ABSTRACT

SAMIH, AHMAD. Towards High-Performance and Energy-Efficient Memory Hierarchy in Current and Future Chip Multi Processors. (Under the direction of Dr. Yan Solihin.)

Chip MultiProcessors (CMPs) are becoming the *de facto* hardware architecture over a range of computing platforms. According to Moore's law, the number of cores in CMPs is expected to keep growing as transistor density continues to shrink. As the number of cores increases, the complexity and trade-offs of current CMP design shift towards the *uncore* part of the chip. Two major components of the uncore subsystem are the Last Level Cache (LLC) and the interconnect.

As for the LLC, it is still an open question of whether it should be private to each core or shared by all cores. A physically shared LLC allows applications to naturally divide-up and share the aggregate cache space. However, a large cache has a high access latency. On the other hand, private per-core LLCs provide low latency accesses to the corresponding core and allow a more scalable multicore configuration. However, CMPs with private LLCs suffer from a cache fragmentation problem; some caches may be over-utilized while others may be under-utilized. To avoid such fragmentation, researchers proposed capacity sharing mechanisms where applications that need additional cache space can place their victim blocks in remote caches. However, we found that allowing victim blocks to be placed on remote caches without considering the temporal locality of the blocks tends to cause a high number of remote cache hits relative to local cache hits. This in turn results in a sub-optimal capacity sharing performance.

Moreover, the rising number of on-chip core counts in CMPs has mandated more scalable interconnects such as Mesh and Torus, which consume an increasing fraction of the total chip power. As technology and operating voltage scale down, the static

power consumes a larger fraction of the total power; reducing it is increasingly important for energy proportional computing. Currently, processor designers strive to send under-utilized cores into deep sleep states in order to reduce idling power and improve overall energy efficiency and energy proportionality. However, even in state-of-the-art CMP designs, the interconnect is always fully active regardless of the number of active cores, thus impacting energy proportionality.

In this dissertation, we address the two problems highlighted above to improve the performance and energy efficiency of the uncore subsystem in current and future CMPs.

First, we show that, in CMPs with private LLCs, current capacity sharing techniques cause a high number of remote LLC hits relative to local LLC hits. We also show that many of such remote cache hits can be converted into local cache hits if we allow newly fetched blocks to be selectively *placed* directly in a remote cache, rather than in the local cache. To demonstrate this, we use future trace information to estimate the near-upperbound performance that can be gained from combined placement and replacement decisions in capacity sharing. Further, we design a simple, predictor-based, scheme called *Adaptive Placement Policy* (APP) that learns from past cache behavior to make a better decision on whether to place a newly fetched block in the local or remote cache. We found that across several multi-programmed workload mixes running on a 4-core CMP, APP's capacity sharing mechanism increases aggregate performance by 29% on average.

Second, to reduce interconnect idling power and achieve better energy proportionality, we propose Router Parking an approach to selectively power-gate routers attached to sleeping cores. Router Parking ensures that network connectivity is maintained, and limits the average interconnect latency impact of packet detouring around parked routers. We present two Router Parking algorithms – an aggressive approach to park as many routers as possible, and a conservative approach that parks a limited set of routers in

order to keep the impact on latency increase minimal. Further, we propose an adaptive policy to choose between the two algorithms at run-time. Evaluations conducted on synthetic and real workloads show that Router Parking can achieve significant savings in the total interconnect energy of up to 41%.

Towards High-Performance and Energy-Efficient Memory Hierarchy in Current and Future Chip Multi Processors

by
Ahmad Samih

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2012

APPROVED BY:

Dr. Gregory T. Byrd	Dr. James M. Tuck
Dr. Tao Xie	Dr. Yan Solihin Chair of Advisory Committee

DEDICATION

To my mohter Salma... and to my father Abdelraouf...

BIOGRAPHY

Ahmad Samih was born in Jordan, on September 1st, 1985. Ahmad received the B.S. degree in computer engineering from the Jordan University of Science and Technology (JUST) in 2007, the MS degree in electrical and computer engineering from North Carolina State University, in 2009. Since then, he has been a PhD candidate in the department of electrical and computer engineering at North Carolina State University under the supervision of Dr. Yan Solihin. During his PhD studies, he has done three research internships at Intel Labs in the Summer and Fall of 2011, and the Summer of 2012. Upon the completion of his PhD program, he will be starting his first full-time job as an Atom Performance Architect at Intel Corporation in Austin, Texas. His research focuses on computer architecture, high-performance and energy efficient memory hierarchy design in multi-core processors, and network-on-chip architectures. More information can be found at http://www4.ncsu.edu/aasamih.

"I seem to have been only like a boy playing on the seashore, and diverting myself in now and then finding a smoother pebble or a prettier shell than ordinary, whilst the great ocean of truth lay all undiscovered before me."

-Sir Isaac Newton

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deep gratitude and appreciation to my research adviser, Dr. Yan Solihin. He has been a great support and a source of motivation and enthusiasm as I progressed through my PhD. His constructive comments and guidance, particularly during brainstorming sessions and before conference submissions played a critical role in developing the skills needed to conduct substantial research.

I would also like to thank my advisory committee members, Dr. Gregory Byrd, Dr. James Tuck and Dr. Tao Xie, for the valuable feedback I received from them during my preliminary examination which helped improve the quality of my dissertation.

I would like to extend special thanks to my PhD collaborator, Anil Krishna who has been a great example to follow during my PhD. Anil has been my unofficial mentor who helped me to set a standard for quality of work. I would also like to thank his wife, Kavita, for the delicious food she used to prepare.

Other friends, who made my journey at NC State enjoyable are Muawya Al-Otoom, Rami Al-Sheikh, and Amro Awad. I had a lot of fun and excitement while interacting with these guys.

I would like to thank all current and former members in ARPERS research group for their friendship and valuable discussions. In particular, special thanks are extended to Xiaowei Jiang (currently at Intel), Siddhartha Chhabra (currently at Intel), Brian Rogers (currently at IBM), Fang Liu (currently at Qualcomm), Ganesh Balakrishnan (currently at IBM), and Devesh Tiwari. I would also like to thank all the members of CESR lab. In particular, Niket Choudhary and his wife Tulika, Sandeep Navada, Rajeshwar Vanka, George Patsilaras, Jayneel Gandhi and Shivam Priyadarshi. I am very thankful to Linda Fontes, Elaine Hardin, Katy Wilson and Kendall Del Rio who were always very helpful

in clarifying any doubts regarding the administrative process and getting all the paper work ready in a timley fashion.

I had fun and very good experience while working at Intel Labs. Many thanks go to my mentors Ren Wang and Christian Maciocco, and to my other group members Sameh, James, Maziar and Mesut.

Last but not least, my parents were a great help throughput. Their indefinite support, love and encouragement, not only through my PhD, but also throughout my life helped shape my personality and given me the strength and knowledge needed to tackle the difficult aspects of life. Similarly, I owe special thanks to my brother, Mohammed, and to my sisters for similar love and support.

Finally, I praise God who helped me complete my dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Chip Multi Processors (CMPs) are becoming ubiquitous hardware architectures over a wide range of computing platforms such as hand-held mobile devices, client-based machines and server-based machines. CMPs are currently being shipped with 8 cores on the same chip [29], and this number is expected to keep growing as transistor density continues to shrink [9]. Such an increasing core count requires more complex and scalable uncore subsystem - specifically at the LLC and the interconnect levels. Due to this increase in complexity, designers are faced with multiple design trade-offs and challenges.

Contemporary CMPs often come with private L1 caches due to their tight timing requirement. However, whether the last level cache (LLC) should be private to each core or shared by all cores is an open question. A physically shared LLC cache allows applications to more fluidly divide up the cache space, and maximizes the aggregate cache capacity because no block is replicated in the LLC cache. However, a large cache has a high access latency. On the other hand, private per-core LLC caches provide low latency accesses to the corresponding core, provide performance isolation between cores, and allow a more scalable multicore configuration. However, private LLCs have

the major disadvantage of *capacity fragmentation*, i.e., when a diverse mix of sequential programs runs on different cores, some programs may over-utilize the LLC cache due to their large working set, while others may under-utilize it due to their small working set. Recent research proposals have employed capacity sharing techniques to better manage the aggregate capacity of private caches. However, such proposals do not consider the temporal locality of the running applications while placing blocks in local and remote caches. *Hence, there is a need to understand the impact of applications' temporal locality on capacity sharing and overall performance.*

Moreover, the rising number of cores in CMPs and the corresponding increase in the number of LLC slices have demanded more reliable and scalable on-chip interconnect topologies such as Mesh and Torus. Such interconnect fabrics consume an increasing fraction of the entire chip power [48] in order to support higher bandwidth requirements and larger number of cores. Recent studies show that, if all cores are 100% utilized, the interconnects in Intel's SCCC [34] and Intel's 80-core chip [27] consume 10% and 36% of the total chip power, respectively. With lower core utilization, the fraction of the power consumed by the interconnect is much higher. As technology and operating voltage continue to scale down, the static power contribution to the total chip power will continue to grow. With low core utilization, even if some cores go to deep sleep state, the system would not be energy-proportional. One reason is due to the fact that the entire interconnect is always on and forwarding packets on behalf of the remaining active cores even though some of the attached cores and LLC slices are power-gated. *Hence, there is a need to reduce interconnect static energy consumption in order to achieve better energy proportionality and reduce overall chip energy.*

In this dissertation, we target the two problems highlighted above to improve the performance and energy efficiency of the uncore subsystem in current and future CMPs,

namely, improving the performance of capacity sharing in CMPs with private LLCs, and reducing the static energy consumption of the state-of-the-art interconnects.

1.1 Managing Capacity Sharing in CMP Architectures with Private Caches via Novel Placement Policies

Recognizing the capacity fragmentation problem in CMPs with private caches, prior work, such as Cooperative Caching [12] (CC) and Dynamic Spill Receive [52] (DSR), has proposed schemes that allow cores to share their local caches with other cores. Such *capacity sharing* mechanisms are enabled by allowing a core to spill a block evicted from the local cache to a remote cache. CC allows any cache to spill into any other cache regardless of cache usage behavior of the applications running on the corresponding cores. While this provides fluid capacity sharing among cores, sometimes it produces an unwanted effect. An application, which cannot benefit from additional cache capacity, may pollute the cache of a core running an application that really needs all the cache space it has. To avoid this drawback, DSR identifies applications that can benefit from additional cache capacity (called *Acceptors*), and applications that have excess cache capacity that can be donated without impacting performance (called *Donors*). Further, it proposes a dynamic mechanism to identify when an application is allowed to spill, and when an application is allowed to receive a cache block.

Both CC and DSR have a common characteristic - remote caches are treated as the victim cache for the local cache. This work investigates if such a strategy is the best way for providing capacity sharing. In particular, we investigate a previously-unexplored

strategy that considers placing newly-fetched blocks directly in remote caches, in addition to considering placing locally-evicted blocks in remote caches. A remote placement strategy is motivated by our observation that the stack/reuse distance of applications that benefit from capacity sharing tends to be high, which causes a lot of remote cache hits compared to local cache hits. By placing incoming blocks in remote caches, we less disturb the local cache population, and many remote hits can be converted into potential local hits. To test the potential performance improvement from remote placement, we developed a hypothetical oracular scheme that, based on a *future-access* trace, it determines whether a newly fetched block should be placed in the local cache or remote cache. In addition, in case it decides to place a newly-fetched block in the local cache it then consults the future-trace again to identify the ideal replacement candidate in the local cache. We call this hypothetical capacity sharing technique Oracular Placement and Replacement Policy (OPRP). Figure 1.1 shows the speedup of the state-of-the-art DSR [52] and the oracle scheme (OPRP) compared to the base case where the CMP has private caches without capacity sharing. A_xD_y represents workloads that have x number of Acceptor applications and y number of Donor applications. The figure shows the average behavior within subsets of workloads grouped by a unique Acceptor-to-Donor ratios, as well as the average across all workloads.

The figure shows that although DSR improves the speedup over the base case, there is a significant performance gap between it and OPRP. In addition to having an oracle replacement policy, OPRP allows newly fetched blocks to be placed at both the local and remote caches. Based on the performance gap between the two schemes, we are motivated to investigate the significance of placement decisions in capacity sharing strategies.

The main differences between these two schemes are: OPRP applies an oracle replacement policy, it also adds

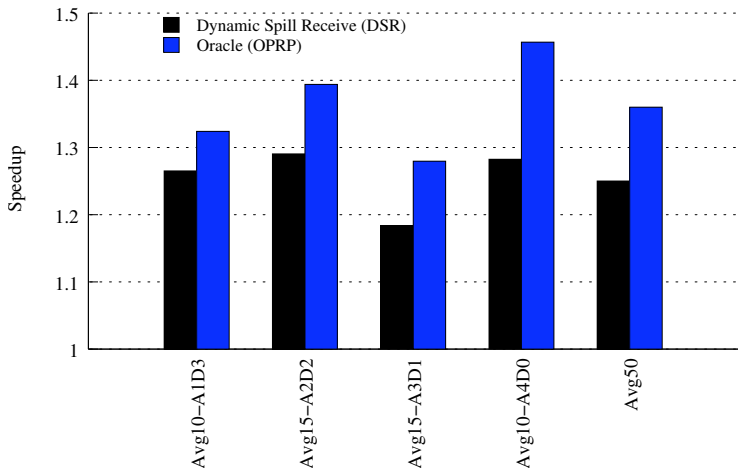


Figure 1.1: Speedups for fifty workload mixes showing the performance of DSR and the oracle scheme over the base case of private caches without capacity sharing. Additional details on simulated machine parameters and workloads can be found in Section 2.3.

Our investigation confirms that applications that benefit from additional cache capacity (Acceptors) are also ones that tend to experience a high number of remote hits in DSR. The high number of remote cache hits is a significant reason for the performance gap between DSR and OPRP. To narrow the gap, we design a simple, predictor-based scheme called *Adaptive Placement Policy* (APP) that learns from an application's past cache behavior to make a remote placement decision. APP's predictor structure is small (32 bytes in size) and has a simple organization. APP tracks whether a local cache block is accessed by the processor while it resides in the local cache and marks blocks that are never accessed as potential candidates for remote placement. Further, APP uses Set-Dueling [52] to dynamically identify applications that should be allowed to place their blocks in remote caches. Despite its simplicity, APP performs robustly across a wide range of workloads and scenarios. We evaluate APP on a quad-core CMP system with 1 MB, 8-way, private last level L2 caches. We use 50 multiprogrammed workload mixes each consisting of 4 SPEC CPU2006 applications. The workload mixes contain varying

ratios of Acceptor and Donor applications. Our evaluation shows that APP improves the performance of a CMP without capacity sharing by 29% on average. APP also outperforms the state-of-the-art capacity sharing technique, DSR, by up to 18.2%, with a maximum degradation of only 0.5%, and an average improvement of 3%. Further, we study the sensitivity of APP and DSR to increasing remote cache hit latency, and a range of L2 cache sizes, as would be expected in larger-scale CMPs. We find that APP continues to perform robustly under these scenarios.

1.2 Energy-Efficient Interconnect via Router Parking

Recent studies show that the interconnects in Intel's SCCC, Sun's Niagara [46], Intel's 80-core chip [27], and MIT's RAW chip [38] consume 10%, 17%, 28% and 36% of the total chip power under 100% core utilization, respectively. With lower core utilization, the fraction of the power consumed by the interconnect is much higher. For example, in Intel's SCCC, interconnect power contribution jumps from 10% to 17% of the total chip power if 50% of the cores are utilized since it will continue to operate at maximum frequency to serve the active cores.

As technology and operating voltages continue to scale down, as evidenced by Intel's 22nm transistor technology [32] and the near-threshold voltage design [9], static power is becoming a larger component of the total power [4, 15, 63]. Figure 1.2(a) shows the breakdown of an interconnect router's dynamic versus static power at 2GHz, with V_{dd} of 1.1 V and 1.0 V, and under three different process technologies – 65nm, 45nm, and 32nm. The results are obtained from the Orion2.0 [36] power model. As shown in the figure, the static power consumption increases rapidly as technology scales down, going

from 11.2% of the total router power in the 65nm (1.1V) technology to 33.6% in the 32nm (1.1V) technology.

Further, Figure 1.2(b) shows the breakdown of the static and dynamic power in the 32nm technology as Vdd decreases from 1.1V to 0.7V. The static power reaches 44.3% of the total power at 0.7V. Overall Figure 1.2 shows that technology scaling [24], accompanied by reduction in the operating voltage [9], significantly increases the contribution of static power to the total power.

Finally, the dynamic power numbers shown in these plots are measured based on 50% load activity. According to International Data Corporation (IDC) study [25], servers achieve utilization of only 10% to 15%. With such low average utilization, the share of static power to the total interconnect power is expected to be even larger.

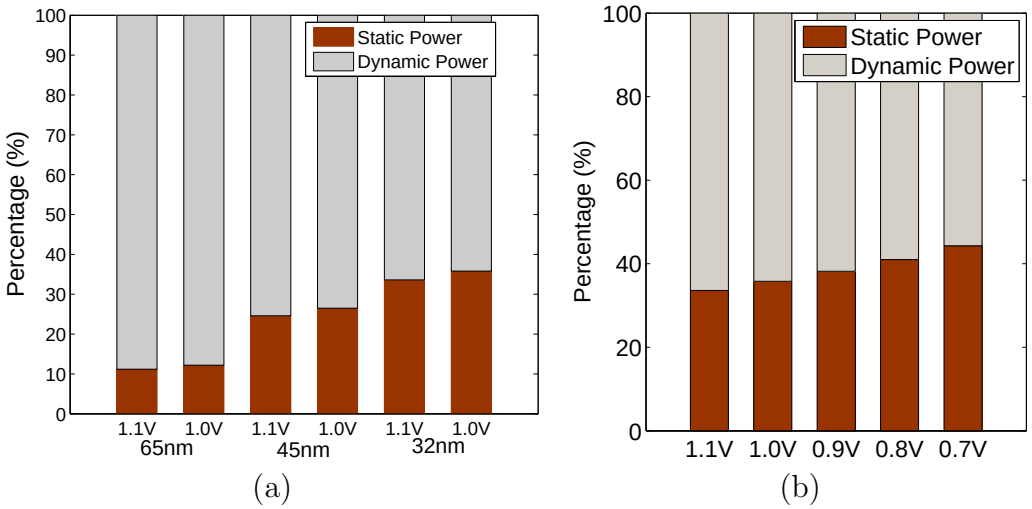


Figure 1.2: Static router power vs. dynamic router power for various technologies, and operating voltages. Dynamic router power is measured at 50% load.

ITRS [33] projects that by 2022, a single chip may include more than 100 cores. The rising number of cores on a single chip, accompanied by limited power envelops and low

system utilization create increasing opportunities to send many of these cores to deep sleep states. Unfortunately, even after exploiting these opportunities, the system would not be energy proportional [5] if the idling power is high. With current interconnects, even when an idle core does not generate any interconnect traffic, the router and links attached to it remain active in order to maintain full interconnect operation.

Prior studies [63, 23] attempted to reduce the interconnect power by reactively power gating router components (e.g. links, ports, or buffers) in response to traffic load. With such an approach, performance may be penalized significantly if the wake-up latency is high or traffic is unpredictable. Hence, only lightweight power saving techniques can be employed. Furthermore, the approach cannot take a full advantage of situations in which a large number of cores are in deep sleep and their associated routers are idle and can be fully power gated without impacting latency by much.

To this end, we propose Router Parking (RP), a novel interconnect power management approach that aims to reduce the static interconnect power. The main idea behind our proposal is to power gate a subset of routers associated with sleeping cores by *proactively* aggregating traffic to the active routers such that it conserves power, does not impact function, and minimizes the impact on performance. Without proactive aggregation, traffic disperses across all routers even in low traffic scenarios. This can keep routers lightly loaded but active – even when their associated cores are parked. Under various topologies, such as Mesh or Torus, there usually exist multiple paths from a given source to a given destination router. Therefore, keeping all routers associated with parked cores active may not be necessary. Parking routers associated with sleeping cores must not cause interconnect partitions, where a given pair of active cores are not able to reach each other. Even after ensuring that no interconnect partitions are created, there are power and performance trade-offs in RP. The static power savings enabled by parking more

routers must be balanced against the increase in interconnect latency due to rerouting around parked routers, which in turn increases run-time and energy.

In order to address this trade-off, we propose and evaluate various RP algorithms. On one end of the solution space is an Aggressive Router Parking (RP-A) approach that strives to park as many routers as possible in order to maximize static power savings. This solution, under heavy traffic, may have an undesirable end-to-end latency impact due to excessive packet detours. The other end of the solution space is a Conservative Router Parking (RP-C) approach that parks fewer, carefully selected, routers in order to minimize the impact of RP on interconnect latency. Motivated by encouraging experimental results on both ends of the spectrum, we design an Adaptive Router Parking (RP-Adp) approach that monitors the interconnect utilization and chooses between RP-A and RP-C at run time, enabling power savings while limiting performance impact.

We evaluate RP on a 64-core tiled CMP using Simics 4.7 [44] with a cycle-accurate interconnect simulator based on the Garnet [19] interconnect model and the Orion2.0 [36] power model. We experiment with both synthetic traffic and real workloads taken from the SPEC CPU2006[65] and PARSEC 2.1[8] benchmark suites. Our evaluations show that for SPEC CPU2006 and PARSEC 2.1 workloads, RP can reduce the total interconnect energy by an average of 41% and 40%, respectively. Finally, for these two workload suites, the performance metric, Weighted Speedup, is reduced by less than 0.2% and 0.4% on average, and 3% and 4% in the worst case, respectively.

1.3 Organization of the Dissertation

The rest of the dissertation is organized as follows.

Chapter 2 motivates our approach to solving the problem of private L2 management,

describes the design and working of APP, and the oracular policies in details, describes our evaluation methodology, provides the results from our evaluations and analyzes the key findings, and describes related work in cache management of Uni- and Multi-processor systems.

Chapter 3 describes the design of Router Parking architecture and algorithms, outlines our evaluation methodology, provides the results and analyzes the finding of our evaluations and sensitivity analysis, and describes prior work that attempted to reduce interconnect energy consumption.

Chapter 4 summarizes the findings and concludes this dissertation.

Chapter 2

Managing Capacity Sharing in CMP Architectures with Private Caches via Novel Placement Policies

This chapter is organized as follows: Section 2.1 motivates our approach to solving the problem of private L2 cache management, Section 2.2 describes the design of our placement algorithms, Section 2.3 describes the evaluation methodology, Section 2.4 shows evaluation results and insights into the effects of placement policies on cache management, and Section 2.5 overviews related work. Section 2.6 concludes this chapter.

2.1 Motivating the Need to Consider Placement Decisions

In Section 1.1, we showed results that show a gap between the performance of DSR, a state-of-the-art capacity sharing technique, and the performance of an oracle scheme

(OPRP). Our hypothesis is that this gap primarily exists because of the oracle scheme's ability to selectively place newly fetched/accessed blocks in remote caches. In this section, we investigate this hypothesis in detail as a first contribution of this work.

If it is true that OPRP outperforms DSR due to its selective placement of incoming blocks in a remote L2 cache, then it must be the case that the incoming blocks are going to be reused by the processor later than blocks that are already stored in the local cache. This implies an anti-LRU reuse pattern, in that blocks that were less recently used (those already in the local L2 cache) are more likely to be accessed in the near future compared to blocks that were more recently used (the most recent local L2 miss). For OPRP to significantly outperform DSR, there has to be a relatively large number of blocks that exhibit this anti-LRU temporal reuse pattern.

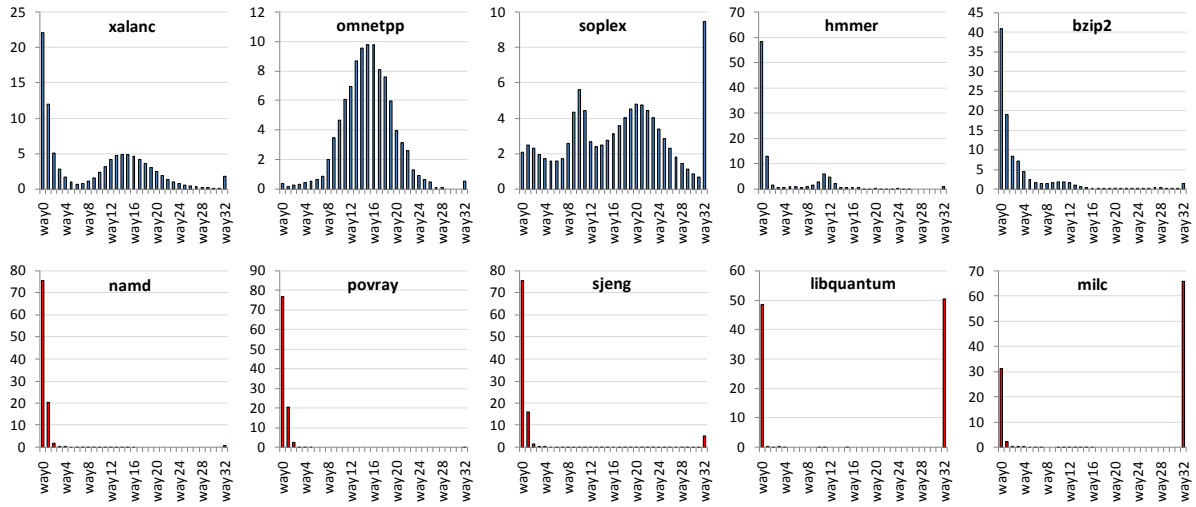


Figure 2.1: Stack Distance Profiles for 10 SPEC CPU2006 applications: x-axis is the cache way number, bars represent the percentage of hits to the cache blocks from the MRU position on the left to the LRU position on the right; 32 ways make up a 4MB cache; right-most bar displays the % of misses.

In order to investigate this anti-LRU temporal reuse pattern behavior, Figure 2.1 shows the *stack distance profile* [41] for 10 SPEC CPU2006 benchmark applications. The L1 cache configuration (Table 2.1) is 32KB in size and 2-way associative, across all experiments. The x-axis in the figure shows the cache ways sorted based on their recency of accesses assuming a 32-way 4MB total cache capacity (hence, each way corresponds to 4MB/32=128KB). The y-axis shows the percentage of total cache accesses that hit in a given stack position. The leftmost bar represents the most-recently used (MRU) position, and the rightmost bar represents fraction of accesses that miss on all 32 ways. Under LRU replacement, a hit rate for a given cache size is represented by the area under the curve from the first until the last stack position the cache can hold. For example, the hit rate of a 1MB cache is the sum of bars way0, way1, ..., way7.

The figure shows two rows of applications, where the top row contains ones that show significant miss rate reduction when the cache capacity is increased from 1MB to 4MB (Acceptors), as evident in the large area under the curve between way 8–31. The second row shows applications that do not benefit from increasing the cache capacity from 1MB to 4MB (Donors). Some of the applications in the second row have very small miss rates (namd, povray, and sjeng), indicating small working sets, while others have very large miss rates (libquantum and milc), indicating either very large working sets or streaming memory access patterns with little temporal reuse. These donor applications not only cannot benefit from larger cache capacity, but also have substantial cache capacity that can be donated without affecting their miss rates.

