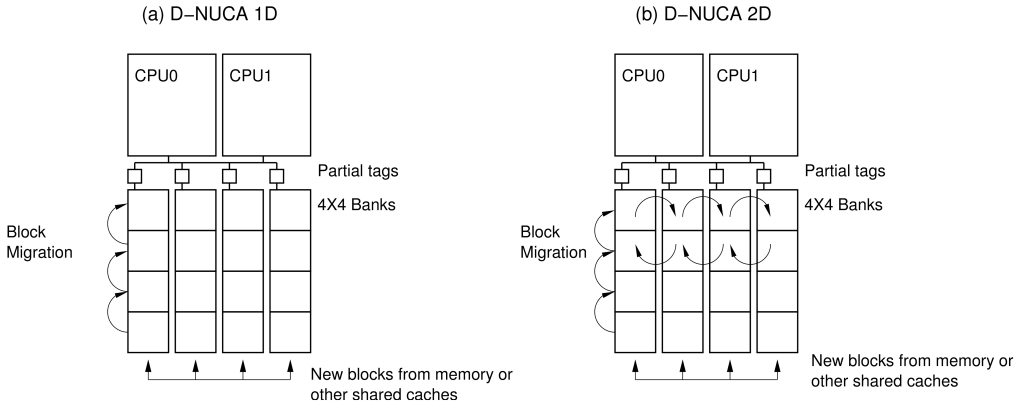


Figure 2.6: D-NUCA Block Migration Policies

In order to achieve both reduced latency and misses, Huh et al. proposed a migration policy and the use of partial tag arrays to speed up the lookup, illustrated in Figure 2.6. The NUCA is logically grouped in columns and each column keeps a special buffer at its end, containing a partial tag array which tracks the state of blocks cached in that column. These partial tags help to detect L2 misses early and reduce the number of requests to banks. Regarding the migration mechanism, two different ones were proposed. The first policy allows blocks to migrate on the vertical direction only (D-NUCA 1D), while the second one allows migration on both directions (D-NUCA 2D), which gives blocks a greater movement freedom but requires a more complex search mechanism. To avoid pointless ping-ponging of data in the case of conflicting promotion caused by different cores asking for the same block, two-bit saturated counters were embedded in the cache tags, which allow a block to migrate only if the relevant counter for that moving direction is saturated. However, both migration mechanisms showed moderate performance improvement.

Huh et al. conclude that no single sharing degree provides the best performance for all the benchmarks. Based on their evaluation, the simplest design seems to be the best: an S-NUCA organization with a sharing degree of two or four. In addition, they point out that considering the complexity of a D-NUCA implementation and the extra energy consumption due to lookups, it is *unlikely* that implementing dynamic migration is justified for CMPs.

However, they also argue that the D-NUCA results still hold promise and research should continue exploring ways to exploit flexible mapping.

2.2.2.2 Replication

As shown in section 2.2.2.1, the consequence of a not fully-shared NUCA organization is that the effective capacity of the L2 cache shrinks and also a L2 coherence mechanism is required. On the other hand, there is also a benefit: access to the replicated data is faster, since the requester can always hit the closest of the copies. This realization led to a new trend in NUCA policies: controlled replication of data.

M. Zhang and K. proposed *Victim Replication* [24], a replication mechanism that handles data in the cache in such a way that the NUCA ends up as a dynamically self-tuning hybrid between private and shared caches. The Victim Replication mechanism is simple. When a processor misses in the *shared* cache, a block is brought in from memory and statically placed - according to its address - somewhere in the shared NUCA, known as the block's global location. The requested block is also directly forwarded to the local bank of the requesting processor. If the local bank's block is later evicted, a copy of the victim block is *probably* kept in the block's global location (if not already evicted), which will reduce subsequent access latency to the same block. A global block is never evicted in favor of a local replica; if that is not possible, no replica is made. Thus the cache's effective capacity does not shrink.

All L1 misses check the local L2 tags first and if a replica is found, the block gets invalidated and moved to its global location. In effect, Victim Replication builds a private victim cache in each local L2 cache, effectively reducing both on-chip communication delay and off-chip traffic. Victim replication manages to combine the advantages of private and shared caches, by implementing a simple and straightforward policy.

Although Victim Replication implements a simple mechanism that improves performance by replicating selected data, its replication policy is static and cannot dynamically adapt to different workload behavior. Beckmann and Wood proposed Adaptive Selective Replication (ASR), a mechanism that dynamically monitors workload behavior to control replication [3]. ASR exploits the fact that the most frequently requested L2 blocks are also the most frequently evicted L1 blocks and focuses on replicating shared read-only blocks, to avoid the overhead of a L2 coherence mechanism.

The choice of the optimal level of replication is a tradeoff between lower hit times for replicated data and greater pressure on the cache, since more replicas reduce its effective capacity. When a L1 cache evicts a shared read-only block and the block is not found in the local L2 cache, the current replication probability determines whether to replicate the block locally. The replication probability is dynamically controlled. ASR monitors the replication's effectiveness and estimates the benefit and cost curves of increasing or decreasing the replication level. The benefit and cost curves are combined to form the *Replication Effectiveness* curve, as illustrated in Figure 2.7, which leads to the final decision for the optimal replication level. The evaluation of ASR showed that it provides better performance to a wider variety of workloads than previously proposed replication mechanisms and both the static shared and private NUCA organizations.

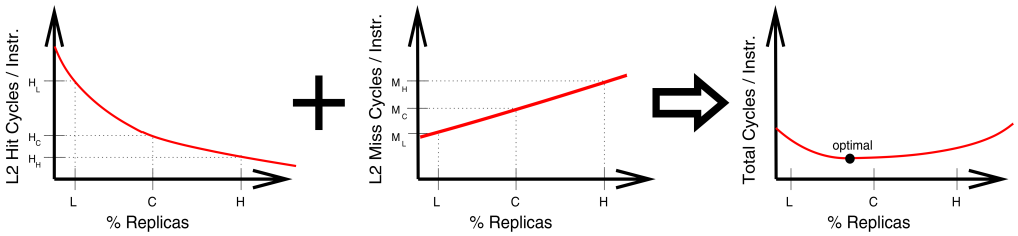


Figure 2.7: ASR computes the potential Replication Effectiveness curve (c) by estimating and combining the replication benefit (a) and cost (b)

2.2.2.3 Dynamic Placement

Optimal placement of blocks in the NUCA is very important, since more local hits can occur, which gets translated into performance. Kandemir et al. [15] claimed that an accurate placement decision cannot be taken during the first allocation of a block in the cache, therefore a migration-based design is needed. The scheme they proposed models the problem of optimal data placement in the L2 cache space as a two-dimensional "post office placement" problem.

Kandemir et al. stress that, whereas the private blocks dominate the requested blocks, most of the block accesses are to shared ones. Consequently, the placement of shared blocks in the caches significantly affects overall performance of diverse applications. Replicating these frequently used shared

blocks would benefit hit times. However, maintaining multiple replicas within the L2 would require the implementation of L2 cache coherence mechanisms, which are undesirable in terms of complexity, energy and network contention. To avoid the overhead of a coherence mechanism, previous works suggested replicating exclusively read-only data. That approach bears two disadvantages: first, correctly and quickly identifying read-only cache blocks is not an easy task and, second, the majority of shared data for some workloads is read-write, thus it cannot be handled by these replication mechanisms. Without duplicating shared L2 cache lines, the only remaining option is to determine a single *ideal* position for each shared line.

To service the majority of cache lines, which are private, Kandemir et al. propose that all blocks should be initially placed in the local bank of the original requester. They rely on the migration mechanism to move the shared cache lines to a better location, later on. To avoid partitioning the NUCA to multiple private caches, thus reducing the utilization of the aggregate LLC space, an interesting *replacement* mechanism is proposed. The metric of *LLC miss intensity* is introduced for each processor, which also characterizes the processor's local tiles, addressed as the processor's cluster.

The LLC miss intensity of a processor is defined as the number of LLC misses that occur during a period of time, e.g. 10 million cycles, and represents the degree of the memory access intensity of the processor's current running thread. According to the miss intensity, which is constantly updated and periodically checked, the processor's cluster is assigned a *priority*. Thus, when a cache line is evicted from a cluster that has high priority, it has higher ability of keeping its victim on-chip by checking more clusters' status and asking one of them to accommodate this victim if possible, as shown in Figure 2.8. On the other hand, if the evicting cluster has the lowest priority, the victim will be directly evicted from the cache. On a last note, a very interesting extension of this priority schema is enforcing Quality of Service for specific applications, by allowing the Operating System or the user to directly set thread priorities, i.e. to set processor clusters' priorities.

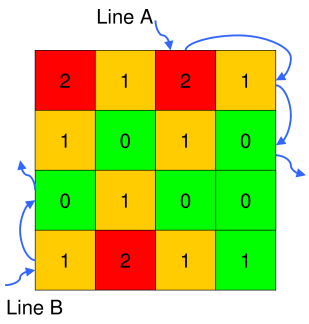


Figure 2.8: Determining the eviction based on cluster priorities

Since data can be placed anywhere in the cache, the lookup mechanism employs a multi-step checking scheme that first checks the local and neighboring

clusters, and then sends requests, if necessary, to remote clusters, until it is determined whether we have an L2 cache hit or miss. The migration mechanism tries to optimize accesses to shared L2 cache lines, by placing the hot lines into ideal positions. To achieve this, each cache line observes the number and origin of requests by processors over a period of time. When the number of total accesses surpasses a certain threshold that characterizes a cache line as "hot", the migration is triggered and the line's target is computed as a weighted median that takes origin and frequency of requests as input.

The evaluation of the design proposed by Kandemir et al. showed that this approach leads to both reduced L2 latencies and IPC improvements on a diverse set of benchmarks.

Many of the proposed NUCA designs can result in banks that have their capacity pushed at its limits, while others banks may be almost empty. Hammoud et al. in their *Cache Equalizer* work [10] proposed a solution to this problem. Temporal pressure at the on-chip L2 cache which is also the LLC, is continuously collected at a group - comprised of cache sets - granularity and periodically recorded at the memory controllers to guide the placement process. Specifically, a pressure array is maintained at the memory controllers of the CMP system. Each slot on the array corresponds to a L2 bank and represents the pressure on that bank, as shown in Figure 2.9.

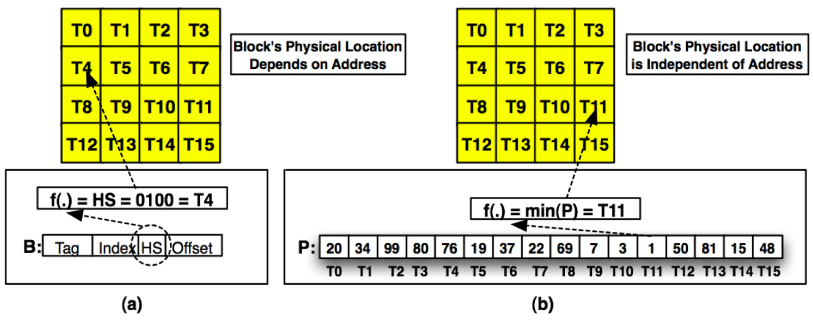


Figure 2.9: Address-based versus pressure-aware placements. (a) Shared scheme strategy. (b) Pressure-aware strategy

On a L2 miss, the main memory is accessed, the pressure array is probed and the block is placed in the tile that exhibits the minimum pressure. Obviously, placing a block to a tile according to the tiles' pressure means that data placement is not statically determined. Thus, it is not possible to mea-

sure cache misses in a meaningful way at L2 banks to keep track of each tile's pressure. Therefore, the pressure value is quantified as the number of lines yielding cache hits during a time interval.

The lookup mechanism implemented was the cache-the-cache-tag (CTCT) [9] policy. CTCT stores two corresponding tracking entries in special tables (TR), at the first-requesting and the static home tiles of the block being placed. Subsequently, when the first-requesting core requests the block again in the future, it locates it quickly with a lookup to the local TR. If another core requests the block, it gets located through the TR of the static home tile, which points to the bank holding the data.

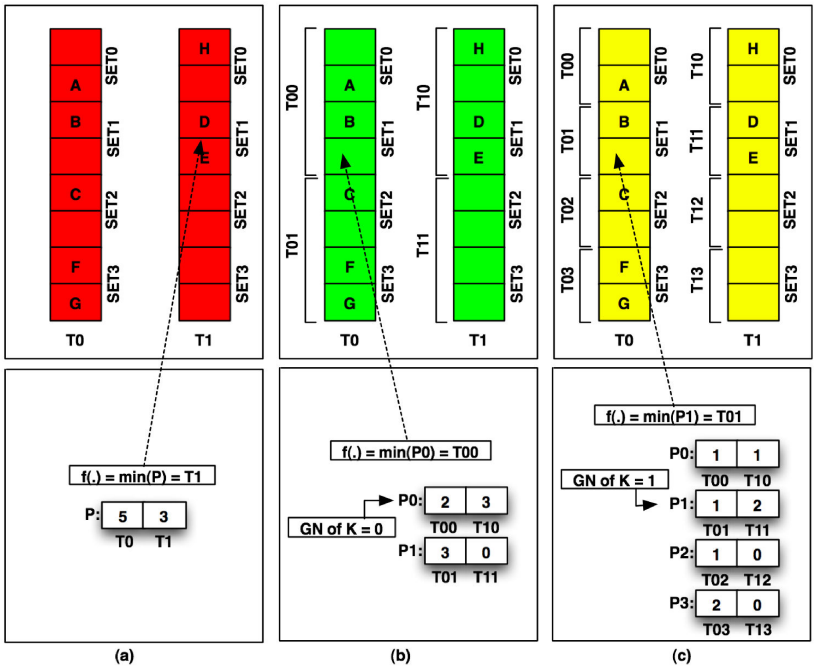


Figure 2.10: Placing block K (with index = 1) using the proposed pressure-aware group-based placement strategy with various granularities. (a) 1-group. (b) 2-group. (c) 4-group

Hummad et al. proposed their own pressure-aware placement by refining the mechanism that keeps track of the banks' pressures. They noticed that collecting pressures at a bank granularity might be relatively imprecise. Therefore, they divided each cache bank into a number of groups in order to gather more detailed, and thus more accurate, pressures from individual sets or groups of sets. Figure 2.10 demonstrates the group-based placement strategy

using different granularities. Over the evaluation procedure, Cache Equalizer was tested with various group granularities and further optimizations using a full system simulator. The results showed that Cache Equalizer outperforms all previously proposed pressure-aware designs and reduces the cache misses of a shared NUCA design by an average of 13.6% and by as much as 46.7%.

