A Study on Hierarchical Cache System Control based on Access Pattern Analysis for Chip Multiprocessor Systems

Huatao ZHAO

October 2020

Waseda University Doctoral Dissertation

A Study on Hierarchical Cache System Control based on Access Pattern Analysis for Chip Multiprocessor Systems

Huatao ZHAO

Graduate School of Information, Production and Systems

Waseda University

October 2020

Abstract

Multilevel hierarchical cache is used to buffer the huge gap on processing speed between on-chip multi-core processor level and off-chip large-scale memory level. As the number of cores integrated on a single chip die recently tends to be dozens or even hundreds, the hierarchical cache loses its ability to cover over the speed gap and fails to optimally interconnect across on-chip components. More seriously, stacked multi-layer systems which have high integration density require a hierarchical cache with higher throughput to fully meet cache access demands among many concurrent threads, and also require more efficient method to guarantee data coherence in hierarchical cache.

To meet the throughput requirement in hierarchical cache, multi-level and private-shared cache structures are used in the recent chip multiprocessor systems (CMPs). Those cache hierarchies can even take a half of overall on-chip area and energy consumption of CMPs. However, many components of a hierarchical cache rarely contribute access hits in the most of execution time, but they waste too much energy for standby. Moreover, shared data which serve to concurrent threads are existing in a hierarchical cache, and coherence maintenances on those data waste too many clock cycles. As a consequence, the current hierarchical cache induces serious issues as follows: (1) Power issue on misallocating cache resources and (2) Data sharing issue. In the issue (1), On-chip caches suffer from high energy consumption overhead and a chip area overhead while such hierarchical cache is failed to satisfy cache resource demands of many threads in a large scope. In the issue (2), cache accesses to shared data among concurrent threads cause extremely expensive coherence maintenance.

The power issue (1) is caused by two reasons. First, increasing of cores requires a large scale of hierarchical cache for ensuring enough spare resources. Second, a demand for cache resource during runtime is locally changed along with processing at cores, which leads to allocation inequality that some threads tend to be in rush traffic but some threads are uncrowded with redundant cache

resources. Rawlins, M. [IEEE T COMPUT, 2013] and Chen, G. [Microproc.Microsyst., 2016] proposed cache tuning based methods to explore optimal allocations on cache resources for each length-fixed interval of instructions, where during runtime, the energy-lowest cache size is explored for the next intervals once behavior changed (i.e., miss rate changed). Adegbija, T. [IEEE T VLSI SYST, 2018] and Wei, W [ACM T ARCHIT CODE OP, 2017] proposed a phase based exploration method on cache resources to save access energy, where firstly a number of phases are classified for each application, and then the optimal phase is explored if miss rate is larger than threshold. However, a demand for the cache resource in any executing period and in any thread could not be optimally satisfied with allocated cache resources appropriately. Thus, the first objective of this thesis is to dynamically allocate cache resources to meet each demand for cache resource for each thread in any executing period, in other words, tune cache bank supply to concurrent threads' demand dynamically in intervals of selected subroutine calls, thereby saving energy by making the utmost of cache utilization.

The data sharing issue (2) is highly related to the fact that some kinds of cache access patterns (i.e., write accesses to shared data) cause expensive maintaining operations among many cores. Shared accesses generated in many concurrent threads may result in serious data inconsistency, and crossed accesses whose target data are existing in other threads will lead to access misses. The access pattern analysis shows that distributions on those harmful patterns can possess a considerable percent of total accesses. Lotfikamran, P. [IEEE HPCA, 2017] proposed a proactive resource allocation method to improve system performance based on shared data traffic profiling, which firstly predict that hot threads require more resources, and then allocate required resources to the threads in each time interval, thereby reducing router stall time and improving performance. Gupta, S. [ICPP, 2015] proposed a spatial locality-based cache partitioning method, which firstly exploits spatial locality in partitioned shared cache, and then, for memory-intensive thread, increases its block size to enlarge shared data re-usage, and save some capacity to other

threads. However, it is still difficult in conventional access paths to detect and deliver shared data, as those paths waste plenty of clock cycles to handle harmful cache accesses. Thus, the second objective of this thesis is to efficiently handle those harmful cache accesses in the proposed concurrent path, which acts as a shortcut path on data sharing accesses among private caches to detect and route shared data in advance.

This thesis is organized as follows:

Chapter 1 [Introduction] introduces the research background of hierarchical cache designs and previous works on the cache optimization, and then describes the outline of the proposed methods.

Chapter 2 [Access Pattern Analysis] represents the detailed experiments on cache access patterns and statistically classifies cache access distributions. Then, those patterns are analyzed to reveal the internal relationships among cache resource demands, locality features and access distributions.

Chapter 3 [Controllable Cache Resource Allocation] proposes a low-power hierarchical cache scheme applying the control theory to give a hardware-based solution for an optimal cache resource allocation in some interval granularity. Firstly, an effective bank allocation policy is proposed for adaptive cache resource allocation. Secondly, preferable intervals which mean proper timing to change resource allocation are designed in fine granularity of per-subroutine. Finally, the controllable cache resource allocation policy is proposed combining the discrete control theory by the PID based controller and cache resource allocation at subroutine-based interval. Experimental results using SPEC benchmark data and Gem5 simulator show that energy consumption on shared cache can be saved by 39.7% on average compared with the conventional equi-interval method, and saved by 11.6% and 18.2% compared with Chen's method [Microproc. Microsyst., 2016] and Adegbija's method [IEEE T VLSI SYST., 2018], respectively.

Chapter 4 [Stacked 3D On-chip Cache Network] proposes a stacked 3D three-layer on-chip architecture consisted of enhanced global- and local- router

networks. The router is improved in cache access detection, sharing data replacement and target data delivery functions. The proposed interaction path in the network can support fast shared data, in which “crossed read” can achieve target data by routing from other caches and both “shared write” and “crossed write” can directly update all copies in virtue of the routing network. Moreover, VLSI layout design of the proposed router architecture is implemented to verify the placement & routing details. And the on-chip design of a stacked 3D structure is evaluated from the viewpoint of thermal affection and estimated chip size. Simulation results indicate that the proposed router-integrated hierarchical cache design of a CMP system improves the system performance by 31.9% and on-chip energy by 17.6%, compared with the base system without a cache network.

Chapter 5 [Conclusion and Future Work] sums up this thesis on achievements and contributions, that is, much energy savings is achieved by the proposed self-adapting cache resource allocation method, and both performance and energy consumption are improved by the proposed router-integrated cache optimization method. Finally, further optimizations on proposed designs are expected in future work.

As a consequence, this thesis represents optimization techniques on hierarchical cache including shared cache level and private cache level. For power issue on misallocating cache resources, a discrete PID based controller is integrated to form a self-adaptive cache allocation method in a novel granularity of per-subroutine based interval. For data sharing issue, a shortcut path on harmful accesses is designed to improve coherence-maintenance efficiency by integrating enhanced router networks. The experimental results show the substantial improvements on both performance and energy consumption.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 Research Background on Hierarchical Cache

1.1.1 Speed Gap between Cores and Memories

In the last decade, on-chip integration density varies from single-core, millions of transistors to dozens of cores, billions of transistors. For example, from Intel Pentium 4 to Xeon Phi [58]. Even though clock frequencies of those processors are almost remaining same (i.e., 2.0 GHz), processing capacities of modern processors behave to be unmatched with their integration scales, where there only have several times of performance improvement rather than expected dozens of times. To look inside of the processor, the highly improved core usually needs to wait for many clock cycles for achieving instructions and datum from hierarchical cache, because the hierarchical cache is much slower than the core, while caches cannot meet the speed demand of linked cores. So that the key reason for such undesired improvement is caused by the speed gap between cores and memories (including on-chip caches and off-chip memory), while such gap has been enlarging since modern computer architecture was proposed.

In Fig. 1-1, transistor scales of some mainstream processor products and private caches are shown from the 1970s to this day. As shown in the figure, the scales of Intel on-chip processors are varying similarly as the description of

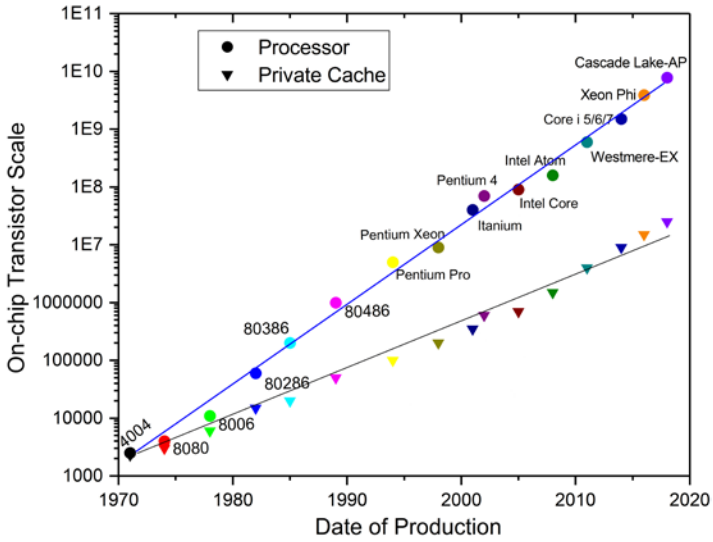


Figure 1-1. On-chip integration scale statistics [29, 58, 63]. The integrated scales on both microprocessor and first level cache transistor are counted among Intel processor series from 1971 to 2018. On-chip transistor scale values are not drown in equidistance.

Moore's law: transistor scales turn to be four times every three years. Meanwhile, linked private cache scales vary in much small amplification (about triple in every four years), in other words, the performance of those processors are much faster than their linked cache hierarchies in the modern system on a chip.

In recently released chip multi-processors (CMP), the number of integrated cores in a chip tends to be 64 or even more. For example, AMD EPYC Rome processor (2019.8) and HUAWEI KP920 processor (2019.10) use have 64-core on chip architecture. To serve so many cores concurrently, large and complex hierarchical cache is employed although it consumes a large portion of on-chip energy. Typically in so-called ‘chiplet’ architecture, many cores require many caches for buffering, and data interaction method tends to shift from traditional ‘Processor to Processor’ to ‘Cache to Cache’ as throughput requirement among threads are greatly increasing. Hence, chip developers try to optimize hierarchical cache in purpose of both low energy consumption and high performance. In this thesis, a lot of efforts have been done on such topic.

1.1.2 Hierarchical Cache Issues

The hierarchical cache usually consists of multi-level caches which can be

classified into two types: private cache and shared cache. Those two cache types form a pyramidal shape in size, energy consumption, and hit rate, while form an inverted-pyramid shape in speed and access frequency. As the integration scales of first-level instruction cache and data cache are limited by speed and cost requests, their sizes are the smallest over other cache levels. Meanwhile, from top to bottom in the pyramid, larger cache sizes make sure the better hit rate, but are enduring with low speed and high energy consumption due to their integration scales. Typically in shared last level cache, it serves many private paths concurrently and acts as the last line of defense on on-chip access misses, while it is criticized for its large size, low speed, and high energy consumption. Ideally, cache hierarchies are desired to just satisfy the hardware demand for maximal access hits while their integration scales are minimally activated for energy saving. Hence, there has a trade-off on cache combinations that are needed to balance performance and energy consumption.