If an application has a *perfect LRU* temporal reuse behavior, i.e., blocks that were accessed more recently have a higher likelihood of being accessed again, the stack distance profile will show *monotonically decreasing* bars. The figure shows all Donor applications show a perfect LRU behavior. However, Acceptors show more interesting behavior.

Four out of five Acceptor applications (xalancbmk, omnetpp, soplex, and hmmer) show imperfect (or anti-) LRU temporal reuse behavior, as evident by significant bumps at way8–32 in the stack distance profile. Only one Acceptor application (bzip2) has a nearly perfect LRU profile. The extent of anti-LRU temporal reuse behavior varies across applications. For example, omnetpp has less than 3.60% hits in the first 8 stack positions (1MB of cache) but 95.90% hits in the next 24 ways (1 to 4MB of cache). The high correlation between Acceptors and anti-LRU behavior makes sense, because with LRU's monotonic decrease, once one bar has a small fraction of access, accesses to higher stack positions must be lower than it. Therefore, overall we can conclude that *Acceptors tend to exhibit some anti-LRU temporal locality behavior, while Donors tend to exhibit perfect LRU temporal locality behavior.*

Note that if a capacity sharing technique only places blocks evicted from the local cache in remote caches, then necessarily the local cache stores more recently used blocks than remote caches. Since all existing capacity sharing techniques (CC and DSR) use remote caches as the victim cache for the local cache, they guarantee that remote caches store less recently used blocks. However, our earlier observation concludes that applications that benefit from additional cache capacity (i.e., Acceptors) are also the ones with anti-LRU behavior. Therefore, by treating remote caches as a victim cache for the local cache, *CC and DSR guarantee the high occurrence of remote cache hits* relative to local cache hits.

While remote hits are cheaper than misses, local cache hits are much preferred over remote hits. Remote L2 cache hits are costly from both a performance and a power point of view. A remote hit has a significantly longer latency compared to a local hit due to the coherence action needed for the remote cache to source the block into the local cache. In addition, it also consumes more power due to the coherence request

posted on the bus, snooping and tag checking activities in all remote caches, and data transfer across the bus. Furthermore, future technology scaling allows more cores to be implemented on a chip, which increases the relative distance of remote caches relative to the local cache, e.g., due to additional on-chip network hops and routing delay. In contrast, the clock frequency growth of the processor may be slowing, hence the off-chip memory access latency is only growing slowly relative to the access latency of on-chip cache. Thus, remote cache hit latency will become a larger problem in the future. It is very important, therefore, to maximize the local hits by converting remote hits into local hits.

The key to converting remote hits into local hits is to *selectively* decide to place blocks locally versus remotely when they are brought into the L2 cache from the lower level memory hierarchy. Remote placement allows blocks that have the highest chance to be accessed by the processor, regardless of their stack positions, to be placed in the local cache.

2.2 Designing Placement Policies for Capacity Sharing

This section discusses how to dynamically identify which and when applications should be allowed to place blocks in remote caches (Section 2.2.1), our oracular placement policy (Section 2.2.2), and a hardware-implementable placement policy (Section 2.2.3).

2.2.1 Dynamic Identification of Spillers and Receivers

Not all applications should be allowed to place their blocks in remote caches, because they may not benefit from increased cache capacity. Evaluating which and when applications should be allowed to place blocks in remote caches should be performed *dynamically and continually*, because temporal reuse behavior is not only application specific, but may also be phase-specific. We refer to a cache (or the application using it) that is allowed to place blocks in remote caches as a *Spiller*, and a cache that can only receive blocks from others as a *Receiver*.

Ideally, we want Acceptor applications to be identified as Spillers and Donor applications as Receivers. However, the identification is not that simple for several reasons. First, an application may switch behavior between Acceptor and Donor dynamically based on its execution phase. Second, an application's classification as a Spiller or a Receiver cannot be determined in isolation. For example, in a workload with many Acceptors and few or no Donors, it may be beneficial for system performance to classify some Acceptors as Receivers so that some of their cache capacity can be donated to other Acceptors which respond much more to additional cache capacity. Finally, applications or threads may migrate from core to core, changing the behavior cores exhibit. Therefore, each application must be classified dynamically and continually as Spiller or Receiver.

To achieve that, we rely on *set dueling* [52], where each cache dedicates a small subset of sets (e.g., 32 randomly-chosen sets) across all caches to always spill, and another subset to always receive. The miss rates of spiller subset and receiver subset are continuously compared to determine which subset achieves a lower all-core miss rate. The policy applied to the rest of the cache sets follows one from the subset that achieves the lowest miss rate. In order to choose a winning policy (spill vs. receive), each cache is augmented

with a 10-bit saturating PSEL counter [52].

2.2.2 Design of Oracular Placement Schemes

In order to study the upper-bound performance improvements from remote placement policies, we developed a hypothetical Oracular Placement and Replacement Policy (OPRP). When a Spiller suffers a cache miss, OPRP determines if the incoming block should be placed in the local L2 or in a remote L2 by looking at a trace of *future* accesses of the Spiller application. This trace is generated prior to simulation run time. Similar to Belady's optimal replacement algorithm [7], all blocks in the local cache set of the missed block, and the miss block itself, are compared against the trace. If the missed block is accessed farther in the future compared to currently cached local blocks, the block is placed on a remote cache. Otherwise, a victim block is selected from the local cache to be spilled into a remote cache. In any case, the missed block is supplied to the processor and its L1 cache.

Placement decision must be made not only when there is a global miss to a new block. It must also be made when there is a remote cache hit, where we must decide whether to bring the block into the local cache. For OPRP, when there is a remote L2 cache hit, the future access trace is consulted again. If the remotely hit block is accessed farther in the future compared to currently cached local blocks, the block is left in the remote cache. Otherwise, a victim block is selected from the local cache to be swapped with the remotely hit block. In any case, the original or swapped remote block is marked as the new most recently used block in the remote cache. The block is, as always, supplied to the processor and its L1 cache.

The optimality of the placement or replacement decision can only be guaranteed for a given application and not globally across a mix of applications. This is because

there is no way to know a priori how traces of accesses from different applications will be interleaved in the actual multiprogrammed execution. Therefore, OPRP only makes oracular decisions from a Spiller's perspective, but not a Receiver's. For Receivers, we simply incorporate performance guard bands so that their performance is not impacted by much (discussed in Section 2.2.5).

Note that OPRP consults the future trace to guide *both* placement and replacement. The decision of whether to place an incoming cache block in the local or remote cache is a placement decision, but what block is victimized from the local cache is a replacement decision. In order to bound the performance improvement attainable from the placement decision alone, we design a new Oracular Placement Policy (OPP). OPP uses the same policy as OPRP in determining whether a block should be placed in the local or remote cache. However, when it is decided that a block should be placed locally, the LRU block is selected as a victim, instead of the block used the farthest in the future. Thus, future trace information is only used for making placement decisions.

2.2.3 Design of APP

Obviously, OPRP and OPP are not implementable as they rely on future trace information. In order to design a placement policy that has a practical hardware implementation, we can approximate *future* information with *past* information. We refer to this scheme as Adaptive Placement Policy (APP). Such a scheme would require a predictor table and logic that determines whether an incoming or remotely hit block should be in the local or remote cache.

APP Predictor Design

To be practical, APP predictor must be low cost and efficient. Each private L2 cache is augmented with the predictor, but the predictor is only used when a core runs an application that is determined to be Spiller.

Without future trace information, APP cannot determine exactly when an incoming block will be accessed again in the future. However, it can record the past history of the block and extrapolate its behavior into the future. APP records whether a block was accessed while it was last resident in the cache. A block that was not accessed during its residence in the local L2 cache either indicates that it will never be accessed again, or more likely its stack/reuse distance is larger than the L2 cache associativity. Thus, if the behavior repeats, such a block should still be fetched on chip, but placed in a remote L2 cache so that it does not pollute the local L2 cache while it is waiting to be accessed again. APP predictor attempts to identify such blocks.

We found the behavior of blocks not being accessed during their residence in the L2 cache to be quite common among Acceptors. One situation that leads to such behavior is when the block's temporal and spatial reuse can be captured completely in the L1 cache. Thus, the block is brought into the L2 cache, but is only accessed at the L1 cache, until the block is evicted from the L2 cache, and a new miss prefetches the block.

In order to provide the block usage history during the block's residence in the local L2 cache, each block in the L2 cache is tagged with a single "Accessed" bit that is reset when the block is initially installed in the L2 cache, and is set when the block is accessed. When the block is evicted, the block's information, along with its access bit, is given to the APP predictor to be recorded. The local APP predictor is looked up when placement decision needs to be made (essentially at every local L2 cache miss), and is updated on

each local block eviction. Therefore, the lookup and update are not on the critical path and should not affect cache access latency.

Next, we will discuss the APP predictor structures.

Instruction-based Predictor. The role of APP predictor is to record the usage behavior of evicted blocks when they were last resident in the local L2 cache. An important question is whether the information should be recorded per instruction, per address, or a combination of them. The first option is to record the usage information per program counter (PC) of the memory instruction that accesses the block. With this option, we keep a per-PC access history in a prediction table. Any blocks that are brought into the cache by a PC will be tagged by this PC, and when any of them is accessed during its residence and is evicted, we set the accessed bit for this PC to record the reuse.

With any limited-size predictor structures, we have to deal with the possibility that multiple PCs map to the same entry in the table. Conflicts between PCs that map to the same entry can be handled by discarding older entries from the table, allowing entries to spill to the main memory, or by allowing them to share a common entry. After experimentation with various designs, we find that a small (4096) and simple (direct-mapped) table works well enough to deliver accurate predictions. Many cache misses are caused by a small number of load/store instructions, keeping a small number of PCs in the table gives good performance.

The predictor table is is indexed by 12 least significant bits of the PC, whereas the remaining PC bits are stored in each entry as tag bits. In addition to tag bits, each entry includes a valid bit, which is used to validate a given PC entry, and a 2-bit saturating counter, which is used to approve a remote versus local prediction. The use of these bits will be discussed in greater detail later.

Address-based Predictor. Another possibility for APP predictor table organiza-

tion is to keep per-address information. In contrast to PCs, where keeping a few PCs that produce most misses is sufficient, we can expect that the number of unique memory block addresses within a program to be very large. Attempting to learn individual behavior of each block is impractical as we need a huge prediction table structure to keep all of them, even when entries can spill to main memory. Allowing new entries to discard older entries still will not allow effective recording when the number of active blocks is much higher than number of table entries. This leaves us with the only one choice left: allowing multiple block addresses to share a common entry. Such a sharing policy works if many blocks share the same behavior, but not when they produce different behavior. Fortunately, in most cases, when an application has a high reuse distance, it affects most of the blocks it touches. Hence, allowing multiple block addresses to share a common entry helps accelerate the learning, rather than hurts performance.

To address the prediction table, a hashed index is generated by *folding* down the cache block address tag into an n-bit index, which addresses a table with 2^n entries. By folding we mean dividing up the address tag into n-bit entities and XORing them (an entity is zero-padded before XORing, if it is not a multiple of *n-bits*). Such hashing does not guarantee that addresses with similar behavior would share a common predictor entry, but because of global access behavior common to many blocks, the choice of hashing function only acts to ensure that entire table is utilized. The global reuse behavior across blocks is corroborated by our experiments, where even a small table with only 256 entries gives the most cost-effective design (Section 2.4.3).

Instruction-and-Address-based Predictor. The final predictor structure we try combines PC and block address behavior history recording (i.e., hybrid predictor). To index the prediction table, both the PC and block address are folded into n-bit entities, which are then XORed together. Similar indexing mechanisms are used in branch predic-

tors [2] to exploit PCs and memory addresses correlation. The structure of this predictor is similar to the address-based one in terms of table size. The difference is only the input used for indexing the table.

Predictor Analysis. Let us compare the complexity of PC-based, address-based, and hybrid predictor structures. First, in most memory architectures, the PC of a load/store instruction is not propagated down to the last-level cache (LLC). Requiring a predictor associated with a LLC to have PC information requires relatively major changes to the memory hierarchy architecture. Second, in order to update the predictor table when a block gets evicted from the cache, the PC of the instruction that *brought* the block on chip should be stored along with each cache line. This enlarges the tag array considerably and increases its access latency. Third, if sequential/stride hardware prefetcher is used at the LLC, the prefetcher issues only addresses. If our predictor requires PC information to work, it will not be able to decide where to place prefetched blocks.

As a result to PC-based predictor's serious drawbacks, we adopt the address-based predictor as a primary design, but we evaluate all three predictor designs in detail: we quantify the hardware costs, predictor accuracy and performance of the three predictor designs in Section 2.4.3.

Predictor Consult and Update Mechanism. Figure 2.2 illustrates how the APP predictor table makes placement decisions, and Figure 2.3 shows how each cache block is tagged with an "Accessed" bit. As mentioned earlier, the table is indexed by folding the cache block address tag into an 8-bit index. The table is tagless.

Each entry in the prediction table contains a saturating counter, which is decremented when an evicted block records that it was accessed during its recent residence in the local L2 cache (its "Accessed" bit is 1), and incremented when the evicted block was never

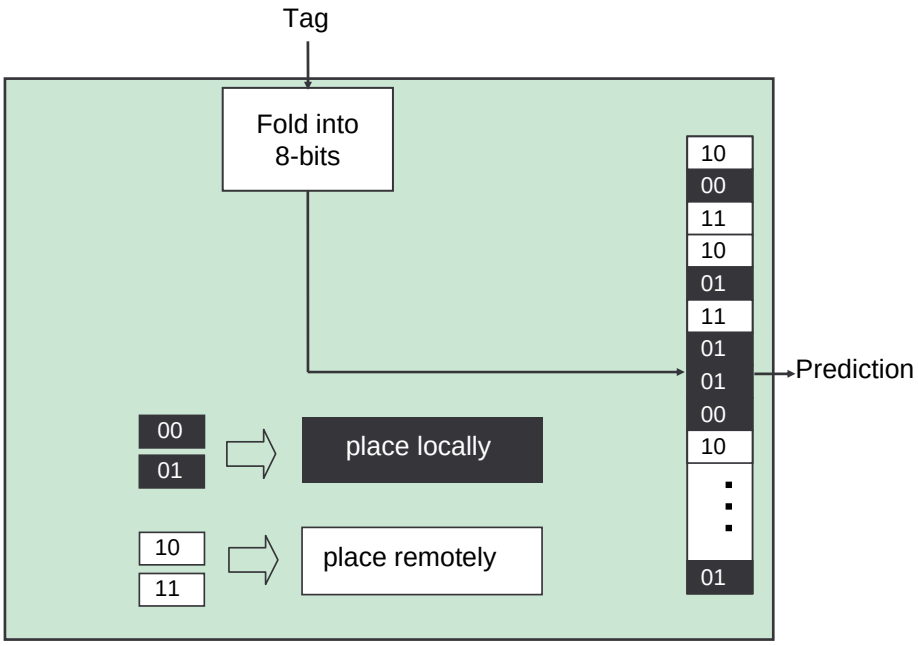


Figure 2.2: Predicting with the APP predictor

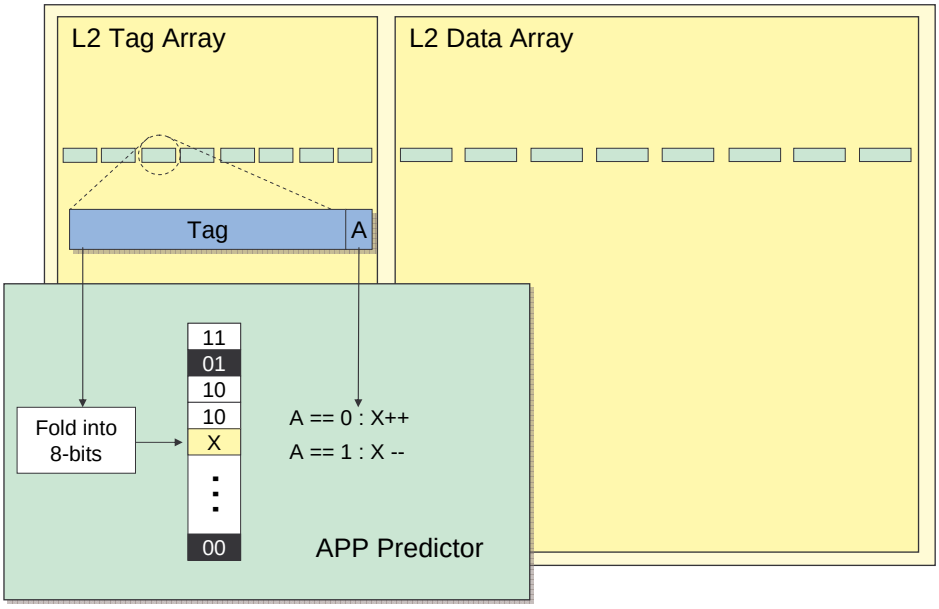


Figure 2.3: Updating the APP predictor

accessed during its recent residence (its "Accessed" bit is 0). When later the local L2 cache suffers a miss to the same block, the prediction table is looked up. If the value in the saturating counter is 1 or higher, the block is placed in a remote L2 cache. Otherwise, the block is placed in the local L2 cache. The initial values of the saturating counters in the predictor table are 0, hence blocks are initially always placed locally.

In our design, we choose 2-bit saturating counters. We tried 1-bit, 2-bit, and 3-bit saturating counters, and found that there is a slight performance advantage in using 2-bit over 1-bit counters, but there is no performance advantage in using 3-bit over 2-bit counters.

The storage overhead for APP predictor table with 256 entries is 512 bits (or 32 bytes). Compared to a 1MB L2 cache with a 64B block size, the total overheads, including the Accessed bits, is $512bits + 16384 \times 1bits = 2.06KBytes$, which is a negligible 0.2% area overhead compared to a 1MB cache.

Handling a Miss and Remote Cache Hit.

Figure 2.4 illustrates how APP handles a global cache miss, i.e., miss in the local and remote caches. While the Spiller's L2 miss is outstanding, the APP predictor is looked up to determine whether the incoming block should be placed in the local L2 or a remote cache (Step 1 in the figure).

If it is determined that the incoming block should be placed in the local L2, the LRU block is victimized for the incoming block (Step 2a). The victim block is moved to a randomly-selected remote Receiver L2 cache. The random selection of a Receiver cache is appropriate in a bus-based CMP we assume, because the cache latency is uniform across all remote caches. For a NUCA CMP, however, a distance-aware selection is warranted. We leave this as future work.

If the APP predictor determines that the incoming block should be placed in a remote

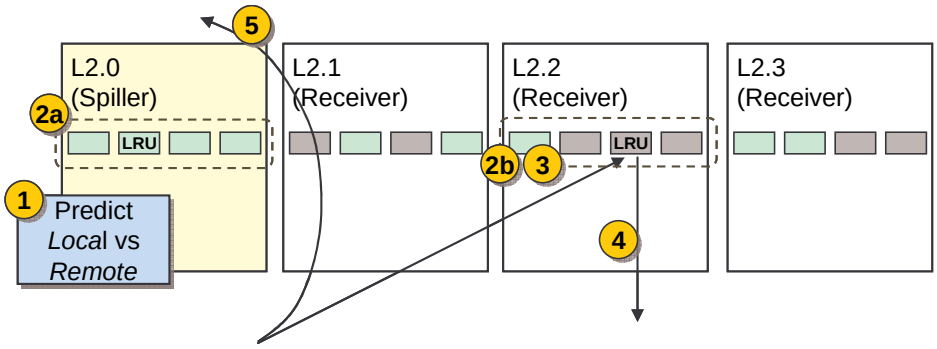


Figure 2.4: Handling cache-misses with APP (assumes placement prediction is *Remote*)

cache, a random remote Receiver cache is selected, and the LRU block there is replaced by the incoming block (Step 2b). The incoming block is marked as the MRU block at the remote cache (Step 3), while the remote victim block is discarded or written back if dirty (Step 4). The incoming block is always supplied to the requesting processor and its L1 cache (Step 5).

Figure 2.5 illustrates how APP handles a remote cache hit (upon a local L2 cache miss). The APP predictor is looked up to determine whether the remotely hit block should be left where it was or be brought into the local L2 cache (Step 1). If the predictor determines the block should be brought into the local L2, the remotely hit block is swapped with the LRU block in the local L2 cache set (Step 2) and marked as the MRU block. Otherwise, the remotely hit block remains in the remote cache, but becomes the MRU block there. The remotely hit block is supplied to the processor and its L1 cache (Step 3).

Handling processor writes and cache write-backs. Maximizing the performance gain of any capacity sharing mechanism requires the local L2 cache to not enforce the inclusion property with the local L1 cache. The reason is that it relies on the ability

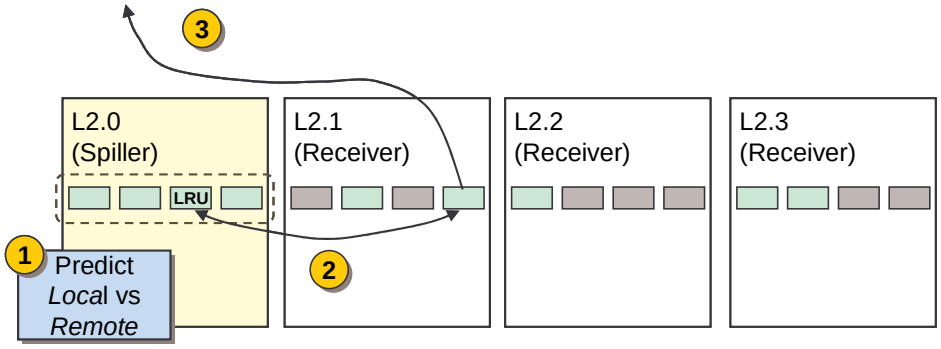


Figure 2.5: Handling remote-hits with APP (assumes placement prediction is *Local*)

to place a block that is still in the L1 cache in a remote L2 cache. Enforcing inclusion unnecessarily limits the mechanism's potential. Our schemes are not an exception.

Not enforcing the inclusion property in general has the benefit of increasing the aggregate cache capacity of multiple levels of caches as well as reducing the number of back invalidates to hot blocks in the L1 cache. However, at the same time, not enforcing the inclusion property introduces the expense of having to duplicate the L1 cache tag array at the L2 level in order to avoid contention-related stalls when the L1 cache tag is accessed by both the core and the L2 cache snooper.

Since a block can exist in more than one tile (private L1 and L2 that belong to two different cores), if the processor issues a write to a block in the L1, the coherence protocol has to post an "invalidate request" on the bus such that all other tiles invalidate their copies. Once the issuing core receives an ack from other tiles, it can go ahead with write request. Note that this does not necessarily impact performance since writes are not on the critical path and can happen in the background.

When the L1 initiates a writeback of a modified block to lower levels, if the block exists in the local L2, the written back block is merged with the local copy, otherwise a

new copy is allocated in the local L2. Note that, on a writeback of a modified block, the block would not exist in other L2 caches since there is only one modified copy on chip.

2.2.4 Coherence Protocol Modifications

We assume that L2 caches are interconnected with a shared bus, kept coherent with a MESI broadcast/snoopy coherence protocol. Upon a local cache miss, the APP predictor is consulted, and the miss is broadcast on the bus.

If the block exists in any remote cache, and APP predictor determined local placement, then a swap transaction is initiated. A swap transaction places the local victim block in a special buffer to create a placeholder for the remote block. Then, the remote block is placed on the bus using a regular cache-to-cache transfer, and is picked up by the requesting cache. Finally, the local victim is pushed to the remote cache. The first two steps are already supported in current MESI protocols. The last step, pushing a block to a remote cache, is already supported in some systems, such as the IBM WireSpeed CMP [21]. These steps do not involve a load or store instruction, so there are no critical path or memory consistency concerns. The only modifications to the cache coherence protocol are: (1) a new swap transaction to distinguish it from regular read miss, (2) a transient state in the coherence protocol to indicate the swap is pending completion, and (3) potentially additional buffers to hold swapped blocks temporarily.

APP can also be adapted (with some modifications) for tile-based CMPs where caches are interconnected with point-to-point links and kept coherent using a directory-based protocol. Such a system has a non-uniform cache architecture (NUCA), and is beyond the scope of this work. However, we will comment on what changes are needed to adapt APP to such a system. One modification is that since cache latency depend on the physical distance from a given core, APP must be modified to take remote cache distance into

account, e.g., by restricting remote placement to near or nearest neighbor caches. In addition, the outcome of APP predictor (local or remote placement) must be sent along the miss request to the directory at the home node. Local placement is handled in a traditional way. Remote placement requires the directory to identify which cache the block should be sent to, considering the physical distance. Finally, the determination for a remote cache hit is more difficult, because the directory information may contain stale information, for example when a clean block in a remote cache is evicted silently. To avoid delaying global miss resolution, a possible solution is for the directory to simultaneously inquire the remote cache and fetch the block from main memory. This avoids delay in fetching from main memory in the case that the remote cache no longer has the block, at the expense of additional power consumption. An alternative solution is to distinguish remotely placed blocks from local blocks, and require the directory to be notified when a remotely placed block is evicted (even when it is clean). Finally, co-ordination with the directory is needed when blocks in the local and remote cache are swapped.

2.2.5 Ensuring Receiver QoS

While capacity sharing is appealing due to significant performance improvement Spiller applications may achieve, it may reduce the performance of Receiver applications sufficiently to violate some Quality-of-Service (QoS) requirements. Such a situation must be avoided.

We augment APP with a *Miss-Rate Monitoring System* (MRMS) for the Receivers. We leverage the fact that each Receiver cache has 32 sets dedicated to spilling. MRMS monitors the miss rates of the spill sets and other sets. The spill sets record the miss rate if the cache does not accept victim blocks from other caches. If the miss rate in other sets is higher by more than 5% compared to the miss rate of the spill sets, the

Receiver disengages from participating in APP and no longer accepts victim blocks from Spiller caches. Receiver accepts victim blocks on the premise that it has excess cache capacity that it does not need. Suffering from $> 5\%$ increase in miss rate indicates that the donated cache capacity is needed for performance. The miss rate comparison is performed at every 3 million cycle boundaries. With our MRMS safeguard, across all cases in our experiments, the execution time perturbation of Receiver applications is negligible. It is worth mentioning that the 5% and the 3 million cycle thresholds are tunable parameters, and the designer can specify the thresholds which best fit the workload environment.

2.3 Methodology

Simulation Model. To evaluate our capacity sharing schemes with oracular placement/replacement, oracular placement, and realizable placement, we build a cycle-accurate multicore machine model on top of Simics [44], a full system simulation platform. We model a 4-core CMP system, where each core has private L1 caches and a private L2 last level cache. Each of the L2 cache is has 1 MB size, 8-way associativity, and access latencies derived from Cacti 6.0 [50]. We also assume that the L2 caches are interconnected with a shared bus, and are kept coherent with a MESI broadcast/snoopy coherence protocol. MESI protocol already has a support for facilitating cache to cache transfer. Table 2.1 lists all relevant configuration parameters used in our experiments.