Hardavellas et al. [11] recognize that access to shared blocks may benefit from migration, but the complex lookup mechanism that a migration mechanism demands is unjustifiable. Instead, in their *Reactive NUCA* work, they focus on efficient placement of data. The proposed scheme of R-NUCA is based on the observation that the cache access patterns of a range of server and scientific workloads can be classified into distinct classes, where each class is amenable to different block placement policies. L2 accesses naturally form three clusters with different distinct characteristics: instructions that are read-only and are shared by all cores in server workloads, shared data that are usually read-write and private data.

- Private blocks are prime candidates for allocation near the requesting tile, ensuring low access time. Since they are always used by the same core, no coherence mechanism is required.

- Instructions are prime candidates for replication across multiple tiles and, again, no coherence mechanism is required since they are only being read. However, uncontrolled replication of instructions is undesirable, since that would increase the cache's capacity pressure and the off-chip miss rate. Therefore, replication is done at a coarser granularity: neighboring slices of the NUCA are logically divided into *clusters*, and replication of instructions is being done at the granularity of clusters instead of individual L2 slices.

- The rest of the shared blocks, i.e. read-only non-instructions and read-write, could also benefit from replication, but that would require a coherence mechanism which, as already stated, is undesirable. Placing this category of data optimally is a challenging problem. However, in server workloads on which R-NUCA focuses, shared data blocks are universally accessed. Therefore, R-NUCA distributes shared data evenly across all tiles, using standard address interleaving. This way, replication and the need for a coherence mechanism are avoided and lookup is trivial and fast, since a block's address uniquely determines its location.

Classification of blocks. Memory accesses are classified at the time of a TLB

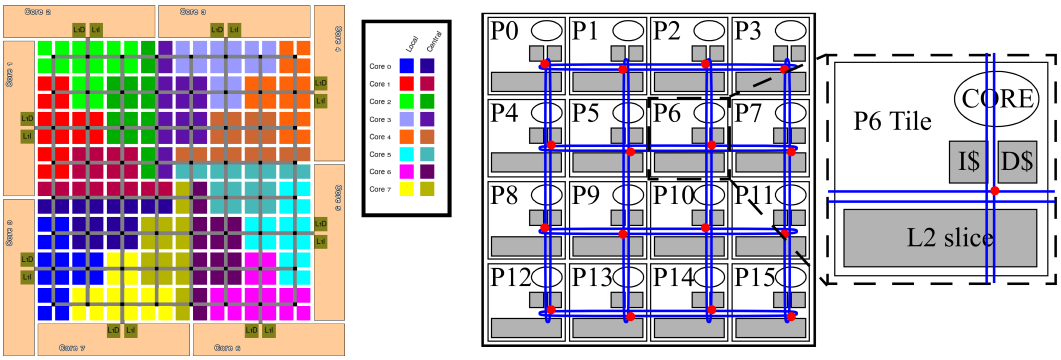
miss. Classification is performed at the OS-page granularity, and communicated to the processors using the standard TLB mechanism. Requests from L1 instruction caches are immediately classified as instructions. All other requests are classified as data requests, and the OS is responsible for distinguishing between private and shared data accesses. Upon the first access, a core encounters a TLB miss and traps to the OS. the OS marks the faulting page as private and the core-ID of the processor is recorded. On a subsequent TLB miss to the page, the OS compares the core-ID in the page table entry with the core-ID of the core encountering the TLB miss. In case of a mismatch, the page is shared by multiple threads an must be reclassified as shared.

In conclusion, R-NUCA avoids block migration in favor of intelligent block placement, eliminating the need for complex lookup algorithms and achieving a 14% performance improvement on average over competing designs, for each workload tested.

Another major difference between R-NUCA and the other works presented here, is that R-NUCA assumes a *tiled* CMP architecture. As shown in Figure 2.11b, the chip consists of multiple tiles, each comprising a processor core, L1 instruction and data caches, a shared-L2 cache slice and network router/switch, which are replicated to fill the die area. On the other hand, the most commonly used NUCA is a multi-banked cache, as the one shown in 2.11a. The cache is split into a lot more banks than the existing processors, thus there are banks that are local to a processor and others that are placed in the center, equally far from all of the chip's processors. Featuring such a great number of banks imposes greater design and control complexities but at the same time provides greater flexibility. For instance, since there are banks that do not belong to any processor, i.e. they are not local to any, are the ideal location for data that is equally shared by all cores.

2.2.2.4 Other approaches for NUCA designs

The *Migration Prefetcher* by Lira et al. [18] introduces an important twist in the migration mechanism that differentiates it from previous proposals. Standard migration attempts to move data to its optimal position within the cache, but still almost half the hits occur within non-optimal banks, since a number of hits must occur before migration is triggered. Inspired by the functionality of the typical data prefetchers, Lira et al. complement the migration scheme with a prefetching technique, in order to recognize access patterns and foretell data migrations, moving data blocks closer to their requester



(a) Multi-banked NUCA

(b) Typical tiled architecture

Figure 2.11: NUCA Architectures

in advance of them being requested. Regular prefetchers and the Migration Prefetcher are orthogonal mechanisms that can be used simultaneously.

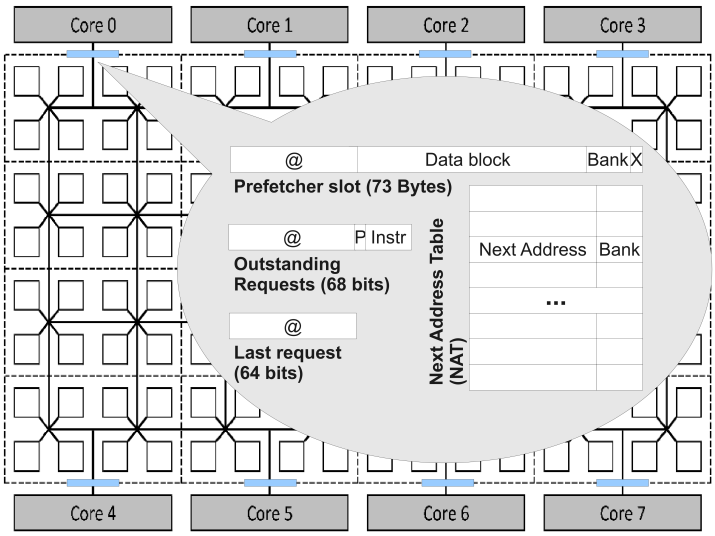


Figure 2.12: Additional structures introduced to enable migration prefetching

The additional structures introduced to enable migration prefetching are shown in Figure 2.12 and include the *prefetcher slot* (PS), which stores the prefetched data, a structure to manage the *outstanding prefetching request* and the *last request* sent from the L1 and the *Next Address Table* (NAT), which keeps track of the data access patterns. For each address requested, the NAT stores the next address to be accessed. The prefetching technique is speculative: it keeps track of access patterns in the NAT and when a known

pattern starts, the prefetcher predicts the next address and then fetches it in the PS. If the prediction was correct, the latency of the access to the prefetched data will be reduced, since it will hit in the local PS instead of a remote bank. When there is a hit in the prefetcher, the cache controller sends the data block to the L1 and then notifies the owner bank that data should migrate one step closer to the requesting core. Lira et al. used *gradual promotion* [4, 17] in their work, but the Migration Prefetcher can be implemented in conjunction with any other migration technique.

The prefetcher is also used to predict the position of data in the NUCA. Each line in the NAT keeps track of the last bankcluster that held the data belonging to the corresponding address. Thus, whenever a lookup is required, a request is sent only to the local bankcluster and the bankcluster suggested by the NAT entry, significantly reducing the network-on-chip traffic as opposed to the contention that a multicast would cause. This lookup method achieved an average accuracy of 70% over the benchmarks selected.

The evaluation procedure showed further that a realistic implementation of the Migration Prefetcher reduces the average NUCA latency by 15%, which translates into an overall speedup of 4% across all benchmarks tested and up to 17% for some. A significant reduction in dynamic energy consumption is also achieved.

2.2.3 To migrate or not to migrate?

As we have seen throughout section 2.2, the first works on NUCA designs investigated migration mechanisms and concluded that an efficient migration mechanism holds great promises for performance improvement in NUCAs [7, 17]. Many later works supported this suggestion and focused on developing efficient migration mechanisms [4, 18, 19]. On the other hand, it was claimed that the performance gains by migrations in the NUCA do not justify the complexity and energy overhead of implementing a migration mechanism [11, 14]. One of the aims of this study is to contribute to the clarification of this still ongoing controversy.

Chapter 3

Tools used

Computer architects have long relied on software simulation to evaluate the functionality and performance of a proposed design. Furthermore, in order to realistically estimate whether the proposed design will be efficient in real-life applications, widely accepted benchmarks are used for its evaluation. In this chapter, the simulation tools and benchmarks that are used in this study will be reviewed.

3.1 Introduction

Software simulation is the only solution for computer architects to evaluate a new design without having to proceed to a time-consuming and expensive hardware implementation. Unfortunately, though, this poses a great bottleneck, since cycle-accurate simulators are several orders of magnitude slower than real hardware. In addition, the growing levels of integration in chip design result in significant increases in computer system size and complexity, thus also increasing the simulator's complexity and the simulation's duration. For instance, simulating a multiprocessor instead of a uniprocessor system does not only suffer the direct overhead of a larger number of simulated cores. It is also essential that additional, complex mechanisms are accurately simulated, such as the cores' interconnection and the coherence mechanism.

Moreover, in order for benchmarking software to provide reliable results, a simulator must be as accurate as possible. This means that simulating the system's basic structures, such as the processors and memory hierarchy, is not enough. A full-system simulation is required, including the simulation of peripheral devices and OS code.

For the above reasons, the simulation throughput is very low, which makes cycle-accurate, full-system simulations especially prohibitive for large-scale multiprocessor systems, because the simulation turnaround for these systems grows almost exponentially with the number of processors. Slow simulation has barred researchers from attempting complete benchmarks and input sets or realistic system sizes on detailed simulators.

3.2 SimFlex

The *SimFlex* project [1, 12] was launched by Carnegie Mellon's CALCM team, in order to provide efficient solutions to the afore-mentioned problems of software simulation. It has developed simulation tools as well as a measurement methodology to enable fast, accurate, and flexible performance evaluation of uni- and multiprocessor systems running unmodified commercial applications. SimFlex is proceeding along two synergistic fronts:

- *SMARTS* applies rigorous statistical sampling theory to reduce simulation turnaround by several orders of magnitude, while achieving high accuracy and confidence in estimates.
- *Flexus* is a powerful and flexible simulator framework that allows full-system simulation that relies heavily on well-defined component interface models to facilitate both model integration and compile-time simulator optimization.

SimFlex combines SMARTS' sampling, to choose application subsets for measurement, with reusable checkpoints of system state, to enable rapid simulation of the selected measurement. Together, these techniques enable 10,000 times reduction in simulation time relative to cycle-accurate simulation without sampling and up to 1000-way simulation parallelism over a cluster of simulation hosts.

3.2.1 SMARTS

The Sampling Microarchitecture Simulation (SMARTS) framework is an approach to enable fast and accurate performance measurements of full-length benchmarks [23]. It prescribes a statistically sound procedure for configuring a systematic sampling simulation run to achieve a desired quantifiable confidence in estimates. The SMARTS framework was developed in order

to address the issues responsible for the intolerably long software simulation durations. It manages to do so, by selectively measuring in detail only an appropriate benchmark subset.

Unlike prior approaches to simulation sampling, the SMARTS framework prescribes an exact and constructive procedure for selecting a minimal subset from a benchmarks instruction execution stream to achieve a desired confidence interval. It uses a measure of variability, i.e. the coefficient of variation, to determine the optimal sample that captures a programs inherent variation. An optimal sample generally consists of a large number of small sampling units. Unbiased measurement of sampling units as small as 1000 instructions is possible by applying careful functional warming – maintaining large microarchitectural state, such as branch predictors and the cache hierarchy – during fast-forwarding between sampling units. Figure 3.1 graphically illustrates how SMARTS functions. The simulation has three distinct phases:

1. Fast functional-only simulation of $[U(k-1)-W]$ instructions
2. Detailed simulation of W warming instructions, without measurement
3. Detailed simulation and measurement of U instructions

As shown in [23], SMARTS' primary contributions are the following:

- **Optimal sampling:** Simulations using the SMARTS framework achieve an average error of only 0.64% on CPI and 0.59% on EPI (energy per instruction) by simulating fewer than 50 million instructions in detail for each of the 41 SPEC2K benchmarks tested.