As there have only several kinds of applications executed in embedded systems, those applications can be classified based on runtime characteristics for fully adapting to current system combinations. For example, a behavior-stable application such as *bzip2* benchmark may only need a small quantity of hierarchical cache while redundant cache accesses can be repeatedly hit on limited cache entries, however, a behavior-unstable application such as *soplex* benchmark may need a much larger hierarchical cache to cover up the cache access distance, otherwise many cache miss accesses will happen and result in serious access delay and energy consumption.

In the first level of hierarchical cache (L1 cache), caches are designed as small size, low associativity, and small line size for the purpose of high access speed, which is almost same as the speed of processor unit. Hence, the L1 cache has very few access entries to cover up application locality or reusing distance and then dozens of percent of all L1 cache accesses tend to be miss accesses, which will be handled in the next level of hierarchical cache. In the last level of hierarchical cache, last-level cache (LLC) is acting as the last line of defense on on-chip cache accesses. Therefore, LLC is designed with large size, high

associativity, and large line size for the purpose of high hit rate. Moreover, each LLC serves with many threads of multi-processor cores concurrently, and plenty of LLC accesses will be occurring in parallel. Inevitably, data coherence in LLC acts as one of the most important issues, because each modification on shared data will cause serious coherence issue among existing copies and brings about difficulty on writing back to off-chip memory. Furthermore, LLCs are designed to satisfy diverse workloads with large enough integrated scale since many processor cores need a number of parts of LLC. Note that each workload has particular features on LLC demands including cache size, associativity, and line size and so on. The demand variations reacted in hit rate or miss rate of LLC accesses are shown out with diverse values. In other words, each workload has its favorite hardware combinations on LLC. To sum up, the cache hierarchies in modern chip multiprocessor systems have the following issues which are aimed at this thesis.

(1) Cache resource demand issue. This issue is generated because the diverse hardware demands of each workload are encountering with fixed and superfluous cache scales. If allocated hardware scales exceed the hardware demands of one workload, extra energy consumption and access latency will be consumed, on the contrary, if allocated hardware scales are insufficient, many miss accesses will occur and waste many clock cycles for off-chip memory access.

(2) LLC partition issue. The LLC should serve with many threads of multi-processor cores, and its components will be partitioned to each thread for the purpose of data pollution prevention. The partitioned scales from LLC need to be dynamically adapted to the hardware demands of current threads.

(3) LLC access sharing issue. Each access request from any thread may have the existing contents in other partitioned parts of LLC, thereby data coherence among copies of target data acts as the access sharing issue which will largely degrade LLC performance to maintain shared data across the entire LLC.

1.2 Related Works on Hierarchical Cache Optimization

In recent years, hierarchical cache optimization is one of the most active topics as the hierarchical cache of commercial mainstream processors can take even half of the on-chip area and energy consumption. Many pieces of research were proposed to improve hierarchical cache in several angles: cache resource re-allocation [1, 6, 8, 9], shared cache partition [11, 12, 18], and parallel accessing [13, 19, 20].

Initially, C. Zhang, et al. [7] proposed the self-tuning method to support hardware-based tuning on cache size, line size and associativity of both private and shared cache, thereby redundant cache components can be shut down in purpose of energy consumption saving. To trace the application variety on cache components, researches in [11, 18] proposed the trace-driven simulation method for efficient management in a multi-thread system. However, count in cache parameters and hierarchical cache, the optimal combination corresponding to one application should be explored from the set which has thousands of combinations. Thus, exploration methods are acclaimed to achieve the target combination rapidly through off-line or runtime exploring. W. Wang, et al. [9, 12] had described the scales of cache combinations for LLC tuning, which tend to be millions of combinations according to a short runtime interval (a runtime period with some dynamic instructions), however, the ideal status is to explore the optimal combination from the large combination set within several or dozens of clock cycles, which seems very difficult to implement. Hence, research in [6] proposed an off-line searching algorithm to explore each combination corresponding to each interval, and then apply all searching results on runtime hierarchical cache allocation. Furthermore, A. Gordon-ross, et al. [24, 25] presented a dynamic allocation mechanism based on phase information of each application interval to achieve the approximate optimal combinations, which showed some energy saving in a two-level hierarchical cache. Moreover, M. Qureshi, et al. [26] integrated the utility theory of economics into cache resource allocation for the purpose of maximizing the total utility of allocated resource

on hit rate or miss rate, thereby hierarchical cache can be apportioned as thread demands. Research in [27] employed analysis models to statically explore the optimal cache combinations of LLC and then the LLC can be partitioned according to evaluation results. H. Cook, et al. [28] proposed the hardware evaluation based LLC partition mechanism to dynamically control partitioning signals for the purpose of LLC resource availability and low energy consumption. Researches in [29] and [30] tried to partition the LLC in the granularity of each cache way with less hardware overhead. And further research in [61] proposed a cache block-based partition method for subtly LLC allocation. To ensure partitioning fairness, researches in [14] and [62] proposed the coordinated allocation methods through evaluating the runtime fairness of partitioned LLC resources. Meanwhile, A. Herdrich et al. [63] proposed the QoS based cache partition method to endow priority for some hot threads for the purpose of high availability. Researches in [64] and [65] saved a large amount of both static and dynamic energy through partitioning LLC based on efficiency criterion.

However, those cache reconfiguration methods and cache partition methods failed to trace the real cache resource demands of processing threads, and their cache allocating operations were done in improper fixed-length based intervals such as ten million dynamic instructions per interval, thereby limiting their efficiencies. Thus, the controllable cache resource allocation method is proposed in chapter 3 for better cache tuning efficiency.

To maintain cache coherence, many recent pieces of research aim to improve coherence mechanisms for the purpose of high scalability, low energy overhead, and high performance. The non-uniform cache architecture (NUCA) [66] allows fast accesses to private or partitioned cache units and cache line migratory is supported in such architecture. Hence, research in [67] proposed a prediction-based coherence communication mechanism to improve shared data communication. Research in [33] designed the hardware of predictor to trace the data sharing. In recent many-core processors, directory-based coherence mechanisms are adopted due to their high scalability [68]. As the majority of

accesses are private rather than shared ones, sparse directory and coarse vector-based directory are proposed to largely decrease directory items, so as to hardware scales [69]. To identify shared or private data, research in [18] proposed a trace-driven reorganization mechanism for the purpose of filtering out shared data for further focused execution. Research in [70] proposed the hot-region based coherence mechanism to allocate the hot region of a directory to the particular core dynamically.

However, those conventional methods neglected the 'cache to cache' interaction requirements under many-core situation. And their solutions needed complex hardware supports while few improvements can be achieved by modifying coherence maintaining paths only. Thus, inspired by network on a chip (NoC) concept, the stacked 3D cache network method is proposed in chapter 4 for much performance improvement.

1.3 Motivations and Proposals

1.3.1 Motivations in this thesis

In order to improve the efficiency of hierarchical cache, both energy consumption and performance metrics should be considered as follows.

(1) For improving energy consumption, the large-scale LLC acted as the major heat source is naturally under-optimized in hierarchical cache. The key design difficulty is how to allocate large LLC for serving so many cores efficiently. In other words, it is difficult to trace the demands of allocated LLC parts for concurrent threads during runtime, and also apply the demands to allocating process dynamically. Thus, one of key motivations in this thesis is to do dynamical shared cache allocation for low power consumption. Ideally, measures to allocate more resources to desired threads, to reduce surplus resources when threads are under idle state, and to set spare resources under low power mode can save many LLC energy consumption.

(2) For improving performance, complex and costly coherence maintenance has great potential to do optimizations. In traditional 'Processor to Processor'

data interaction method, some data sharing related cache accesses can cause dozens of time access latency for updating new data to all old data copies or borrowing target data across entire hierarchical cache. Thus, the other key motivation in this thesis is to build a shortcut coherence maintaining path for actualizing fast 'Cache to Cache' data interaction, thereby improving system performance greatly.

1.3.2 Proposals and contributions

In this thesis, aiming at LLC access issues including resource allocation, partitioning, and access pattern optimization, novel hardware-based mechanisms are proposed to allocate LLC resources efficiently and further extend from chip multiprocessor systems to multi-layer stacked systems. In order to make full usage of LLC banks, a dynamic allocation method that is integrated with the discrete control theory is proposed to control runtime cache bank allocation among novel and refined control intervals. To look through state of the art in LLC allocation techniques, this is the first hardware-based achievement which is integrated with discrete Proportion-Integration-Differentiation controllers on runtime LLC allocation. To sum up, the key concept of this controllable LLC allocation mechanism is to tune bank supply of shared cache to each thread's demand by using online PID control in intervals of selected subroutine calls. And contributions of such design are described as follows.

(1) A novel evaluative criterion on allocating LLC resource is proposed based on the combined metric which counts in both hit rate tendency and energy per access tendency along with cache parameter varying, and then energy consumptions can form a concave curve, which is applicable in feedback based discrete control. And the searching complexity of combination exploring can be greatly reduced by focusing on the extreme points of each curve.

(2) The relationship of application locality and influenced dynamic instruction flow period is discovered to establish the behave-stable control periods. Through sampling the runtime flow on frequently used subroutines, the

relationship can be described that repeated calls on the same subroutine behave in an approximately stable locality and calls shifting on other subroutines may lead to locality change as the internal architectures of subroutines have various characteristics.

(3) A feedback based allocation mechanism is firstly implemented into the LLC for the purpose of dynamic runtime combination control. Based on dynamic energy and latency counting, all discretized results are designed to connect with LLC resource allocation, thereby the difference of energy or latency change can be employed as the feedback value of a controlled variable for further reflexing to the difference of allocated LLC resource. Moreover, an off-line sampling method is designed to explore each control parameter set corresponding to each control interval, and then the controllable mechanism is implemented to dynamically allocate LLC resources for the purpose of quite a number of energy-saving.

To further improve performance of hierarchical cache, the allocation mechanism should consider the issue of reusing previous hits in the granularity of per access. Moreover, as the throughput capacity of hierarchical cache is imperative in the many-core system, information interaction among partitioned LLC parts should be ensured to transport vast amounts of data in the same parallel layer and upper/lower layers. Thus, aiming at relieving the interconnection traffic to LLC and optimize the data stream of coherence maintenance requests, a filter-based router network is proposed to integrate into multi-layer three-dimensional stacked system structure for the purpose of filtering large amounts of data shared based accesses, and further interact shared data in virtue of routing network. To sum up, the key concept of proposed cache network design is to build shortcut path on data sharing accesses among private caches by using a dedicated network for detecting and routing shared data in advance. And such cache network design contains innovative contributions as follows.

(1) A novel classification method is proposed to make differences among

cache accesses including repeated accesses, shared accesses and crossed accesses, and further those accesses are classified into read situation and write situation, counted together, six kinds of access types are marked to represent particular requirements, thereby each access type is endowed with enhanced and efficient access path for latency reduction.