Multi-Programmed Workloads. Before constructing workloads for this study, we first separate potential Donors from potential Acceptors among 19 C/C++ SPEC CPU2006 [65] applications. If they benefit from $> 10\%$ increase in L2 cache hit rates when the L2 cache capacity is increased from 1MB to 4MB, they are categorized as Accep-

Table 2.1: System Configuration

Cores	3GHz 4cores, in-order single issue
Private L1 I/D caches	32KB, 2-way, 2-cycle access latency 64B block size
Private L2 caches	1MB, 8-way, 10-cycle local hit (or 512KB, 8-way, 7-cycle local hit) (or 2MB, 8-way, 12-cycle local hit) 40-cycle remote L2 cache hit 64B block size LRU replacement policy
Memory	300-cycle access latency

tors, otherwise they are categorized as Donors. Acceptors display two distinct temporal locality patterns. Some Acceptors show a *perfect-LRU* temporal locality behavior, where more recently used blocks have a greater reuse probability than less recently used blocks. Other Acceptors have some *anti-LRU* behavior, where recency of use mostly does not correlate with reuse potential. Figure 2.1 in section 2.1 shows applications with each of these two temporal locality behaviors. Donors, on the other hand, cannot benefit from an increase in the cache space from 1MB to 4MB. This is either because they have small working sets that fit in a 1MB L2 Cache, or they have a streaming data access pattern which suffers a high miss rates regardless of any additional cache capacity. Consequently, we divide Donors further into two groups - small-working-set and streaming.

The applications and the category they fall into are shown in Table 2.2. The first two rows contain *all* applications we can identify as Acceptors in SPEC CPU2006. The next two rows contain a subset of applications we can identify as Donors in SPEC CPU2006. We chose only a subset of all Donors because we observed that Donors behave similarly from the point of view of the insights we hope to gain from this study. All Donors with a small working set share the same trend (L2 cache miss rate of less than 2%); therefore,

we use a small representative set for such Donors. Similarly, all Donors with a streaming data access pattern behave similarly (less than 2% reduction in miss rate upon increasing the cache from 1M to 4MB).

Table 2.2: Acceptor and Donor applications

Acceptors (anti-LRU)	omnetpp, xalancbmk, soplex, hmmer
Acceptors (perf-LRU)	bzip2
Donors (small WS)	namd, povray, sjeng
Donors (streaming)	milc, libquantum

In order to cover all distinct workload scenarios, we choose fifty 4-benchmark workloads that cover different ratios of Acceptors and Donors, different types of Acceptors (anti- and perfect-LRU), and different types of Donors (small-working-set and streaming). The workloads are shown in Table 2.3. A workload denoted as A_xD_y represents a workload with x Acceptors and y Donors. The first ten workload mixes have one Acceptor and three Donors. The next fifteen workload mixes have two Acceptors and two Donors. The next fifteen workload mixes have three Acceptors and one Donor. The last ten workload mixes have four Acceptors and no Donors. Acceptor applications are distributed nearly uniformly across all workloads (i.e., each Acceptor appears in almost the same number of workloads). Donors are selected in a similar fashion. However, since all Donors behave the same from the point of view of this study, they might as well be selected randomly.

To test each workload, each of the four applications in the workload are fast-forwarded

Table 2.3: Workloads (Grouped by the Acceptor to Donor Ratio)

Group	Workloads in the group
1 Acceptor 3 Donors (A1D3)	^{M1}{omnetpp, namd, povray, milc}, ^{M2}{xalancbmk, libquantum, namd, namd}, ^{M3}{soplex, milc, povray, namd}, ^{M4}{bzip2, sjeng, namd, namd}, ^{M5}{omnetpp, milc, milc, namd} ^{M6}{xalancbmk, povray, namd, sjeng}, ^{M7}{soplex, libquantum, libquantum, milc}, ^{M8}{bzip2, milc, sjeng, libquantum}, ^{M9}{hmmer, namd, namd sjeng}, ^{M10}{omnetpp, milc, povray, lib}
2 Acceptors 2 Donors (A2D2)	^{M11}{omnetpp, xalancbmk, namd, povray}, ^{M12}{omnetpp, hmmer, milc, namd}, ^{M13}{xalancbmk, bzip2, libquantum, sjeng}, ^{M14}{omnetpp, bzip2, namd, sjeng}, ^{M15}{xalancbmk, hmmer, povray, libquantum} ^{M16}{soplex, bzip2, milc, povray}, ^{M17}{soplex, hmmer, sjeng, sjeng}, ^{M18}{omnetpp, omnetpp, povray, povray}, ^{M19}{xalancbmk, xalancbmk, namd, milc}, ^{M20}{bzip2, bzip2, milc, libquantum}, ^{M21}{omnetpp, hmmer, libquantum, milc}, ^{M22}{xalancbmk, omnetpp, milc, namd}, ^{M23}{soplex, xalancbmk, libquantum, povray}, ^{M24}{bzip2, hmmer, namd, sjeng}, ^{M25}{hmmer, hmmer, sjeng, sjeng}
3 Acceptors 1 Donor (A3D1)	^{M26}{soplex, xalan, hmmer, povray}, ^{M27}{omnetpp, hmmer, bzip2, povray}, ^{M28}{soplex, xalancbmk, hmmer, milc}, ^{M29}{xalancbmk, bzip2, milc, hmmer}, ^{M30}{omnetpp, xalancbmk, xalancbmk, namd} ^{M31}{xalancbmk, xalancbmk, xalancbmk, namd}, ^{M32}{xalancbmk, hmmer, hmmer, povray}, ^{M33}{soplex, bzip2, hmmer, milc}, ^{M34}{soplex, bzip2, xalancbmk, libqunatum}, ^{M35}{soplex, sjeng, xalancbmk, libquantum} ^{M36}{xalancbmk, bzip2, bzip2, milc}, ^{M37}{omnetpp, xalancbmk, bzip2, libquantum}, ^{M38}{soplex, soplex, xalancbmk, povray}, ^{M39}{bzip2, hmmer, soplex, sjeng}, ^{M40}{omnetpp, libquantum, soplex, soplex}
4 Acceptors 0 Donors (A4D0)	^{M41}{omnetpp, xalanckbmk, xalancbmk, hmmer}, ^{M42}{bzip2, hmmer, hmmer, xalancbmk}, ^{M43}{omnetpp, soplex, soplex, soplex}, ^{M44}{omnetpp, hmmer, omnetpp, omnetpp}, ^{M45}{bzip2, omnetpp, bzip2, xalancbmk} ^{M46}{hmmer, bzip2, omnetpp, hmmer}, ^{M47}{bzip2, soplex, soplex, hmmer}, ^{M48}{hmmer, hmmer, hmmer, hmmer}, ^{M49}{bzip2, bzip2, bzip2, omnetpp}, ^{M50}{xalancbmk, xalancbmk, xalancbmk, xalancbmk}

for 10 billion instructions in order to skip their initialization phase. After that but prior to statistics collection, the cache models are warmed for 1 billion cycles. Finally, timing simulation is started and each workload is run until the slowest application runs for 250M instructions.

2.4 Results and Analysis

In this section we evaluate the performance of APP against a base case of private caches without capacity sharing (Base), Cooperative Caching (CC) [12], Dynamic Spill Receive (DSR) [52], Oracular Placement Policy (OPP), and Oracular Placement-Replacement Policy (OPRP).

We use the following metrics to evaluate performance and fairness of the 50 multi-programmed workloads: Weighted Speedup, Harmonic Mean, and Throughput. These three metrics and their significance have been described in detail in prior work [20]. Table 2.4 defines these metrics.

Table 2.4: Performance metrics used and their definitions. $IPC_{i,base}$ represents the IPC of an application running alone on core i without capacity sharing enabled, while IPC_i represents the IPC of an application running on core i with capacity sharing enabled.

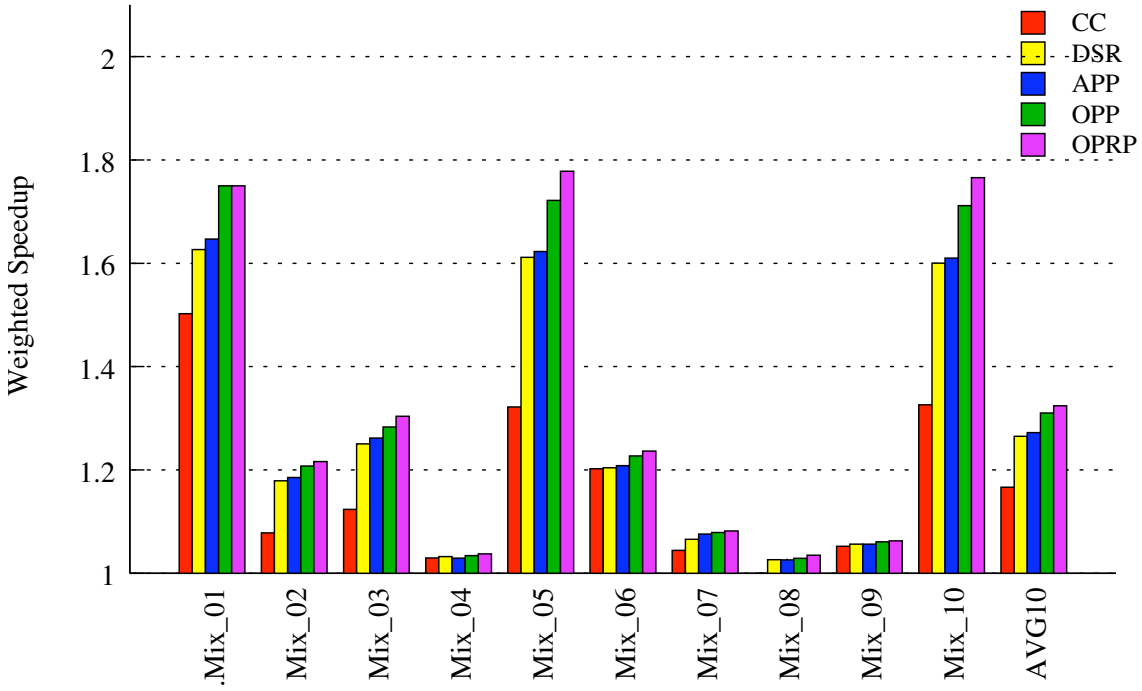
Metric	**Formula**
Weighted Speedup	$\sum_{i=1}^{n}(IPC_i/IPC_{i,base})$
Harmonic Mean	$n/\sum_{i=1}^{n}(IPC_{i,base}/IPC_i)$
Throughput	$\sum_{i=1}^{n} IPC_i$

Of these metrics, Weighted Speedup is the preferred metric [20, 62]. It corresponds to a physical, system-level measure of performance - the number of instructions executed across *all* applications in a multiprogram mix per unit of time. This metric equalizes the

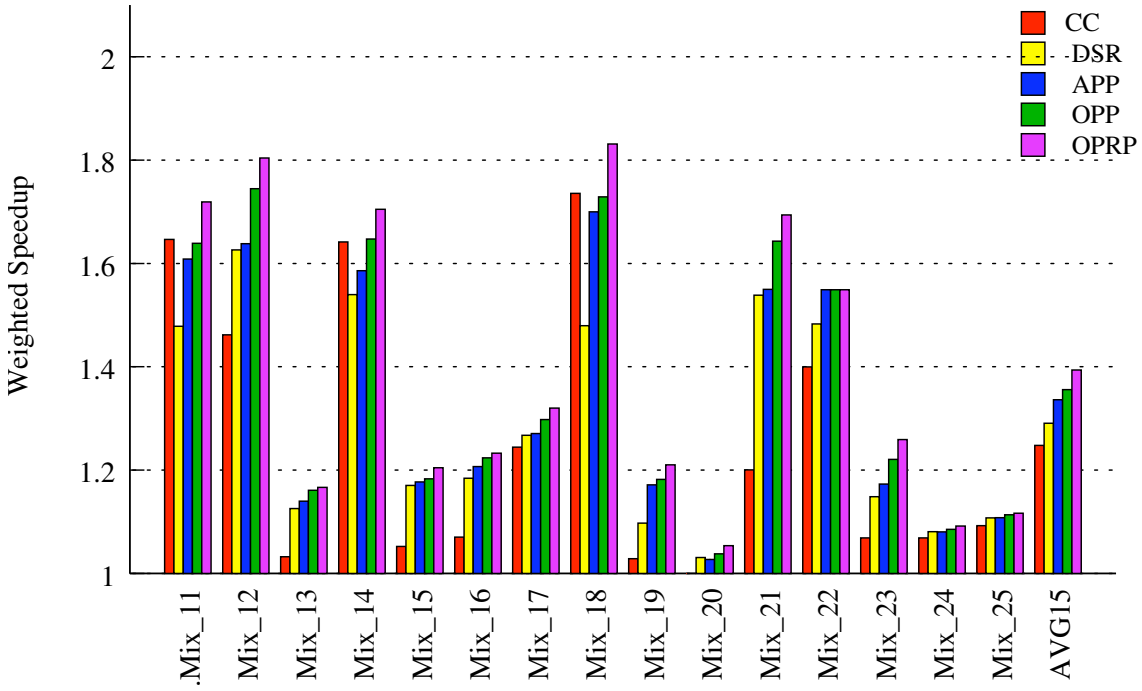
contribution of each program in the mix by normalizing its performance in the mix to its performance when run in isolation. Weighted Speedup does not bias the measured performance by favoring high-IPC applications.

2.4.1 Performance of APP, OPP, and OPRP

Figure 2.6 and Figure 2.7 show the Weighted Speedup results for Cooperative Caching (CC), Dynamic Spill Receive (DSR), our scheme APP, and our two oracular schemes - oracular placement (OPP) and oracular placement-replacement (OPRP). Recall that CC provides capacity sharing where all applications can spill their victim blocks to other caches, while DSR improves upon it by detecting Acceptor applications and only allowing them to spill to other caches. The DSR implementation is based on the code provided by the authors of DSR [52], integrated into our simulation infrastructure. The results are arranged into four charts based on the Acceptor to Donor ratio in each group of workloads. The weighted speedups are normalized to the base case of a CMP with private caches without capacity sharing. The average weighted speedups over all 50 workloads are shown in the last set of bars.

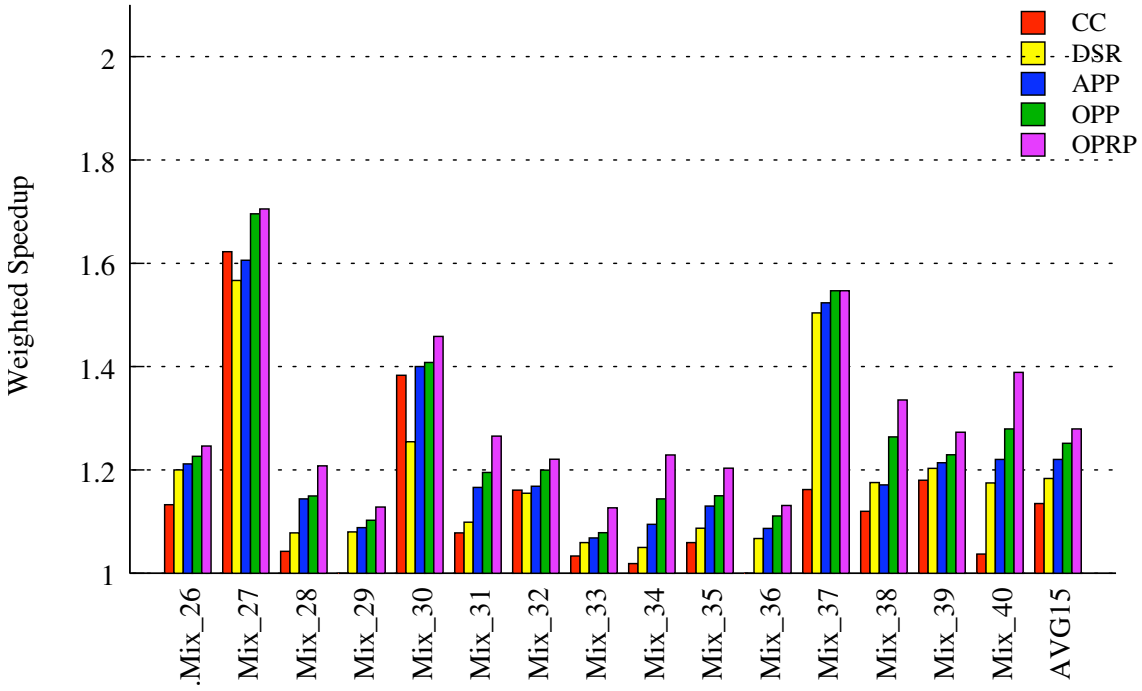


(a) A1D3 workloads

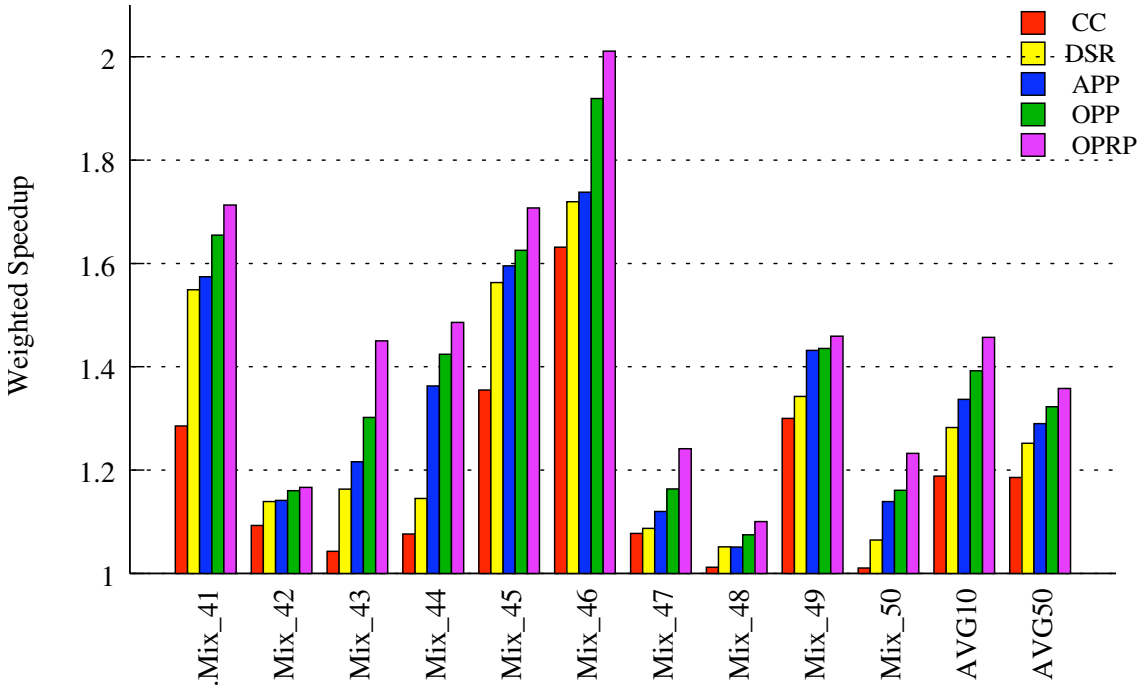


(b) A2D2 workloads

Figure 2.6: Normalized Weighted Speedup for A1D3 and A2D3 workloads compared to the base case of private caches without capacity sharing.



(a) A3D1 workloads



(b) A4D0 workloads

Figure 2.7: Normalized Weighted Speedup for A3D1 and A4D0 workloads compared to the base case of private caches without capacity sharing, with average for all 50 workloads shown as the last set of bars in part (b).

There are two main observations that can be made from the figures. First, all schemes improve the Weighted Speedup for all workloads compared to the base case of private caches without capacity sharing. There is, however, a varying gap between the Weighted Speedup improvements across the various schemes. On average, over fifty workloads, CC and DSR improve the Weighted Speedup by 19% and 25% respectively, while the hypothetical OPP and OPRP improve the Weighted Speedup by 32% and 36% respectively. The Weighted Speedup improvement from the hardware-implementable APP falls right between DSR and OPP, averaging 29%.

Second, across different workload groups, the performance improvement of APP compared to DSR varies. The performance improvement of APP over DSR for A1D3 is the smallest (27.3% versus 26.5%) because only one Acceptor application benefits from APP's remote block placement, whereas the Weighted Speedup reflects the average improvement for all applications running together. Thus, the more acceptor applications in a workload, the more the Weighted Speedup is improved with APP compared to DSR. APP's average Weighted Speedup improvement for A2D2 is 33.6% versus DSR's 29%. For A3D1, APP achieves an average Weighted Speedup of 22% compared to DSR's 18.2%. Finally, for A4D0, APP achieves 33.7% improvement in Weighted Speedup while DSR achieves 28.2%.

At first glance, the average improvement of Weighted Speedups achieved by APP over DSR may seem modest. However, there are three important points to consider. First, improving the performance over the state-of-the-art technique is not a minor feat. While APP's improvement over DSR may seem modest, DSR's improvement upon CC, which was the state-of-the-art of the time, may similarly seem modest (25% vs. 19%).

Second, recall that our workloads are designed to cover a wide range of workload mixes and behavior. For five workloads, APP outperforms DSR by more than 5%, and

sometimes by significantly more (18%). Such a difference is significant considering that over DSR, APP relies on reducing cache hit latencies by only 30 cycles (converting a modest remote cache hit latency of 40 cycles to local cache hits latency of 10 cycles). As the number of cores on a chip grows, we expect that the ratio of remote cache hit latency to local cache hit latency will grow substantially, causing APP's benefit to increase as well. We quantify this in a later section when we vary the remote cache hit latencies (Section 2.4.5).

Finally, in all the workloads, APP is almost never outperformed by DSR, suggesting that APP rarely makes mistakes that result in the decrease of local hit rates versus DSR.

2.4.2 In Depth Analysis of APP's Performance

To understand why APP outperforms CC and DSR but is outperformed by OPP and OPRP, Figure 2.8 shows the average *local cache hit rate* for each group of workloads. The hit rate for Acceptors in each group of workloads is calculated by averaging the hit rates of all Acceptor applications.

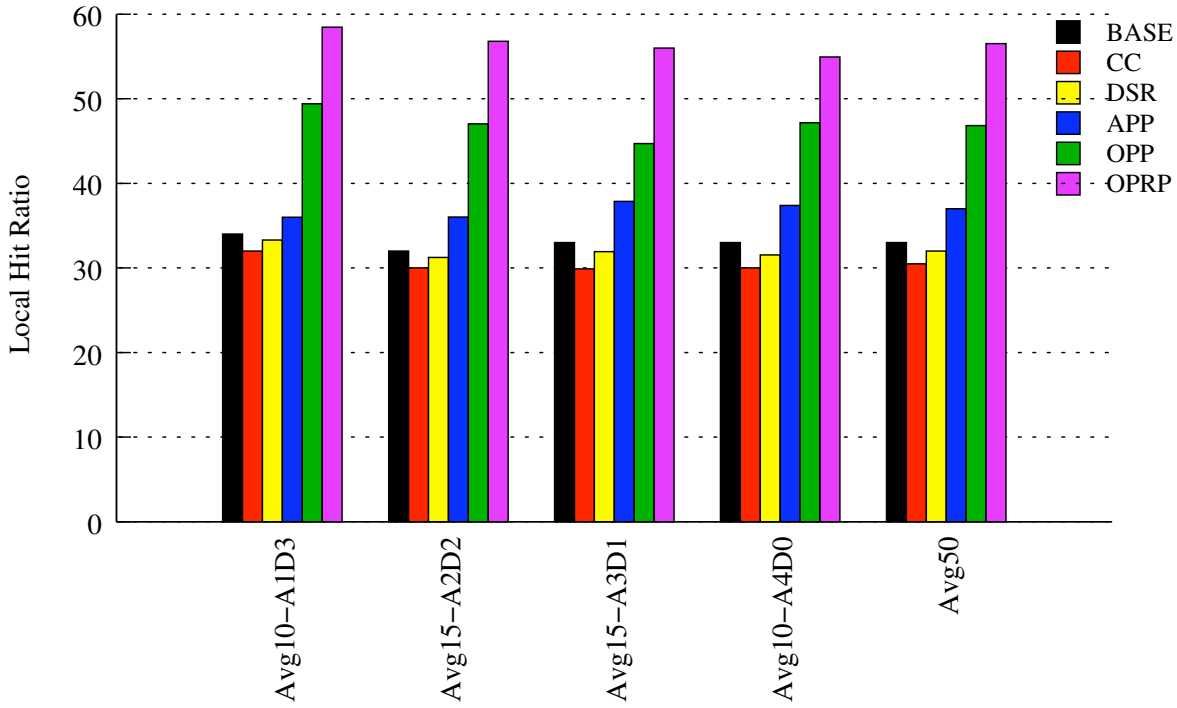


Figure 2.8: Local L2 hit rate comparison for all groups of workloads for the base case of private caches without capacity sharing (BASE), CC, DSR, APP, OPP, and OPRP

We can make the following observations from Figure 2.8. First, we can see that CC experiences the lowest local L2 hit rates (30.5%). Recall that in CC any core is allowed to spill victim blocks to any core regardless of their cache demand. Hence, the hit rates of Acceptors decrease due to the cache pollution from Donors, compared to the base case of private caches.

Second, DSR and the Base case experience similar local L2 hit rates (32%), with slightly lower hit rates in DSR for all groups of workloads. The similarity in local hit rates is due to the fact that in both policies, a block requested by the Spiller application is always brought into the local cache, and the only difference between them is in how to handle blocks evicted from the local cache. Hence hit rates between them should be

identical if everything else is equal. However, since DSR dedicates 32 sets in each cache to always receive (this is necessary to dynamically identify Spillers/Receivers), DSR's local hit rates are always slightly lower than the Base case.

Third, compared to DSR, APP's local L2 hit rates for Acceptors are about 5% higher for all workload groups (37% vs. 32%). This result demonstrates that by selectively placing incoming blocks in remote caches, APP is successful in retaining more useful blocks in the local L2 cache, thus improving the local L2 hit rates. The increase in local L2 cache hit rates is responsible for the performance improvement APP achieves over DSR.

Finally, while APP improves the hit rate significantly over the base case, there remains a gap between Base and OPP hit rates (37% vs. 46.5%). The gap is caused by OPP making placement decisions based on future accesses, whereas APP makes decisions based on past access patterns. In addition, APP uses a relatively simple predictor design. More sophisticated predictor designs can narrow the gap with OPP significantly, at the expense of increased complexities (Section 2.4.3).

Comparing the average local L2 cache hit rates achieved by OPP versus DSR (46.5% vs. 32%) and versus OPRP (46.5% vs. 56%), we can infer that the majority (roughly 60%) of the performance gap between DSR and OPRP is due to the intelligent placement of blocks, while the remaining is due to perfect versus LRU replacement policies. This corroborates our initial hypothesis that selective placement of cache blocks plays a major role in boosting the performance of capacity sharing schemes, and that our priority should be in providing intelligent placement of blocks, ahead of improving the cache replacement policies.