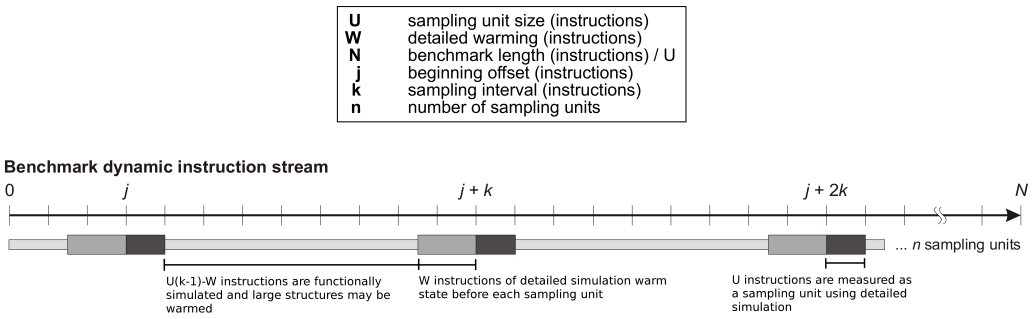


Figure 3.1: Systematic sampling in SMARTS

- **Simulation speedup:** On a 2 GHz Pentium 4, SMARTSim can achieve average speedups of 35 and 60 relative to *sim-outorder* for 8-way and 16-way super-scalar processor models, respectively. Sim-outorder was one of the most widely used simulators at the time the SimFlex project was launched.

3.2.2 Flexus

Flexus is a family of component-based C++ computer architecture simulators that enables full-system timing-accurate simulation of uni- and multiprocessor systems running unmodified commercial applications and operating systems.

Flexus encompasses both a simulation infrastructure and default simulation models. A simulator is composed of individual modules that are hooked together during compilation. A module is often the equivalent of a single hardware structure - for example, a branch predictor or a cache. A key strength of Flexus is its isolation of components: one implementation of a particular module can be swapped for a different implementation without requiring changes to any other modules.

This flexibility also allows a simulator to be tailored to the needs of a specific research hypothesis. If memory system performance is being evaluated, a simple bandwidth-based processor pipeline might be sufficient. Conversely, a study that closely examines microarchitecture could employ a simple memory model. The Flexus core provides services, such as scheduling and statistics, that are common and useful to all simulators.

Flexus builds on *Virtutech Simics* [20], enhancing its functionality with advanced timing models. Simics enables full-system simulation. It is a simulator that allows unmodified commercial operating systems and applications to boot and run. However, Simics provides only functional simulation; it does not attempt to model the passage of time accurately. Flexus hooks into Simics and monitors the instruction stream that the simulated system would executes. In addition, Flexus can control Simics' timing, so as to model out-of-order effects and speculative techniques.

Flexus is designed to support the simulation sampling and checkpointing methodologies developed by the SimFlex research project. The keys to this support are *flexpoints*, checkpoints that store the snapshots of the state of

Flexus components alongside Simics checkpoints of programmer-visible state. Flexus' component-based design enables easy creation of several components that all model the same hardware at various levels of timing fidelity. These components all share the same format for storing state in flexpoints. This way, a simple, fast simulator can rapidly construct a flexpoint library, which can then be measured using a more detailed simulator.

For the purposes of this study, Flexus 4.0.0 is used, which was the latest release at the time. Flexus 4.0.0 includes the following simulation models:

- **Uniprocessor simulators**
  - **UP.Trace:** A non-timing uniprocessor simulator
  - **UP.OoO:** Out-of-order uniprocessor simulator
  - **UP.Inorder:** In-order uniprocessor simulator
- **Multiprocessor simulators**
  - **CMP.L2Shared.Trace:** A non-timing chip-multiprocessor simulator with shared L2
  - **CMP.L2SharedNUCA.Inorder:** In-order chip-multiprocessor, private L1s, shared NUCA L2, based loosely on Compaq Piranha
  - **CMP.L2SharedNUCA.OoO:** Similar to CMP.L2SharedNUCA.Inorder with out-of-order processor cores
  - **CMP.MT4.L2Shared.Trace:** A non-timing chip-multithreaded simulator
  - **CMP.MT4.L2SharedNUCA.OoO:** Out-of-order chip-multithreaded processor with private L1s and shared NUCA L2

There are two distinct categories of simulators: the trace and the timing simulators. Each timing simulator has a corresponding trace simulator. The trace simulators can be used to study cache miss rates but their most significant purpose is to construct flexpoints, which in turn will be used by the timing simulators. Flexpoints reduce timing simulation duration significantly. As illustrated in Figure 3.2, a flexpoint load can replace a very long functional warming period that precedes a measurement point. The flexpoint functionality and creation procedure are further explained in Section 3.3.

In this study, CMP.L2Shared.Trace simulator has been used for flexpoint generation and the CMP.L2SharedNUCA.OoO simulator for the timing simulation. CMP.L2SharedNUCA.OoO is a timing-accurate simulator simulates CMP systems with a shared L2 Tiled Cache (NUCA) as the last cache

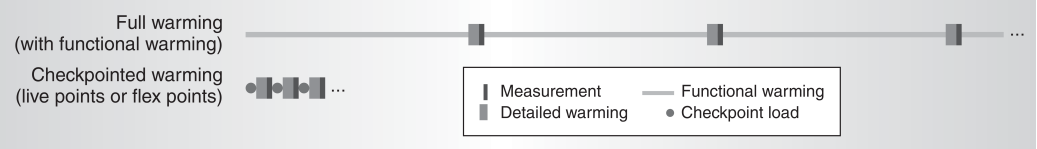


Figure 3.2: Warming approaches for simulation sampling. Checkpointed warming greatly accelerates simulation sampling while maintaining the same accuracy as full warming.

level, using processor cores which support out-of-order instruction execution. The full-system timing simulation includes cores, the whole memory hierarchy and their interconnection network.

3.3 The experimental procedure

Prior to recording any results during a timing simulation, functional warming is required in order to acquire an accurate performance estimation with SMARTS. Functional warming dominates simulation time because SMARTS must functionally simulate the entire benchmark's execution, even though it will simulate only a tiny fraction of the execution using detailed microarchitecture timing models. Functional warming may occupy more than 99 percent of the simulation runtime. Furthermore, functional warming requires simulation time proportional to benchmark length rather than sample size. As a result, the overall runtime of a SMARTS experiment remains constant even when we relax the statistical confidence requirements by reducing the measured sample size.

Flexpoints provide an alternative to functional warming that reduces simulation turnaround time without sacrificing accuracy. A flexpoint stores the necessary data to reconstruct warm state for a simulation sampling execution window upon loading it. Checkpoint implementations of most modern computer architecture simulators have two limitations:

- They don't provide complete microarchitectural model state.
- They cannot scale to the required checkpoint library size (about 10,000 checkpoints per benchmark), which would require multiple terabytes of storage.

Flexpoints manage to address both limitations by storing only selected microarchitectural state in flexpoints. The key challenge lies in storing microar-

chitectural state such that flexpoints can still simulate the range of microarchitectural configurations of interest. Fortunately, with the exception of the branch predictor and memory hierarchy, most microarchitectural state can be reconstructed dynamically with minimal simulation, i.e. a few thousand instructions of detailed warming, and thus needs not be stored. The size of conventional checkpoints is shrinked by three orders of magnitude through storing only the subset of state necessary for limited execution windows in the flexpoints. Figure 3.2 illustrates how flexpoints replace functional warming.

3.3.1 Preparing a new workload

To prepare a workload for simulation, we must investigate its performance variability to design an optimal sample - one that minimizes total simulation time for a desired confidence level. Then we construct a flexpoint library for the optimal sample. The following steps detail how we construct these libraries [22].

1. *Create preliminary sample of flexpoints.* First, we construct a small preliminary sample, for example 30 flexpoints, which we use to characterize the applications variability and warming requirements. Although this sample is insufficient to provide high-confidence simulation results, it typically provides a good estimate of target metric variance.

2. *Determine detailed warming requirement.* The preliminary sample is measured for intervals several times longer than our expected detailed warming at a measurement granularity several times finer than our expectations for sampling unit size. Using these extended, fine-grained measurements, we perform the empirical warming analysis illustrated in Figure 3.3, which helps us determine the required length for the detailed warming.

3. *Create flexpoint library.* Now that we have a desired sample at hand, we can launch the flexpoint creation to spread the final sample over a known-representative execution interval. If the workload we investigate has regions of execution with different characteristics, we have to make sure to create a range of flexpoints over all of these different regions, in order to capture the full spectrum of the workload's behavior. Flexpoint generation can be done using one of Flexus' Trace simulators. Since all of our workloads are CMP workloads, we used the CMP.L2Shared.Trace simulator for this matter.

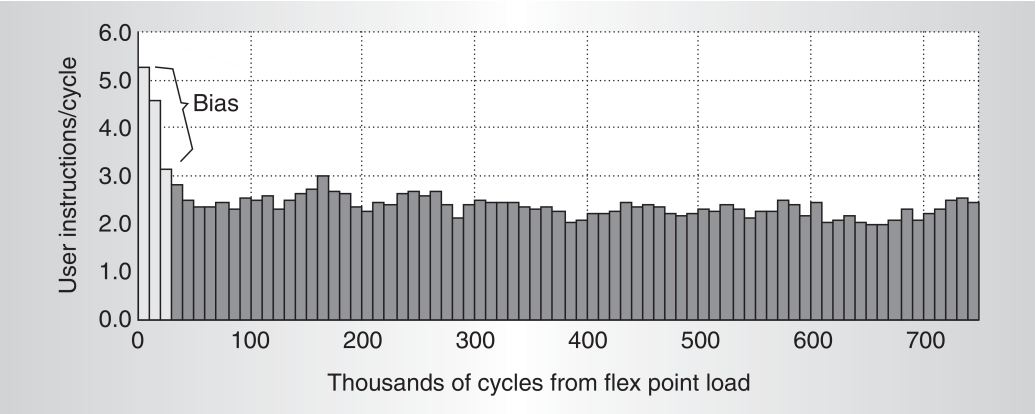


Figure 3.3: Empirical warming determination for the OLTP (DB2) benchmark. Each bar in this graph represents the mean User-IPC from 50 sampling units. The sampling units are 10,000 cycles long, and their offset from a flexpoint load is plotted on the x-axis. This benchmark requires at least 30,000 cycles of detailed warming.

In this study, there was no need to follow steps 1 and 2, since the SimFlex team already provides the empirical sample size that suits the workloads we tested. For the workloads we used, the proposed values are:

- 25,000,000 cycles interval between flexpoints.
- 50,000 cycles of sampling after 100,000 cycles of detailed warming for each flexpoint.
- A variable number of flexpoints, depending on the specific workload, in order to capture all the execution phases.

Having created a flexpoint library for each workload we investigate, we can now repeatedly run timing simulations, using the same flexpoints. The timing simulator, the CMP.L2SharedNUCA.OoO for instance, loads the state of the simulated system from a flexpoint, executes a very short functional warming period and very quickly enters the sampling region. In addition, every flexpoint is independent; it can be loaded and used without any interaction with the rest of the flexpoints of the flexpoint library. This allows massive simulation parallelism: a whole timing simulation that consists of hundreds of flexpoints can be distributed over many host machines to be executed in parallel. This way, a timing simulation only takes a few minutes instead of hours or even days that a classic timing simulation with the full

warming phase would last.

The need for a new flexpoint library emerges only if we drastically alter the microarchitecture configuration; for instance, if we change the size of the caches. Timing parameters can be freely altered without a need for new flexpoints. Thus, flexpoints do not only significantly reduce the timing simulation's duration, but are also highly reusable.

3.4 Benchmarks

A benchmark is a program that is used in order to assess the relative performance of a proposed configuration. Benchmarks provide a method of comparing the performance of different system architectures and configurations.

3.4.1 Common benchmark classes

Applications can be categorized according to their characteristics and system requirements. Therefore, a wide range of benchmarks exists, in order to represent this range of different applications. To fully evaluate a proposed design, benchmarks from different categories should be used, to ensure the design's efficiency on a wide range of applications.

A few popular categories of benchmarks are:

- *Generic CPU performance benchmarks.* This is the most general-purpose benchmark class. The widely used SPEC CPU suite is its primary representative [2].
- *Recognition, Mining, and Synthesis benchmarks (RMS) [8].* A class of benchmarks for evaluating various aspects of modern high performance systems, such as thread-level parallelism and transactional memory. A popular benchmark suite that includes RMS benchmarks is PARSEC [6].
- *Scientific benchmarks.* This class features applications performing highly intensive scientific calculations.
- *Server & Database benchmarks.* Used to evaluate a system handling a great load of transactions targeting a server or DBMS such as Apache, Zeus, Oracle and IBM DB2.

3.4.2 Benchmarks for Flexus

Flexus comes with a set of of general purpose, scientific and server benchmarks. The ones used in this study are described in this section.

3.4.2.1 General purpose benchmarks

Flexus uses the **SPEC CPU2000** benchmark suite [13]. SPEC CPU2000 is an industry-standardized CPU-intensive benchmark suite. SPEC designed CPU2000 to provide a comparative measure of compute intensive performance across the widest practical range of hardware. The implementation resulted in source code benchmarks developed from real user applications. These benchmarks measure the performance of the processor, memory and compiler on the tested system. CPU2000 has nowadays retired and has been replaced by CPU2006 in 2007. Flexus 4.0.0, however, comes with preconfigured CPU2000 benchmarks. We used four SPEC CPU2000 benchmarks combined on a CMP to evaluate the system's performance on multi-programmed workloads:

- **gcc:** An integer benchmark based on gcc Version 2.7.2.2. It generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled. gcc has had its inlining heuristics altered slightly so that it will spend more time analyzing it's source code inputs, and use more memory.

- **twolf:** The TimberWolfSC placement and global routing package is an integer benchmark and is used in the process of creating the lithography artwork needed for the production of microchips. Specifically, it determines the placement and global connections for groups of transistors, known as standard cells, which constitute the microchip. TimberWolfSC has been customized for SPEC to often create cache misses.

- **mcf:** A benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation. The program is written in C, the benchmark version uses almost exclusively integer arithmetic. It is designed for the solution of single-depot vehicle scheduling (sub-)problems occurring in the planning process of public transportation companies.