(2) The proposed router architecture is firstly integrated with three functional modules to support target access capturing on characteristic accesses, maintaining on shared data coherence and access transporting among entire network, and then many accesses which are existing in other partitioned banks or have stored in partitioned banks can be directly handled in virtue of enhanced router network. Furthermore, a great deal of data sharing based accesses can be maintained by enhanced coherence logic in the router network rather than in the LLC coherence controller, hence access latency of proposed cache network design shows much more reduction over conventional LLC access pipeline.

(3) The proposed router architecture is implemented in IC Compiler to represent the hardware overhead including energy consumption, integration area and access path latency. Meanwhile, the router network is firstly integrated into the first layer for improving access throughput traffic and access latency.

Chapter 2

ACCESS PATTERN ANALYSIS

2.1 Overview on Cache Access Patterns Analysis

Access pattern in this thesis is defined as the cache access set, which represents specific characteristics under certain evaluation matrixes (i.e., access latency, energy, hit rate). And one set can be classified in different profiling granularities, for example, fixed number of dynamic instructions acts as one profiling interval. Thus, application features of SPEC-cpu-2006 benchmark suite [16] and PARSEC benchmark suite [45] can be profiled to show internal locality features in detail. In order to study the features of those benchmarks, seven kinds of target features are analyzed including access reusing distance features, access locality features, parallel processing features on shared data [54], hierarchy traffic features, access distribution features, access hit features and comprehensive evaluation features. All those features are highly related to cache resource demand of particular application and particular clock period, and the final target is to dynamically match the resource allocation with resource demand in the granularity of per-application, per-interval, and further per-access. Moreover, the tendencies of overall energy consumption, up-down interconnecting traffic, and throughput performance are precisely explored along with allocated resource scale increasing, consequently, the much more reasonable searching algorithm on the optimal LLC combinations can be

proposed with far less searching space and far improved fast searching speed which is suitable for implementing on dynamic LLC allocation.

In this chapter, firstly, Ch. 2.2 describes the pre-experiment setups. Ch. 2.3 and 2.4 discuss cache access patterns to explore the hierarchical cache optimization methods proposed in Ch. 3 and Ch. 4, respectively. That is, based on those access pattern analysis results, the optimization method for cache bank allocation at shared-level cache and the solutions for cache coherence problem in private/shared-levels are described.

2.2 Pre-experiment Setups

In order to represent above-mentioned seven features clearly, each feature type will be tested in particular system setups, including a single-core-multi-cache level setup, multi-core-shared hierarchical cache setup. The simulator platform employs the Gem5 [53] simulator for a trace-driven detailed and precise module on-chip multi-processors and an event-driven Ruby module on hierarchical cache which consists of cache levels, interconnecting networks, and off-chip memory. Moreover, a cache power and area scale modeling tool CACTIv6.5 [55] is integrated into the test platform for replenishing hierarchical cache simulation, which supports energy, latency and area simulations on hierarchical cache. And this tool can support to simulate static non-uniform cache architecture (SNUCA) [38] which is applied on cache allocation proposal, and also supports to simulate dynamic non-uniform cache architecture (DNUCA) [15] which is applied on cache process optimization proposal. As to instruction flow analysis, a customized application analysis tool named PIN [17] is employed for detailed monitoring runtime status of hierarchical cache accesses. Thus, seven target features are desired to represent as follows.

(1) Access reusing distance features: Use large enough record directory to count the time period between the first access and the next hit on the same data which is stored at a particular cache level. Hence, distance feature is the crucial

adjective behavior on allocating cache entry number, which should cover the majority of access distances in the purpose of access reusing.

(2) Access locality feature: The locality feature is expected to show the runtime change tendency of dynamic instruction flow, and further make a thorough inquiry on when there is a steady period or a locality-changed period and what factors are related to locality steady or changed. Thus, such a feature can help to design suitable allocating intervals.

(3) Parallel processing features on shared data: In the LLC, the on-chip system benefits from highly-parallel arithmetic, while the percentage of data sharing acts as a key factor in system efficiency. Furthermore, copies of shared data across the entire LLC will lead to serious coherence problems. Thus, those features are desired in designing the degree of parallel operation and maintaining LLC coherence.

(4) Hierarchy traffic features: The access traffic from upper to lower is regarded as the access demand which requests hardware support with plenty of access entries. Moreover, traffic of a particular hierarchy level is fluctuant in dynamic instruction flow corresponding to each application or even each interval, which acts as the working task set on LLC resource allocation.

(5) Access distribution features: To analyze all LLC accesses in the granularity of per access, the distribution of each access is counted together for representing the target locations in LLC, which can be employed to allocate suitable resources and help to modify or enhance the conventional access path.

(6) Access hit features: The hit rate along with allocated LLC resource increasing will vary to form a particular curve corresponding to each application, and such feature acts as one criterion on the allocated amounts of LLC resource. Moreover, hit distribution in small granularity (i.e., per interval) is symbolized to indicate the access traffic and locality status precisely.

(7) Comprehensive evaluation features: Instead of counting on typical hierarchical cache, the energy consumptions of top-bottom system are firstly calculated into the pattern of overall energy consumption per access, which is

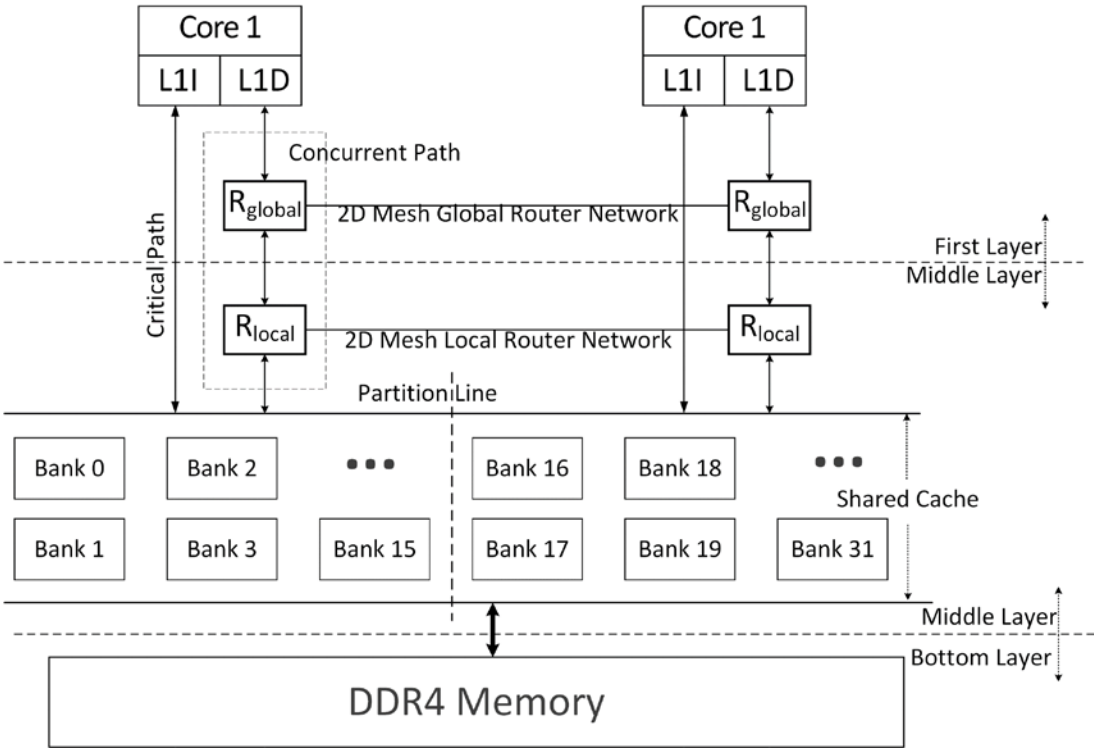


Figure 2-1. Proposed architecture example. Each 3D on-chip component is stacked into different layers, where private core units and global routers are stacked into the first layer, local router and shared cache are stacked into the middle layer, and the DDR4 memory is stacked into the bottom layer. The shared cache is partitioned into bank groups.

benefited to represent the tendency of energy consumption along with allocated LLC resource varying. And further such pattern is adaptive in playing the part of controlled quantity.

2.2.1 Simulation Platform Architecture

Based on the key purpose of each pre-experimental analysis, two kinds of simulation platforms which can be combined with reconfigurable combinations of processors and cache hierarchies are designed to adapt to diverse simulation targets.

(1) Single-processor-multi-level-cache based platform (SP-MLC).

To explore the features of a particular application, the SP-MLC platform shows high veracity and simplicity on revealing typical features in a particular

application [75]. For example, each cache resource combination can be simulated in this platform for filtering interference from other threads. Hence, this platform is designed with same setups just as the ones of multi-processor platform including both processor setups and hierarchical cache setups.

(2) Multi-processor-shared-cache based platform (MP-SC).

As shown in Fig. 2-1, a stacked 3D on-chip architecture is shown as an example to represent the multi-processor-shared-cache based platform. In the MP-SC platform, each private processor unit links to one router, and then each router is interconnected with each partitioned shared cache bank group through TSVs. And the partition lines averagely allocate shared cache banks to each processing thread, where each bank group is one-to-one correspondence with each router, and each group stretches out to the other router network, and further links to the off-chip memory in the bottom layer.

For feature profiling purpose, the SP-MLC architecture is employed to avoid access disturbances from other parallel threads. But for data sharing experiments, the MP-SC architecture is employed to study distributing features of shared accesses. As shown in Table 2-1, the setup details of the SP-MLC and MP-SC platforms are listed. To reduce the number of variable LLC combinations and simplify exploration workload, each processor only contains a single thread and hierarchical cache is designed to simulate level one cache as the private cache and level two cache as the shared cache, and all parameters of

Table 2-1. System Configuration list.

Component	Configurations
Processor Unit	2.0 GHz, 2 cores, single thread per core, 1.1 V supply voltage, 128 IW entries, 32 nm technology library, 30 cycle TLB miss latency.
L1I/L1D Cache	32 KB instruction cache, 32 KB data cache for a core (private), 4-way, 64 B line size, 2 cycle latency.
Shared L2 Cache	2 MB total size S/D-NUCA, 32 banks, 1 MB partitioned for a core (shared), 16-way, 128 B line size, 20 cycle latency.
Main Memory	1 GB Double Data Tate (DDR4 2133MHz, 1.2V), 8 KB page size, 120 cycle latency.

both processors and caches are contrivable as the ones of the commercial mainstream chips.