Workloads that enjoy significant performance improvement in APP (e.g., M11, M31, and M44) are also ones that are dominated by Acceptor applications with anti-LRU

temporal reuse patterns. The average local hit rate of the Acceptors in each workload for APP versus DSR are 15% vs. 7% (M11), 20.3% vs. 9.3% (M31), and 22% vs. 5% (M44), respectively. Since remote placement of blocks is the only difference between APP and DSR, the results clearly point to the significant role placement decisions play in improving local cache hit rates of Acceptor applications. The increase in local hit rates also demonstrates that by allowing anti-LRU incoming blocks to be placed in remote caches, existing blocks in the local L2 cache enjoy better temporal locality from less cache perturbation.

A small number of workloads show almost negligible performance improvement, or a slight degradation over DSR (e.g., M25, and M04). Acceptors in these workloads (example bzip2 and hmmer) show an almost perfect-LRU behavior and very little anti-LRU behavior (Figure 2.1). These workloads already experience high local hit rates (66% for M25, and 82% for M04), hence most of the performance improvement can only come from additional cache capacity rather than converting remote cache hits into local cache hits.

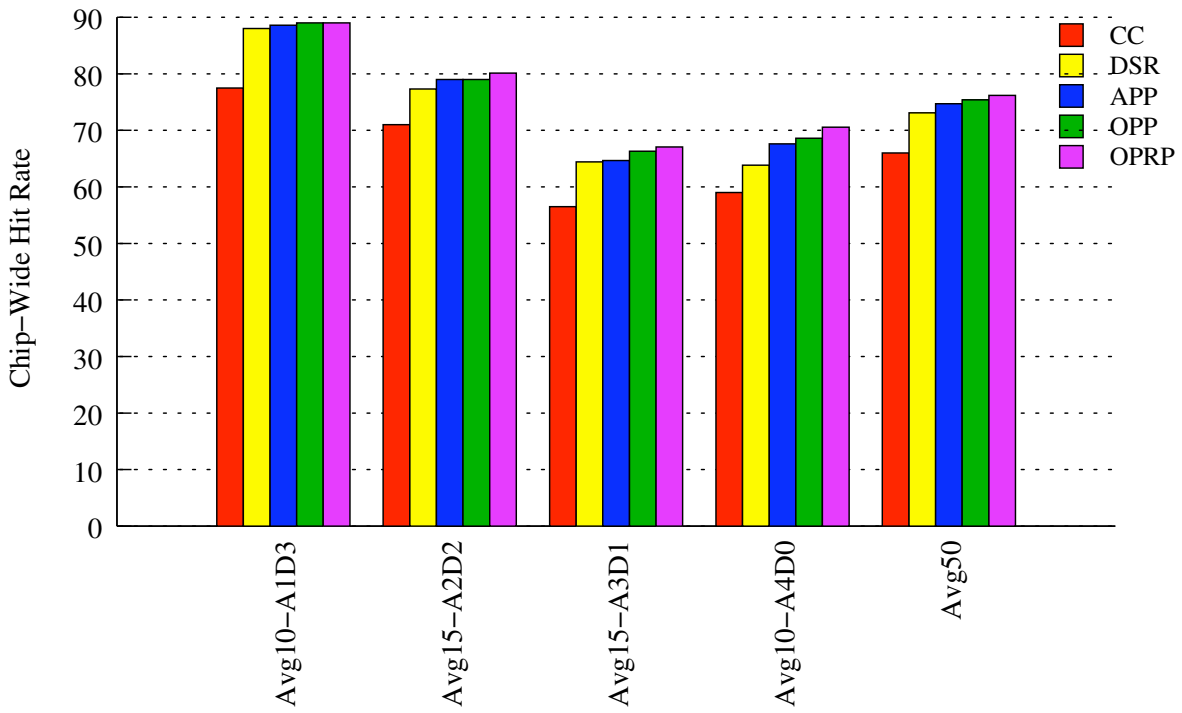


Figure 2.9: Chip-Wide L2 hit rate comparison for all groups of workloads for CC, DSR, APP, OPP, and OPRP

Chip-Wide Hit Rate. Earlier, we have shown that APP increases the local L2 cache hit rate for Acceptor applications by about 5% compared to DSR. Such an increase in the local cache hit rate is sufficient to make APP's performance better than DSR's. However, this assumption holds true if and only if the increase in the local cache hit rate *does not* come at the expense of an inordinate drop in remote cache hits.

Figure 2.9 shows the average *chip-wide L2 hit rates* (the total number of L2 hits that are satisfied by the local or remote caches, divided by the total number of L2 cache accesses) for all Acceptors in a given group of workloads. This metric considers both local and remote hits.

CC manages to bridge most of the gap by improving the chip-wide hit rates significantly over the base - from 33% (base) to 66% (CC). DSR improves upon CC and achieves

an average of 73%. Not only is APP able to maintain this chip-wide hit rate, but it also manages to increase it slightly. It is interesting to note that APP continues to outperform DSR even in this metric. The reason for this is that by matching the temporal locality of each cache block with the its placement position (local vs. remote), APP is able to reduce perturbation to both the local and the remote cache space, resulting in a slight improvement in chip-wide hit rates.

Worst Case Performance of APP. We discussed in Section 2.4.1 that applications with an almost perfect-LRU behavior are not expected to benefit from APP since all incoming lines in such an application should be placed locally. However, if APP cannot improve the performance of some applications compared to DSR, it should not degrade them either. Our results confirm that APP's maximum performance degradation compared to DSR, across all workloads, is a negligible 0.5% (M20). This is in contrast to APP outperforming DSR by 18.2% in the best case (M44), and by 3% on average.

Finally, Applications in a given workload mix are continually adapting to act as either Spillers or Receivers. Therefore, it is critically important that the performance improvement for Spiller caches does not come at the expense of degrading any Receiver cache's performance. We find that the threshold of APP's Miss-Rate Monitoring System (Section 2.2.5) is rarely triggered for the QoS level we chose (5% hit-rate decline), confirming the resilience of most Receiver caches to losing some of their cache space. Further, only 4% of all applications (8 out of all the 200 applications in the 50 workload mixes) suffered an IPC degradation of more than 4% compared to the base case of no capacity sharing. Even for these applications, the worst-case slowdown for a Receiver is only 14%, which is more than completely offset by the speedup improvement in Spiller applications. In comparison, the worst-case slowdown for a Receiver in DSR is 13%.

Overall, the experimental results stress that APP outperforms local block placement

schemes because of the improvement in local miss rate (more blocks are found in the local L2 cache), as well as the slight improvement in the chip-wide miss rate (more blocks are found in the L2 caches on chip). Further, the performance gain in anti-LRU applications does not come at the expense of degrading perfect-LRU applications. Finally, APP safeguards Receiver applications from being adversely affected as they donate excess cache capacity to Spiller applications.

2.4.3 Impact of Predictor Design

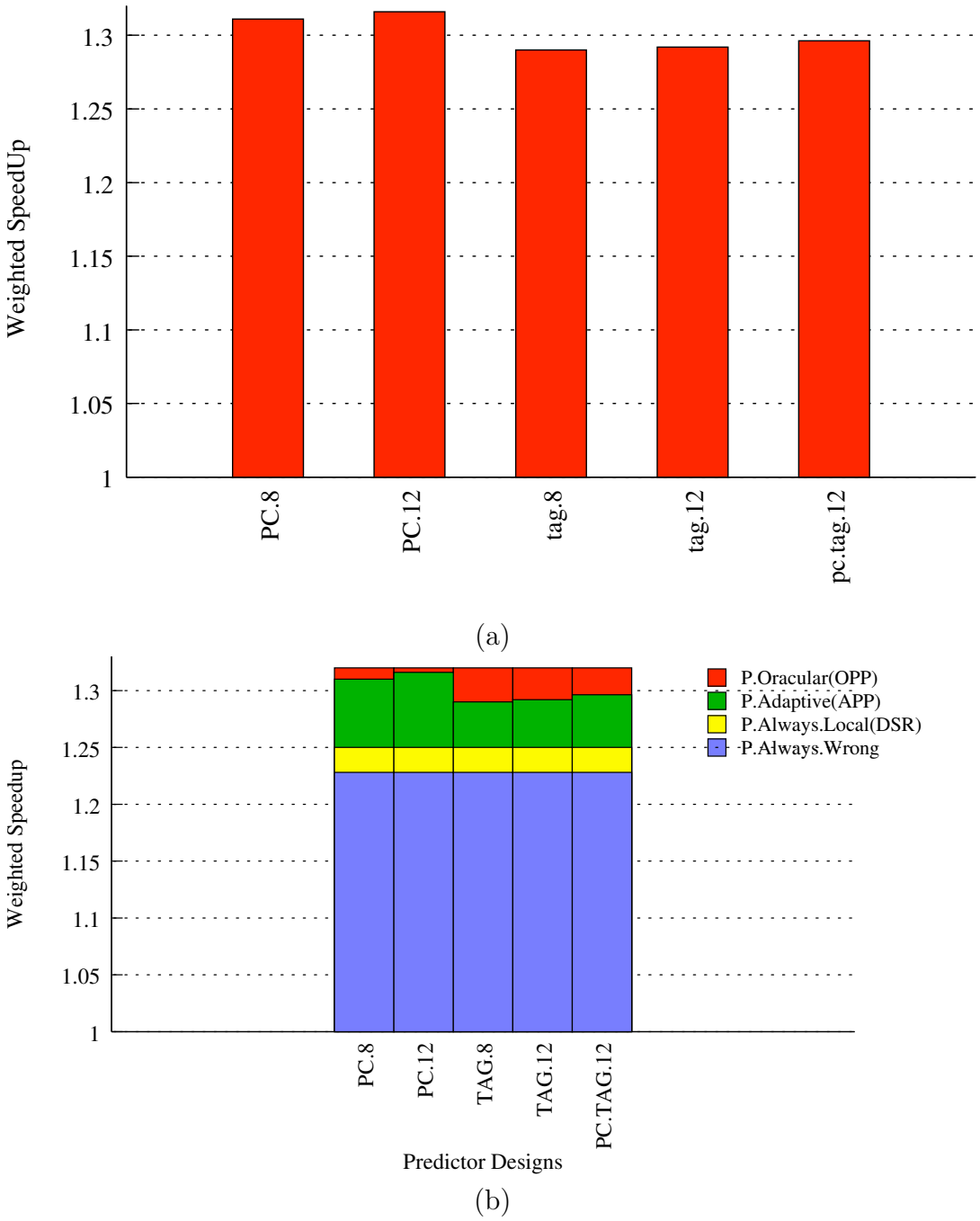
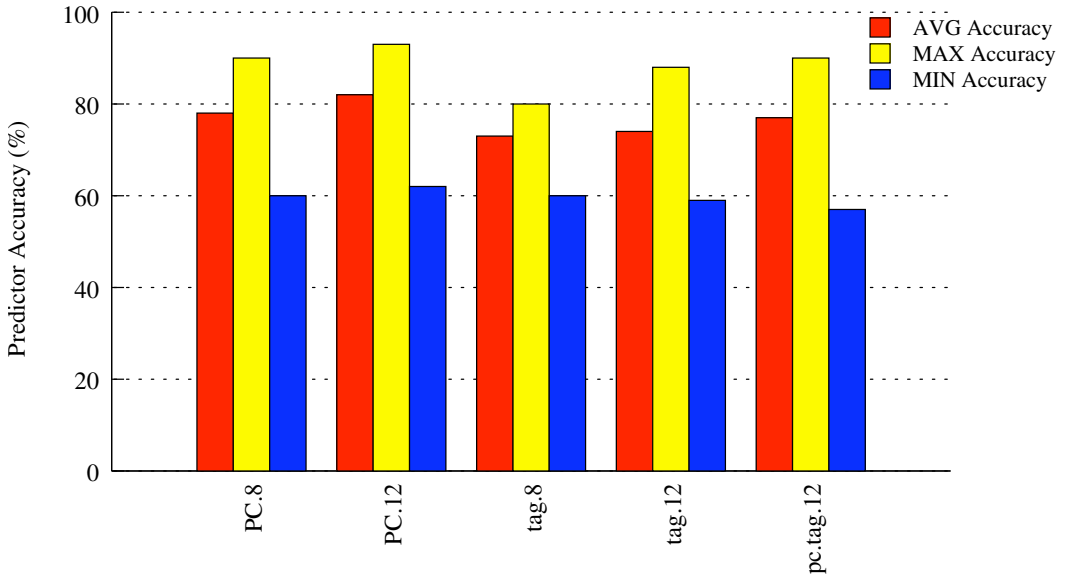
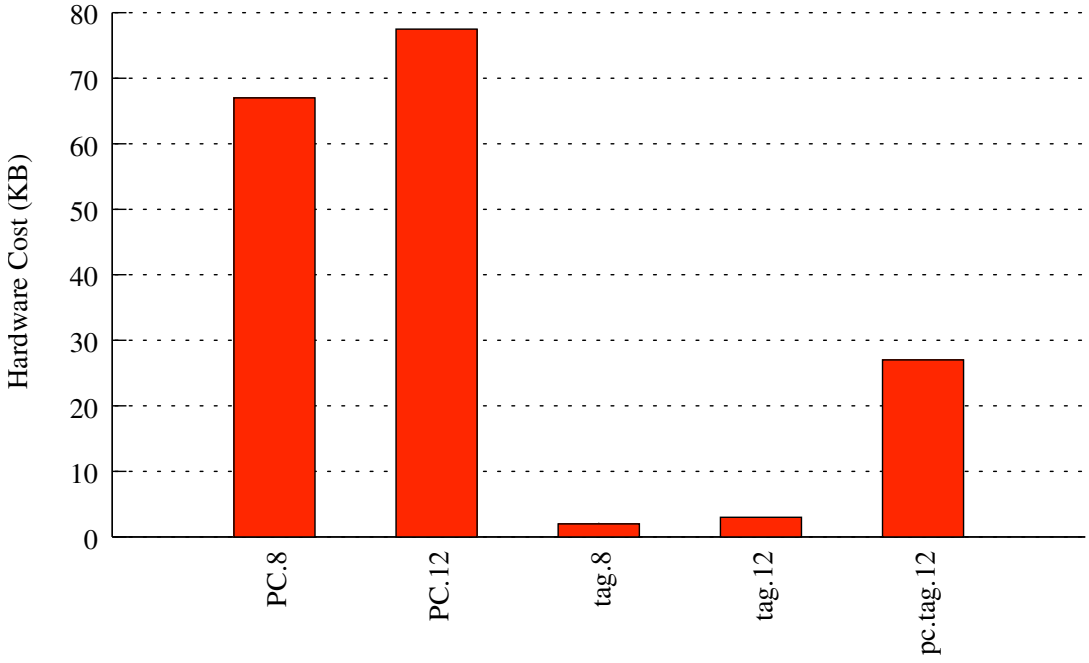


Figure 2.10: Average weighted speedup over base (a), breakdown of speedup (b).



(a)



(b)

Figure 2.11: Average, minimum, and maximum accuracy for various predictors (a), and hardware cost for various predictors (b).

In Section 3.1, we described three possible implementations for APP predictor. Figure 2.10 and Figure 2.11 show the performance of the various designs with other parameters fixed according to Table 2.1. Each predictor is denoted as X_Y where X is the predictor type (PC for PC-based predictor, tag for address-based predictor, and PC.tag for a hybrid PC and address-based predictor); while Y is the number of index bits and the size of the table (2^Y entries).

Figure 2.10(a) shows the average weighted speedup across all 50 workloads normalized to the base case of private caches. It shows that PC-based predictors outperform address-based and hybrid predictors. To analyze the source of performance improvements, Figure 2.10(b) breaks down the contributions to speedups from various factors. The lowest component of bars represents a dummy predictor that always makes the opposite placement decision (local vs. remote) vs. OPP. The second component represents the additional speedup from DSR (always placing blocks locally). The third component represents the additional speedup from APP with various predictor designs, whereas the last component represents the additional speedup from OPP. The figure shows that the headroom for improving speedup from the default APP predictor (TAG.8) is relatively small. The PC.12 predictor almost completely bridges the gap of APP and OPP, while PC.8 predictor shows a close performance as well.

The reason why PC-based predictors perform better than address-based and hybrid predictors can be seen in the predictor accuracies in Figure 2.11(a). PC.12's average accuracy is 82% (93% maximum and 62% minimum), while TAG.8's average accuracy is 73% (80% maximum and 60% minimum). The superiority of PC-based design can be attributed to the few number of PCs in a program phase contributing to most of cache misses, and the uniform temporal reuse behavior for all blocks accessed by the PCs. The address-based predictors, on the other hand, map many more block addresses to a small

number of prediction table entries, hence accuracy relies on the blocks showing identical reuse distance behavior.

However, PC-based predictor's performance superiority must be evaluated against its cost and complexity. Figure 2.11(b) shows the total hardware overheads of the prediction table and the required additional tag information for cached blocks. PC.12 requires 11.5KB prediction table, 33-bit additional tag information per cache block, for a total of 77.5 KB, representing a 7.5% area overhead for a 1 MB cache. TAG.8 is much simpler, only requiring 32-byte prediction table, and 1-bit additional tag information for the "Accessed" bit per cache block, for a total of 2KB, representing a 0.2% area overhead for a 1MB cache. Considering the tiny hardware overhead for address-based predictor, and that it still performs within 3 percentage points in weighted speedup compared to PC-based predictor, we view address-based predictors as more cost effective. Furthermore, a PC-based predictor requires a relatively major change to the entire memory hierarchy, as PC is not normally available at the last level cache (Section 2.2.3).

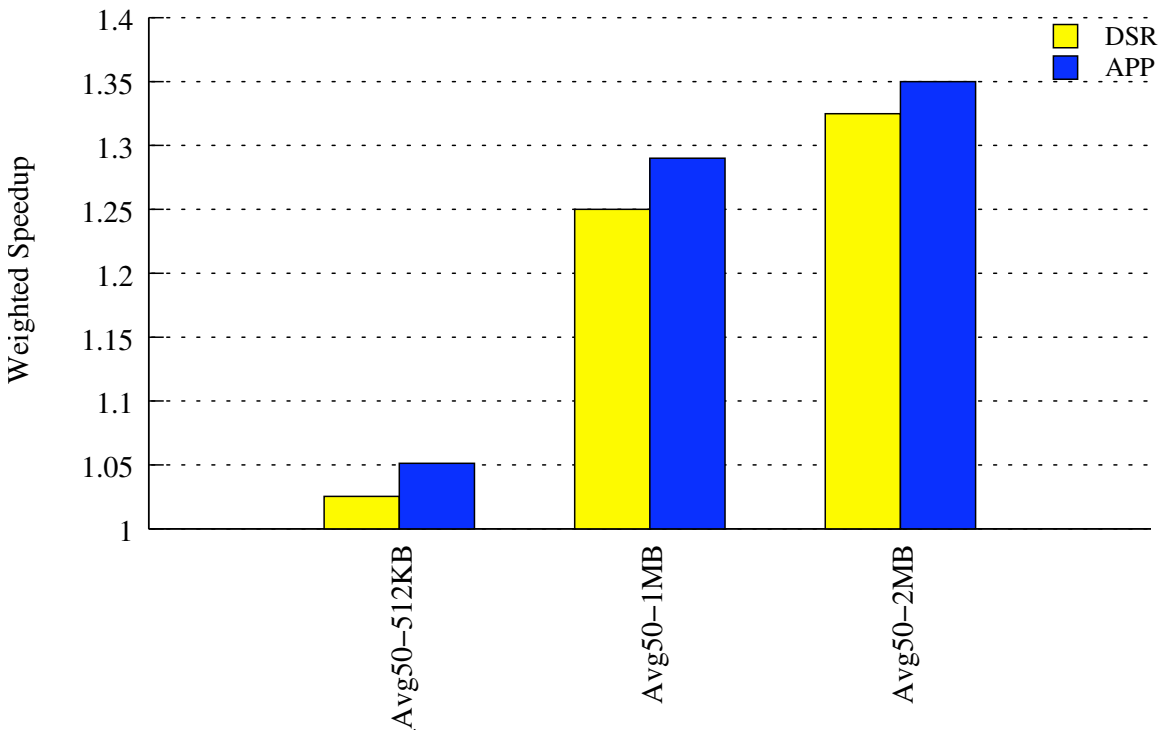


Figure 2.12: Weighted Speedup improvements for APP vs. DSR for different cache sizes compared to the base case of private caches

2.4.4 APP's sensitivity to Cache Size

In this section, we evaluate APP performance when the L2 cache size is varied from 512KB, 1MB (default in other sections), and 2MB. Latencies for various cache sizes are shown in Table 2.1. Figure 2.12 shows the weighted speedup results across various cache sizes, normalized to the base case of no capacity sharing.

The figure shows that as the cache size increases, the average performance improvement from APP (and DSR) increases. The reason is that with larger caches, there is more excess cache capacity that can be donated by Receivers, and there are more benchmarks which change from Spillers to Receivers as their working sets now fit in the cache. The increase in excess cache capacity in the system increase the number of Spillers and

improves the performance of each Spiller.

The figure also shows that across all cache sizes, APP consistently outperforms DSR. However, the relative gap between them narrows with larger cache sizes. The reason is that larger caches also convert many remote cache hits in DSR into local cache hits as the local cache can hold more of the working set of the Acceptor applications. Thus, in a way, larger caches compete with APP in tackling the same problem, which is, improving the local hit rates. However, averages in this case can be misleading, because with larger caches, some Acceptor applications become Donor applications as their working sets now fit in the local cache. Since APP only benefits Acceptor applications, the average performance improvement reflects fewer benchmarks that enjoy significant performance improvement.

2.4.5 APP's Sensitivity to Remote Cache Hit Latency

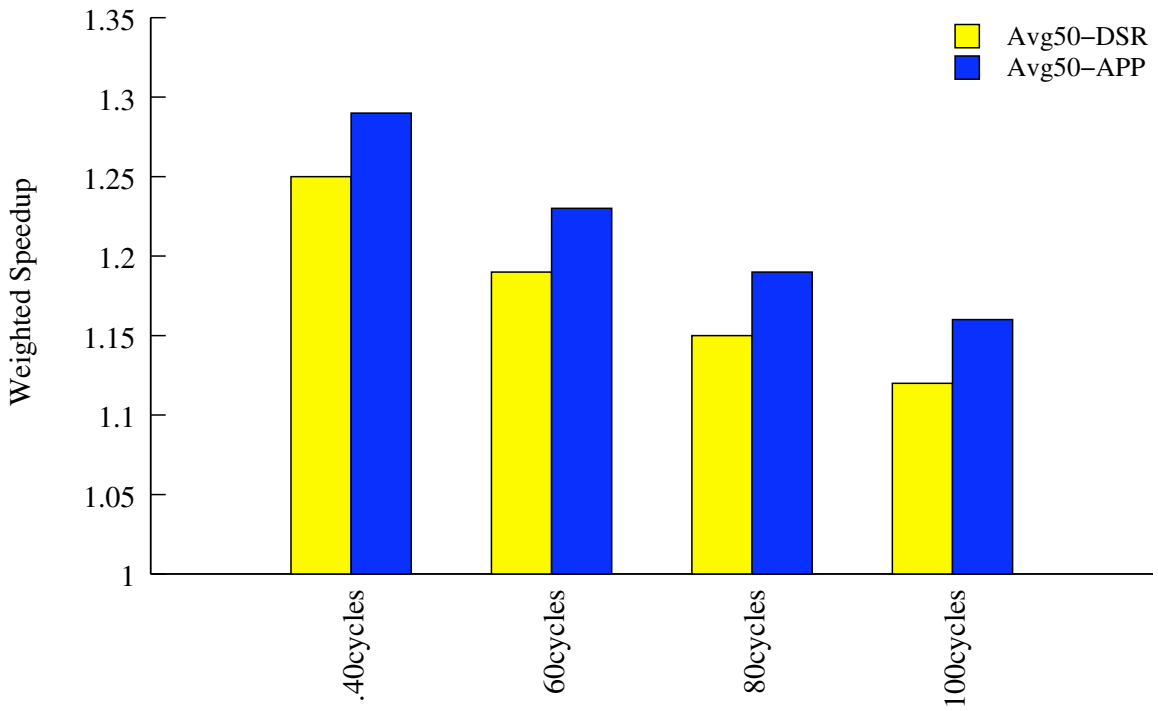


Figure 2.13: Average Weighted Speedup across fifty workloads for APP and DSR with various remote hit latencies (40, 60, 80, and 100 cycles). Other latencies are unchanged.

Thus far, all experiments assume a fixed remote cache hit latency of 40 cycles. As more cores can be integrated on a single CMP, the average remote cache hit latency can be expected to grow relative to the local cache hit latency. For example, consider a 64-core system connected by an 8×8 mesh. Round-trip communication between two diagonal cores requires 28 hops, which translates to a total remote hit latency of 140 cycles, if we assume a 5-cycle hop latency.

Figure 2.13 shows the average weighted speedup for APP and DSR, with various remote cache hit latencies: 40 (default), 60, 80, and 100 cycles. The figure shows that while the absolute spread between APP and DSR remains constant, the relative improvement of APP over DSR grows as the remote hit latency increases. The reason for this is that

with higher remote hit latencies, converting remote hits into local hits becomes a relatively more important factor for performance improvement, whereas converting global misses to remote cache hits becomes a relatively less important factor for performance improvement.