- **art:** The Adaptive Resonance Theory 2 neural network is a floating-point application, used to recognize objects in a thermal image. The objects are a helicopter and an airplane. The neural network is first

trained on the objects. After training is complete, the neural network attempts to match the learned images with similar objects in the scan-field image.

3.4.2.2 Server benchmarks

Flexus features a wide range of server workloads, since the SimFlex development team's research focuses on servers. Server applications have certain characteristics:

- multi-threaded execution that runs on multi-processor architectures
- large memory footprint
- intensive I/O
- performance of the Operating System matters
- client-server schema
- complicated workload setup & tuning
- non-deterministic behavior

The SPEC CPU benchmark suite satisfies none of these characteristics, thus it cannot be used for server performance evaluation. Therefore, Flexus uses a server benchmark suite, which includes an IBM DB2 and an Oracle DBMS running the TPC-C OLTP benchmark, an IBM DB2 running the TPC-H DSS benchmark, an Apache and a Zeus Web server running SPECweb99:

- **TPC-C** is online transaction processing (OLTP) application that simulates a complete computing environment where a population of users executes transactions against a database. OLTP is a class of program that facilitates and manages transaction-oriented applications, typically for data entry and retrieval transactions in a number of industries, including banking, airlines, mailorder, supermarkets, and manufacturers. TPC-C is popular for evaluating OLTP performance and is centered around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. We evaluate TPC-C performance on an Oracle database, a relational database management system (RDBMS) from Oracle Corporation.

- **TPC-H** is a decision support benchmark (DSS), i.e. a computer-based information system that supports business or organizational decision-making activities. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. TPC-H illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. We evaluate two out of TPC-H's twenty-two queries on DB2, an RDBMS provided by IBM.

- **SPECweb99** is a SPEC benchmark for evaluating the performance of Web Servers. SPECweb99 continues the SPEC tradition of giving Web users the most objective and representative benchmark for measuring a system's ability to act as a web server. In response to rapidly advancing Web technology, the SPECweb99 benchmark includes many sophisticated and state-of-the-art enhancements to meet the modern demands of Web users. SPECweb99 has nowadays retired and has been replaced by SPECweb2005 in 2005. Flexus 4.0.0, however, comes with preconfigured SPECweb99 benchmarks. We evaluate SPECweb99 performance on two different web servers:

  - **Apache Web Server:** an open-source HTTP server for modern operating systems including UNIX and Windows NT. It provides a secure, efficient and extensible server that provides HTTP services in sync with the current HTTP standards. Apache has been the most popular web server on the Internet since April 1996.

  - **Zeus Web Server** is a proprietary web server for Unix and Unix-like platforms. It is designed to be a high-performance web server, competing with other commercial web servers, while also claiming a high degree of compatibility with the Apache Web Server.

3.4.2.3 Scientific benchmarks

In this study, we used one scientific benchmark, **em3d**, which models the interaction of electric and magnetic fields on a 3D object. The data access patterns of this application are quite irregular, presenting minimal locality. The em3d application takes several parameters:

- *Number of nodes:* Number of electric field nodes and number of magnetic field nodes on EACH processor. Thus, the computation scales as the number of processors scales.
- *Degree of nodes:* average number of nodes that a node depends on
- *Remote probability:* the probability that a given dependence will be remote (owned by another processor)
- *Distance span:* the number of processors "away" that a node with remote dependencies can depend on. For example, a distance span of 3 means that processor 0's remote dependencies may depend on processors 1, 2, 3, n, n-1 and n-2.

3.4.2.4 Benchmark selection summary

The workloads used in this study are summarized in Table 3.1.

OLTP Online Transaction Processing (TPC-C v3.0)	
Oracle	Oracle 10g Enterprise Database Server 100 warehouses (10GB), 16 clients, 1.4 GB SGA
Web Server (SPECweb99)	
Apache	Apache HTTP Server v2.0. 16K connections, fastCGI, worker threading model
Zeus	16K connections, fastCGI
DSS Decision Support Systems (TPC-H)	
Qry 2, 17	IBM DB2 v8 ESE, 480 MB buffer pool, 1GB database
Scientific	
em3d	768K nodes, degree 2, span 5, 15% remote
Multi-programmed (SPEC CPU2000)	
Mix	2 copies from each of gcc, twolf, mcf, art; reference inputs

Table 3.1: Application parameters

Chapter 4

Developed Dynamic Placement Model

This chapter presents the dynamic NUCA policies investigated in this study, as well as the extensions that are required on the basic structure of the simulator for their implementation.

4.1 System architecture

Flexus' timing CMP simulators implement a *tiled* system architecture. The chip is physically organized in tiles, each comprising a processor core, L1 instruction and data caches, a shared-L2 cache slice and a network router/switch, which are replicated to fill the die area. A typical tiled architecture is shown in Figure 4.1. The baseline simulator, places a data block in the L2 cache slice that is statically determined by the block's address. In order to investigate dynamic NUCA policies, this default function had to be extended.

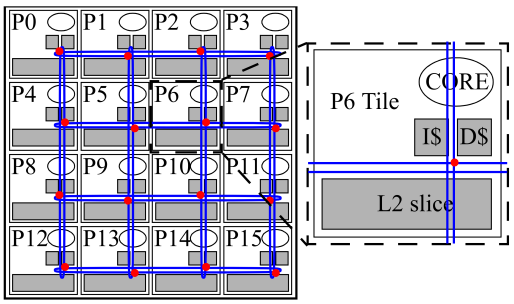


Figure 4.1: Typical tiled architecture

4.2 Motivation

As we have seen in Chapter 2, a static NUCA provides varying access latencies instead of a single, uniform latency for each cache access. The static NUCA design significantly outperforms the classic UCA design, without significant control mechanism overheads. However, it still lacks flexibility, since data is always placed in a single location inside the cache, that is statically determined by its address. Therefore, the full potential of the non-uniform access latencies is not exploited, since accesses to a requester's local, fastest bank are not maximized. To illustrate this, we execute four of the workloads described in Section 3.4.2.4: mix, em3d, apache and Qry17.

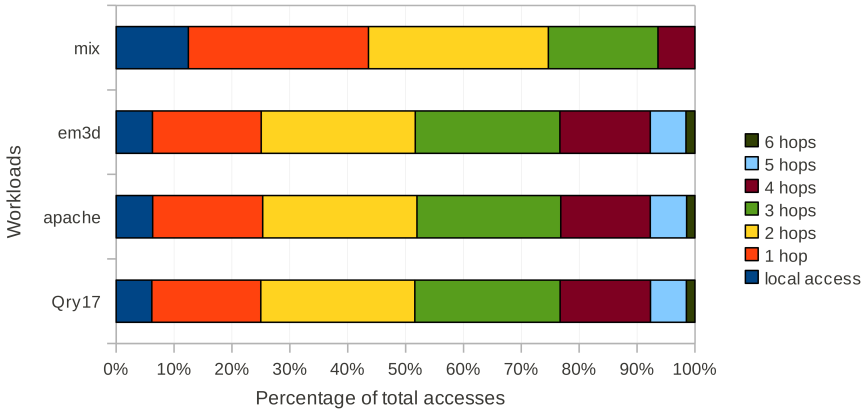
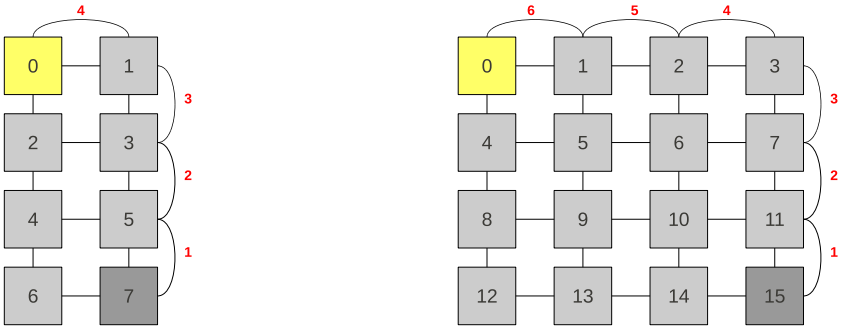


Figure 4.2: Hops required for the accesses to the NUCA

Figure 4.2 shows the distribution of accesses to a shared 16MB L2 NUCA for these workloads. Cache accesses are categorized according to their proximity to the requester. Mix is executed on a simulated 8-core tiled architecture, which means a worst-case access require four hops. Apache, Qry17 and em3d are executed on a 16-core tiled architecture, thus the worst-case access requires six hops. Figure 4.3a illustrates a worst-case access scenario for an 8-core system, where tile 0 asks for a block that is found in tile 7's L2 bank, while Figure 4.3b shows the same case for a 16-core system, where the requested data resides in tile 15's L2 bank.

Figure 4.2 illustrates that remote accesses dominate the total number of accesses: only a 7% of accesses to the L2 on average are local. In other words, the majority of accesses to the L2 cache are not serviced at the minimum latency, i.e. by the local bank. Table 4.1 shows that the high percentage of



(a) 8-core tiled architecture (b) 16-core tiled architecture

Figure 4.3: Worst-case access scenario

remote hits significantly limits NUCA's performance: a L2 access suffers an average latency of up to 2.5 hops. A typical hop latency lays between 4 and 10 cycles, thus each additional hop adds a considerable cost to the overall access latency.

	Benchmark			
	Mix	em3d	apache	Qry17
Local access	12.5%	6.3%	6.3%	6.2%
1 hop	31.1%	18.8%	19.0%	18.8%
2 hops	31.1%	26.6%	26.7%	26.6%
3 hops	18.9%	25.0%	24.9%	25.1%
4 hops	6.4%	15.6%	15.4%	15.6%
5 hops	–	6.2%	6.3%	6.2%
6 hops	–	1.5%	1.5%	1.5%
Avg. hops	1.76	2.5	2.49	2.5

Table 4.1: Impact of remote accesses to the NUCA.

Thus, the static NUCA leaves plenty of opportunities for improvement. Managing to localize the majority of the requests to the NUCA should reduce the average hops required to service a request per cache access. This should translate into a direct reduction of cycles required to service memory requests, which, in turn, should result in a boost to overall system performance. To achieve that, however, we need a more flexible mechanism for placing data in the cache.

4.3 Dynamic NUCA policies

4.3.1 Dynamic Placement policy

The first step towards a dynamic NUCA design is the decoupling of data from its address. Our dynamic placement policy places new data, i.e. data that is fetched in case of an L2 miss, always in the requester's local L2 bank. The motivation behind this decision is our aim to achieve optimal access to private data.

However, this placement policy has an important drawback: the NUCA ends up behaving as a cache hierarchy with private L2 caches. This results in a significant reduction of the effective L2 cache size that each core can utilize. For instance, in a 16-core tiled architecture, each core can only use 1/16 of the L2 cache's total size. We try to mitigate this problem by using an appropriate replacement policy.

4.3.2 Replacement policy

The aim of our replacement policy is to enable tiles to spill data to their neighboring tiles, thus being to able to store each core's data on a greater fraction of the cache. To describe the implemented replacement policy, we introduce the following terms:

- *Neighbors* are the investigated tile's directly adjacent tiles. Thus, each tile has a minimum of two neighbors, if it is a corner tile of the mesh, and, for a 16-core configuration, a maximum of four neighbors. For an 8-core configuration, the maximum number of neighbors is three.
- *Core Request Intensity* (CRI) is the number of a core's requests to the L2 cache as a whole. Core request intensity is used as a metric of how memory-intensive the thread executed by a core is. The higher the core request intensity, the higher we assume the utilization of the core's local L2 bank is.
- *Bank HitRate* (BHR) is the overall hitrate achieved in each bank. This includes all of a bank's incoming requests, regardless of their origin, thus, both remote and local hits are accounted. Bank hitrate is used as a metric of how efficiently the data that resides in a L2 bank is utilized. If a bank has a high hitrate, it is assumed that it can endure a small downgrade in its current effective capacity by evicting some of its own blocks, in favor of its neighbors.

The replacement policy mechanism is triggered every time a new block is brought to the L2 cache from the main memory. The new block replaces the LRU block of the L2 bank, which in turn is either evicted to the main memory, or spilled to a neighbor. This decision depends on our replacement policy, which is based on a two-level criterion:

1. The tile that is evicting the block checks the CRI of its neighboring tiles and compares these values to its own CRI. Neighbors with a lower CRI value than the evicting tile are the candidates to receive the evicted block.

2. Among the candidates that were chosen on the previous step, the *victim tile* that will have to accommodate the evicted block is the one with the highest BHR. An extra limitation applies to the previous procedure: the evicting tile's BHR must be lower than the victim tile's.

In case no neighboring tile satisfies both steps of the replacement criterion, the block is directly evicted from the L2 cache.

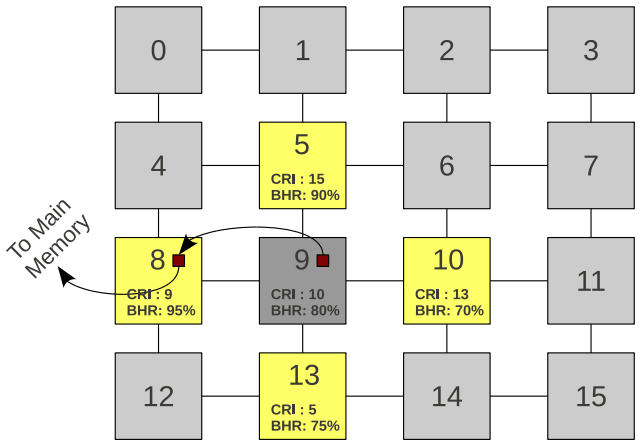


Figure 4.4: The replacement decision

Figure 4.4 shows an example that illustrates the replacement procedure. An LRU block from tile 9, whose CRI is 10, is about to get evicted. Neighboring tiles 5, 8, 10 and 13 are checked, which have a CRI of 15, 9, 13 and 5 respectively. After step 1, only tiles 8 and 13 remain as candidates, since tiles 5 and 10 have a higher CRI than tile 9. On step 2, tile 8 is found to have the highest BHR among the candidates: 95% as opposed to 75% of tile 13 and 80% of tile 9. Thus, tile 8 is picked as the victim tile to allocate tile 9's evicted data. To provide free space for the incoming block, tile 8 evicts the

LRU cached block of the corresponding set.