2.2.2 Performance and Energy Parameter Modeling

According to the process of executing a cache access request, the overall energy consumption is calculated as static and dynamic power of hierarchical cache, routers, crossbars, and off-chip memory during executing period of some numbers of access requests ($N_{request}$) [74]. The period is defined as clock cycles from the clock cycle of the first cache access request arriving at the clock cycle of the last cache access result returned. As shown in Eq. 2-1, the average value named energy consumption per access (E_{access}) can be calculated by using overall energy to divide request number. Similarly, the performance of all tests are normalized as the average instruction per clock (IPC, stands for dynamic instruction numbers $N_{dyn\text{-}ins}$ divided overall cycle numbers N_{cycle}), while cycle numbers are counted in the period from the clock cycle of first cache access request arriving at the clock cycle of last cache access result returned. Furthermore, the IPC in the MP-SC platform represents that all dynamic instructions executed in all processors are dividing by the unified clock cycles of the platform rather than cycles of all processors.

$$E_{access} = \{\sum E_{cache} + \sum E_{router} + \sum E_{crossbar} + \sum E_{off-chip-momory}\} / N_{request} \quad (2\text{-}1)$$

$$IPC = N_{dyn\text{-}ins} / N_{cycle} \quad (2\text{-}2)$$

Where E_{cache}, E_{router}, $E_{crossbar}$ and $E_{off\text{-}chip\text{-}memory}$ stand for the average energy of cache, router, crossbar and off-chip memory during a cache access, respectively. Based on the detailed cache modeling tool CACTIv6.5 [55], the static and dynamic power of hierarchical cache are accumulated with selected cache components including tag array, data array, output logic, and pre-charge

logic. And further counting in the of-chip memory accesses, the access latency of one cache request ($L_{request}$) can be calculated by Eq. 2-3.

$$L_{request} = R_{hit} * L_{access\_hit} + R_{miss} * L_{access\_miss} \quad (2\text{-}3)$$

Where R_{hit} and R_{miss} represent the hit rate and miss rate in current cache, and L_{access_hit} and L_{access_miss} stand for the latency of cache access hit and miss, respectively, which can be represented as follows.

$$L_{access\_hit} = \max(L_{data-array}, L_{tag-array}) + L_{charge} + L_{output} \quad (2\text{-}4)$$

$$L_{access\_miss} = L_{tag-array} + L_{output} + L_{off-chip-memory} \quad (2\text{-}5)$$

Where all access latency values on cache components and off-chip memory are set by cache model analysis which counts the access path latency based on transistor level integration with the aid of CACTI tool, in like manner, both static power and dynamic power of all cache components and off-chip memory can be accumulated on their integrated and activated transistors. Hence, the E_{cache} of any cache level is defined as Eq. 2-6.

$$E_{cache} = R_{hit} * L_{access-hit} * \left\{ \sum P_{cache-static} + \sum \alpha * P_{cache-dynamic} \right\} + R_{miss} * L_{access-miss} * \{ \sum P_{cache-miss} + \sum \beta * P_{cache-dynamic} \} \quad (2\text{-}6)$$

$$E_{off-chip-memory} = R_{miss} * L_{access-miss} * \{ \sum P_{memory-static} + \sum \gamma * P_{memory-dynamic} \} \quad (2\text{-}7)$$

Where P stands for the corresponding power, and α, β and γ ($0 \leqslant \alpha, \beta, \gamma \leqslant 1$) represent the rate sets of cache resource activation in the access path.

Moreover, the energy consumption in crossbars can be directly contained in the simulator. The static power ($P_{router-static}$) and dynamic power ($P_{router-dynamic}$) of proposed router architecture are generated by IC Compiler (details see Chapter 5), and the energy consumption of a router is described as Eq. 2-8.

$$E_{router} = L_{request} * \{ P_{router-static} + \delta * P_{router-dynamic} \} \quad (2\text{-}8)$$

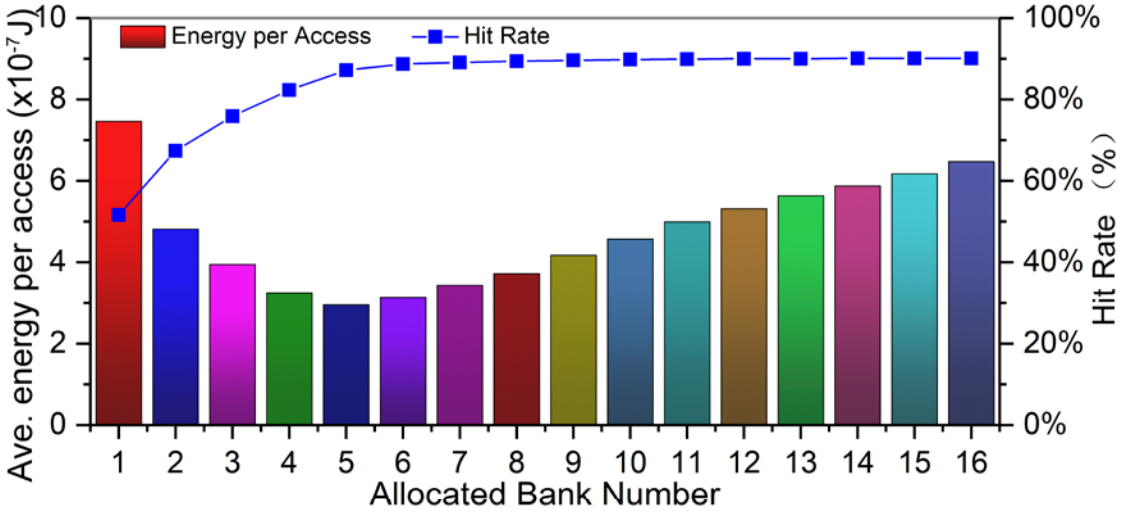


Figure 2-2. Energy sampling with different cache banks. All tests are simulated on *403.gcc* benchmark, where operational cache bank number is sampled from one to sixteen.

Where δ $(0 \leqslant \delta \leqslant 1)$ stands for hardware activation rate in the router.

2.3 Analysis on Hierarchical Cache Demands

2.3.1 Cache Resource Demand

To analyze the resource demand corresponding to a particular workload, two key parameters are employed for representing demand change along with allocated cache bank varying. The first parameter is the hit rate, and the other one is the average energy consumption which is calculated in the granularity of each LLC access. All tests are implemented in the SP-MLC platform.

As shown in Fig. 2-2, the hit rate of *gcc* benchmark displays a rising curve along with allocated cache bank increasing [73]. With one bank allocated, hit rate is about 51%, which means that almost half of all LLC accesses will be access miss and further off-chip memory accesses are required with far longer access latency than access hit. Correspondingly, energy consumption tends to be very large as those LLC accesses are last for a long time in handling access misses and a large hardware scale of off-chip memory is activated to support miss restoration. Hence, cache resource demand is not satisfied with the

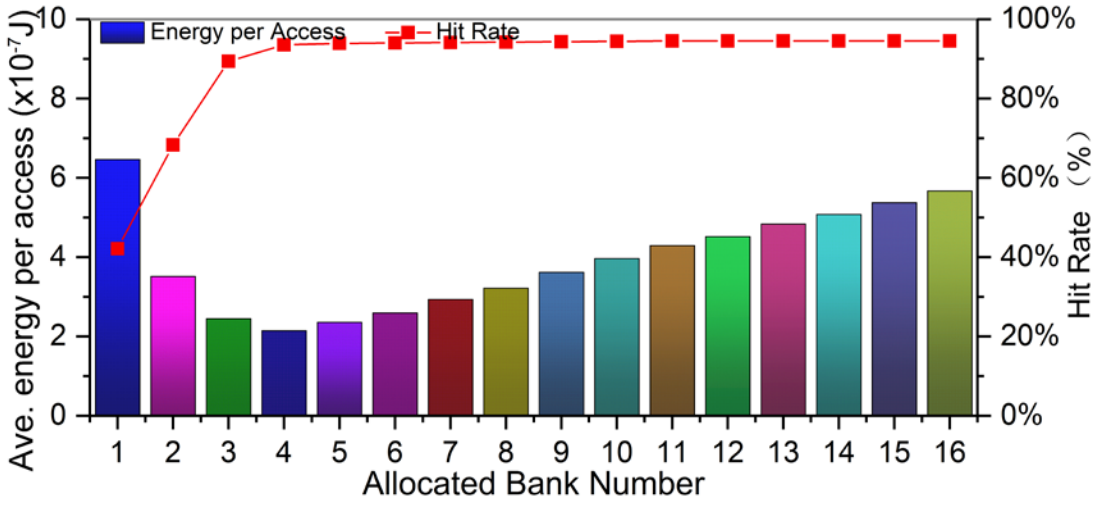


Figure 2-3. Energy sampling on *401.bzip2* benchmark. Operational cache bank number is sampled from one to sixteen.

allocation of one bank, which will result in low hit rate and high energy consumption. If one more bank is allocated, hit rate can be greatly improved by about 16%, and such improvement on hit rate continues along with allocated bank number increasing. However, the amplification of such improvement tends to be decreasing progressively, where there are very few improvements in the hit rate achieved after allocating more banks to the 6-bank situation. And the new allocated banks in this situation contribute a bit of hit rate improvement, on the contrary, their hardware scale wastes some energy, resulting in an increment of energy per access value. Typically, there is the lowest value of energy per access in the situation of allocating five banks, where the hardware overhead of all allocated banks seems to generate the resultant force with energy consumption saving due to hit rate improvement, thereby the rise and fall in energy consumption are balanced at the energy lowest trade-off situation.

In a similar manifestation, the hit rate of *bzip2* benchmark form a trend curve that improvement in allocating first several banks grows rapidly and then remains stable along with allocated bank number increasing. As shown in Fig. 2-3, the hit rate in allocating one bank situation is only about 42%, however, hit rate is greatly improved by 89% as there are two more banks allocated. In other

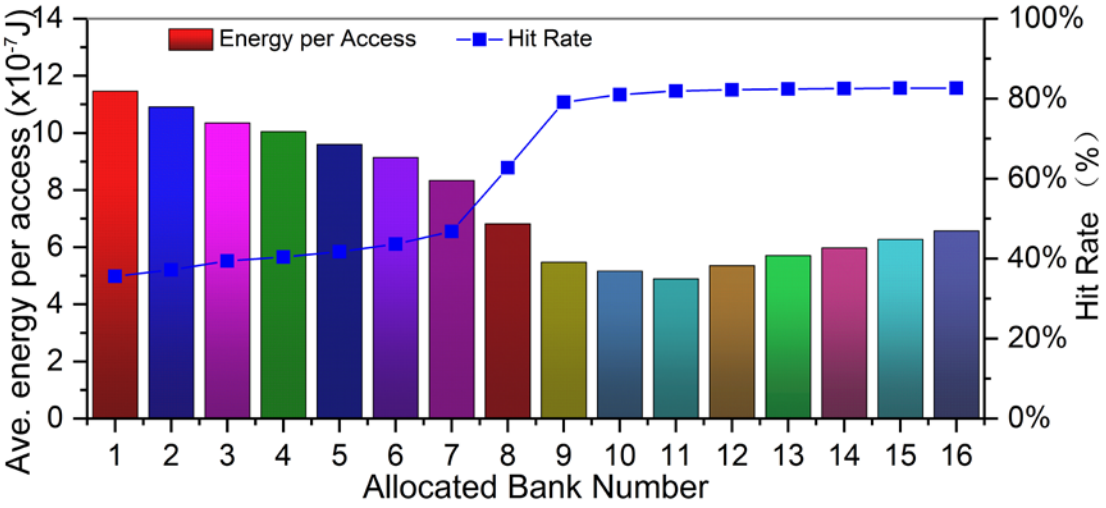


Figure 2-4. Energy sampling on *450.soplex* benchmark. Operational cache bank number is sampled from one to sixteen.

words, hit rate curve will become stable rapidly along with allocated bank number increasing, and allocating more banks just wastes some cache resource while energy consumption is increased due to that activated hardware.