2.4.6 Other Performance Metrics - Throughput and Fairness

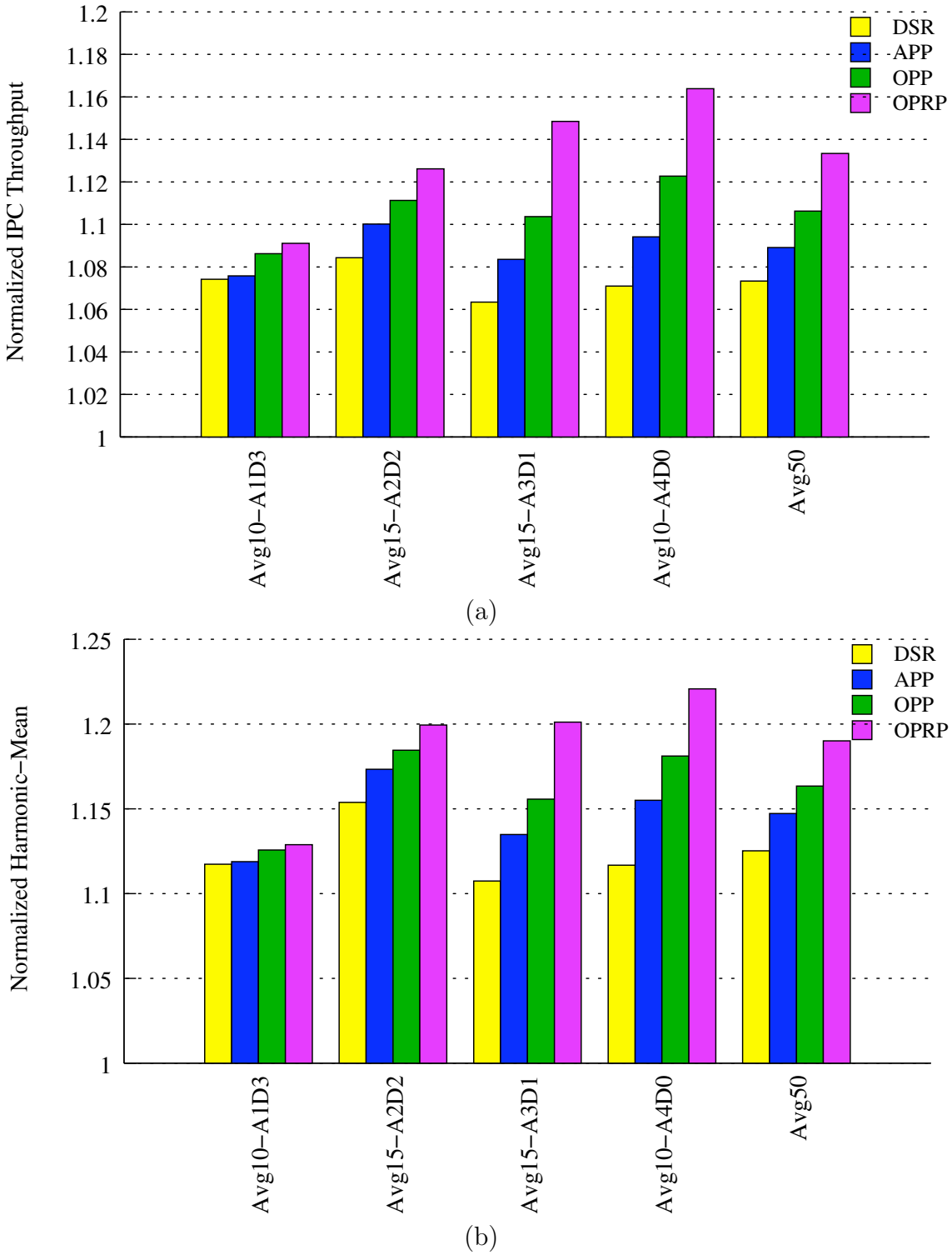


Figure 2.14: IPC throughput (a) and harmonic mean (b) for various schemes, normalized to private caches with no capacity sharing.

Figure 2.14 shows IPC throughput (a) and harmonic mean of speedups (b) of DSR, APP, OPP and OPRP, normalized to no capacity sharing. The figures show results that are consistent with the earlier figure showing weighted speedups.

2.5 Related Work

Management of shared caches in CMPs. There is a rich body of work that has studied the problem of effectively managing shared on-chip cache resources in CMPs. Solutions include software-controlled schemes [56, 67, 3], and hardware-only techniques. Hardware techniques differ in their approach: partitioning cache ways [54, 66], partitioning cache sets [18, 64], partitioning groups of lines [40], thread-aware replacement policy [42, 14], thread-aware insertion policy [53, 35], and thread-aware insertion and promotion policies [69].

Dynamic Insertion Policy (DIP) [53] and Thread-Aware Dynamic Insertion Policy (TADIP) [35] motivates the need for careful insertion of a block into a cache set. Another recent proposal, Pseudo-LIFO [14] effectively impacts the insertion position of a block by utilizing both the recency stack and the fill stack, which ranks lines based on the order in which they were inserted into a cache set. This body of work explores the impact insertion (and replacement) decisions can make in *shared* caches.

Private caches present a new set of challenges and opportunities. Placement across caches, not insertion in the local cache at a particular stack position, should be considered. Placement decisions, even on a hit, can significantly impact the block's access latency. In addition, there is typically no global recency or fill stack information across corresponding sets in the private caches.

Management of private/distributed caches in CMPs. Victim Replication

(VR) [70] and Adaptive Selective Replication (ASR) [6] studied techniques to allow remotely-homed blocks to be replicated in the local cache. These schemes primarily apply to multi-threaded workloads, where blocks identified to be more important are replicated locally (or victimized remotely). With multi-programmed workloads, which is the focus of this study, there is no significant data sharing and therefore no reason to replicate data.

The work that comes closest to ours in terms of the scope of the problem is work that studies how capacity can be shared among distributed caches. We refer to such a technique as *capacity sharing* [57, 58]. Capacity sharing techniques in prior studies include Cooperative Caching (CC) [12] and Dynamic Spill Receive (DSR) [52]. CC was the first work to bring the notion of capacity sharing across private caches. Prior to this, CMP NU-RAPID [16] proposed Capacity Stealing (CS) in order to manage the capacity of a shared cache's banks.

CS and CC allow each core to spill an evicted block to a remote private cache. Such capacity sharing allows any core to spill into other caches regardless of the temporal locality of the application running on that core. DSR improves upon CC, and identifies the existence of Acceptors and Donors. Further, it presents a dynamic mechanism to classify which application can spill, and which application can receive at any given time. Cooperative Cache Partitioning (CCP) [13] also builds upon CC by prioritizing cooperative caching opportunities, across applications in a coschedule that are competing for cache space, in a time-sliced manner. CS, CC, CCP and DSR explore remote block placement on eviction of a local block, In contrast, in this work we study remote block placement when a block is first brought on chip.

2.6 Conclusions

Capacity sharing is a technique for reducing cache fragmentation in a CMP system with private last-level caches; it allows applications that need additional cache space to place their blocks in remote caches. Current capacity sharing mechanisms treat remote caches as the victim cache for the local cache. We have shown that such a strategy guarantees a high number of remote cache hits relative to local cache hits for applications that exhibit anti-LRU behavior, which are usually the same applications that benefit from additional cache capacity.

In this chapter, we have investigated strategies that consider placing not only locally-evicted blocks in remote caches, but also *newly-fetched blocks* in remote caches. We investigate the upperbound performance that can be gained from combined placement and replacement decisions in capacity sharing, by using future trace information to make the decisions. Based on our findings, we propose a scheme that is implementable in hardware with little hardware overhead. We show that this scheme, APP, improves performance by 29% on average compared to a baseline with no capacity sharing, across 50 multiprogrammed workloads consisting of four SPEC CPU2006 applications. APP outperforms DSR, the state-of-the-art capacity sharing mechanism that only places local victim blocks in remote caches, by up to 18.2%, with an average improvement of 3%. APP dynamically identifies which applications should be allowed to place their blocks in remote caches, and which applications should not. In addition to improving aggregate performance significantly, APP also has safeguards to ensure that applications whose caches are accepting blocks from other cores are not slowed down by much.

Chapter 3

Energy-Efficient Interconnect via Router Parking

This chapter is organized as follows: Section 3.1 describes the design and algorithms we evaluate for our Router-Parking approach in details. Section 3.4 describes our evaluation methodology. Section 3.5 provides the results from our evaluations and analyzes the findings. Section 3.7 describes related work in reducing interconnect traffic. Section 3.8 summerizes our findings and concludes this chapter.

3.1 Router Parking Design

Figure 3.1 shows a high-level architecture of the Router Parking (RP) framework. The system consists of n tiles interconnected together via an interconnect fabric with a topology such as a Mesh [1] or a Torus. The figure shows three types of tiles - some where the core and the router on the tile are both active (grey), some where the cores are

[1] In this work we implement and evaluate our algorithms on a 2D-Mesh topology; the RP approach can, however, be applied to other interconnects with minimal modifications.

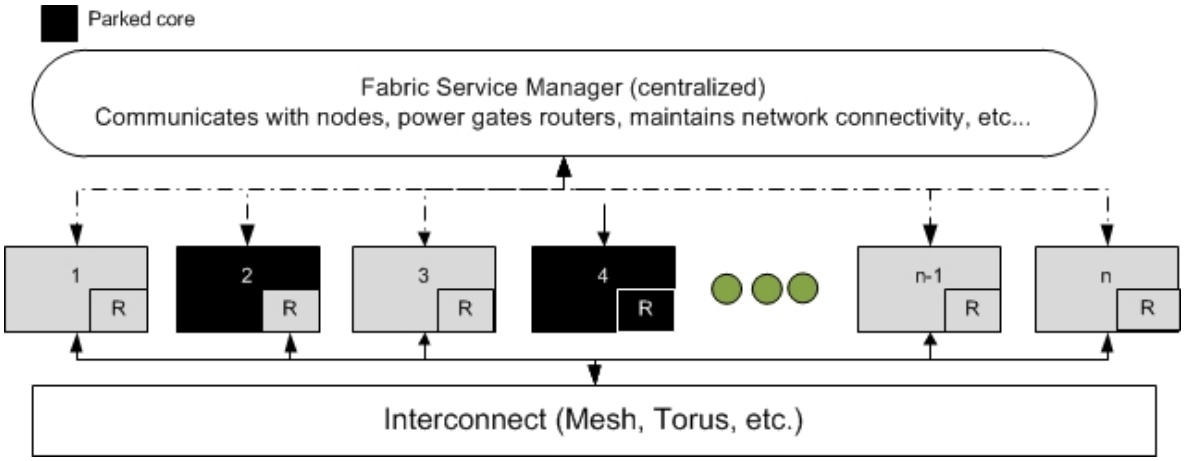


Figure 3.1: High-level view of the Router Parking architecture - nodes, routers, interconnect fabric and a centralized Fabric Manager.

parked (black) and the routers are active (grey), and some where the cores and routers are both parked (black). The figure also shows a component called the *Fabric Manager (FM) - centralized* logic responsible for configuring, monitoring and re-configuring the interconnect in order to implement RP.

A centralized FM is essential for efficient and effective interconnect power management as seen in many recent products, e.g., Intel Sandy Bridge architecture employs a centralized System Agent to handle power management [28]. Further, changes in the interconnect configuration warrant changes in the routing tables of individual routers. This requires globally consistent view of the active network to be visible to all nodes. While this could be achieved via a distributed approach, it increases the operational complexity and reconfiguration latency as pointed out by Aisopos et al. [1].

The FM could be either added as a functionality in firmware to one of the tile's LLC cache controllers, or to one of the on-chip memory controllers. It could also be implemented as a dedicated engine located in the fabric, similar to Intel's optical fabric (LightPeak) FM [30] which holds several fabric management responsibilities, such as,

setting up source-to-destination channels and handling topology changes. Further, the code of the FM could be executed on one of the cores in the system. In this work, we employ and evaluate an on-chip dedicated engine for the FM and install it in a tile positioned in the middle of the Mesh network. This ensures that the average distance to other nodes in the network is minimized.

Epochs Reconfiguration is attempted after periodic intervals called epochs. An important parameter is the length of an epoch, which must take into account the frequency of node configuration changes and the FM operation overhead. We empirically determine the epoch length based on the reconfiguration overhead (Section 3.5.2). However, if a dynamic epoch determination is desired, one may detect the stability of configurations and change the epoch length dynamically. For example, suppose that S_i measures the fraction of nodes that change their state (i.e., sleep to active or active to sleep) in the i^{th} epoch. To detect whether the rate of change is stable or not, we may use a moving average weighted more heavily towards recent data [39]; $S_l = \alpha S_i + (1 - \alpha)S_l$; where S_l is the long term stability of the configuration; α is a parameter that dictates how much recent data is weighted compared to older data. When S_l is high (close to 1), indicating a relatively stable configuration, FM gradually increases the length of the epoch.

At the end of each epoch, the FM follows a sequence of actions to prepare for a potential network reconfiguration.

Gathering node information. At the end of each epoch, the FM performs a multicast to all *active* routers asking for the status of their associated nodes. Parked routers are guaranteed to have their associated nodes parked as well, and need not be involved in this process. Nodes with active routers reply to the FM's broadcast with information about their status (parked or active). There are three ways for parked nodes to supply their status to the FM without themselves waking up. First, a small logic in the node

is responsible for communicating with the FM using the available communication links; therefore, there is no need to wake up the whole computing node to reply to the FM's query. Second, the router detects node inactivity by a timeout method and replies on behalf of the node. Third, the node notifies the FM before it goes to sleep; the FM, in turn, saves this information for future use.

Processing gathered information. Once the FM receives the node status information, it processes the information to decide which routers should be parked during the next epoch. The FM uses the following objectives while generating a new network configuration. First, the new configuration should not partition the network. This requirement might be relaxed for special situations – for example, several functional partitions may not need to communicate with each other due to job independence. Second, the algorithm should be able to park as many routers as possible in order to achieve the highest static energy savings. Third, the algorithm should minimize the latency impact due to the necessary detours around parked routers introduced by RP. Finally, the algorithm should be fast. Section 3.2 describes the algorithms we evaluate in detail.

Populating new configuration. All nodes continue to follow the old network configuration and routing decisions until they receive a new configuration from the FM. Once the FM creates the bitmap corresponding to the new configuration, it sends it to all involved nodes in the system. If a given router finds that it is permitted to park, it will power gate its ports, crossbar, and arbiters. Note to ensure packet deliverability, the router may need to wait until it flushes out all packets first. If on the other hand, a given router finds that it remains active, it updates its routing table to reflect the new network configuration. Recall that the interconnect does not freeze when a transition occurs from one epoch to the next. Some routers may be operating under the old routing configuration, while others may be operating under the new routing configuration. The

combination may cause undeliverable packets and deadlocks. We discuss how to handle these corner cases in Section 3.3.2.

If one computing node or core (along with its router) goes alive within an epoch, then the FM could either delay a network reconfiguration until the end of the current epoch or perform a network reconfiguration immediately. Since the epoch length is much shorter than the core residency in a deep sleep state (or the wake-up latency from such a deep sleep state, Section 3.5.2), postponing a network reconfiguration until the end of the epoch is a reasonable assumption. Further, note that a router's resume latency is hidden by the latency of a core being resumed, and a network configuration could start as soon as involved routers are alive regardless of whether their associated cores have come back completely or not.

3.2 Router Parking Algorithms

As mentioned in the previous section, the FM has a solution space that consists of mainly three parameters – the number of routers parked, the additional latency introduced, and maintaining the network connectivity. There is a wide range of algorithms that could be employed at the FM that differ in their aggressiveness regarding the number of parked routers and what trade off to enforce between the number of routers parked and introduced latency. In this section, we describe two basic RP algorithms - Aggressive Router Parking and Conservative Router Parking. We then describe an Adaptive Router Parking algorithm that chooses between the two basic algorithms according to run time interconnect condition.

3.2.1 Aggressive Router Parking Algorithm

The goal of an Aggressive Router Parking (RP-A) algorithm is to park as many routers as possible while maintaining the network connectivity. However, it may lead to an increased packet delivery latency when packets have to travel more hops to avoid parked routers. The delayed packet delivery may increase dynamic router energy which may offset the static energy saving from router parking. Thus, this approach is likely attractive when the interconnect packet injection rate is low because only few packets are affected and it is unlikely for them to impact performance. With a low injection rate, static energy saving may significantly exceed the additional dynamic energy consumption, making it worthwhile. Fortunately, many applications incur a low packet injection rate. For example, our analysis shows that for SPEC CPU2006 applications running on a tiled-CMP with reasonably-sized private L1 and L2 caches, the average packet injection rate of L2 misses onto the interconnect fabric is typically low: 9 misses per 1000 instructions on average, with a maximum of 45 misses per 1000 instructions.

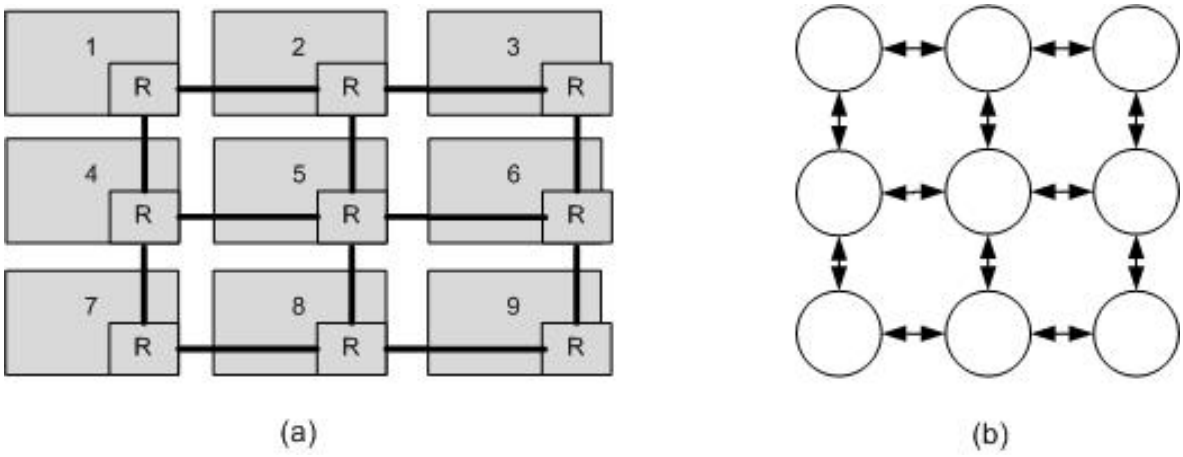


Figure 3.2: A 9-node 2D-Mesh interconnect (left) and a graph G(V,E) representation of the interconnected routers.

Under this approach, the FM views the interconnected routers as a graph; G(V,E), where routers are depicted as vertices V, connected to each other via bidirectional edges E. Figure 3.2 shows an example of 9-node 2D-Mesh interconnect. The FM processes the graph and evaluates its connectivity. We borrow Tarjan's Strongly Connected Components [55] algorithm from graph theory for this step. Tarjan's algorithm calculates the number of strongly connected components in a Graph(V,E) with a linear complexity of O(n); where n is the number of routers. Note that in a Mesh interconnect, *any* connected component is also strongly connected since all links are bidirectional. Therefore, we optimize Tarjan's algorithm for a Mesh topology such that we only identify the number of connected components instead of the original, heavier algorithm, that locates strongly connected components.

The pseudo-code of our optimized Tarjan algorithm is shown in Algorithm 1. The algorithm initially marks all routers associated with parked nodes as potential candidates for parking, and assumes they are parked. The algorithm then identifies the number of connected components. A connected component count of 1 means that parking all potentially marked routers does not partition the network. A connected component count greater than 1 means that parking all marked routers will create disjoint partitions in the network. In the latter case, the FM needs to decide how to connect the partitions. Finding the optimal path (i.e., the one that involves waking up the least number of routers) to connect the two partitions could increase the complexity of this solution to O(n^2).

In order to avoid the O(n^2) complexity, we avoid testing all potential combinations that connect partitions to find the optimal configuration. Instead, we let the FM to randomly pick an edge router (i.e. router that directly connects to a parked router) from every partition and create a path between that router and the FM itself by turning

Algorithm 1 A pseudo-code for the RP-A algorithm

```
measure_network_connectivity()
for all routers {v}; that belong to active nodes{n} do
  if v.visited then
    continue
  end if
  strong_cnctd_comps(v)
  num_strong_cnctd_comps ← num_strong_cnctd_comps + 1
  create a new strong cnctd comps list
  while stack_is_not_empty do
    w ← pop()
    add w to new strong cnctd comps list
  end while
end for
if num_of_cnctd_comps = 1 then
  return TRUE        //network connectivity is maintained
end if
return False

strong_cnctd_comps(v)
push(v)
v.visited ← 1
//Do a depth − first search
for all v's successors; {w} do
  strong_cnctd_comps(w)
end for
```

on routers along the path. Since all partitions can reach the FM, the network becomes strongly connected. Note that the chosen routers may not lead to the least amount of routers turned on. Therefore, we let the FM repeat the process k times, and then select the paths that turn on the least number of routers. How good the final solution depends on the value of k, the number of partitions, and the size of the network. In the network used in our study, we found that $k = 8$ leads to a very good configuration.

The example shown in Figure 3.3 helps illustrate this process. Part (a) of the figure shows a 16-tiles connected via a 2D Mesh topology. Tiles in black are parked (core is in deep sleep state). Once the FM receives the status of these cores, it marks routers associated with parked tiles as potential candidates for router parking. To apply our Tarjan-like algorithm, the FM views connected routers as a connected graph, as shown in part (b). The figure also shows that no edges connect to parked routers (black circles). After applying the algorithm, it will result in two network partitions – partition 1, 2, 3, 5, 7, 9, 13 and partition 12, 15, 16. In this case, unparking any of the routers 8, 11, and 14 can connect the partitions. Part (c) of the figure shows the two partitions connected by excluding router 11 from parking. Part (d) of the figure shows the final state of the interconnect once the new configuration is populated to all routers.

3.2.2 Conservative Router-Parking Algorithm

Under moderate to high packet injection rates, more packet rerouting due to parked routers could lead to high dynamic energy consumption which would offset the savings in the static energy achieved by RP. Conservative Router Parking (RP-C) addresses this trade-off.

RP-C relies on a subtle observation - network partitions and long detouring occur

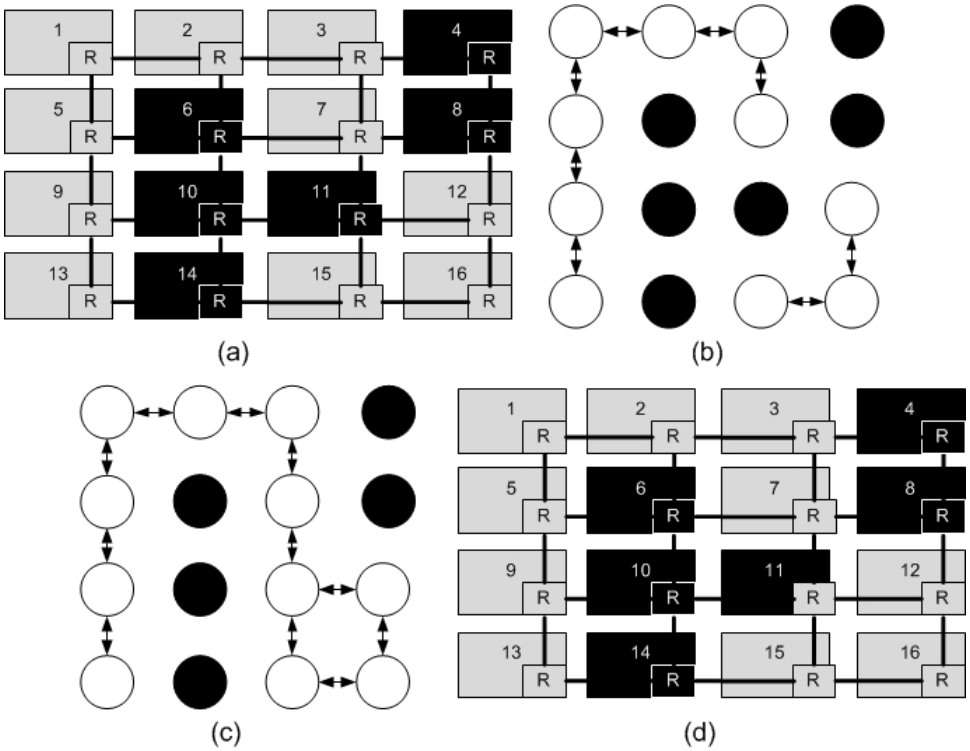


Figure 3.3: An example 16-node Mesh network

when a series of direct[2] and/or indirect[3] neighbors are parked forming a contiguous disconnect (e.g., diagonal, vertical, horizontal, staircase, loops, etc.). Such series could partition the network and introduce longer detouring latency. RP-C disallows any direct or indirect neighbor routers to be parked at the same time, thereby avoiding both partitions and long detours. The pseudo-code for the conservative algorithm is shown in 2.

Algorithm 2 A pseudo-code for the RP-C algorithm

```
park_routers_conservatively()
for all routers {v}; that belong to parked nodes{p} do
  park_this_router ← (any_neighbor_routers_parked() OR
  any_immediate_diagonal_routers_parked())
  if park_this_router then
    bit_map[router_number] ← 1
  end if
end for
```

To understand this, let's apply RP-C to the network shown in Figure 3.4(a). The figure shows 16-tiles connected via a 2D-Mesh topology (nodes in black are parked), and routers in black are candidates for parking. RP-C will park a router if none of its direct or indirect neighbors is parked. If the sweep starts at tile 1, the resulting network configuration is shown in Figure 3.4(b). The network ended up with only 3 routers parked (4, 6, and 14); the remaining three are left on. The benefit it that the average latency between any two nodes under RP-C is much less than in RP-A. In addition to better handling high packet injection rates, RP-C is more suited to systems undergoing frequent reconfiguration. As an optimization, outer edge routers are allowed to park on

[2]East, West, North, and South neighboring routers.
[3]Northeast, Northwest, Southeast, and Southwest.

series since they do not cause interconnect partitions.

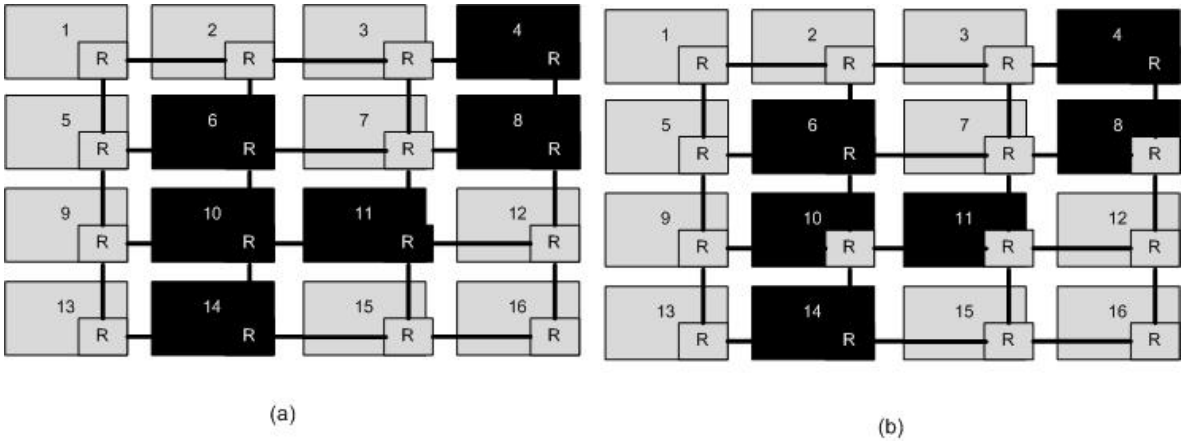


Figure 3.4: An example of a 16-node Mesh network, with 6 nodes parked, and their routers being marked as potential candidates for Router Parking (a), the final configuration of the network showing that no series of routers are allowed to be parked (b).

3.2.3 Adaptive Router-Parking Algorithm

Interconnect traffic is crucial parameter in dictating whether RP-A, RP-C or no Router Parking (No-RP) is the better approach. We propose and evaluate an adaptive Router-Parking (RP-Adp) algorithm that monitors the run time interconnect utilization and chooses between these options. The algorithm is illustrated in 3.5.