An additional benefit of this replacement policy is the fact that spilling data to a bank with a higher hitrate makes the policy self-tuning: a bank will stop receiving its neighbors' evicted blocks if it reaches a point that the great number of incoming spills dramatically drops its hitrate.

4.3.3 Lookup mechanism

Designing an efficient and at the same time realistic search mechanism for a dynamic NUCA is not a trivial task. In order to simplify its complexity, placement of data is usually not fully dynamic; a data block cannot be placed *anywhere* in the cache but only in a subset of the NUCA's banks. However, since the aim of this study is to qualitatively evaluate the potential of a fully dynamic design and whether the flexibility it provides justifies the extra complexity, a *centralized directory* was used, in order to solve the lookup problem. This directory includes all addresses of data blocks that are in the NUCA at any time, along with the tile ID where it resides. Thus, before sending any request to the L2 cache level, this directory has to be checked, in order to retrieve the block's location inside the NUCA.

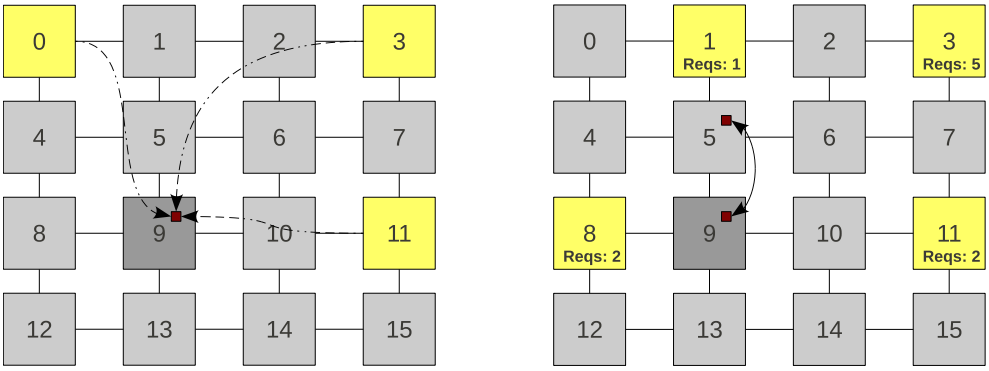
The centralized directory is being used as an ideal lookup mechanism, in order to focus on the rest of the dynamic NUCA policies and evaluate the upper bound of the potential performance improvement.

4.3.4 Migration policy

Implementing a replacement policy that prevents shrinking the available L2 capacity of a core to the size of its local L2 bank gives birth to another challenge. While the effective capacity of the L2 for each core is extended by spilling data to the tile's neighbors, the aspiration of maximizing the number of local accesses remains. Thus, another mechanism is needed, responsible for moving the data closer to its requester's tile, in order to minimize the access time of that core's subsequent requests for the same block. To achieve this functionality, a block migration policy is implemented.

Our dynamic NUCA scheme features a gradual migration policy, which is illustrated in Figure 4.5b and is similar to the one proposed by Beckmann and Wood [4]. In order to estimate the gradual migration's target, we keep

spatial statistics about the origin of the requests for each block. Specifically, two counters are used: a North-South (NS) counter, that keeps track of the average up-down offset of the tile of the requester, relative to the current position of the block and an East-West (EW) counter, that, similar to the NS, keeps track of the average right-left offset. These statistics are being accounted while a block receives requests. The block is free to migrate once there are no on-the-fly requests for it. At this point, the L2 cache controller determines the direction which the most requests came from, by using the information provided by the NS and EW counters. To make this concept clear, an example is given, in Figure 4.5a.



(a) Data block receiving requests from tiles 0, 3, and 11

(b) The gradual migration target decision

Figure 4.5: Migration caused by requests to a block in tile 9

In this example, tile 0 searches for a block that is currently placed in tile 9's L2 bank. The block's NS and EW counters are initially set to zero. The centralized directory forwards the request to tile 9's L2 cache controller, and it also updates the two spatial counters of the block: the NS counter is increased, while the EW is decreased, since the block received a request from a remote tile that is located up and left of the current tile, thus $NS = 1, EW = -1$. Now tile 9's L2 cache controller receives another request for the same block, this time from tile 11. The block's NS counter remains unchanged, while the EW counter is increased ($NS = 1, EW = 0$). Finally, a third request comes from tile 3, which results in $NS = 2, EW = 1$. If tile 9's block is free to migrate after the third request, it will be moved one step upwards, thus to tile 5, since $|NS| > |EW|$ and $NS > 0$.

As illustrated in the previous example, the migration target of a block is the tile that is located one step towards the direction of the most remote hits.

When a block is migrated from tile A to tile B, one of tile B's blocks has to be replaced. Instead of evicting tile B's block in favor of placing tile A's migrating block, the two blocks of the two interacting tiles are swapped. Tile B's block that participates in the migration procedure is not a random block; it is the LRU block of the corresponding set. Figure 4.5b illustrates the migration target decision for a block that is located in tile 9 and has received a variable number of requests from remote tiles 1, 3, 8 and 11. The block will migrate one step to the "north" to tile 5, since previous requests have set the NS and EW counters to 6 and 5 respectively, and it will be swapped with tile 5's LRU block of the same set.

The migration mechanism aims to improve average access latency to shared blocks. Since blocks are dynamically placed in the local bank of their initial requester, a block that is shared by many cores may be initially placed in a non-optimal location. The migration policy will gradually move the shared block to a location where the average access latency for all its requesters is improved.

4.4 Implementation of the dynamic NUCA

In this section, the implementation of the policies analyzed in Section 4.3 are described.

4.4.1 System architecture

For the purposes of this study, Flexus' CMP.L2SharedNUCA.OoO timing simulator was used. This simulator features a CMP system with out-of-order processor cores and a static shared L2 NUCA cache as the system's LLC. Each core has its own L1 data and instruction cache while L1-L2 coherence is enforced by a inclusive MESI protocol. The system's tiles are interconnected in a mesh topology.

4.4.2 The baseline simulator

Figure 4.6 illustrates the life cycle of a memory request that ends up to the L2 cache.

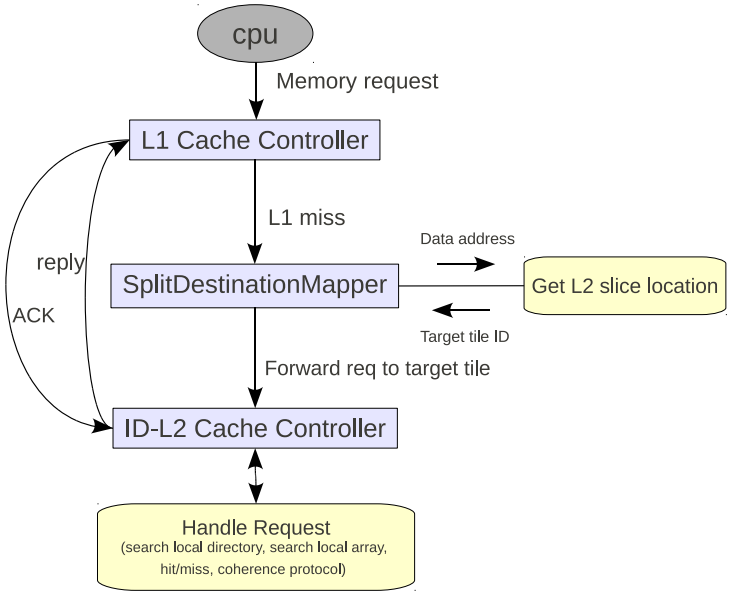


Figure 4.6: Default procedure for a L2 request

After the occurrence of a L1 miss, a packet containing the memory request is forwarded through the interconnection network to the next cache level, i.e. the L2 cache. However, the L2 cache is split into banks and distributed across the chip's tiles, since a tiled architecture is simulated. The decision of *which* tile hosts the L2 bank that should receive the memory request packet must be taken prior to dispatching it to any L2 target. Therefore, it is first sent to the SplitDestinationMapper (SDM) component.

The SDM statically decides, according to the data's address, in which L2 bank the requested data is expected to be found and the request is forwarded to the appropriate tile. The target's L2 cache controller receives the request and the typical procedure of servicing a memory request follows: if the data is not on-chip, hence a L2 miss occurs, a request for the data is sent to the memory. If the data is available, i.e. data is already on-chip or the memory has replied with the desired data, the L2 controller sends a reply to the initial L1 requester. The transaction is considered to be complete once an ACK message is sent from the L1 controller that initiated the request back to the L2 bank that serviced the request.

In the dynamic NUCA we investigate, data location in the L2 is independent of its address and can dynamically change. Therefore, the simulator's default mechanism is not sufficient, and new components have to be im-

plemented in order to support such a flexibility. We extend the baseline CMP.L2SharedNUCA.OoO simulator to create the CMP.L2DynamicNUCA.OoO simulator, which implements our dynamic NUCA scheme.

4.4.3 The centralized directory component

The centralized directory introduced in Section 4.3.3 is implemented as an extension to the baseline simulator, in order to locate data in the NUCA after decoupling its location from its address.

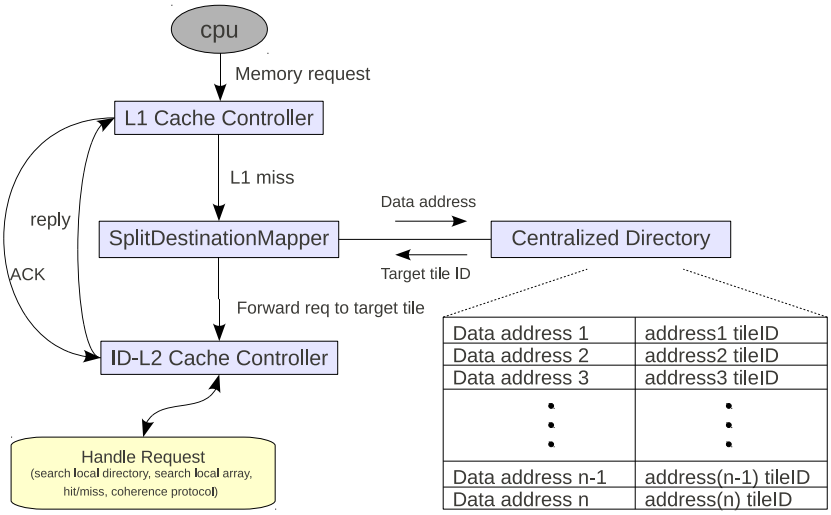


Figure 4.7: The centralized directory is used to keep track of data location in the NUCA

As shown in Figure 4.7, the centralized directory, overrides the static function of the SDM that determines the data location in the L2 cache, making the decoupling of data from its address possible. The centralized directory is implemented as a hash map, therefore lookups are very fast and they do not affect the simulations' duration significantly.

4.4.4 Dynamic placement policy implementation

The centralized directory's functionality makes a fully dynamic placement mechanism possible. Every time the SDM receives a packet with a request to the L2 cache, the data's address is looked up in the centralized directory.

If it is found, the data is already allocated in a L2 bank, so the request is forwarded to the tile hosting that bank. In the opposite case, a new directory entry is created for that address and the request is forwarded to the requester's local L2 bank, i.e. the L2 bank that resides in the same tile as the requesting processor. This request will cause a L2 miss in the local L2 bank and the data will be allocated in it.

The functionality of the centralized directory, and the fact that it is accessed every time before a request is sent from an L1 to the L2 cache, ensures the following:

- If the data is on-chip, it will be found and the request will be forwarded to the data's holder.
- If the data is not on-chip, it will be automatically placed in the requester's local L2 bank.
- No duplicates of data can occur in the L2 cache at any time. Thus, no modifications to the default coherence protocol of the simulator are needed in order to maintain the consistency of data in the L2 cache.

4.4.5 Replacement policy implementation

As described in Section 5.4, our replacement policy makes data spill decisions based on the metrics of core request intensity (CRI) and bank hitrate (BHR). Therefore, three new counters are required per tile. Each tile is enhanced with:

- A counter that keeps track of its core's L2 requests, for the CRI.
- A counter that accounts the number of L2 hits that occurred to the tile's L2 bank, and a similar counter for the L2 misses. These counters are needed for the calculation of the BHR.

Every time a block is about to get evicted from a L2 bank, the dynamic NUCA's replacement policy is triggered. The evicting bank checks and compares the core intensity and bank hitrate of itself and its neighbors and the replacement decision is taken, according to the criteria analyzed in Section 5.4. If a data block is spilled to a neighboring tile, the centralized directory is updated with the new location of that block.

The timing of a data block spill to a neighboring tile has not been modeled, as the impact on the system's performance should be minimal anyway. Spilling

data to neighboring tiles is an action that is not on the critical execution path and its only impact might be a slight increase of contention on the tile interconnection network.

4.4.6 Migration policy implementation

Migration is a completely new mechanism to the simulator. Figure 4.8 summarizes the migration policy's function. As described in Section 4.4.2, a transaction is accounted as completed when the L2 cache controller that served a request receives an ACK from the requester's L1 cache controller. After the completion of the transaction, the L2 cache controller of the tile triggers the migration procedure. The mechanism's function is strongly based on functionalities provided by the centralized directory.