Moreover, there are several benchmarks that show quite different characteristics on the value varying tendency of hit rate and energy consumption. For example, the hit rate of *soplex* benchmark will be improved very few in allocating first several banks until allocated bank number exceeds a threshold, while exceeding such threshold can bring extra hit rate improvement and then hit rate remains stable again at a much larger value. As shown in Fig. 2-4, hit rate in allocating the first seven banks is increasing gently, meanwhile, allocating two extra banks will bring about more than 25% hit rate improvement. And then, hit rate remains stable even with extra banks allocated. Relatively, the lowest energy consumption will be screened out at the situation of just exceeding the threshold.

2.3.2 Discussions on Demand Variation

To sum up, all benchmarks can be classified into three patterns while each pattern represents a particular amplitude of improvement variation along with allocated bank increasing. As shown in three examples, *gcc* benchmark behaves

a little insensitivity with bank allocation, *bzip2* benchmark is sensitive with extra bank allocation and access pattern on *soplex* benchmark shows a phase-step based curve characteristic. As a result, both hit rate and energy consumptions are highly related on cache resource allocation, and two appearances can be observed as follows.

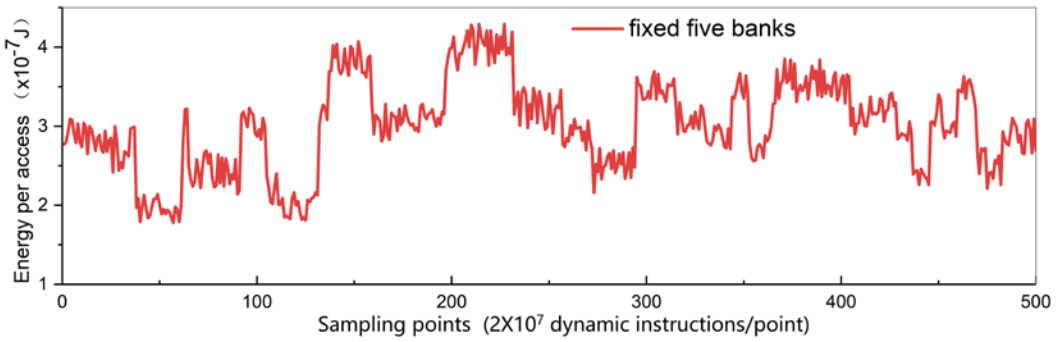
(1) There exists an optimal bank allocation point for each benchmark from the viewpoint of energy consumption, that is, energy consumption tends to be the lowest one while the hit rate tends to be remaining stable at the approximate maximum value.

(2) As to allocating bank number from one to the optimal number incrementally, hit rate is increasing rapidly and energy consumptions follow to be decreasing in case of *gcc* and *bzip2* benchmarks while the *soplex* benchmark behaves similar but it ranges from several banks later to the optimal one. Moreover, the hit rate of all three patterns almost remain stable from the optimal bank number to all bank allocated, and energy consumptions are increasing as the hardware overhead scale.

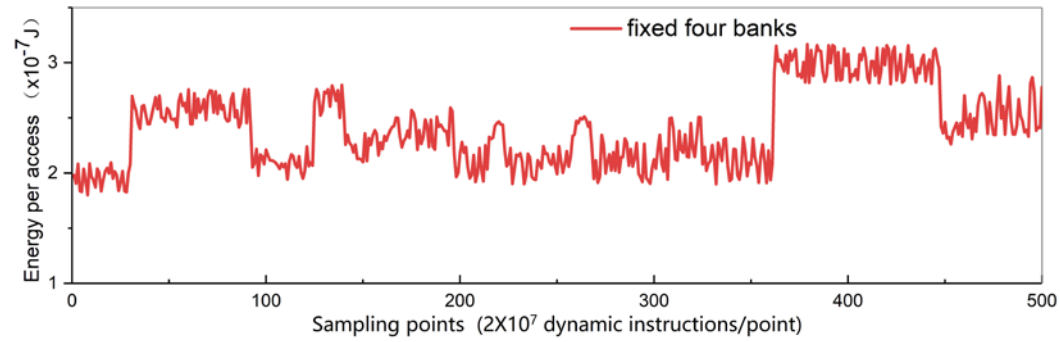
Thus, the cache resource demand can be described as the optimal cache allocation to achieve both low energy consumption and access latency. In other words, more banks are needed in case that allocated bank number is less than the resource demand, and in case of exceeding the resource demand, redundant banks should be removed off the current allocation. Moreover, the profiling curve of energy consumption based on resource allocation shows an analogous concave parabolic, which behaves similar as the reversed feedback-based controller, where output response of controlled parameter will be convergent to the steady-state value and a disturbance can be handled in the negative feedback control path. As to LLC allocation, each disturbance on energy consumption will generate feedback to bank allocator for the purpose of converging to optimal cache allocation, thereby satisfying current cache resource demand.

2.3.3 Access Locality Analysis in Fine Granularity

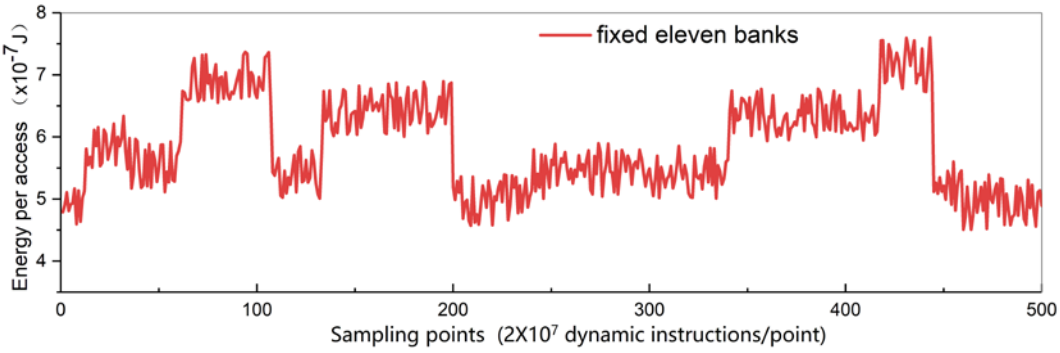
In order to represent locality features more accurately, the successive approximation method is applied for exploring fine granularity for the purpose of LLC allocation. Firstly, entire dynamic instruction flow with some number of bank allocation is treated as one candidate for the optimal cache allocation. Thus, there are sixteen candidates existing in the case of sixteen available LLC banks, and the quasi-optimal cache allocation can be selected by comparing



（a）Energy per access profiling with *gcc* benchmark.



（b）Energy per access profiling with *bzip2* benchmark.



（c）Energy per access profiling with *soplex* benchmark.

Figure 2-5. Energy sampling curves. (a) *403.gcc* benchmark. (b) *401.bzip2* benchmark. (c) *450.soplex* benchmark. Each sampling point in X-axis stands for a set of twenty million dynamic instructions.

energy consumptions of all sixteen candidates. As a result, quasi-optimal bank numbers for *gcc*, *bzip2,* and *soplex* benchmarks are selected as five banks, four banks and eleven banks as known from Figs. 2-2, 2-3 and 2-4, respectively. Secondly, the operating granularity of LLC allocation is further reduced as each small period partitioned on dynamic instruction flow. For example, twenty million dynamic instructions per period are adopted. As shown in Fig. 2-5, five hundred sampling periods are selected to calculate the energy consumptions in each period, and then all energy values together form the tendency curves of energy consumption per access values while a fixed number of quasi-optimal bank allocation is allocated in the platform corresponding to each benchmark. Thus, sampled curves can show the locality change features along with executed dynamic instruction scale increasing.

As shown in Fig. 2-5, energy values of sampling periods vary acutely with a large variation range. Two key appearances are emphasized as follows.

(1) Energy consumption changes sharply in some periods. For example, the energy consumption in the 200-th period around varies roughly from three to four as shown in Fig. 2-5(a). And such variation must be caused by locality change, as a result, allocated bank number may be unsuitable for the current period.

(2) Energy consumption may have an approximately stable locality in some periods and those energy consumptions of sampling periods tend to be little difference. For example in Fig. 2-5(a), the energy values between the 210-th period and 230-th period around are almost same. In those periods, locality change seems to be very slight, and also energy consumptions remain stable.

To further analyze why locality behaves alike with those appearances, the application architecture should be fully considered as three key reasons as follows.

(1) The proportion of cache-access related instructions counted from the entire instruction set acts as a key influence factor on the throughput of dynamic

Table 2-2. Instruction sampling in statistical distribution.

Benchmark	Load Ins. (%)	Store Ins. (%)	Benchmark	Load Ins. (%)	Store Ins. (%)
400.perlbench	27.1	15.1	453.povray	35.1	14.5
401.bzip2	35.8	12.7	454.calculix	37.6	11.2
403.gcc	27.2	14.8	456.hmmer	41.3	14.2
410.bwaves	48.1	7.6	458.sjeng	28.5	14.9
416.gamess	40.9	12.6	459.GemsFDTD	51.6	9.1
429.mcf	36.2	11.8	462.libquantum	33.2	11.5
433.milc	37.5	12.5	464.h264ref	40.7	13.8
434.zeusmp	34.8	12.3	465.tonto	43.2	12.5
435.gromacs	26.9	16.7	470.lbm	35.4	10.2
436.cactusADM	45.3	13.1	471.omnetpp	34.2	15.5
437.leslie3d	42.6	10.5	473.astar	37.7	14.3
444.namd	36.2	9.1	481.wrf	43.9	9.6
445.gobmk	29.5	14.0	482.sphinx3	34.7	6.1
447.dealII	40.7	12.5	483.xalancbmk	35.5	9.7
450.soplex	38.5	7.9			

instruction flow. If all instructions are classified into cache-related instructions and cache-irrelevant instructions, cache-related instructions typically in load and store instructions will occupy the majority of execution time and also consume the majority of energy debit, even the proportion of those instructions usually takes less than half of total instructions. For example, a load instruction will be executed within several cycles if such access encounters a cache hit, however, if such access encounters a cache miss, dozens of cycles or even hundreds of cycles are needed to conduct off-chip memory access. Although access latency of this access can be partly hidden in the pipelined execution path, such latency is far larger than the latency of executing a cache-irrelevant instruction.

To achieve the distribution of dynamic instruction statistics, three counters are set in the platform to record the total number of dynamic instructions, load instructions, and store instructions. Thus, the distribution of load and store instructions are defined by the ratios of load or store instructions, which are described as Eq. 2-9.