The average router dynamic power P_d is proportional to the interconnect utilization (number of packets switched per time unit). When P_d is smaller than a certain threshold, MIN, indicating very light traffic, RP-A is the most desirable algorithm. On the other hand, if P_d is larger than another threshold, MAX, indicating very heavy traffic (approaching the interconnect saturation point), router parking should be turned off. When P_d is between MIN and MAX thresholds, where the interconnect traffic is moderate,

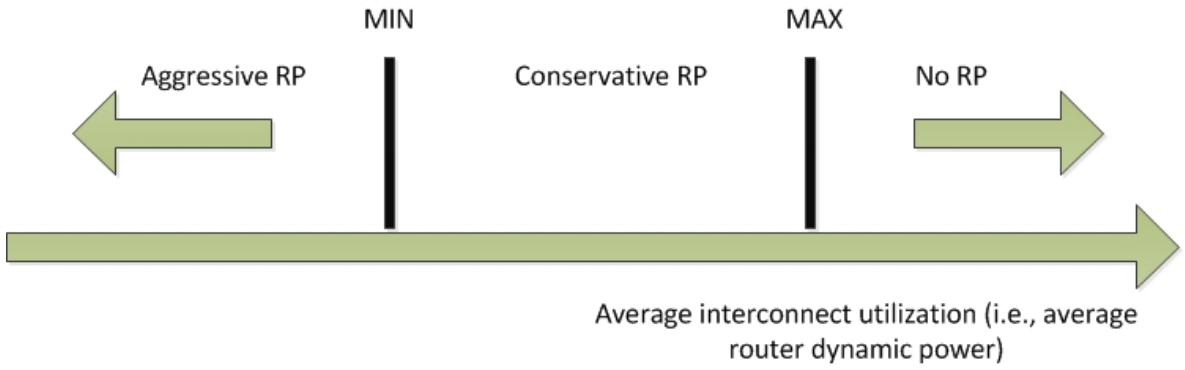


Figure 3.5: An illustration of the Adaptive Router Parking algorithm.

RP-A may not be the right choice since the reduction in static power is offset by increase in dynamic power due to heavy rerouting. Hence, the adaptive algorithm opts for the RP-C algorithm in this region.

The MAX and MIN thresholds are important parameters for the stability of RP-Adp. Here we describe how to choose the two thresholds, as well as the rationale for the choice.

Let P_s denote the router's static power, which is a design parameter for a specific interconnect. *RP-A must be selected when the dynamic power introduced due to the extra hops (H_e) is less than the static power savings due to parking routers.* This can be expressed as the following inequality;

$$H_e * P_d < R_p * P_s \tag{3.1}$$

where R_p is the number of additional routers parked in RP-A compared to RP-C; and H_e is the number of extra hops in RP-A compared to RP-C.

According to our design space experiments, under RP-A, H_e is usually smaller than R_p, (i.e., the number of average extra hops is usually smaller than the additional routers parked under RP-A compared to RP-C). This is because packets can usually find other

minimal or near-minimal paths. Hence, if $P_d < P_s$, the inequality will be met. This means that P_s is a reasonable value for the threshold MIN. In other words, when $P_d < P_s$, RP-Adp chooses RP-A, else it chooses no-RP or RP-C (contingent on comparing P_d to MAX threshold).

When choosing the MAX threshold to switch between No-RP and RP-C, the same rationale applies. That is, if $P_d > \frac{R_p}{H_e} P_s$, router parking should be turned off. H_e corresponds to the additional hops introduced due to parking routers under RP-C compared to No-RP. R_p corresponds to the routers parked by the conservative approach.

These MIN and MAX parameters could either be chosen empirically, or at run time. In the latter case the algorithms to be compared (say, RP-C vs. No-RP) are applied in successive epochs, and the necessary statistics (average interconnect latency, average dynamic and static power, and in case of RP-C, the number of routers parked) are collected in each epoch. Applying the above inequality identifies the better algorithm for RP-Adp to employ.

3.3 Steady/Transient State Issues

Having looked at the architecture and algorithms associated with Router Parking at a high level, we will now focus on implementation issues that must be considered.

3.3.1 Deadlock Avoidance and Recovery

We use minimum routing algorithm for normal packet forwarding so that performance is not impacted significantly compared to the baseline of no router parking. However, even with low traffic rate, there exists a probability, albeit very small, for deadlocks to happen. We opt for deadlock recovery approach using an escape channel, similar to

what proposed in [17]. Our approach requires minimal resources, i.e., an additional per-router flit buffer and a routing table used by the escape channel to route a deadlocked packet. Deadlock occurrence is detected by a timeout mechanism. Upon detection, a deadlocked packet is injected into a flit buffer and routed through the escape channel to its final destination. The escape channel employs up*/down* [59] deadlock-free routing algorithm. The routing tables are updated whenever an interconnect configuration takes places since the nodes comprising the escape channel may change. Details are discussed in next section.

3.3.2 Transient state behavior

While a reconfiguration is underway, some routers may be operating as per the old network topology, while others may be operating under the new topology. Further, if routers are allowed to park immediately once they receive a new configuration, a case may arise where some packets are rendered undeliverable. The packets cannot be routed to destination in such a case because the new topology and routing function cannot handle them. Moreover, the transition from the old to the new routing tables may create additional dependencies among network resources, causing what is referred as *reconfiguration-induced deadlock* or *ghost dependency* [43].

Many current techniques handle the ghost dependency problem using static configuration methods. Using static configurations, packet injection is prohibited and all in-flight packets have to be drained from the entire network before the new topology and configuration is deployed. During the configuration process, no new packet injections are allowed [1]. The network resumes to normal operation when the reconfiguration is completed. This approach applies to all network topologies and routing functions. It easily avoids any reconfiguration-induced deadlocks either during or after the reconfiguration

process, since there are no packets during the reconfiguration period, and no packets are subjected to more than one routing functions. However, this approach may reduce network availability if reconfiguration events occur frequently. Hence, it is may not be the preferred approach. We, on the other hand, employ the following 4-phase dynamic interconnect reconfiguration protocol that does not stall the network, avoids undeliverable packets and handles the reconfiguration induced deadlock problem.

Phase 1: Update normal operation routing tables, wake up to-be-turned-on routers. (Step 1) FM wakes up routers that will be online in the next epoch. (Step 2) FM sends control packets including the new routing table to all routers that are active during the next epoch. (Step 3) Routers will update their routing tables once they receive the new reconfiguration and start operating based on the new routing information. Meanwhile, if a deadlock occurs, nodes will continue to use the *old* escape path to recover. (Step 4) FM sends a propagation order spanning tree (POST) [17] control message to all routers and awaits responses regarding the completion of the routing tables update. Once response is received, phase 2 starts. Note that, to-be-parked routers can not park after Phase 1, since they may still provide an escape path for deadlocked packets.

Phase 2: Halt injecting user-level packets onto the escape network and let it drain. Since the escape path is the only way to resolve a deadlock, it is critical to update this path without introducing deadlocks. To handle this, once Phase 1 ends, we stop using the escape path even if a deadlock occurs (i.e., postpone deadlock recovery), and drain the escape path. We initiate the drainage by broadcasting a POST message rooted at the fabric manager and reaching the leaf nodes. The drainage takes place in two sub phases. (Step 1) All up* channels are emptied from leaf routers to root router (i.e. FM). (Step 2) all down* channels are emptied from root router to leaf routers. This ensures that the secondary path is drained of packets.

Phase 3: Globally activate the new routing function in the escape path. The escape path has one input channel, once its drained of old packets (Phase 2) and no user-packets are being injected, the FM updates all routing tables. The update is performed at one level of the tree at a time (up*/down* tree), the FM awaits a response from the first level, then updates the second level and so on. This sequence is needed since there might be leaf nodes unreachable by their neighbors (used to be parked), so upper levels have to be updated first so these nodes are reachable. Once all levels are updated, all primary and secondary routers have been updated. Note that, phase 3 does not necessarily need to wait until Phase 2 is finished. Instead, Phase 3 can overlap with down* channels drainage (Step 2 in Phase 2).

Phase 4: Packet injection onto the escape path is allowed to resume and to-be-parked routers can park. Once all routing tables are updated, the FM signals all nodes to inform them that reconfiguration is over. The to-be-parked routers can safely park at that point, and other routers can use the escape path if they encounter a deadlock.

3.4 Router Parking Evaluation Methodology

3.4.1 System Configuration

We use a cycle-accurate multicore simulation infrastructure based on Simics [44], a full system simulation platform with an interconnect model based on Garnet [19], and the Orion 2.0 power model [36]. Table 3.1 lists the relevant configuration parameters of the system. Throughout the evaluations, the default configuration is a 64-core (8x8) CMP connected via a 2D-Mesh.

Table 3.1: System and Interconnect Configuration

Cores	64 in-order cores, 2GHz
Private L1 I/D, L2 caches	(L1) 32KB, 2-way, 2-cycle latency, 64B/block Private L2 caches with Capacity Sharing [12] Each private L2 cache: 1MB, 8-way, 10-cycle latency, 64B block, LRU policy
Memory	300-cycle access latency
Fabric Manager (FM)	40mW Installed in middle tile
Routers	32nm, 2GHz @ 1.0 V_{dd}, 4-stage (4-cycle) pipeline, 5 I/P ports, 5 O/P ports, 4 Virtual Channels/port, 8 flits/VC, 1 flit buffer for escape channel, 2 routing tables, 64 entries, 3-bit/entry 10-cycle wake-up latency, overlapped with core wake-up
Links	16B, 1-cycle latency,
Energy	Router dynamic energy/access = 2.38e-10J Router static energy/cycle = 1.32e-10J static energy breakdown: Buffer = 0.9187e-10J other components (crossbar, switch arbiter, vc arbiter, and clk) = 0.402e-10J Link dynamic energy/access = 7.89103e-13J Link static energy = 0.0 Power-gating overhead = 2.3pJ
Routing algorithm	baseline: xy w/o routing tables router parking: minimum routing for normal forwarding and up*/down* for escape channel

3.4.2 Traffic Generation

We use a combination of synthetic traffic, as well as real workloads' traffic taken from PARSEC 2.1 and SPEC CPU2006 benchmark suits. Real applications tend to have low packet injection rates; synthetic traffic allows us to parameterize the injection rate and stress test RP algorithms.

Synthetic Traffic

For the synthetic traffic, we augment each one of the 64 nodes with a traffic generator module that injects traffic based on a statistical distribution. The destination node is selected based on the *traffic patterns* [61]. We provide a brief description of the traffic patterns in Table 3.2. We experiment with various fractions of parked cores (10-80%). For each run, we keep the *fraction* of parked cores constant; however, we change the *set* of parked cores every sampling period. *This allows us to introduce the reconfiguration overhead.* We run each node for 100,000 cycles. Our synthetic traffic methodology is in line with prior work [49, 48].

Table 3.2: Traffic Distribution Pattern

Traffic Distribution	Uniform (λ); λ: packet injection rate/node/cycle
Traffic Patterns	Uniform Random (UR); destination is randomly selected, Transpose (TP); (x,y) sends to (y,x) Tornado (TOR); (x,y) sends to (x+k/2 -1, y); k is dimension
Packet size	Synthetic Traffic: 2 flits/packet (e.g., 0.06 pkt/node/cycle = 0.12 flit/node/cycle) Real Workloads: 5 flits/packet (1 control + 4 data)

Real Applications

Our baseline (as shown above in Table 3.1) assumes private L2 caches with *capacity sharing* [12]. In this architecture, L2 victims are evicted from the local cache to a randomly selected L2 cache belonging to an unparked core, rather than being dropped off-chip. Upon a miss in the local tile, the directory is consulted and the block is supplied from the remote cache to local cache via a cache-to-cache transfer.

We construct a multi-programmed workload, consisting of 64, randomly-selected benchmarks from the C/C++/Fortran SPEC CPU2006 benchmarks [65]. Similar to the synthetic traffic, we run the workload with various fractions of parked cores, and change the *set* of parked cores every sampling period. We fast forward all active applications for 10 billion instructions. We then attach the cache models and enable detailed timing simulation. We warm each cache for 1 billion cycles, before running each application in the workload for "at-least" 150M instructions. At that point, the required statistics are collected and the application continues to run and contend for cache space.

We also experiment with 12 of the 13 benchmarks form the PARSEC 2.1 multi-threaded benchmark suite (*raytrace* is excluded due to compilation problem). In each run, we vary the number of threads of each benchmark based on the fraction of parked cores (e.g., for 10% parked cores, we run 90% (out of 64) threads rounded to the nearest integer). We create checkpoints at the start of the main work loop (PARSEC source codes clearly identifies this region). We run each thread for 150M instructions or till the end of the region of interest, whichever happens sooner.

To introduce reconfiguration overhead, the epoch length was initialized to 10,000 cycles for the synthetic traffic, and to 50,000 cycles for SPEC and PARSEC experiments. Further, the overhead of the Fabric Manager and the communication overhead between

the Fabric Manager and the routers are included in the evaluation.

3.4.3 Evaluation Metrics

For both synthetic and real workloads, we compare the energy savings of the different Router Parking algorithms by measuring the interconnect static, dynamic and total energy. To measure the Router Parking performance impact on the real workloads, we measure the Weighted Speedup (WS). The significance of WS has been described in detail in prior work [20].

Finally, in order to measure the trade-off between energy savings of the interconnect (accounting for energy loss introduced by the Fabric manager and the associated communication overhead), and the performance impact on the workload, we measure the Energy-Delay Product (EDP). The delay corresponds to the total number of cycles the slowest application took to finish execution (equivalent to the number of cycles the interconnect was exercised until the slowest application finished execution). Table 3.3 defines these metrics.

Table 3.3: Performance and Energy metrics $IPC_{i,base}$ represents the IPC of an application running on core i without RP, while IPC_i represents the IPC of an application running on core i with RP.

Metric	**Formula**
EDP	Interconnect Energy * number of cycles
Weighted Speedup	$\sum_{i=1}^{n}(IPC_i/IPC_{i,base})$

3.5 Evaluation and Analysis

3.5.1 Synthetic Traffic Evaluation.

Significance of Static Energy. Figure 3.6 shows how the static energy, dynamic energy, and average network latency (y-axes) change as the packet injection rate increases (x-axis). As can be seen, the traffic injection range can be divided into four virtual regions. First, under relatively low packet injection rates (< 0.015 pkt/node/cycle, equivalent to 0.03 flit/node/cycle), the static energy consumed exceeds its dynamic energy counterpart. Second, under moderate injection rates ($> 0.015, < 0.035$), the dynamic energy starts taking over the static energy. Third, under relatively high packet injection rates ($> 0.035, < 0.060$), the dynamic energy is an order of magnitude more than static energy. Finally, under very heavy packet injection rates (> 0.060), the interconnect saturates and queuing delay starts showing up, making the average network latency exceed thousands of cycles. These observations give us insights regarding which range would be best for each RP algorithm.

For the first region, aggressive algorithm (RP-A) is more desirable since significant static energy savings can be achieved without impacting the latency and dynamic energy much; the amount of traffic being detoured is small. At the other extreme, when the interconnect is saturated, any RP algorithm would make the situation worse; hence, RP should not be applied and all routers should be active.

The regions in the middle requires trade-off analysis. For example, for relatively high packet injection rates – say 0.06 pkt/node/cycle (or alternatively 0.12 flit/node/cycle), RP-A may not be the best option since static energy savings by parked routers can be easily offset by an increase in dynamic energy due to heavy traffic rerouting. Hence, conservative algorithm (RP-C) could potentially be more appealing in this case. On the

other hand, if packet injection is, say, 0.02, then the increase in dynamic energy due to rerouted traffic could still be less than the static energy savings, giving the RP-A an appealing use case. RP-Adp can decide the winning algorithm. We evaluated a wide range of packet injection ratios in each of the four regions and found that the results in each region are consistent. Due to space limitations, throughout the upcoming sections, we restrict the results to three packet injection scenarios – 0.01 (static energy > dynamic energy), 0.04 (static energy < dynamic energy), and 0.06 (static energy << dynamic energy).

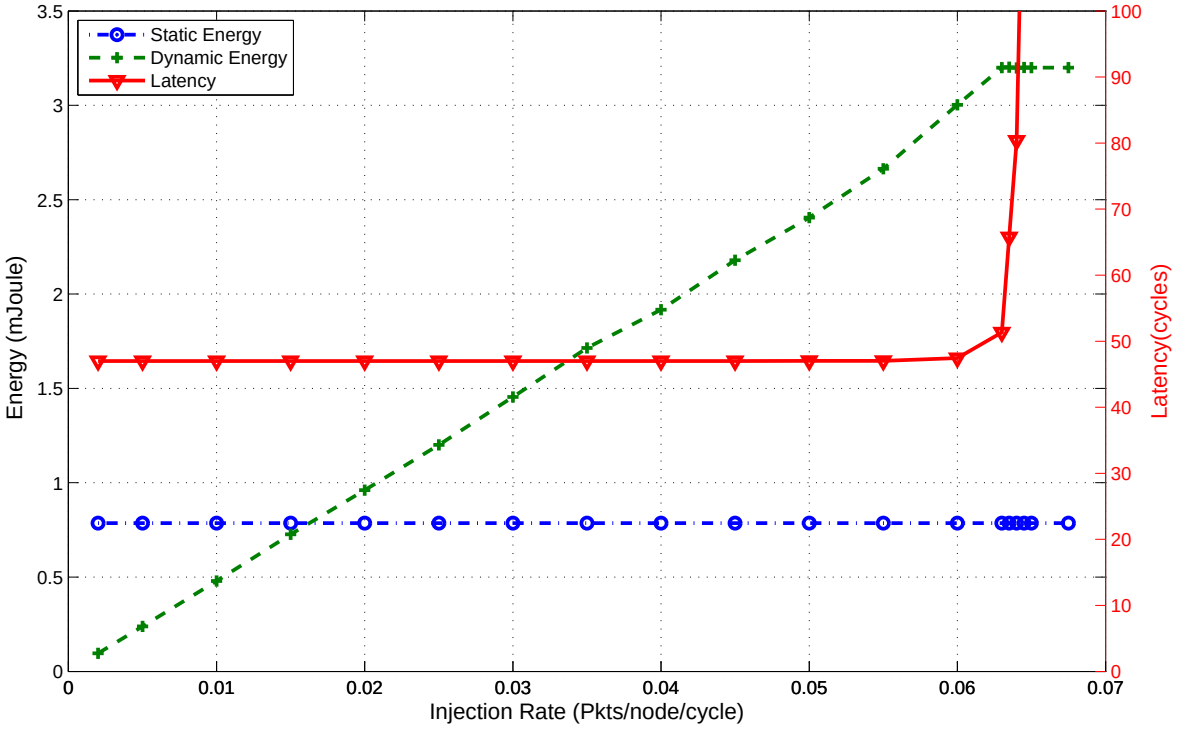
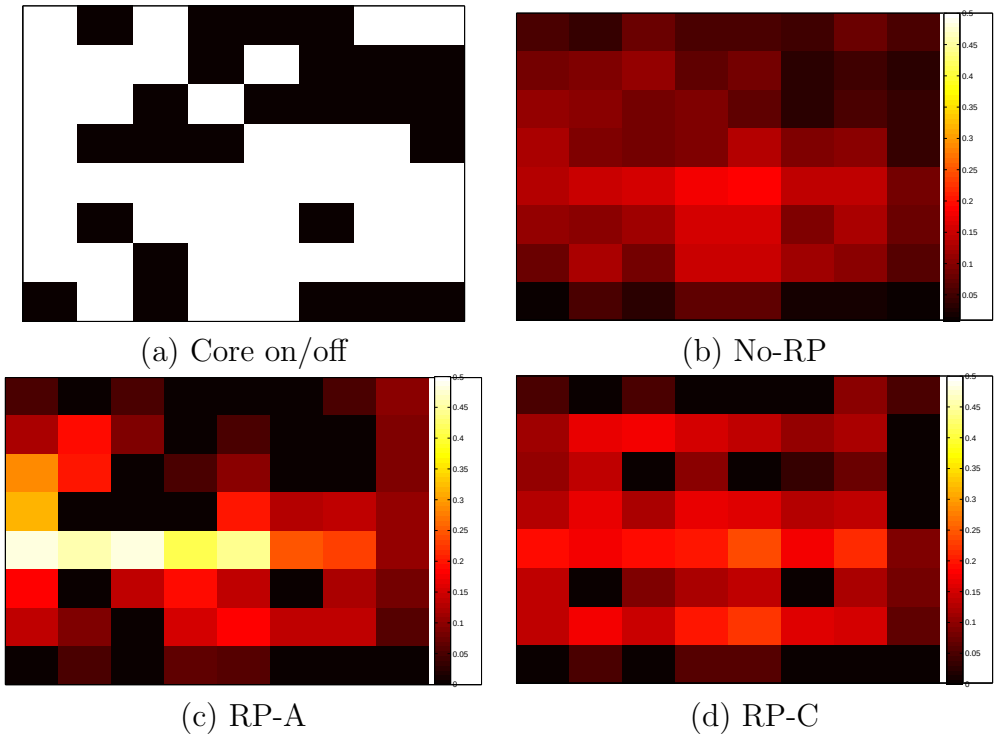


Figure 3.6: Interconnect static energy, dynamic energy, and average latency for uniform random traffic pattern, for various packet injection rates; each packet is 2 flits.

In-depth Analysis of Router Parking Algorithms. Figure 3.7(a) shows the

distribution of 40% randomly parked cores (in black) in the 8x8 2D Mesh network. As shown, letting all parked cores park their routers may cause network partitions, hence the need for intelligent Router-Parking algorithms. Parts (b,c,d) show the heat maps for the 8x8 router utilization under the No-RP, RP-A, and RP-C, for the same 40% parked cores ratio, and 0.04 packet injection rates. The color of a tile corresponds to the average router utilization, with bright colors indicating higher router utilization, dark colors indicating low router utilization, and black colors indicating no traffic going through. As shown in the No-RP, since all routers are active, the average utilization is low, as depicted by relatively dark-colored routers. Note that, even in the No-RP case, with the 40% fraction of parked cores, some middle routers are darker since they are not generating traffic but merely forwarding packets for other nodes. When RP-A is applied (part c), black routers indicate that they are parked. Further, since the number of active routers is less than No-RP, the average utilization is higher as shown by the bright colors. Moreover, since in RP-A, a cluster of routers can be parked as long as no network partitions are created, long detours may be introduced. This can be visualized around the cluster of routers in middle of network. Under the RP-C case (part d), the number of parked routers is not as many, hence, the average utilization is in between no-RP, and RP-A cases. Further, the RP-C figure shows that no clusters of parked routers are created. This leads to little latency impact in RP-C case as compared to RP-A.



(a) Core on/off (b) No-RP

(c) RP-A (d) RP-C

Figure 3.7: Core on/off bitmap (a), Heat Maps for the 2D Mesh, 64 routers (Brighter is higher temperature), under the three cases, No RP (b), RP-A (c), and RP-C (d). Heat in the figure corresponds to router utilization.

Figure 3.10 shows three rows for different packet injection rates (top row 0.01, middle row 0.04, and bottom row 0.06 pkt/node/cycle). Each row shows the interconnect static energy, dynamic energy (including the FM overhead), average latency, and the total (static + dynamic) interconnect energy normalized to the base case of no Router Parking. The x-axes correspond to the fraction of parked cores. The figures compare the three algorithm RP-A, RP-C, and RP-Adp to the base case of no Router Parking (no-RP).

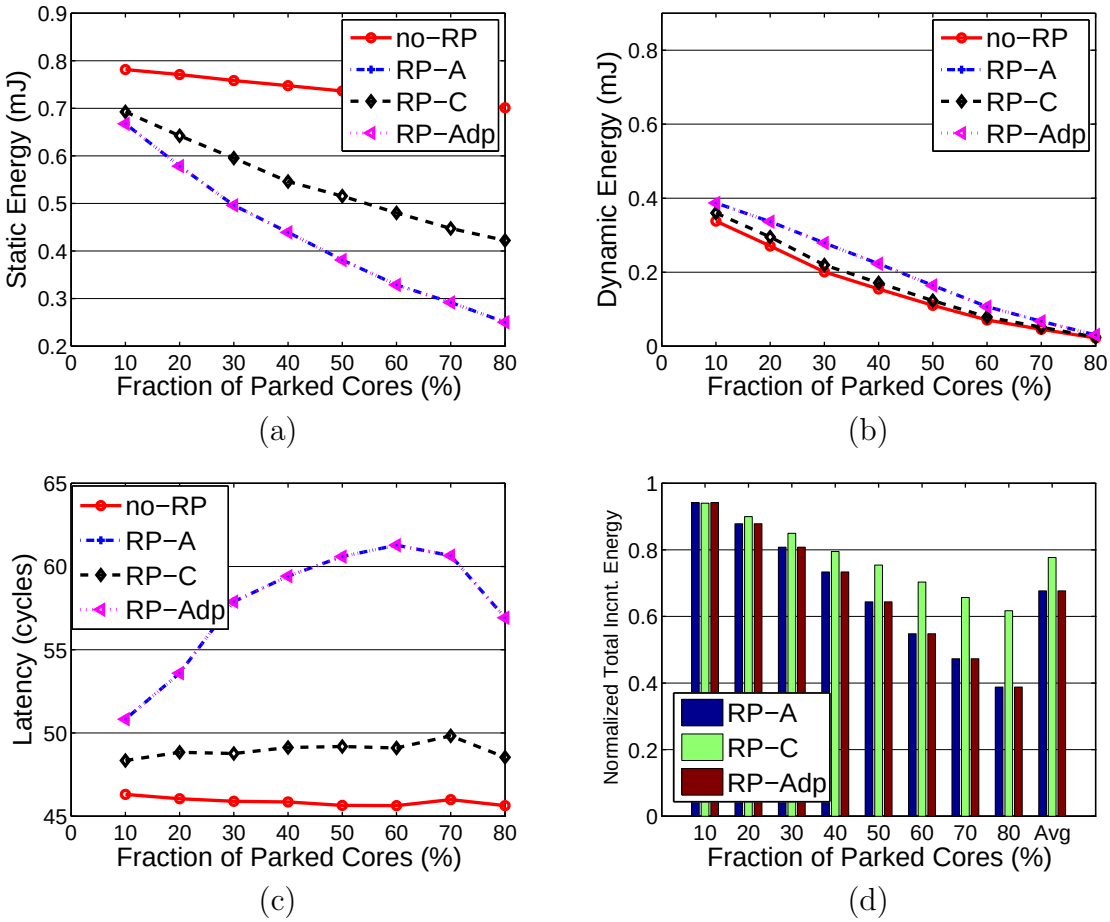


Figure 3.8: Interconnect static energy (a), interconnect dynamic energy (b), average interconnect latency (c), and total (static + dynamic) interconnect energy normalized to the base case of no Router Parking (d) for packet injection rates of 0.01 pkt/node/cycle (or alternatively 0.02 flit/node/cycle)

.