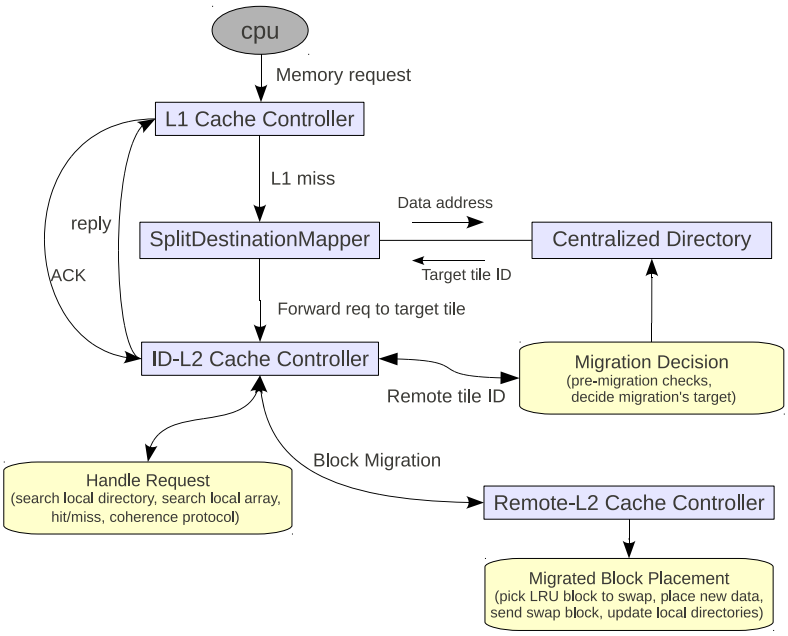


Figure 4.8: Adapted procedure for a L2 request and migration mechanism

The steps followed by a cache controller after the completion of each transaction are the following:

1. Check whether the block is free to migrate. A block is considered to be free for migration only if there are no pending requests for it by any core. This *locking* mechanism is enforced through the centralized directory. Each directory entry that corresponds to a block's address has a

counter. This counter is increased each time a request to the L2 for this block comes from the L1 and decreased by the L2 cache controllers for every ACK received for this block, thus for every completed transaction for this block. This counter is used as a lock: a block is considered to be free to migrate only when this counter gets down to zero. This strategy prevents the unwanted scenario of on-the-fly requests reaching their target L2 bank only to find out that the requested data is not there anymore, because of a preceding migration.

2. For the purpose of our migration policy decisions, the centralized directory keeps some spatial statistics about the origin of the requests for the block. Therefore, each tracking entry of the centralized directory that corresponds to a cache block is augmented with a NS and EW counter, which are updated and used as described in Section 4.3.4.

3. When the block is free to migrate, the two counters, NS and EW, are checked in order to decide the migration's target tile. The target choice depends on the migration policy. Since a one-step-at-a-time gradual migration policy is applied, the block will migrate one hop up, down, left or right, to a neighbouring tile's L2, depending on the values of the NS and EW counters.

The overhead of the data block swap between the two L2 banks required by a migration, has not been modeled. In any case, its impact on overall performance is not expected to be important, since the migration procedure is not on the critical path of servicing a request. When a decision for a migration is taken, the swap happens instantly. This serves two purposes:

- *Simplicity.* No special care needed to be taken for false misses, i.e. misses occurring while requesting a data block at the moment it is on-the-fly, because of a previous migration. To avoid false misses, some additional mechanisms have to be implemented, such as lazy migration [4], which adds complexity to the migration policy design.

- *Ideal migration modeling.* Since migrating blocks has no cost, the upper border of the implemented migration strategy can be evaluated.

4.5 Customizing the flexpoints

The changes to the baseline timing simulator that were described in Section 4.4 are not enough to evaluate the new design in Flexus. As described

in sections 3.2.2 and 3.3, a great contribution of Flexus is that the long functional warming periods are replaced by a checkpointed warming, a flexibility provided by the flexpoints. However, since the functionality of the L2 cache has been drastically changed, the flexpoints used by the default static NUCA timing simulator cannot be used unchanged. Certain changes have to be made to the flexpoint creation and loading procedures to deliver a more consistent state of the cache to the timing simulator, with respect to the dynamic NUCA policies.

Flexpoints for the baseline CMP.L2SharedNUCA.OoO timing simulator are created by the CMP.L2Shared.Trace simulator. The latter, models the L2 cache as a classic, unified cache and runs a fast trace simulation that keeps track of the data traffic in the cache. When the warm-up period of 25 million cycles reaches its end, the trace simulator stores the state of the microarchitectural structures, such as caches and branch predictors, along with the coherence directory state, in special output files. These output files comprise the flexpoint's flexstate.

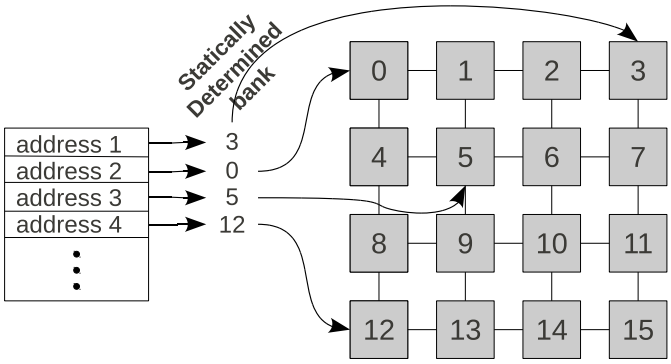


Figure 4.9: The default loading procedure of the L2 NUCA using the flexpoint's saved flexstate

When the timing simulator starts, one of its first actions is to load the system's initial component states from the flexpoint's flexstate files. For the CMP.L2SharedNUCA.OoO simulator, the L2 load procedure includes reading the L2 flexstate file and statically placing data in the cache's banks according to their address. As shown in Figure 4.9, the L2 cache's contents are loaded from the flexpoint and a static function determines each block's location in the NUCA, according to its address.

Bringing the L2 cache to a dynamic-NUCA-consistent state at the beginning of each timing simulation, means approximating the cache state that would occur if we were executing a full functional warming as precisely as possible, using Flexus' checkpointed warming method. Thus, the loading phase had to be adapted in order to be as consistent as possible with our dynamic NUCA's dynamic placement, replacement and migration policies. Therefore, the CMP.L2SharedNUCA.OoO simulator's loading phase for the L2 cache was changed for our CMP.L2DynamicNUCA.OoO simulator, for which the following criteria are applied:

- Data blocks with more than one L1 sharers are placed in the tile that provides the optimal average distance for its sharers. This decision was taken assuming that the migration mechanism would eventually move such a shared block in a location that suits all of the block's requesters.
- Data blocks with no L1 sharers are placed in the tile of their last requester. This information is not available by default in the flexpoint's L2 cache flexstate, therefore the trace simulator was extended to track the last requester for each block and also store this information in its flexstate output files.
- The previous two criteria do not account for the limited capacity of each L2 bank. Thus, a larger number of blocks than a bank's available slots may be directed to a single bank. When a block does not fit in its target bank, it is spilled to the target's closest L2 bank that still has a free slot in the corresponding cache set.

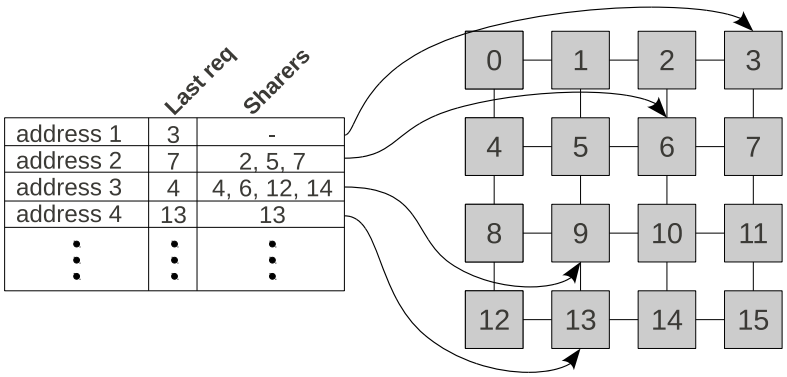


Figure 4.10: The adapted loading procedure of the L2 NUCA using the modified flexpoint's saved flexstate

An example for the new L2 cache loading procedure for the CMP.L2Dynamic-NUCA.OoO simulator is illustrated in Figure 4.10. Data block "Address 1"

does not have any L1 sharers, therefore we decide to place it in the local L2 bank of its last requester, which is core 3. Block "Address 2" has three L1 sharers: 2,5 and 7. Therefore, it is placed in the tile that provides optimal average access for all three sharers. In this case, the optimal choice is tile 6. The same procedure applies for block "Address 3", which has four sharers, 4, 6, 12 and 14, for which the tile that provides optimal average acess is 9. Finally, "Address 4" has only one sharer, therefore it is placed in that sharers tile. Note that the last requester field plays a role in initial placement only when a block has no sharers.

Chapter 5

Experimental evaluation

In this chapter, we evaluate the dynamic NUCA policies described in Chapter 4. We compare our developed dynamic design, implemented by the CMP.L2DynamicNUCA.OoO simulator, to the baseline static shared NUCA design, implemented by the baseline CMP.L2SharedNUCA.OoO simulator.

5.1 Simulated system's parameters

A 16-core architecture was used for the evaluation of the scientific and server workloads, while an 8-core tiled architecture was used for the the multi-programmed workload. The system parameters of the simulated system for both the 8-core and the 16-core CMPs are shown in Table 5.1.

Processing Cores	UltraSPARC III ISA; 2GHz, OoO cores Inorder Execution 8-stage pipeline, 2-wide dispatch/retirement
L1 caches	split I/D, 64KB 4-way, 2 ports 64-byte blocks, 32 MSHRs, 16-entry victim cache
L2 NUCA cache	1MB per core for 16-core system, 2MB per core for 8-core 16-way, 12-cycle hit 64-byte blocks, 64 MSHRs, 16-entry victim cache
Main Memory	3 GB memory, 8KB pages, 90 ns latency
Memory Controllers	one per 4 cores, round-robin page interleaving
Interconnect	2D mesh (4x4 for 16-core, 4x2 for 8-core) 5-cycle hop: 3-cycle link latency, 2-cycle router

Table 5.1: System parameters

5.2 Performance metrics

Throughput application performance is typically reported in terms of transactions per second. However, transactions are too long for Flexus' timing simulators to execute, as shown in Figure 5.1, and their completion rate has a high coefficient of variation. Therefore, another metric has to be used, that is proportional to transaction throughput but has lower variance at smaller measurement sizes is needed.

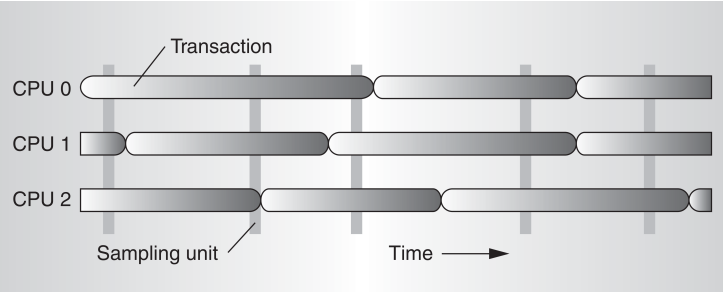


Figure 5.1: Sampling a throughput application

Wenisch et al. [22], observe that although the time to complete a particular number of transactions varies greatly, the amount of work a database or Web server process must perform to complete a certain transaction type does not vary. As a result, the rate at which user-mode instructions complete is linearly proportional to transaction throughput. Thus, they propose that this linear relationship between user-instruction throughput and transaction throughput provides the opportunity to sample *User-Instructions Per Cycle* (UIPC) to assess transaction throughput.

UIPC is defined as the number of commitef user-mode instructions, divided by the total number of measured cycles. The commonly used metric of *Instructions Per Cycle* (IPC) cannot be used because it is not proportional to transaction throughput, as it includes many system instructions that do not contribute to forward progress. Using UIPC as the target metric saves three orders of magnitude in simulation time over using transaction throughput [22].

Performance improvement is reported in the form of speedup, i.e. UIPC achieved by our dynamic design relative to the UIPC achieved by the baseline configuration. In order to reduce the impact of performance variability between flexpoints, the following performance evaluation is applied:

1. For each flexpoint, UIPC is collected for both the baseline and the dynamic configuration.

2. The speedup achieved in each flexpoint is computed.

3. Average speedup is the average of the speedups achieved in all flexpoints.

4. The confidence for the average speedup is computed and represented in the performance graph as an error bar, thus, performance improvement's confidence interval is equal to average speedup ± confidence. With a 95% confidence level and assuming a normal distribution, the confidence interval is:

$$\bar{x} \pm 1.96 \frac{\sigma}{\sqrt{n}}$$

where $\bar{x}$ is the average speedup, σ is the standard deviation of flexpoint speedups and n is the sample size.

5.3 Performance evaluation

Using UIPC as the performance metric, we evaluate the performance of our dynamic design for the selected workload set. Figure 5.2 illustrates the speedup of the dynamic design, compared to the baseline. Our dynamic NUCA achieves a 7.7% average speedup on all workloads and a maximum speedup of 15.6% for the multi-programmed workload. This is expected, since the multi-programmed workload benefits the most from the dynamic placement, as all data used is private.

On the other hand, server workloads are dominated by shared data accesses. Nevertheless, our dynamic NUCA designs improves performance for these workloads too; the proportion of data that is private is accessed fast, while the migration mechanism manages to move shared data to a location that benefits all requesters.