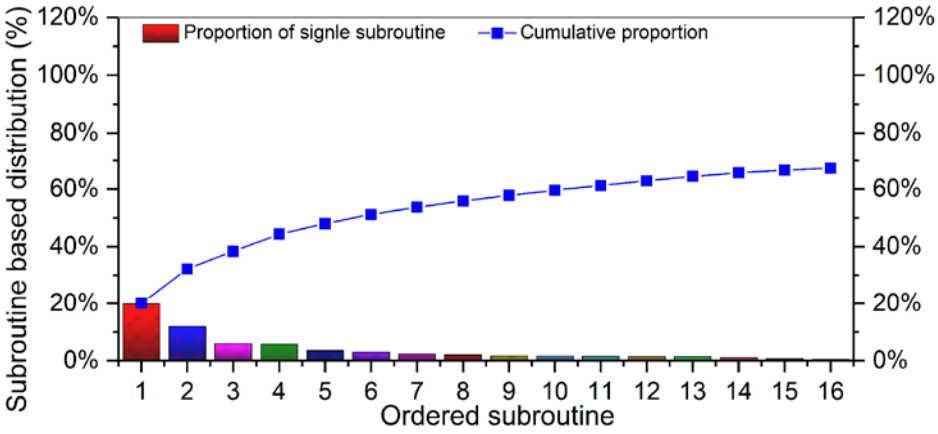
$$C_{load\ ins.} = \frac{\sum counted\ load\ instructions}{\sum all\ dynamic\ instructions} \tag{2-9}$$

$$C_{store\ ins.} = \frac{\sum counted\ store\ instructions}{\sum all\ dynamic\ instructions}$$

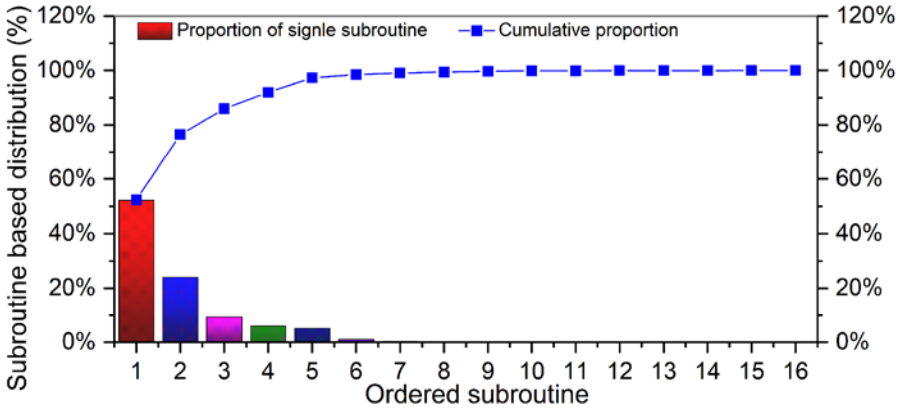
During each statistic, dynamic instructions are counted in the SP-MLC platform. As shown in Table 2-2, all the benchmarks from the SPEC benchmark suite are analyzed, and the average values of the load instruction and store instruction distributions are 37.24% and 12.08%, respectively. As it is almost impossible to trace the lifetime of counting load and store instructions in the pipelined instruction execution sequence, the possible lifetimes of single load and store instruction executing paths are analyzed to represent their latency impacts. For example, the lifetime of a write hit on LLC may hold twenty cycles, the lifetime of a write miss on LLC may hold two hundred cycles, however, the lifetime of a cache-irrelevant instruction is usually handled in a single cycle.

(2) The reuse distance distribution of instruction architecture will affect the access time in a decisive manner. Typically in reusing previous cache-related data, if allocated LLC banks have enough entries to cover up the reuse distance, few off-chip memory-based accesses are generated, otherwise, hundreds of cycles will be consumed to handle a single LLC miss. Thus, it is crucial to allocate sufficient LLC entries for the purpose of reusing the majority of previous data.

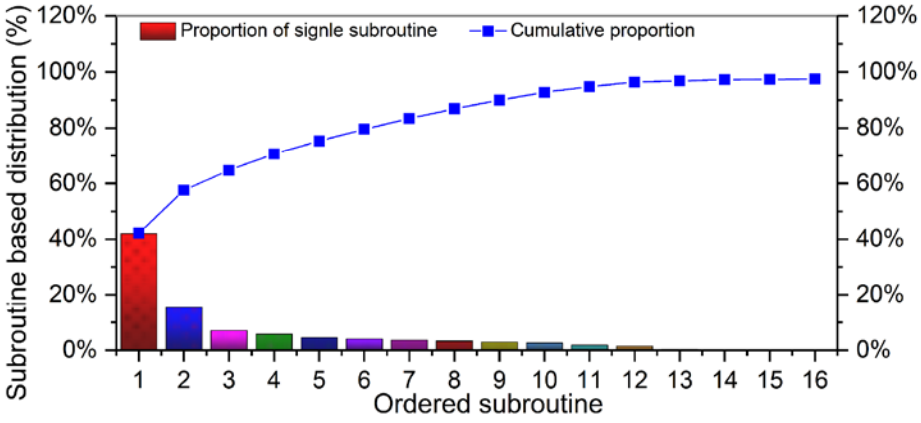
(3) Some kinds of instruction clusters are repeatedly executed in an execution period, and then such a period shows stable locality in both energy consumption and performance. For example, if there is a length of static instructions which is executed hundreds of times to form an execution period, the energy consumption and performance values in this period will be stable as each executed time has a similar or even same dynamic instruction sequence. Note that each call on same subroutine may have a highly similar dynamic instruction sequence and also executing sequence, thereby their cache resource demands are similar and so as to the energy consumption and performance. Thus, subroutines of a benchmark are expected to act as more suitable allocation intervals rather than intervals based on equalizing the dynamic instruction sequence.



(a) Subroutine proportion of *gcc* benchmark



(b) Subroutine proportion of *bzip2* benchmark



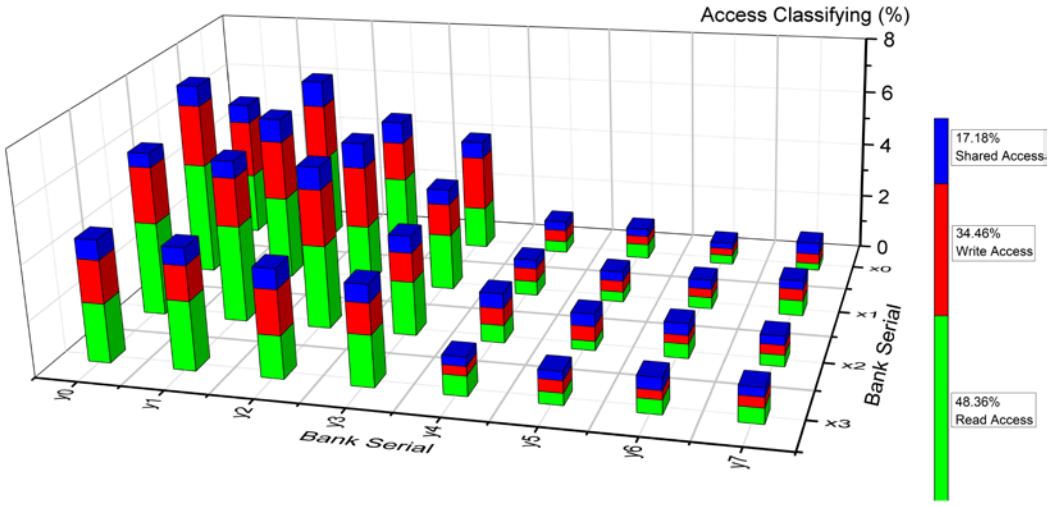
(c) Subroutine proportion of *soplex* benchmark

Figure 2-6. Sampling curves of selected subroutine distribution. (a) *403.gcc* benchmark. (b) *401.bzip2* benchmark. (c) *450.soplex* benchmark.

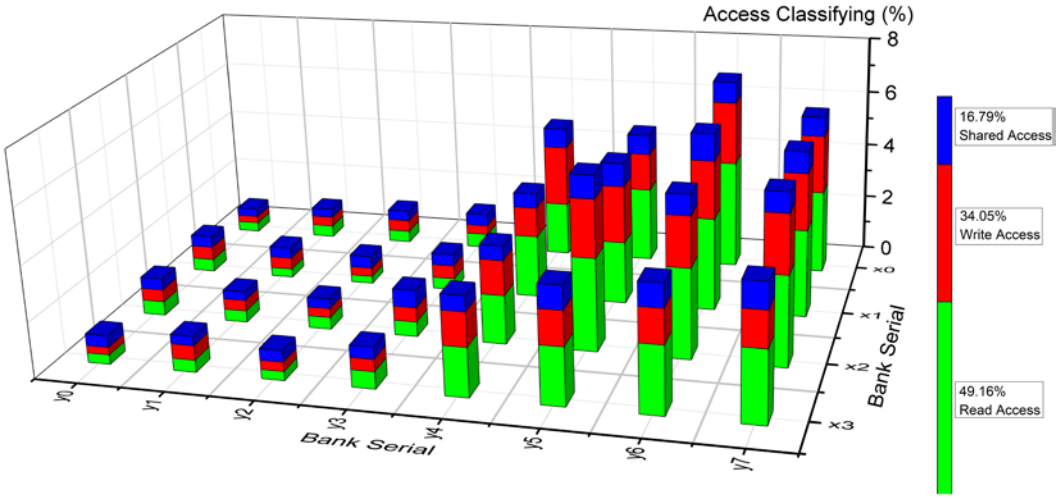
To apply subroutine based intervals in LLC allocation, the proportion of dynamic instructions in subroutine calls over all dynamic instructions needs to be explored because that ideal allocation interval should cover up the majority of dynamic instruction sequences. Thus, two markers are assigned in the front

and back positions of each subroutine for the purpose of tracing every calls and further doing statistics on the proportion of dynamic instructions scale. With the help of the PIN tool [17], the first sixteen subroutines are selected to reveal their proportion over the overall dynamic instruction scale, while those proportions counting together will account for most of the entire dynamic instruction flow. As shown in Fig. 2-6, dynamic instructions counting in sixteen most-frequent-invoked subroutines from *gcc*, *bzip2* and *soplex* benchmarks can take the proportion of total dynamic instructions in the percentage of 67.3, 99.9 and 97.5, respectively. Moreover, it can be observed that those selected subroutines are invoked by many times and the top several subroutines can nearly form the particular locality features in the current benchmark. Thus, calls on those so-called hot subroutines are naturally suitable for acting as cache allocation intervals rather than equal-division-based intervals.

To sum up, cache resource demands of any application are analyzed from different angles. Firstly, in the granularity of per-application based simulations, the pre-experimental results show that each application behaves with a particular locality feature and further has particular cache resource demand. And there is a tradeoff on allocating cache resources for such demand, in which allocating more cache resources over its demand leads to energy waste and allocating fewer cache resources over its demand leads to many access misses. Secondly, in the granularity of per-interval based simulations, locality within a dynamic instruction flow varies acutely, where locality change points separate locality stable periods into different energy consumptions. Thus, those locality features infer that each selected subroutine has the optimal cache resource demand and further cache allocation method can refine in the granularity of each subroutine call. Moreover, the statistical results of selected subroutine calls show that the majority of dynamic instruction flow can be covered up in those subroutine calls.



(a) Access distributions on requests from Core0.



(b) Access distributions on requests from Core1.