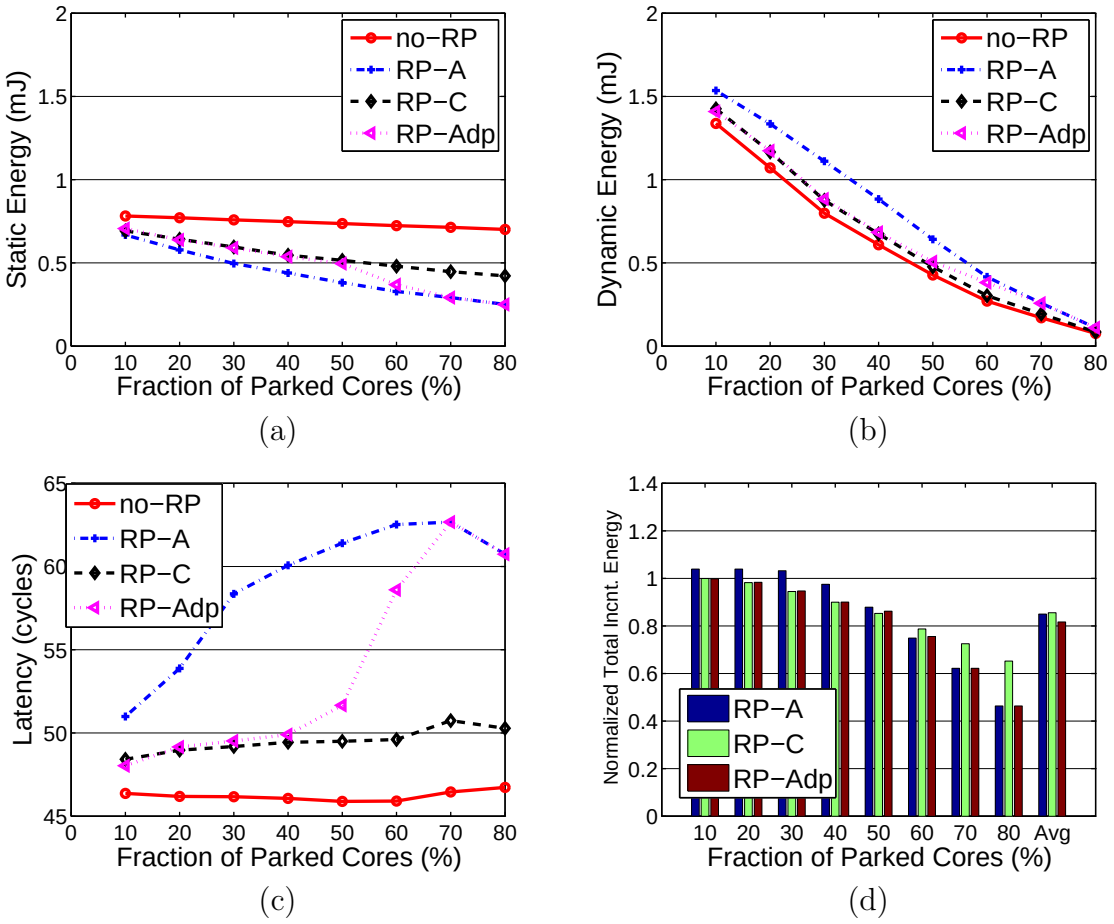


Figure 3.9: Interconnect static energy (a), interconnect dynamic energy (b), average interconnect latency (c), and total (static + dynamic) interconnect energy normalized to the base case of no Router Parking (d) for packet injection rates of 0.04 pkt/node/cycle (or alternatively 0.08 flit/node/cycle)

.

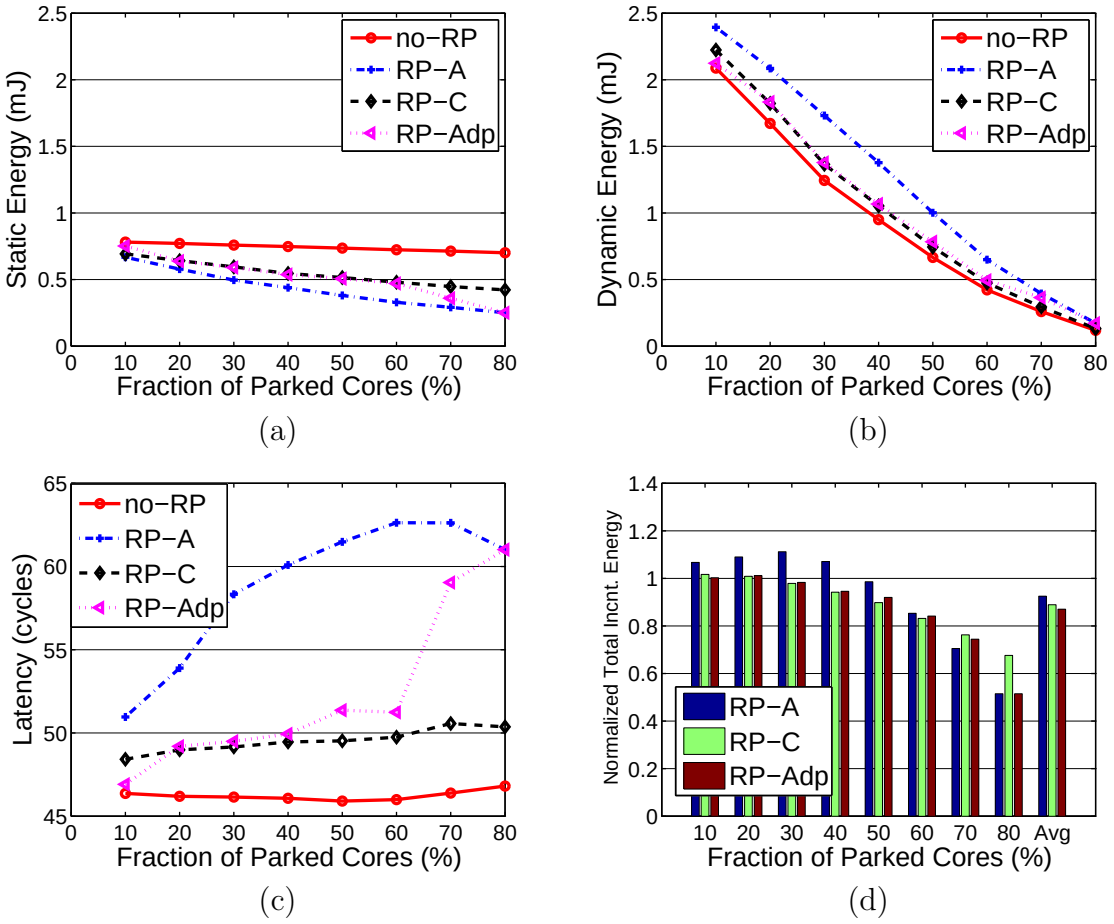


Figure 3.10: Interconnect static energy (a), interconnect dynamic energy (b), average interconnect latency (c), and total (static + dynamic) interconnect energy normalized to the base case of no Router Parking (d) for packet injection rates of 0.06 pkt/node/cycle (or alternatively 0.12 flit/node/cycle)

.

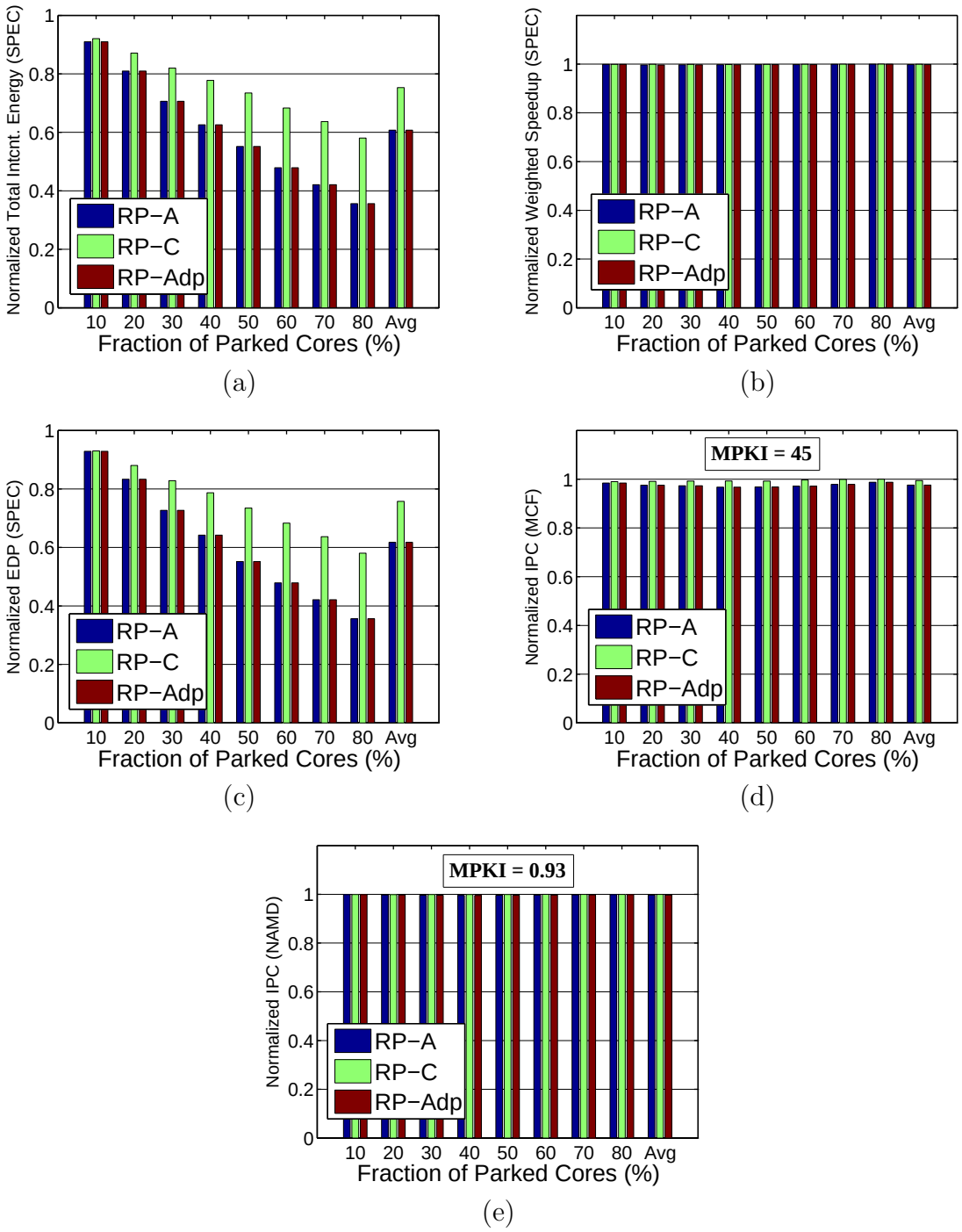


Figure 3.11: Total interconnect energy (a), Weighted Speedup (b) and EDP (c) for the SPEC CPU2006 workload. IPC Speedup for one memory intensive application *mcf* (d), and IPC Speedup for a non-memory intensive application *namd* (e). Each figure shows RP-A, RP-C and RP-Adp with varying fraction of parked cores, with the average being the last set of bars in each figure. All are normalized to the base case of no Router Parking.

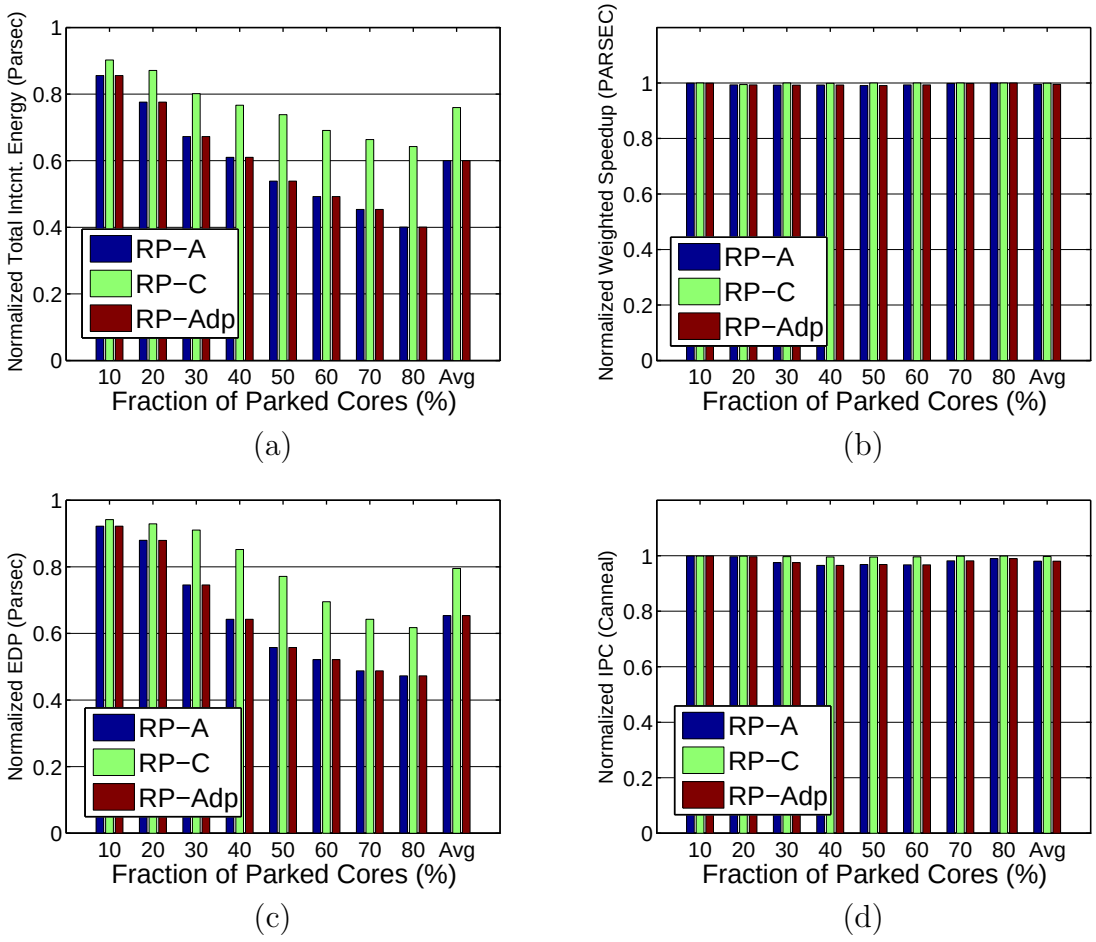


Figure 3.12: Total interconnect energy (a), Weighted Speedup (b) and EDP (c) for the PARSEC2.1 workload. IPC Speedup for a memory intensive application *canneal* (d). All are normalized to the base case of no Router Parking.

Several interesting observations can be drawn from these figures. First, RP-A - due to its ability to park many routers, achieves significant static energy reduction for the three packet injection scenarios. Further, this reduction is accompanied by modest dynamic energy increase for the low packet injection rate case (top row), hence, resulting in significant *total* energy savings (RP-A reduces total interconnect energy by an average of 32% and up to 61% compared to no-RP case). However, as the packet injection

rate increases (from middle to bottom row), *and* the fraction of parked cores is below 30% for 0.04 pkt/node/cycle and below 40% for 0.06 pkt/node/cycle, the increase in the dynamic energy partially offsets the reduction in the static energy, hence, increasing the total interconnect power slightly (as shown in part d). As the fraction of parked cores grows, the static energy reduction becomes more visible and leads to total energy savings. For example, even for high packet injection rates (bottom row), RP-A still reduces *total* interconnect energy by an average of 8%.

Second, as mentioned in Section 3.1, RP-A parks more routers than RP-C, hence, it has a bigger impact on interconnect latency (part c) due to longer detours. This produces heavier traffic rerouting, which may lead to an increase in the dynamic energy offsetting any savings in the static energy. This indicates that RP-A is less attractive when the traffic injection rate is high and the fraction of parked cores is small. In such cases, RP-C is more favorable.

Therefore, neither RP-A nor RP-C can fit the wide spectrum of run-time dynamics. This is why we need an adaptive design that can choose either one dynamically, based on the run-time network conditions. For example, with low injection rate (top row), and since the overall network traffic is light and dynamic energy is small, RP-Adp selects RP-A in order to park as many routers as possible for power saving (RP-Adp is overlaid on top of RP-A in the top row of the figure). On the other hand, with packet injection rate of 0.06 (bottom row), when the parked nodes are few (ratio below 10%), with more active nodes injecting traffic, average traffic in the network is relatively heavy. In this case, RP-Adp closely follows no-RP behavior. As the ratio of parked nodes gradually increases, indicating average traffic in the network becomes progressively lighter, we can see that RP-Adp first diverts (around 10% to 20% ratio) from no-RP to RP-C. Then as the ratio increases further(from 60% to 80%), indicating even fewer injecting nodes and lighter

traffic in the network, RP-Adp gradually converges to RP-A. These set of results show that our RP-Adp can capture the network dynamics and reacts accordingly, achieving the design objectives with significant gains (an average of 32%, 19%, and 11%, for the three packet injection rates, respectively) and almost no reduction in total interconnect energy in the case of high number of active nodes accompanied by high traffic loads, as shown in Figure 3.10(d).

Finally, although the average interconnect latency impact may seem high with large fraction of parked cores, (32% increase in latency for RP-A under 60% fraction of parked cores as shown in part c), it does not necessarily transform in performance slowdown. The reason is that it highly depends on the amount of traffic being rerouted. For example, if a given workload posts very few interconnect requests over its entire run time, even if these requests take a long detour, this will not impact the application's performance by much. Hence, the lower the packet injection rate, the lower the performance impact. We will show in the next section, that with typical workloads (SPEC CPU2006 and PARSEC2.1), the observed impact on performance is limited due to the fact that these workloads have relatively low injection rates and the interconnect is not the bottleneck.

3.5.2 Real Workload Evaluation

Figure 3.11 shows the various metrics for the SPEC CPU2006 workload mix, running RP-A, RP-C, and RP-Adp, normalized to the base case of no Router Parking. The x-axes in the figures show the varying fraction of parked cores. As shown in part (a) of the figure, RP-A and RP-C achieve significant *total* interconnect energy reductions reaching up to 61% and 42% for 80% fraction of parked cores, respectively. On average, RP-A and RP-C can achieve 40%, and 25% reduction in total interconnect energy. The reason why RP-A consistently outperforms RP-C in reducing total interconnect energy is due

to the marginal increase in interconnect dynamic energy RP-A introduces while parking more routers. Part (b) of the figure confirms this observation by showing the Weighted Speedup normalized to the base case of no-RP. As shown, the performance is impacted by less than 0.2%. Hence, the associated dynamic energy increase is expected to be low. The normalized EDP (part c) follows a similar trend to the normalized energy reduction since the delay is marginally impacted showing significant reduction in EDP. RP-Adp was able, at run time, to detect this and pick RP-A.

Recall that weighted speedup is calculated across all 64 applications in the workload. To gain better understanding into how RP impacts individual applications, the figure part (d) shows a memory intensive applications out of the 64 applications comprising the workload mix (i.e, *mcf*). *mcf* is impacted the most across all applications. This is expected since *mcf* posts many interconnect misses (MPKI=45) and therefore the interconnect latency might become a bottleneck. *mcf* notices slowdown of up to 3%. This is compared to another non-memory intensive application; i.e., part (e), *namd* (MPKI=0.93), which shows almost no noticeable impact since it does not post as many interconnect misses. Many SPEC CPU2006 applications follow *namd* behavior. *To summarize, interconnect latency is not the bottleneck for many SPEC CPU2006 workloads. Hence, although RP increases average interconnect latency, it does not impact the applications' performance significantly.*

Similar trends are shown for the PARSEC 2.1 workload as depicted in Figure 3.12. On average, RP-Adp achieves 41% reduction in total interconnect energy with almost less than 0.4% slowdown in Weighted Speedup. Hence achieving an average reduction of 38% in EDP. *canneal* (part d) was the application impacted the most by Router Parking. It is slowed down by up to 4%. Similar to above, RP-Adp picked the RP-A as a run-time choice since the savings in static energy offset the dynamic energy increase.

Impact of prolonged execution on energy savings. It is critically important that the slowdown in application's performance (and the corresponding increase in processor run time) does not have an offsetting impact on energy. In order to illustrate this, Figure 3.13 (a,b) shows the total interconnect energy for RP-Adp with and without considering the extra energy consumed by active cores, normalized to the base case of no Router Parking, for SPEC CPU2006 and PARSEC2.1 workloads, respectively. The cores considered in this experiment are similar to those existing in Intel's SCCC with 2W TDP/core. As shown in the figure, the energy impact due to cores running longer time had negligible impact on the overall interconnect energy savings. This shows the potential of RP to reduce *total chip* energy.

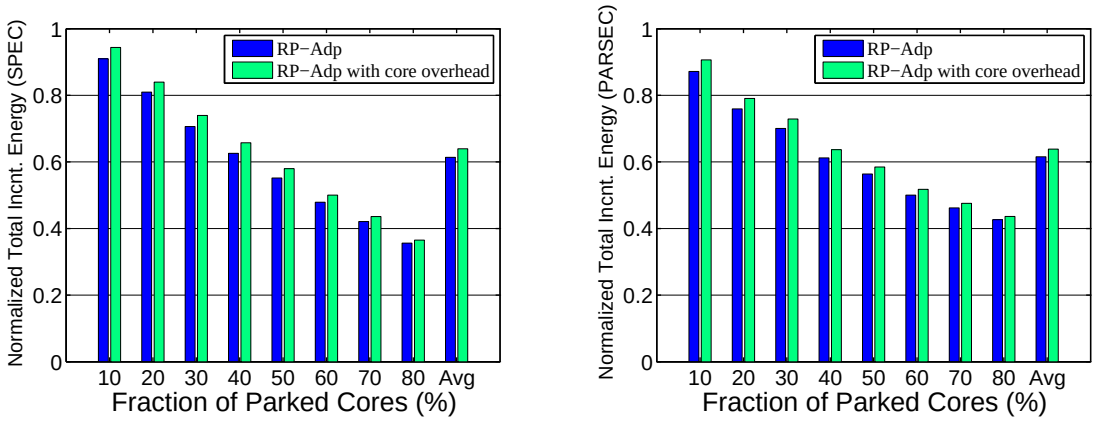


Figure 3.13: Total interconnect energy for RP-Adp, with and without extra core energy consumed by longer runs, for SPEC CPU2006 (a), and PARSEC2.1 (b), normalized to the base case of no Router Parking.

Impact of epoch length on energy savings. Figure 3.14 illustrates the interconnect energy of RP-Adp for SPEC CPU2006 and PARSEC 2.1 workloads, normalized to the base case of no Router Parking, under different epoch lengths; 1k cycles, 10k cycles

and 50k cycles. As shown in the figures, for very short epoch length (1000 cycles) - albeit unrealistic - the energy consumed while re-configuring the interconnect is significant such that, for small fraction of parked cores, it not only offsets the interconnect energy savings, but also increases the interconnect energy compared to the base case of no Router Parking by up to 18%, making such reconfiguration frequency undesirable. As the epoch length increases, the energy overhead becomes relatively small, hence, it is easily offset by energy savings. These observations give us the guideline that the epoch length should be in the order of tens of thousands of cycles or larger.

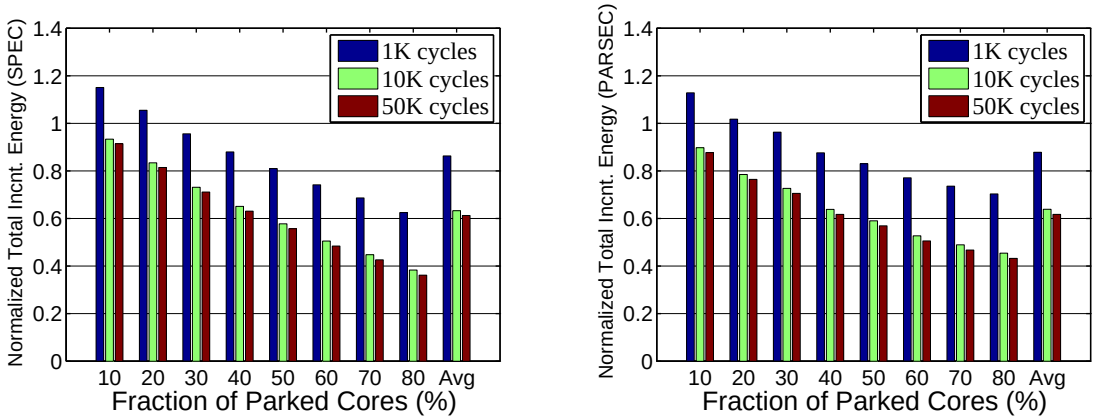


Figure 3.14: Total interconnect energy for RP-Adp with various epoch lengths, for SPEC CPU2006 (a) and PARSEC 2.1 (b), normalized to the base case of no Router Parking.

Core sleep frequency and its impact on epoch length. Realistically, cores sleep for long enough periods ($> 560\mu$s for Atom [26] and $> 800\mu$s for Nehalem [31]) in order to amortize the overhead of power gating and resumption. This translates into interconnect reconfiguration at the same order. Based on the above discussed epoch length sensitivity results, the reconfiguration overhead is negligible if the reconfiguration is done

"at-least" every 50k cycles, which is about 50μs on a 1GHz processor. In reality, we expect reconfiguration to occur much less frequently, leading to negligible reconfiguration overhead.

Impact of Routing Tables on energy savings. As mentioned in Section 3.4, our system incorporates two routing tables in each routers. Such routing tables are not needed in a 2D-Mesh interconnect that uses dimension-ordered routing (e.g., XY). Figure 3.15 shows the total interconnect energy of RP-Adp, RP-Adp with extra core energy due to prolonged runs (shown in the previous figure), and additional energy due to the introduced routing tables, for SPEC CPU2006, normalized to the base case of no Router Parking. As shown in the figure, the average interconnect energy savings have gone down from 40% to 37% to 36%. This shows that the energy consumed by the extra Routing Tables do not impact the energy savings of Router Parking algorithms by much. Further, this energy is well spent to enable significant energy savings via Router Parking. Note that, in case there are no available routers to park (i.e., all cores are active), RP architecture would consume more power than the baseline case due to these additional routing tables. In such a case, the FM can send out notifications to all nodes to turn off their routing tables, and revert to XY routing.

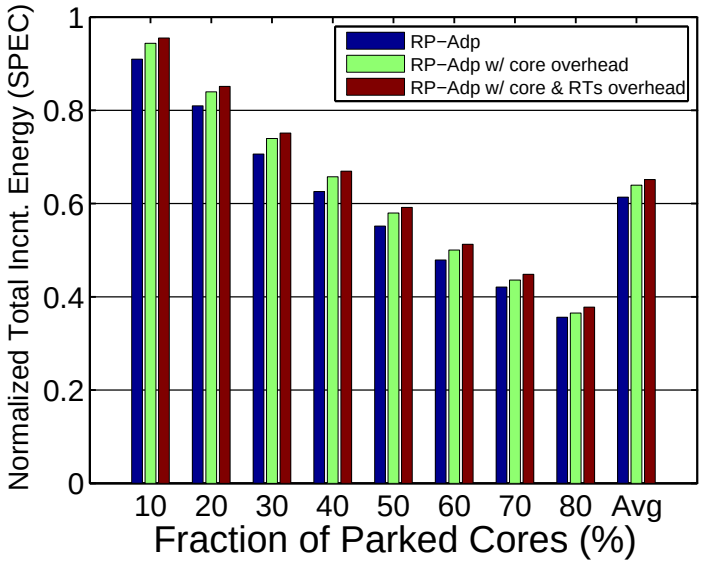


Figure 3.15: Total interconnect energy for RP-Adp, RP-Adp with extra core energy consumed by longer runs, and with extra core energy and extra energy consumed due to Routing Tables for SPEC CPU2006 normalized to the base case of no Router Parking.