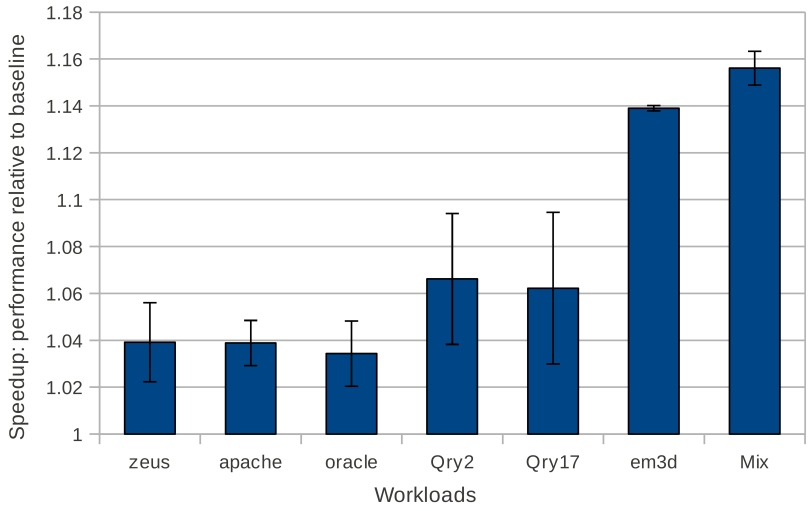


Figure 5.2: Performance speedup for the dynamic NUCA design, relative to the baseline static NUCA

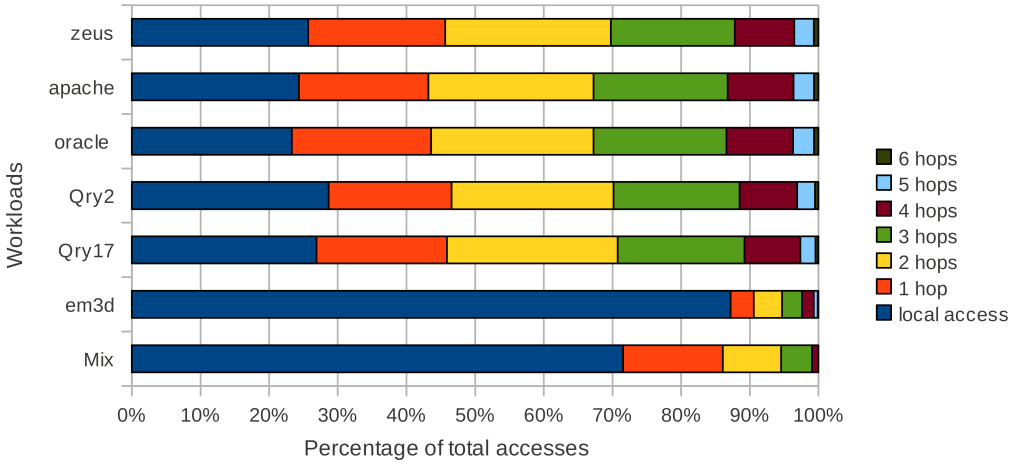


Figure 5.3: Hops required for the accesses to the NUCA in the dynamic design

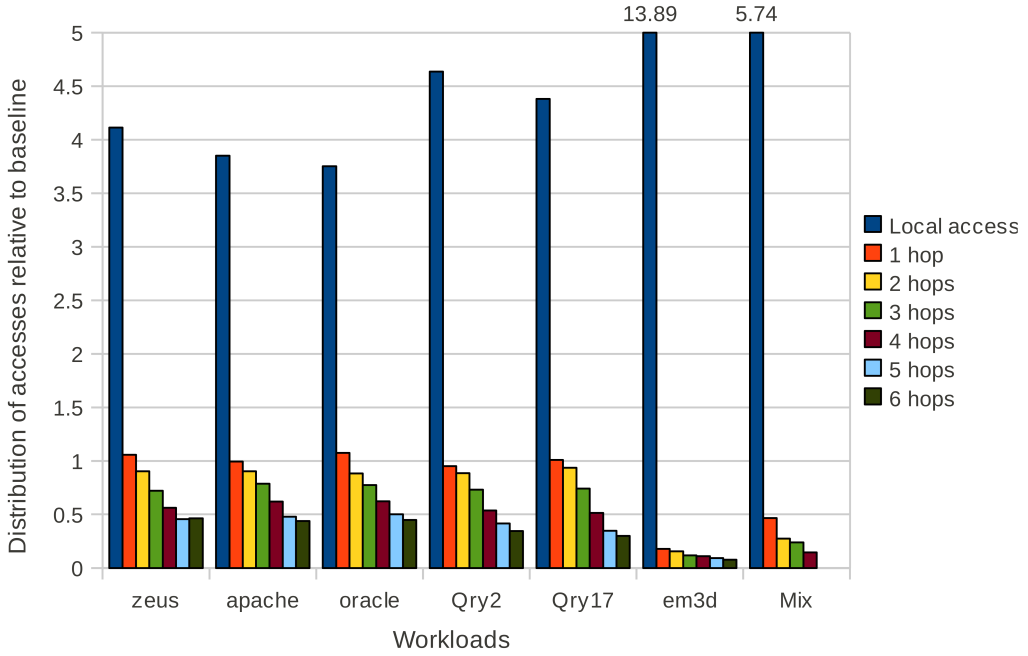


Figure 5.4: Distribution of hops required per L2 cache access, relative to the baseline design

The main reason for the speedups observed in Figure 5.2 is that our dynamic NUCA manages to reduce the average hops required for each L2 access, by increasing the portion of local L2 accesses. Figure 5.3 illustrates the distribution of L2 accesses in the dynamic NUCA design. Local accesses are greatly increased as opposed to the baseline NUCA, presented in Section 4.2, and for Mix and em3d, they represent the majority of L2 accesses. Figure 5.4 shows the distribution of hops required per access for the dynamic design, relative to the baseline.

Table 5.2 shows the average hops per L2 access for the baseline and the dynamic design. Our design achieves a great reduction of the average hops per access; an average of 40.7% among all workloads and a maximum of 87.7% for em3d. This reduction translates into an overall system performance boost.

Workload	Average hops		Avg. hop reduction
	basic	dynamic	
Mix	1.76	0.49	72.2%
em3d	2.50	0.31	87.7%
oracle	2.49	1.83	26.6%
Qry17	2.50	1.70	32.0%
Qry2	2.50	1.70	32.0%
apache	2.49	1.84	26.2%
zeus	2.49	1.75	29.6%

Table 5.2: Comparison of average hops per L2 cache access between the baseline and the dynamic design

5.4 Replacement policy evaluation

The implemented replacement policy spills data to the evicting tile's neighbors, in order to increase the effective L2 capacity that each thread running on a core can utilize. The metrics used for making a spill decision are *core request intensity* (CRI) and *bank hitrate* (BHR), introduced in Section 5.4, as well as *core hitrate* (CHR) which is defined as a core's ratio of L2 hits to L2 misses. This additional metric expresses the reuse of data for a specific thread that runs on a core.

Using these three metrics, eight different policies were evaluated:

- *Policy 1:* Among the neighbors with a lower CRI pick the one with the highest BHR.
- *Policy 2:* Directly evict the data block or pick a neighbor at random.
- *Policy 3:* Find two neighbors with the lowest CRI and pick the one with the highest BHR.
- *Policy 4:* Pick neighbor with the highest BHR.
- *Policy 5:* Pick neighbor with the lowest BHR.
- *Policy 6:* Find two neighbors with the highest BHR and pick the one with the lowest CRI.
- *Policy 7:* Pick neighbor with the highest CHR.
- *Policy 8:* Pick neighbor with the lowest CHR.

Our dynamic design, evaluated in Section 5.3, features replacement Policy 1. Figure 5.5 illustrates the performance for all policies and workloads, relative to Policy 1. Figure 5.6 gives a cumulative overview of the average speedup achieved on all workloads for each policy, relative to the average speedup achieved by Policy 1. Policy 5 seems to have an almost negligible advantage over Policy 1 in average. However, Policy 5 always picks the neighbor with the lowest BHR to spill data. That way, a certain bank may be constantly burdened with other banks' data that is not reused, e.g., data from a streaming application, thus further lowering its BHR. That would result in a trashing of the cache with useless data. Therefore, we prefer Policy 1 that is self-tuning: a bank will stop receiving its neighbors' evicted blocks if it reaches a point that the great number of incoming spills dramatically drops its BHR.

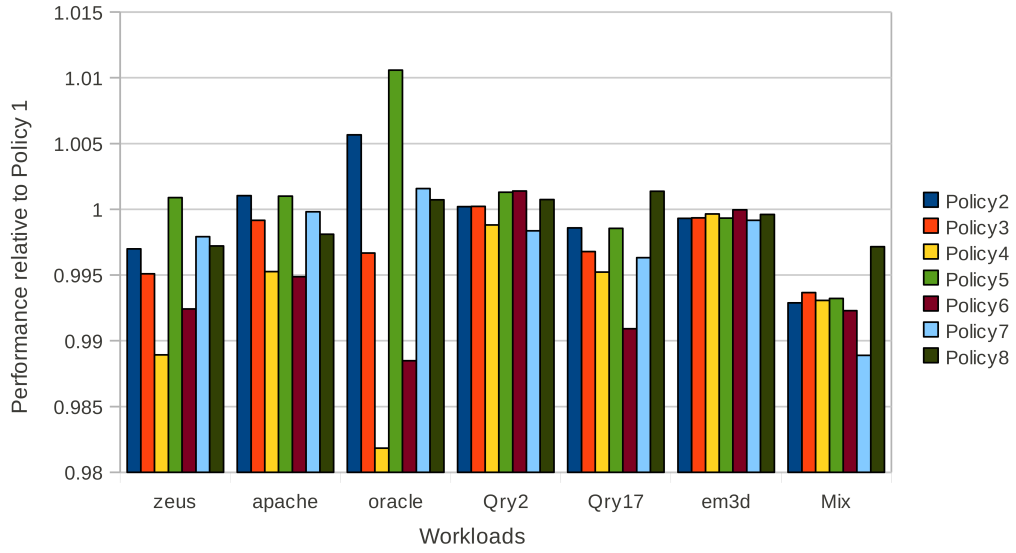


Figure 5.5: Replacement policies performance, relative to policy 1

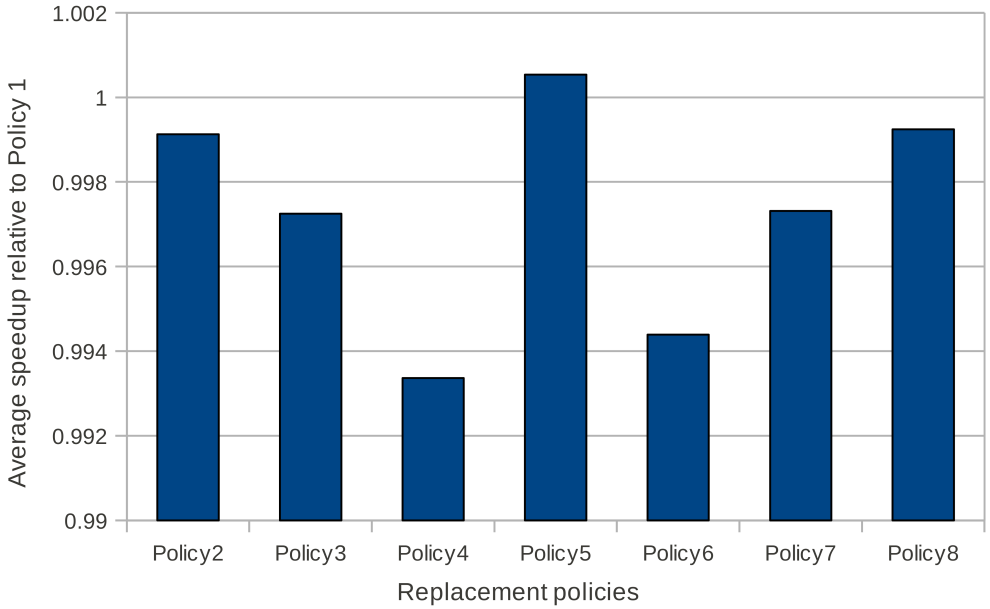


Figure 5.6: Average performance speedup for all replacement policies, relative to the average performance speedup achieved by policy 1

However, Figures 5.5 and 5.6 imply that choosing any of the compared replacement policies has a very small impact on overall performance. The fact that even the Random spill policy achieves similar results suggests that the criteria we use may be not sufficient to make the best spill decisions. A more insightful spill decision would require statistical data collected over monitoring periods, during execution. For instance, Kandemir et al. [15] propose a replacement policy, similar to the one we used in our dynamic NUCA. However, they decide data spills according to each tile's core miss intensity, after monitoring periods of 10 million cycles. Unfortunately, Flexus' timing periods only last a short period of 150,000 cycles and, therefore, we are unable to implement policies that require such extensive monitoring periods.

5.5 Overhead estimation

Our dynamic design achieves performance improvements for all workloads. However, it assumes a practically unrealistic component, the centralized directory. In this section we will try to estimate and address its cost.

The hardware overhead of the centralized directory is not prohibitive. Implementing a basic centralized directory in hardware, without the extra counters introduced for the migration policy, practically means duplication of the tag array, plus $\log_2 N$ bits per entry to store the ID of the data's location, for a N-tiled architecture. On a 64-bit, 16-core system with a 16 MB 16-way associative L2 cache with 64-byte blocks, the size of the centralized directory is about 1.5 MB. This translates to a hardware overhead of less than 10%, which is tolerable, if the benefits we get in turn are great.

The most unrealistic assumption that has been made is that a block is always located in the NUCA through the centralized directory, at no cost.However, the lookup procedure interferes with the critical path of servicing L2 requests. To approach a more realistic design, we assume that each lookup in the centralized directory, has a certain cost in cycles. The centralized directory can be placed in the center of the CMP, where it will be reachable by all tiles in one hop. In the timing simulations, the per-hop latency between tiles is set to 5 cycles, consisting of a 3-cycle channel latency plus a 2-cycle router latency. Assuming the same channel cost for reaching the centralized directory, no router latency, since we have point-to-point communication, and a lookup latency of 2 cycles, the overall cost of retrieving the location of a block from the centralized directory is estimated at 8 cycles. This "naive" access design is illustrated in Figure 5.7.