Figure 2-7. Distributions of shared accesses. The test platform consists of two processing cores and thirty-two shared cache banks, where banks in y0, y1, y2 and y3 axes are allocated to Core0 and rest banks are allocated to Core1.

2.4 Analysis on Shared Cache Access Pattern

2.4.1 Overview on Shared Cache Access Pattern

To further analyze the shared cache access distributions in granularity of each instruction, we employ one benchmark from PARSEC benchmark suite (*dedup*) as an example, and this benchmark is scheduled to a test platform which has two

cores in the first layer, thirty-two DNUCA banks in the second layer (interconnected with 2D mesh routers) and two TSVs for linking each core to their partitioned banks respectively (other details see Section 4). Then, all shared cache accesses from two processing cores to all the banks are counted, and those accesses are classified into their destination banks, thereby those accesses from each core can be drawn into frequency-based access distributions on each bank. Moreover, each access is classified into several types based on where the access request comes from and which banks will be the destination of this access request. And further, each access distribution is considered that its effect degree of current access type affects the energy consumption and performance of this application.

2.4.2 Results of Access Pattern Distribution

As shown in Fig. 2-7, shared accesses are spreading on banks with proportions from 0.27% (Bank[x0,y6]) to 1.03% (Bank[x1,y2]), while the average shared accesses take about 19.76 % of all accesses. Meanwhile, read access proportions range from 0.29% (Bank[x0,y7]) to 4.62% (Bank[x1,y0]), while all read accesses take about 45.96% of total accesses. As to write accesses, the proportions range from 0.67% (Bank[x0,y6]) to 2.86% (Bank[x1,y0]), while all write accesses take average 34.28% of total accesses. Corresponding with three issues, we can confirm: 1) Almost 19.76% of accesses may potentially overlap their access duration which may cause a long pause duration. 2) From bank serial y4 to y7, about 21.53% of total accesses hit at partitioned banks of Core 1, mean that those accesses will cause long interconnecting latency. 3) Read accesses are dispersed at fringe banks (i.e., Bank[x3,y2]), resulting in more routing traffic. And even more serious, continuous writes may occur frequently in Bank[x0,y3] because write accesses even take a half proportion of total accesses in this bank, result that many cycles are wasted on dealing with those write congestions.

The above appearances of shared cache access distributions indicate that many potential improvements on both access latency and throughput can be

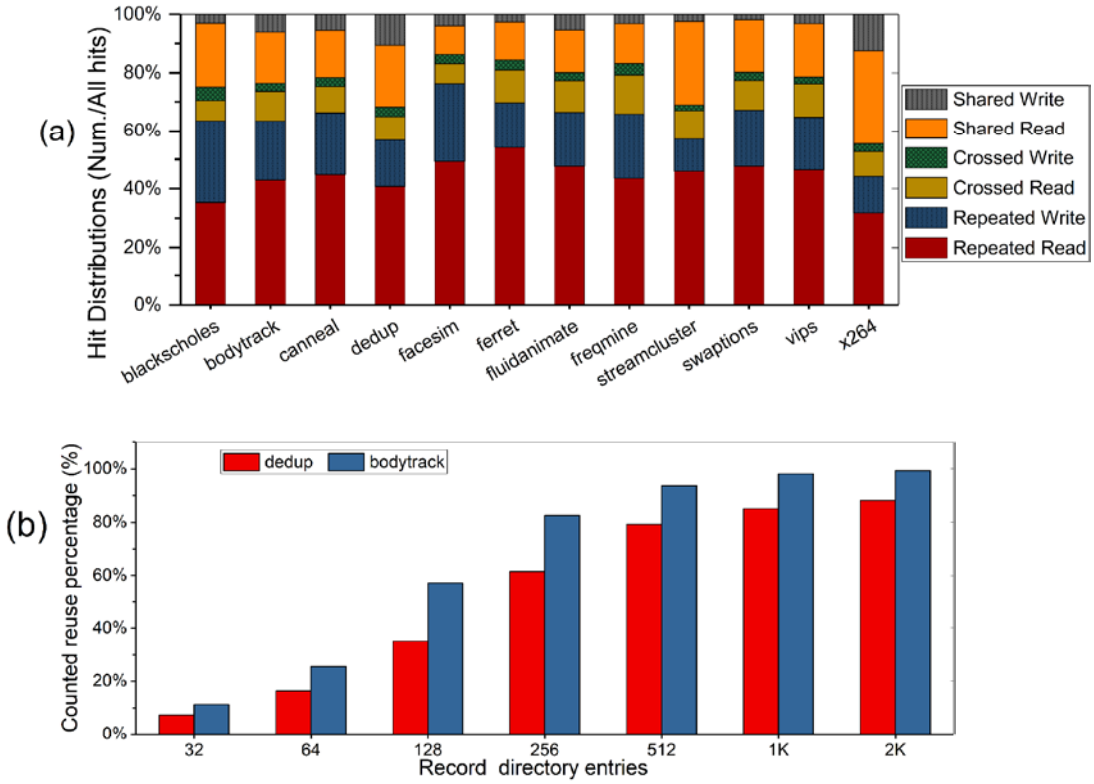


Figure 2-8. Access pattern distributions. (a) Access hit distributions of each access pattern with different benchmarks, where miss accesses are ignored; (b) Reuse distance sampling with different record entries allocated, where the recorded contents are replaced with first-in-first-out algorithm.

achieved if some of those accesses can be handled in an efficient access path (access path in this thesis is defined as the processing path from emit an cache access to receive an echo). So that we are motivated to manage all those accesses to each bank for reasonable allocation strategies as follows.

Ideally, shared data should adequately supply for cores, read accesses should be convergent to some key banks, and some write accesses should be dispersed to idle banks for preventing congestions. To actualize those strategies, a novel router network is proposed in Ch. 4 for fast tracing potential harmful accesses on the shared cache and handling them in a concurrent path.

Furthermore, the access distributions are counted in six access types (each access type is divided into read access and write access), which are classified as follows [72].

(1) Shared accesses: Stand that current shared cache access has objective data in both cache bank groups of two processing cores.

(2) Crossed accesses: Stand that current shared cache access only has objective data in the cache bank group of the other processing core.

(3) Repeated accesses: Stand that current shared cache access only has objective data in the cache bank group of current processing core.

Thus, sixteen record counters for a bank group composed of 16 banks are employed to trace every shared cache accesses which are requested from two processing cores. All the benchmarks from PARSEC benchmark suite are simulated to account runtime accesses in a test platform that has two processing cores, sixteen shared cache banks. Fig. 2-8(a) displays that shared write accesses can take the proportion of percentage from the lowest 1.8 (*swaptions*) up to the highest 12.4 (*x264*), while the average value is 5.1 %. The hit distributions of shared read accesses take the proportion of percentage from the lowest 9.7 (*facesim*) up to the highest 32.2 (*x264*), while the average value is 18.7 %. The hit distributions of crossed write accesses take the proportion of percentage from the lowest 2.1 (*streamcluster*) up to the highest 4.7 (*blackscholes*), while the average value is 3.1 %. The hit distributions of crossed read accesses take the proportion of percentage from the lowest 6.8 (*facesim*) up to the highest 13.4 (*freqmine*), while the average value is 9.6 %. The hit distributions of repeated write accesses take the proportion of percentage from the lowest 11.2 (*streamcluster*) up to the highest 28.1 (*blackscholes*), while the average value is 19.2 %. The hit distributions of repeated read accesses take the proportion of percentage from the lowest 31.7 (*x264*) up to the highest 54.8 (*ferret*), while the average value is 44.3 %. Based on the above results, three appearances can be revealed as follows.

(1) Shared accesses take the proportion of 23.8% overall accesses, typically in shared write accesses, which may cause serious access conflictions in shared data.

(2) Crossed accesses take about 12.7% of overall accesses, and those accesses will encounter misses in their partitioned bank group but hit at the other group.

(3) Repeated accesses take about 63.5% of overall accesses, and those accesses will quite frequently hit at the partitioned bank group.

The above appearances indicate that a great deal of improvement on access latency can be achieved if shared accesses can be linked cross bank groups, crossed accesses can get target data from the other bank group and repeated accesses can be filtered into a recording table for data reusing. Thus, the upper-layer router network acts as a suitable operation object to bridge across bank groups, because all data and requests are interconnected by the router network. Meanwhile, shared cache accesses can be relayed in a concurrent access path, and some selected cache accesses are recorded and further handled for efficient bypassing. However, the efficiency of access filtering is highly related to the reuse distance of those accesses, so that counted reuse percentages of each application should be identified for exploring the suitable record entry numbers. Thus, several first-in-first-out replacement policies based record directories are employed to analyze the reuse percentage over all accesses along with record directory entry increasing. As shown in Fig. 2-8(b), only 7.2% of overall accesses can be recorded with 32 record directory entries in case of *dedup* benchmark. With 512 record directory entries, the proportion of reused accesses is 79.3%, and even with doubled entries number, the proportion value only gets a 5.8% improvement. Hence, there is a tradeoff on determining the integrated scale of record entry.

2.4.3 Discussions on Access Pattern Features

Since the access patterns are explored in the granularity of each shared cache access, experimental results show that three different access types represent diverse features on distributions among overall accesses, which are highly related to the hit rate and energy consumption of one application. Firstly, distribution percentages of shared accesses represent the possibility of data

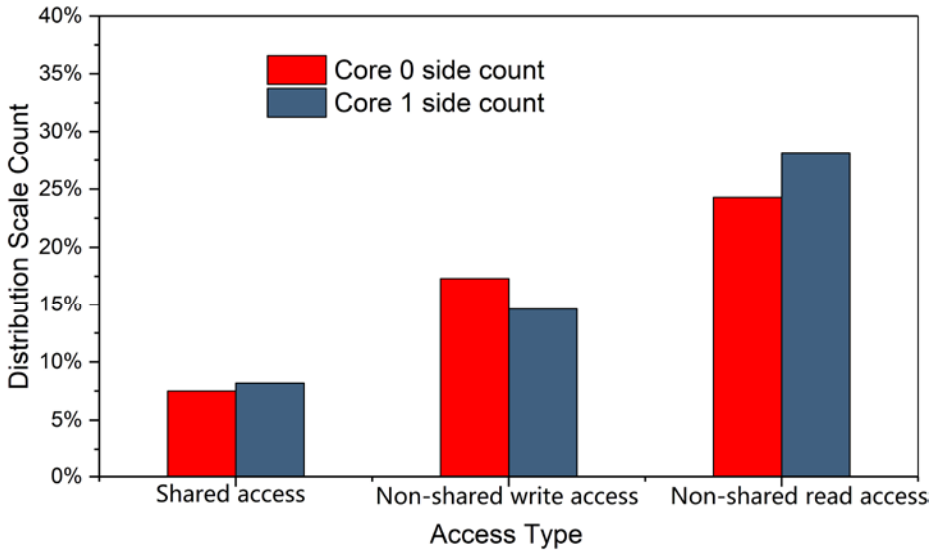


Figure 2-9. Trace statistic of access distributions. All shared cache accesses are classified into each processing core.

inconsistency which requests complex hardware and wastes plenty of clock cycles to maintain data coherence. Secondly, distribution percentages of crossed accesses show the great opportunity on improving access latency, where crossed accesses can employ the previous target data from the other partitioned bank group in the purpose of access hits rather than shared cache misses in current partitioned bank group. In other words, the previous data in other bank groups are desired to deliverer from those groups to the current group, and further return to current processing core with limited latency, otherwise, access misses happen to cost large access latency on an access to off-chip memory. Thirdly, repeated accesses take the largest proportion over other access types, and such proportion value indicates that many shared cache accesses will occur again in limited reuse distance and once the record entry number exceeds the reuse distance, current access can be handled by the record entries very fast rather than shared cache access. Furthermore, continuous writes on same cache block will lead to write congestion if the former write has not finished, however, continuous writes on record entries can be buffered that the second write can temporarily write on the record entries instead of waiting for finishing the first write access.