3.5.3 Sensitivity Analysis

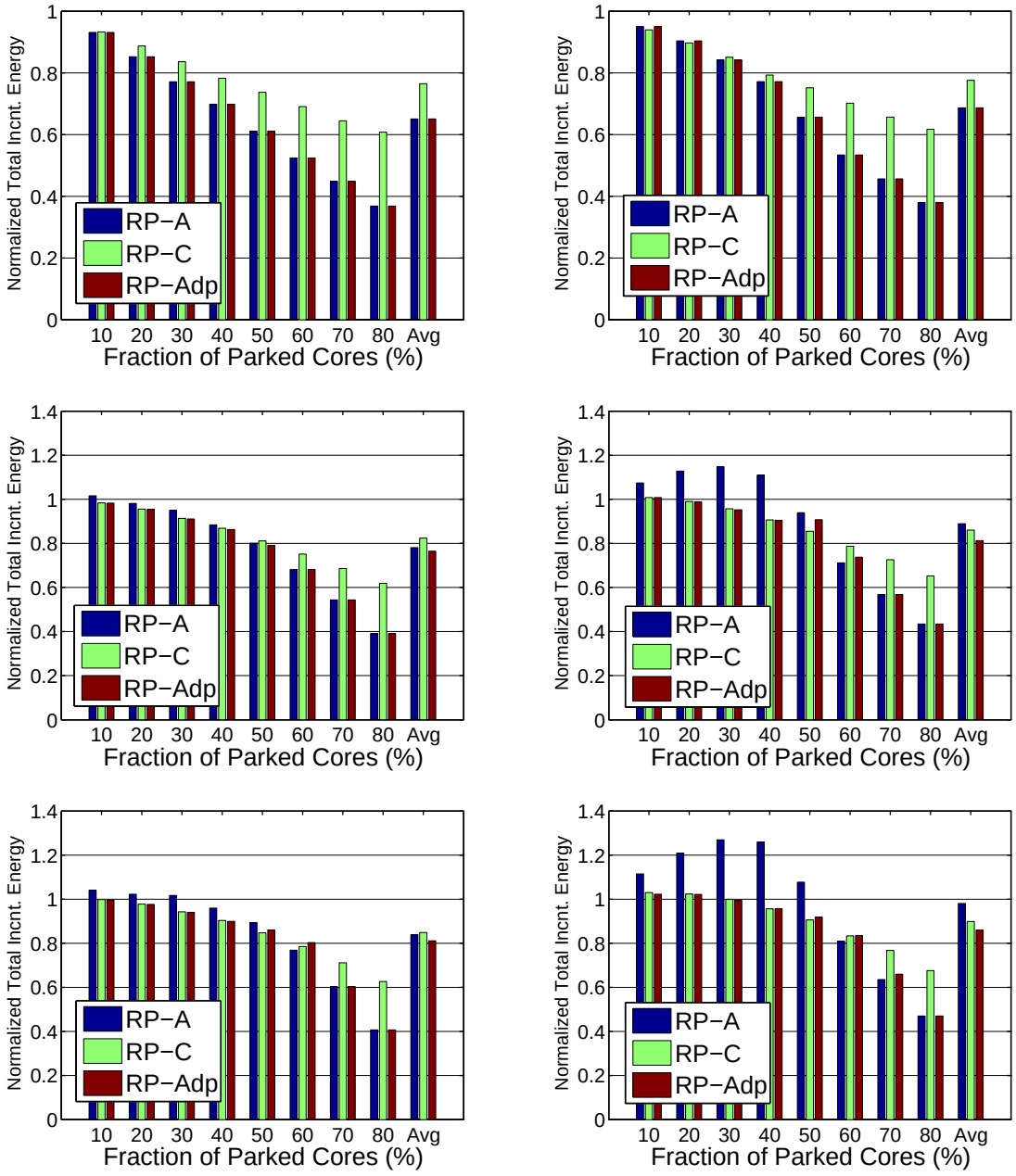
Sensitivity of Router-Parking Algorithms to Various Traffic Patterns. In Section 3.5.1, we analyzed the Router Parking algorithms behavior under uniform random traffic pattern. In this section, we evaluate Router Parking algorithms (RP-A, RP-C and RP-Adp) under different traffic patterns.

Figures 3.16 (a)(b) show the total interconnect energy under two different traffic patterns (TP and TOR), for various packet injection rates (top row 0.01, middle row 0.04. bottom row 0.06), normalized to the base case of no Router Parking. The major takeaways from this series of data are the following:

First, the traffic pattern plays an important role in how well each Router Parking algorithm performs. For Tornado (TOR) traffic pattern, each packet has only one single

minimal path. Any parked router between the source and destination causes packets detour. The situation is even more severe with RP-A, which may configure adjacent routers to be parked, making a packet travel several extra hops. On the other hand, For Transpose (TP), there are multiple minimal paths between source/destination pairs, (e.g, 9 minimal paths from (1,3) to (3,1)). Which means, even if RP algorithms park some routers in between, there is still a good chance that packets will follow another minimal path so the latency is not increased. Due to this reason, with the same injection rate, both RP-A and RP-C perform better under TP traffic pattern than TOR pattern, especially RP-A.

Second, the results show that under high packet injection rates (e.g., 0.06) with small fraction of parked cores ($< 30\%$), if the default latency is small (e.g., TOR-like traffic pattern), the default policy of no-RP is more efficient than applying either RP-A, or RP-C (Figure (c) bottom row). Thus, our adaptive algorithm (RP-Adp) adapted to such condition and produced negligible impact on interconnect energy.



(a) TP (b) TOR

Figure 3.16: Total interconnect energy for different injection rates 0.01 (top row), 0.04 (middle row), and 0.06 (bottom row) for Transpose (TP) and Tornado (TOR) traffic patterns, normalized to the base case of no Router Parking.

Sensitivity to running on RTL vs. on one of the cores. As discussed in Section 3.1, the FM functionality could be run on a dedicated RTL engine (our baseline assumption), or it could be run on one of the available cores. These two approaches have several advantages and disadvantages. While running on an RTL is much faster and consumes less power, it requires changes to the hardware and introduces area overhead. On the other hand, running on one of the cores does not require major hardware changes, however, at the cost of slower operation and more power consumption. Figure 3.17 shows the total interconnect savings for RP-Adp with an ideal FM (consumes no power), a FM that runs on an RTL (our baseline assumption), and a FM that runs on one of on-chip cores. The data corresponds to SPEC CPU2006 and is normalized to the base case of no Router Parking. As shown in the figure, running on an RTL engine has almost negligible power overhead if compared to an ideal FM. Further, running on one of the cores (2W TDP/core) does not reduce the average interconnect savings by much (40% to 37%) on average. This set of data shows that both approaches are fruitful.

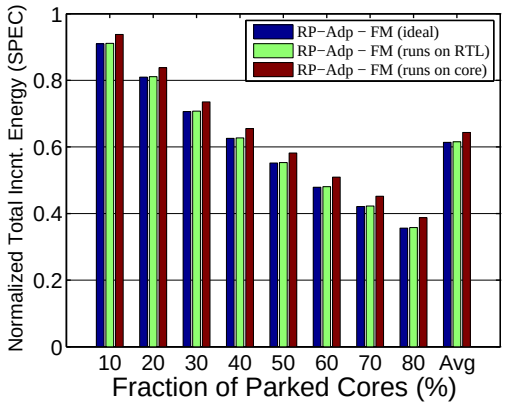


Figure 3.17: Total interconnect energy for RP-Adp with an ideal FM, RTL-based FM, and a FM whose functionality is run on one of the existing cores, for SPEC CPU2006, normalized to the base case of no Router Parking.

3.6 Other Issues

Shared LLC CMPs. Many contemporary CMPs are designed with a *distributed but shared* Last Level Cache (LLC). For routers in such CMPs, even if a core is parked, its slice of the distributed LLC and the associated router cannot be power gated. This is because the contents of the cache slice must be retained to service mapped cache requests and maintain cache coherency. Although Router Parking does not apply to such architectures *directly*, Router Parking can be augmented with a distributed LLC power management policy. This power management policy power-gates selective LLC slices dynamically by changing the hash function that determines the home LLC slice. This is outside the scope of this dissertation and we leave it as future work.

Scalability The Fabric Manager does not necessarily need to communicate with all nodes in the system following a network reconfiguration. It could perform localized network re-configuration such that it limits the reconfiguration overhead. Further, online connectivity algorithms such as Online Tarjan algorithm can re-evaluate the connectivity of the interconnect if a few nodes change their status without having to reevaluate the entire network from scratch. Employing localized reconfiguration and online algorithms improve the scalability of the centralized Fabric Manager. Finally, with a much larger number of cores, an n-level hierarchy of Fabric Managers could be introduced, with each FM taking responsibility of an island of cores. All first level FMs connect to a second level FM, and so on. Recall that as the number of cores grows, the opportunity of core parking grows, hence larger static energy savings are realized. We believe that RP will continue to be a fruitful architecture for future many-core CMPs.

How does Router Parking interact with the OS? Thus far, we have discussed an OS-agnostic Router Parking architecture. Although an OS-aware counterpart offers less

portability, it does provide additional operational flexibility. First, instead of defining maintaining network connectivity as *any active core can communicate with any other active core*, it could be defined as *any thread can communicate with other collaborating threads (e.g., multiple threads working on a certain job.* Having the OS passing down hints to the FM about the running jobs and the threads associated with each job can help reduce interconnect reconfiguration complexity and overhead. Second, in an OS-agnostic environment, if the OS migrates a certain thread from an active core to an idle one, the FM has to adapt to the new state by doing an interconnect reconfiguration. If, on the other hand, the FM communicates with the OS, it can postpone or prepone network reconfiguration to be in harmony with OS migration decisions. Finally, if the OS clusters active threads in one interconnect portion - although not favorable due to creating hot spots, Router Parking can benefit from this by simply shutting down all routers associated with inactive cores without the need to evaluate connectivity (i.e., by running the Router Parking algorithms) since all active cores are connected to each other.

3.7 Related Work

There is a rich body of work that has studied the problem of reducing interconnect traffic in CMPs. We divide this work into the following main categories.

Reducing Static Power Consumption Soteriou et al. [63] proposed a component-based router power gating technique that works by monitoring the interconnect link utilization and power gating components such as ports and links in response to bursts and dips in network traffic. Although this approach shows promising result, it has several drawbacks. First, performance may suffer significantly when the wake-up latency is

high or traffic is unpredictable. In order to react to changes in traffic quickly, only lightweight power saving techniques can be employed. For example, it power gates only certain router components, hence missing an opportunity to power gate entire routers. Second, the approach is agnostic to the core sleep state. It misses an opportunity for power gating mostly idle routers attached to sleeping cores. Finally, in addition to the previous issues, this approach assumes that outbound and inbound links have different physical ports for the convenience of maintaining network connectivity. Whereas many modern interconnects such as Intel's LightPeak [30] have their inbound and outbound links connected to the same physical port, and are hence either both on or both off. To the best of our knowledge, Router Parking is the first work that takes a *holistic* view of the system by *parking* selective routers based on the sleep state of their associated cores. By doing this *proactively*, traffic is aggregated away from sleeping routers, hence we avoid the need for a fast wake-up latency. At the same time, power gating entire routers enable saving more power. However, component-based power-gating may complement Router Parking (RP) if it is applied to active routers.

Reducing Dynamic Power Consumption Matsutani et al. [45] proposed decreasing router operating frequency and supply voltage in order to reduce the router power when necessary. Link Dynamic Voltage Scaling [60] explored scaling link frequency and voltage to reduce link power consumption under low utilization. Dynamic Voltage and Frequency Scaling (DVFS) is also exploited [47] to tune the frequency of on-chip routers to improve power and performance. These approaches mainly aim at reducing the dynamic power consumed by active routers and links while our work reduces static power. This body of work is orthogonal to our Router Parking techniques and can be applied to further reduce power of the remaining active routers.

Reducing Dynamic and Static Power Consumption Researchers have also proposed techniques to save both dynamic and static interconnect power [49, 23, 37]. Bufferless routers [49], observes that buffers consume a significant fraction of the total router power consumption, and therefore bufferless routing schemes can reduce power. These works all focus on power managing router components by reacting to the traffic pattern. In contrast, we take a holistic approach to adaptively and proactively manage the traffic and enable the entire router to enter low power state. Researchers also proposed more sophisticated topologies, e.g., Parallel Concreted Mesh (PC-Mesh) [10] and Homogeneous Parallel Concentrated Mesh (HPC-Mesh) [11] that enable richer connections thus providing more opportunities for power saving and fault tolerance.

How is Router Parking different from Fault Tolerant Networks. There has been substantial research on Fault Tolerant Networks (FTN) [1, 68, 22]. While one may imagine applying FTN techniques to route around parked routers, there are fundamental differences between FTN and our router parking. In RP, a centralized manager is preferred because it provides an ability to perform *holistic* power management. In contrast, FTN does not manage power and prefers distributed algorithms to tolerate unpredictable failures. The need for holistic power management is recognized by the industry and is incorporated into various products [28], and other NoC studies [51]. Another difference is that RP operation must be fast in order to accommodate frequent interconnect changes, while faults in FTN are rare cases. Hence, there is no need for FTN algorithms to be fast. Finally, RP must ensure full network connectivity, whereas in FTNs, faulty routers may cause disconnected network partitions.

3.8 Conclusions

In this chapter, we proposed and motivated Router Parking (RP), which selectively power gates interconnect routers in large Chip Multi Processors (CMPs), as a step towards energy proportional computing. We evaluated three RP algorithms - an aggressive approach (RP-A) that parks as many routers as it can while maintaining connectivity between active cores, a conservative approach (RP-C) that tries to reduce the performance impact of rerouting traffic around parked routers, and an adaptive approach (RP-Adp) that dynamically selects the better of the first two approaches. RP-A works better for light traffic because it gets the most static power savings with only a small performance impact due to increased packet routing latency. RP-C parks fewer routers in order to keep detours around parked routers short. RP-C works better as the traffic increases - it ensures that the dynamic energy consumed due to the longer detour latency does not overtake the static energy savings due to parked routers. Overall, we found that RP reduces the total interconnect energy by 32%, 40% and 41% respectively, while only reducing the performance by less than 0.5%.

Chapter 4

Conclusion

Current and future Chip MultiProcessor (CMP) systems pose interesting design challenges and trade-offs in several parts of the uncore subsystem, particularly the Last Level Cache (LLC) and the interconnect.

Private LLCs suffer a capacity fragmentation problem; some caches are over-utilized while others are under-utilized. Capacity Sharing is a technique for reducing such fragmentation; it allows applications that need additional cache space to place their victim blocks on remote caches. However, current capacity sharing mechanisms treat remote caches as the victim cache for the local cache space without considering the temporal reuse of the running application. In the first part of this dissertation, we studied the aforementioned problem. We have shown that treating remote caches as victim caches guarantees a high number of remote cache hits relative to local cache hits for applications that exhibit anti-LRU behavior, which are usually the same applications that benefit from additional cache capacity. To address this problem, we investigated strategies that consider placing, not only locally-evicted blocks in remote caches, but also *newly-fetched* blocks in remote caches. We demonstrated an upperbound performance that could be

obtained from combined placement and replacement decisions in capacity sharing, by using future trace information to make such decisions. Based on our findings, we proposed a scheme that is implementable in hardware with little hardware overhead. We showed that this scheme, APP, improves performance by 29% on average compared to a baseline with no capacity sharing, across 50 multi-programmed workloads consisting of four SPEC CPU2006 applications. APP outperforms DSR, the state-of-the-art capacity sharing mechanism that only places local victim blocks in remote caches, by up to 18.2%, with an average improvement of 3%. APP dynamically identifies which applications should be allowed to place their blocks in remote caches, and which applications should not. In addition to improving aggregate performance significantly, APP also has safeguards to ensure that applications whose caches are accepting blocks from other cores are not slowed down by much.

On another front, state-of-the-art interconnects such as Mesh and Torus are the interconnects of choice to provide a scalable interconnect medium to accommodate the rising number of cores (and LLC slices) in CMPs. With such interconnects, even if some cores go to deep sleep states, the interconnect remains fully active to service the remaining active cores. Unfortunately, this increases the idling power and reduces energy proportionality significantly. In the second part of this dissertation, we studied this problem. We have shown that in current interconnect architectures, routers never get the chance to go deep sleep states when their attached cores (and LLC slices) are in deep sleep state. Hence, in order to maintain energy proportionality, we proposed to *power-gate* routers attached to parked cores using a technique we refer to as Router-Parking. We have discussed and evaluated two modes of operation that differ in how aggressively they park routers, an aggressive approach (RP-A) and a conservative one (RP-C). The aggressiveness determines the average interconnect latency impact and the increase in the dynamic

energy due to packet detouring around parked routers. Further, in order to adapt to run time changes, we further presented an adaptive policy (RP-Adp) that employs the better algorithm based on run-time conditions. We showed that RP-A works better for light traffic because it gets the most static power savings with only a small performance impact due to increased packet routing latency. RP-C parks fewer routers in order to keep detours around parked routers short. RP-C works better as the traffic increases - it ensures that the dynamic energy consumed due to the longer detouring latency does not offset the static energy savings due to parked routers. We experimented with both, synthetic traffic of different patterns, and real workloads taken from the SPEC CPU2006 and PARSEC 2.1 benchmark suits. We found that RP reduces the total interconnect energy by 32%, 40% and 41% respectively, while only reducing the performance by less than 0.5%.

REFERENCES

[1] Konstantinos Aisopos, Andrew DeOrio, Li-Shiuan Peh, and Valeria Bertacco. Ariadne: Agnostic reconfiguration in a disconnected network environment. PACT '11, Washington, DC, USA, 2011.

[2] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. Exact: explicit dynamic-branch prediction with active updates. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 165–176, New York, NY, USA, 2010. ACM.

[3] Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John Carter. Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches. In *HPCA '09: Proceedings of the 2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[4] Arnab Banerjee, Robert Mullins, and Simon Moore. A power and energy exploration of network-on-chip architectures. NOCS '07, pages 163–172, Washington, DC, USA, 2007.

[5] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40:33–37, December 2007.

[6] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006.

[7] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems journal*, 5(2), 1966.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

[9] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54:67–77, May 2011.

[10] J. Camacho, J. Flich, A. Roca, and J. Duato. Pc-mesh: A dynamic parallel concentrated mesh. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 642 –651, sept. 2011.

[11] Jesus Camacho and Jose Flich. Hpc-mesh: A homogeneous parallel concentrated mesh for fault-tolerance and energy savings. ANCS'11, Washington, DC, USA, 2011.

[12] Jichuan Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *ISCA'06*, 2006.

[13] Jichuan Chang and Gurindar S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS '07*, pages 242–252, New York, NY, USA, 2007. ACM.

[14] Mainak Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches. In *MICRO*, 2009.

[15] Xuning Chen and Li-Shiuan Peh. Leakage power modeling and optimization in interconnection networks. ISLPED '03, pages 90–95, New York, NY, USA, 2003. ACM.

[16] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication and Capacity Allocation in CMPs,. In *In the 32th ISCA*, June 2005.

[17] J. Duato and T.M. Pinkston. A general theory for deadlock-free adaptive routing using a mixed set of resources. *Parallel and Distributed Systems, IEEE Transactions on*, 12(12):1219 –1235, dec 2001.

[18] Ramon Doallo Dyer Rolan, Basilio B. Fraguela. Adaptive Line Placement with the Set Balancing Cache. In *MICRO*, 2009.

[19] N. Agarwal et. al. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. 2009.

[20] Stijn Eyerman and Lieven Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3), 2008.

[21] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development*, 54(1):3:1 –3:11, january-february 2010.

[22] M.E. Gomez, J. Duato, J. Flich, P. Lopez, A. Robles, N.A. Nordbotten, O. Lysne, and T. Skeie. An efficient fault-tolerant routing methodology for meshes and tori. *Computer Architecture Letters*, 3(1):3, january-december 2004.

[23] H. Matsutani, et al. Performance, area, and power evaluations of ultrafine-grained run-time power-gating routers for cmps. *CAD of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):520 –533, april 2011.

[24] Wei Huang, M.R. Stan, S. Gurumurthi, R.J. Ribando, and K. Skadron. Interaction of scaling trends in processor architecture and cooling. pages 198 –204, feb. 2010.

[25] IDC. Future of Virtualization. *http://www.vmware.com/files/pdf/analysts/Future-of-virtualization-IDC.pdf*, 2008.

[26] Intel Atom. Atom Idle governor. *https://build.pub.meego.com/package/view_file?file=linux-2.6.35-intel-idle.patch&package=dlm-kernel-netbook&project=home%3Alazhar*, 2010.

[27] Intel Corp. Teraflops Research Chip. *download.intel.com*, 2007.

[28] Intel Corp. Core Micorarchitecture. *http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.19.9-Desktop-CPUs/HC23.19.911-Sandy-Bridge-Lempel-Intel-Rev%207.pdf*, 2008.

[29] Intel Corp. Intel Xeon E7 Processor. *http://ark.intel.com/products/53580/Intel-Xeon-Processor-E7-8870-(30M-Cache-2_40-GHz-6_40-GTs-Intel-QPI)*, 2011.

[30] Intel Corp. Thunderbolt Technology. *http://techresearch.intel.com/spaw2/uploads/files/*, 2011.

[31] Intel Nehalem. Nehalem Idle governor. *http://lists.linaro.org/pipermail/linaro-dev/2012-February/009954.html*, 2010.

[32] Intel Pressroom. Intel 22nm 3-d tri-gate transistor technolog, 2011.

[33] ITRS. International Technology Roadmap for Semiconductors: 2008 Edition, Assembly and packaging. In *http://www.itrs.net/Links/2008ITRS/AP2008.pdf*, 2008.

[34] S. Dighe J. Howard. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *(ISCC)*, ISSCC10, 2010.

[35] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive Insertion Policies for Managing Shared Caches. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008.

[36] A.B. Kahng, Bin Li, Li-Shiuan Peh, and K. Samadi. Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration. april 2009.

[37] Gwangsun Kim, J. Kim, and Sungjoo Yoo. Flexibuffer: Reducing leakage power in on-chip network routers. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 936 –941, june 2011.

[38] Jason Sungtae Kim, Michael Bedford Taylor, Jason Miller, and David Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. ISLPED '03, New York, NY, USA, 2003. ACM.

[39] Minkyong Kim and Brian Noble. Mobile network estimation. MobiCom '01, New York, NY, USA, 2001. ACM.

[40] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.

[41] J. Lee, Y. Solihin, and J. Torrellas. Automatically Mapping Code on an Intelligent Memory Architecture. In *In 7th Intl. Symp. on High Performance Computer Architecture*, 2001.

[42] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008.

[43] O. Lysne, J.M. Montanana, J. Flich, J. Duato, T.M. Pinkston, and T. Skeie. An efficient and deadlock-free network reconfiguration protocol. *Computers, IEEE Transactions on*, 57(6):762 –779, june 2008.

[44] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2), 2002.

[45] H. Matsutani, M. Koibuchi, Daihan Wang, and H. Amano. Adding slow-silent virtual channels for low-power on-chip networks. In *NoCS 2008. Second ACM/IEEE International Symposium on*, pages 23 –32, april 2008.

[46] Harlan McGhan. Niagara 2. *Microprocessor Report*, November 2006.

[47] Asit K. Mishra, Reetuparna Das, Soumya Eachempati, Ravi Iyer, N. Vijaykrishnan, and Chita R. Das. A case for dynamic frequency tuning in on-chip networks. MICRO 42, New York, NY, USA, 2009. ACM.

[48] Asit K. Mishra, N. Vijaykrishnan, and Chita R. Das. A case for heterogeneous on-chip interconnects for cmps. ISCA '11, New York, NY, USA, 2011. ACM.

[49] Thomas Moscibroda and Onur Mutlu. A case for bufferless routing in on-chip networks. ISCA '09, New York, NY, USA, 2009. ACM.

[50] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[51] George P. Nychis, Chris Fallin, Thomas Moscibroda, Onur Mutlu, and Srinivasan Seshan. On-chip networks from a networking perspective: congestion and scalability in many-core interconnects. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 407–418, New York, NY, USA, 2012. ACM.

[52] M.K. Qureshi. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, Feb. 2009.

[53] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. ACM, 2007.

[54] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006.

[55] R. Tarjan. Algorithm Design. *Communications of the ACM*, 1987.

[56] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 2006.

[57] Ahmad Samih, Anil Krishna, and Yan Solihin. Understanding the limits of capacity sharing, 2009.

[58] Ahmad Samih, Yan Solihin, and Anil Krishna. Evaluating placement policies for managing capacity sharing in cmp architectures with private caches. *ACM Trans. Archit. Code Optim.*, 8(3):15:1–15:23, October 2011.

[59] M.D. Schroeder, A.D. Birrell, M. Burrows, H. Murray, R.M. Needham, T.L. Rodeheffer, E.H. Satterthwaite, and C.P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. *Selected Areas in Communications, IEEE Journal on*, 9(8):1318 –1335, oct 1991.

[60] Li Shang, Li-Shiuan Peh, and N.K. Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *HPCA-9 2003.*, pages 91 – 102, feb. 2003.

[61] Arjun Singh, William J Dally, Amit K Gupta, and Brian Towles. Goal: A load-balanced adaptive routing algorithm for torus networks. In *International Symposium on Computer Architecture (ISCA*, pages 194–205. ACM Press, 2003.

[62] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *ASPLOS-IX*, pages 234–244, New York, NY, USA, 2000. ACM.

[63] Vassos Soteriou and Li-Shiuan Peh. Exploring the design space of self-regulating power-aware on/off interconnection networks. *IEEE Trans. Parallel Distrib. Syst.*, 18:393–408, March 2007.

[64] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. ACM, 2008.

[65] Standard Performance Evaluation Corporation. *http://www.specbench.org*, 2006.

[66] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *J. Supercomput.*, 28(1):7–26, 2004.

[67] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. *SIGPLAN Not.*, 44(3), 2009.

[68] Jie Wu. A fault-tolerant and deadlock-free routing protocol in 2d meshes based on odd-even turn model. *Computers, IEEE Transactions on*, 52(9):1154 – 1169, sept. 2003.

[69] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. *SIGARCH Comput. Archit. News*, 37(3), 2009.

[70] Michael Zhang and Krste Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. IEEE Computer Society, 2005.