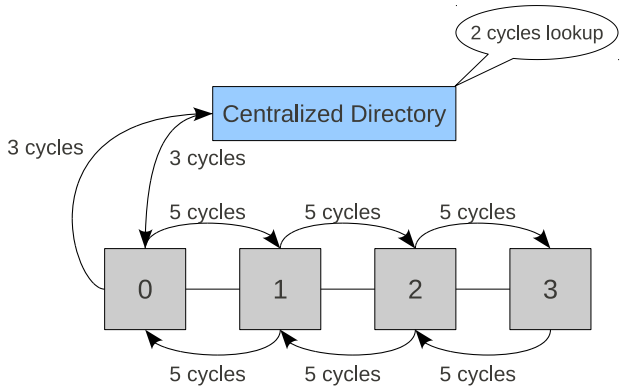


Figure 5.7: Network latencies for the naive access design

Since our dynamic placement policy manages to significantly increase local bank L2 hits, an improved approach would be accessing the local tile's L2 bank in parallel with dispatching the request to the centralized directory. Figure 5.8 shows the performance of the naive and the local parallel access

designs relative to the baseline performance. Performance for the dynamic design without any cost estimation, which was presented in Section 5.3, is also included in Figure 5.8, named as "no cost", for comparison. Approaching the cost of the centralized directory with the naive design, we observe an important performance degradation. The parallel local access design is noticeably better than the naive, but is still significantly outperformed by the no cost centralized directory.

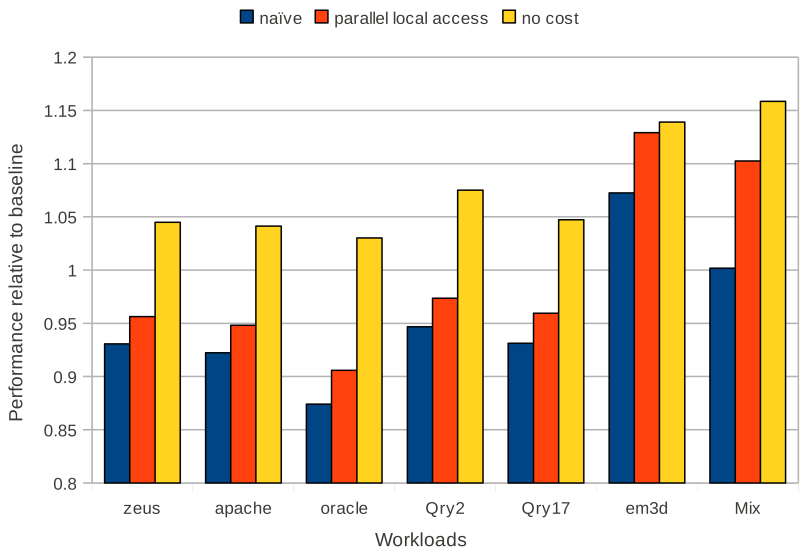


Figure 5.8: Performance of the dynamic NUCA design relative to the baseline static NUCA for the naive, parallel local access and no cost centralized directory designs

Figure 5.7 reveals a great opportunity that is left unexploited. Since all tiles are directly connected to the centralized directory, the multi-hop travel of the request to its target tile can be avoided, if the centralized directory forwards it directly to the target. A request from tile 0 to tile 3 would require a total of 38 cycles, as illustrated in Figure 5.7. The improved design, which is shown in Figure 5.9, manages to avoid the 15-cycle travel of the request from tile 0 to tile 3, reducing the overall request duration to 23 cycles.

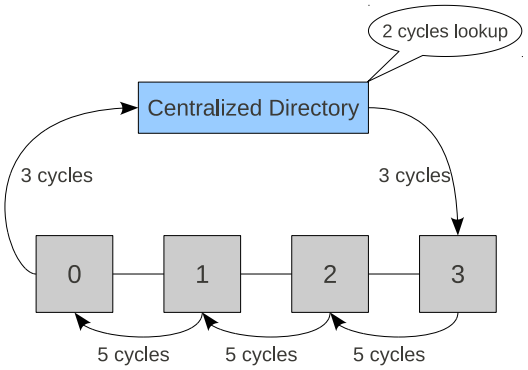


Figure 5.9: Network latencies for the improved access design. Hop latencies required to forward the request from tile 0 to tile 3 are avoided.

Request type	Total hop cost	Centralized dir cost	Benefit
1-hop	5	8	-3
2-hop	10	8	2
3-hop	15	8	7
4-hop	20	8	13
5-hop	25	8	18
6-hop	30	8	22

Table 5.3: Benefit from the improved access design for each request type. Requests are grouped by the distance in hops between the requesting and servicing tile. Costs and benefit are expressed in cycles.

The improved access design provides access latency improvement, proportionate to the distance between the requesting and the servicing tile. The longer the distance, the greater the benefit, as shown in Table 5.3. The improved access design improves overall access latency for all remote accesses that require more than one hop.

Figure 5.10 illustrates the performance speedup for the parallel local access, the improved design and the no cost design, relative to the baseline NUCA's performance. An interesting observation in Figure 5.10, is that the improved access through the centralized directory slightly outperforms the no cost design for most of the workloads, where the impact of a centralized directory is not modeled at all. However, the impact of the centralized directory's impact is slightly underrated:

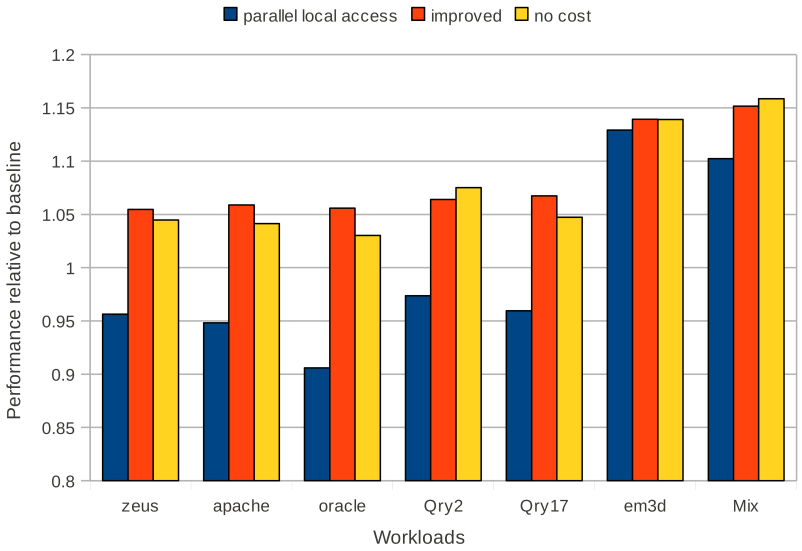


Figure 5.10: Performance of the dynamic NUCA design relative to the baseline static NUCA for three different centralized directory access designs

- Channel latency for the wires connecting the centralized directory to each tile was assumed to be equal to the channel latency of the wires connecting the tiles. However, the greater average length of these wires would also increase their latency.
- The high contention in the centralized directory caused by the simultaneous requests coming from different tiles was not modeled. Every request was assumed to require a 2-cycle lookup latency.

In general, former studies have shown that the use of a centralized directory for the lookup mechanism is inefficient in real systems. Realistic search mechanisms use more complex techniques, as we have seen in Section 2.2, where previous approaches were discussed. Since our dynamic placement policy ensures that a great fraction of overall L2 accesses are to the requester's local or neighboring tiles, a realistic approach could be a two-phase search, similar to the one used by Beckmann and Wood [4]: first, send a multi-cast to the requester's neighbors, and, if the block is not found, send a broadcast to the remaining tiles. Other works proposed other smart search mechanisms that could be employed. Kim et al. [17] and Huh et al. [14] used partial tags distributed among the cache banks, which summarize each bank's addresses of contents. Hammoud et al. [10] used another distributed search technique called *cache-the-cache-tag*, which stores the location of each data block both in the tile that is statically determined by its address and in the tile of its

initial requester to ensure both low-cost and fast data lookup.

Chapter 6

Conclusions

In this study, we investigated NUCA designs. We explained the limitations of the UCA design and how a transition from the classic, monolithic UCA design to a distributed, NUCA design manages to overcome them. A simple, static NUCA design addresses the performance bottleneck of a long, unified cache access latency, but, nevertheless, it provides limited flexibility, since a data block can only be placed in a single location determined by its address. To exploit the full potential of a NUCA, advanced policies can be applied for placement, replacement, migration and search of data in the cache, thus turning the static design to dynamic.

6.1 Our dynamic NUCA design

This study focuses mainly on the investigation of a dynamic placement policy. However, a complete dynamic NUCA design requires the implementation of all four aforementioned policies. A replacement and a migration mechanism have also been designed, in order to complement the function of the placement policy. The search policy is a very important part of the dynamic NUCA that is crucial for its efficiency. Designing a smart, effective and low-cost lookup mechanism is very challenging. In this study, things have been simplified by replacing the need for a complex search mechanism with a simple centralized directory that serves the lookup needs in the NUCA. For the evaluation, a suite of benchmarks containing server, scientific and multi-programmed workloads was used. Our dynamic NUCA scheme achieves an average speedup of 7.7% relative to a static NUCA and a maximum speedup of 15.6%. The maximum speedup was achieved for the multi-programmed workload, where each core's working set is private. Since our NUCA design

attempts to place all incoming to the L2 blocks in the bank of their initial requester, this result was expected. Server workloads, on the other hand, are dominated by shared data accesses. Despite that fact, our design achieved a performance speedup for these workloads too.

6.2 Discussion - Future work

Even though some unrealistic assumptions have been made, the great performance improvements show that similar dynamic NUCA designs hold promise. To approach reality, we also provided an overhead analysis, which shows that the access cost to a centralized directory structure such as the one used, does degrade performance improvements. However, certain optimizations were offered to mitigate this overhead. Despite any possible optimization though, the use of a centralized directory is a naive approach for a real-world NUCA lookup mechanism. To reach a realistic design, the centralized directory needs to be replaced by a smart distributed lookup mechanism, as proposed in the related literature.

Further investigation on the migration and replacement policies might also bring greater performance improvements. The migration policy used in our dynamic NUCA is a gradual promotion scheme of one step-at-a-time, based on the general direction of the preceding requests' location origin. Another interesting policy that could prove to be beneficial is to track the physical distance of the requests from the servicing tile and allow longer-range migrations, in case the majority of requests comes from distant tiles. In addition, data ping-pong effects have not been investigated. It is, however, an interesting matter that may be used as a metric for the effectiveness of migration decisions.

The usual solution to avoid pointless data movement in the cache, is a migration threshold and the impact of this threshold on overall performance can be great. A very low value for the threshold may lead to increased ping-pong phenomena. On the other hand, a very high threshold value may prevent potentially beneficial migrations to occur. Therefore, the threshold's value must be carefully chosen and the optimal value may vary, depending on each chip's and workload's characteristics. For example, a multi-threaded workload which uses much shared data would probably benefit from a rather high migration threshold value, whereas a very low threshold value would improve the performance for a multi-programmed workload, where

each thread's working set consists of private data. An ideal approach would be the development of smart mechanisms that allow dynamic manipulation of the threshold and, subsequently, migration intensity.

We also investigated various replacement policies, using a few different criteria. However, as discussed in section 5.4, it seems that taking good replacement decisions requires monitoring during execution. Unfortunately, to use monitoring periods, additional extensions to the used toolchain are needed. Another interesting extension to the data spill mechanism would be to allow cascade data movements, thus creating a wave of moving data between banks in the cache instead of a single transition. Such an approach increases the replacement policy's complexity, making it a challenging future work topic.

Bibliography

[1] Simflex: Fast, accurate & flexible computer architecture simulation. http://parsa.epfl.ch/simflex/.

[2] Spec cpu benchmark suites. http://www.spec.org/cpu/.

[3] B.M. Beckmann and D.A. Wood. ASR: Adaptive selective replication for cmp caches. In *Proceedings of the 39th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-39)*, 2006.

[4] B.M. Beckmann and D.A. Wood. Managing wire delay in large chip-multiprocessor caches. In *37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[5] S. Behling, R. Bell, P. Farrell, H. Holthoff, F. O'Connell, and W. Weir. The power4 processor introduction and tuning guide. *Redbooks*, 2001.

[6] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] Z. Chishti, M.D. Powell, and T.N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, December 2003.

[8] P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology@Intel Magazine*, 2005.

[9] M. Hammoud, S. Cho, and R. Melhem. ACM: An efficient approach for managing shared caches in chip multiprocessors. In *HiPEAC*, 2009.

[10] M. Hammoud, S. Cho, and R. Melhem. Cache equalizer: A placement mechanism for chip multiprocessor distributed shared caches. In *HiPEAC*, 2011.

[11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

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

[13] J.L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *COMPUTER*, July 2009.

[14] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler. A nuca substrate for flexible cmp cache sharing. In *Proceedings of the 19th International Conference on Supercomputing*, June 2005.

[15] M. Kandemir, F. Li, M.J. Irwin, and S.W. Son. A novel migration-based NUCA design for chip multiprocessors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008.

[16] R.E. Kessler, R. Joos, A.Lebeck, and M.D. Hill. Inexpensive implementations of set-associativity. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, May 1989.

[17] C. Kim, D. Burger, and S. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS*, 2002.

[18] J. Lira, T.M. Jones, C. Molina, and A. Gonzalez. The migration prefetcher: Anticipating data promotion in dynamic NUCA caches. In *HiPEAC*, 2012.

[19] J. Lira, C. Molina, and A. Gonzalez. Analysis of non-uniform cache architecture policies for chip-multiprocessors using the parsec benchmark suite. In *MMCS*, 2009.

[20] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 2002.

[21] P. Shivakumar and N.P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. In *Western Research Laboratory Research Report*, 2001/2.

[22] T.F. Wenisch, R.E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J.C. Hoe. Simflex: Statistical sampling of computer architecture simulation. In *IEEE Micro special issue on Computer Architecture Simulation*, July/August 2006.

[23] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[24] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.