Moreover, access distributions on each bank are represented to infer the cache resource utilization in the granularity of per access level, while all shared

cache accesses generated by two processing cores are counted with the aid of the PIN tool.

As shown in Fig. 2-9, in case of target data existing in both bank groups, such kind of shared cache accesses are counted in a proportion of 7.6%. In the case of target data existing in its own bank group which means 'non-shared write access', they take the proportion values of 17.3 and 14.9, respectively. And non-shared read accesses take the proportion values of 24.7 and 28.1, respectively. It can be observed that there are a few difference quantities between two processing cores. However, those accesses hit at shared cache banks, and reveal quite diverse distributions among banks. Under the least-recently-used based cache replacement policy, frequently invoked banks tend to record many access hits and some banks are rarely employed to contribute some access hits. As a result, some shared cache banks are redundant and can be re-allocated to other needful threads.

In the view of another angle on access pattern features, each access type including shared, crossed and repeated accesses has different processing pipeline path, and so as to access latency. For example, a repeated read access only needs several clock cycles, but a crossed read access needs to get the target data from off-chip memory, obviously, the access latency of the crossed read access is far larger than the one of the repeated read access. Although some latency can be partly concealed into pipelined execution sequence, the rest parts of access latency will retard the on-chip execution speed greatly. Moreover, some access types such as crossed read and write take the largish proportions of overall accesses, so that the system performance is largely decreased due to resultant effect of those costly shared cache accesses. Reasonably, some instruction clusters show particular energy consumption and performance values, which are highly related to the proportions of those costly accesses.

2.5 Summary on Access Pattern Analysis

In this chapter, access patterns are analyzed in the granularity of per-application based intervals, per-subroutine-call based intervals, and single shared cache access. Based on experimental results, locality features are explored along with cache resource varying, thereby achieving the runtime cache resource demand under the current application. Furthermore, evaluation metric of energy consumption is employed to represent the relationship between energy consumption and allocated cache resource and reveal that there is an approximately optimal cache resource allocation point for each application, while exceeding this point only results in resource wasting and lack of this point will result in many access misses and enlarged access latency. Thus, all benchmarks can be pre-classified for further employment in controllable cache allocation design.

The locality features are also analyzed in a much precise granularity to explain the internal reasons for locality change and stable. Firstly, subroutines of one application are analyzed to describe that dynamic instructions that come from subroutine calls will take the most proportion of overall dynamic instruction scale. And the same subroutine is repeated invoked to form a locality stable period, but different subroutines are invoked severally, which may lead to locality change on the point of subroutine calls converting. Secondly, calls on same subroutine show very similar behavior on energy consumption and performance appearances, so that subroutine calls act as the optimal intervals to dynamically allocate cache resources. Moreover, the concept of balancing between supply and demand on cache resource is analyzed to be integrated into cache resource allocation: if the supply of cache resource is less than demand, allocate more resources, and if the supply is more than demand, then remove the redundant resource. Similarly, such co-ordination of supply and demand acts as the same process of feedback-based control process. Thus, the analyzed pattern features can be fully employed in the next chapter to propose a self-controlled dynamic cache allocation method.

To analyze the access pattern in the minimum granularity of each shared cache access, those accesses are classified into several access pattern types for the purpose of representing proportions among access types and features on the pipelined path of processing those access types. Thus, repeated access type takes the largest proportion of overall accesses and shows two potential improvable points that some hottest data can be filtered out to a fast record router for the purpose of fast request responding and continuous write can be buffered in the record router also. As to crossed accesses, there is a great opportunity to reduce many off-chip memory accesses through bridging the target data to current partitioned bank group. For shared accesses, data coherence issue can be potentially handled in case that all bank groups can be interconnected with the aid of the router network. Thus, those features in access types are desired to be employed for proposing an enhanced router network to improve access latency greatly.

To sum up, access pattern features are employed by two proposal chapters, respectively. Features on hierarchical cache demands represent the energy and hit rate characteristics in per interval granularity, which inspire to conduct controllable cache resource allocation design in chapter 3 by dynamic meeting cache resources to current resource demands, thereby saving some on-chip energy. As to distribution features of access patterns in single access granularity, some number of data sharing accesses need to be handled in complex and costly processing paths, which inspire to establish 'cache to cache' interaction network in chapter 4 for fast shared data routing among threads, thereby reducing access

Table 2-3. Summaries on access pattern features.

Quantity	Contents in Section 2.3	Contents in Section 2.4
Applying proposal	Ch.3 Controllable cache resource allocation	Ch.4 Stacked 3D on-chip cache network
Profiling granularity	Per interval	Single cache access
Characteristics	Energy and hit rate	Access latency
Applying cache level	Shared cache	Shared or private cache

latency greatly. As shown in Table 2-3, contents in this chapter are classified into two applying proposals.

Chapter 3

CONTROLLABLE CACHE RESOURCE ALLOCATION

3.1 Introduction on Cache Resource Optimization Design

On-chip system architectures tend to integrate more and more large scale of both processing cores and hierarchical cache. And the integrated processing core number is increasing as dozens or even hundreds. As a result, the shared cache which is used for bridging the speed gap of processing cores and off-chip memory is desired to have a large enough size for the purpose of providing sufficient interconnecting bandwidth to serve so many processing cores. However, the shared cache will consume about half of on-chip energy and occupy about half of the on-chip implementation area also. Actually, only a part of cache resources will be employed during runtime and in the very rare case that all cache resources are fully used at the same time. Thus, there is an opportunity that some shared cache resources can be re-allocated to other threads which desire more cache resources, thereby both energy consumption and performance will be improved.

Many pieces of research focus on the shared cache allocation in purpose to get the utmost out of cache resources, based on the observation that most of cache resources only contribute very few access hits during their entire lifetime

[1, 8], on the contrary, those resources waste a large quantity of energy to keep themselves activated. To prevent energy waste on those resources, research in [9] proposed a shut-down based method to put some resources at minimal activated status, which consumes a small amount of energy but can be fast activated in case of resource reusing. Nevertheless, the number of redundant cache resources or scarce cache resources should be accurately explored because that removing too many resources will lead to many access misses and adding too many resources will result in extra energy consumption also. Hence, some previous researches try to reconfigure operational cache resource by searching for the optimal cache resource allocation corresponding to every benchmarks [6, 20], but such reconfigurable cache method needs to alter used cache parts frequently, leading to unnecessary data erasure. Some researches try to identify the requested cache resources based on locality features and further propose the access-phase-aware based cache architectures [11, 18], but their tuning intervals by equally dividing dynamic instruction flow or executing time (length-fixed intervals) are unable to classify phases precisely. And some researchers proposed exploration algorithms to support the software-based OS-level cache allocation [13, 19], or to achieve the optimal resource allocation combinations [12], but their exploration processes are very costly and some explored results are outdated after long exploration time. Therefore, novel intervals are needed to fit phase variation rather than length-fixed intervals, and also fast tuning method is desired to replace costly exploration method.

In this chapter, a novel cache resource allocation method [73] is proposed to dynamically allocate cache resources during runtime and accurate resource demand can be explored based on locality analysis, and further the cache resource allocation process is working in the granularity of interval level, which can provide much more precise locality tracing. Firstly, according to pre-experimental analysis on access distributions discussed in chapter 2.4, energy consumption will vary closely with allocated cache resources increasing and there is an approximately optimal amount of allocated cache resources. Thus, the cache allocation process is shifting to find every approximately optimal

amount of allocated cache resource corresponding to each benchmark. Secondly, based on locality analysis in granularity of per subroutine call, some most-frequently-used subroutines can be selected as suitable instruction clusters that repeated calls on those subroutines can take a majority of entire instruction set and the appearances of subroutine calls form the behaviors of locality, thereby those calls can be the prefect intervals on cache resource allocation. Finally, since the control loop and control intervals are proposed, the self-controllable cache resource allocation method can be actualized in purpose of maximizing resource utilization. Note that there are few kinds of workloads executed in embedded systems, so that it is realizable to explore control parameters used in each benchmark. Thus, the discrete Proportion-Integration-Differentiation (PID) control method can be integrated with the runtime cache resource allocation process. As a result, each thread can always be allocated the optimal amount of cache resource in the granularity of per subroutine call dynamically, thereby saving plenty of on-chip energy consumption without any performance degradation.

In the rest of this chapter, motivations of this design are firstly described. And then the controllable cache resource allocation method is proposed in detail including control interval design, PID parameters setting and controllable cache resource allocation path design.

3.2 Motivations Applied from Hierarchical Cache Demands

As described in chapter 2.3, two key appearances are emphasized to represent the motivations of proposed cache resource allocation method as follows. (1) There is an approximately optimal demand existing among different allocated cache resource scales on one benchmark. (2) Locality stable and locality change affect the cache resource demand.

In the first appearance, the approximately optimal demand is explored by comparing energy consumptions in case of allocated shared cache banks increasing. If there is only one bank allocated, many access misses are

happening to enlarge the overall latency due to off-chip memory accesses. In pace with allocated banks rising, increment of hit rate is becoming smaller and further remains almost stable. Suppose that there is a balance point, in which the benefit of energy consumption from hit rate improvement is equal to the energy cost of extra allocated cache bank. Similar to the steady-state value in control theory [23], extra allocated banks will bring out more energy consumption and then feedback output will generate negative bank number to further reduce some banks in the next interval until the balance point is achieved. Hence, the cache resource allocation process can integrate the control theory as the appearance follows the same process of feedback-based allocation loop, and the objective of cache allocation is to converge at the balance point. In case of locality change (similar to a disturbance in control loop), the cache allocation tends to converge at a new balance point.

In the second appearance, the cache resource demand is varying along with dynamic instruction flow and any selected instruction cluster will behave in different energy consumptions. Furthermore, this cluster may have its own optimal cache resource allocation corresponding to the current period. As described in chapter 2.3, the sampled results on different periods show quite various energy consumptions and there are two appearances including many locality-change points and locality-stable periods exciting. However, the traditional partition method which divides entire dynamic instruction sequence into isometric intervals or partitions the execution time into isometric intervals cannot trace locality variation and obviously, little improvement can be achieved. Hence, a novel interval is proposed based on subroutine calls. Since the pre-experimental results have shown that dynamic instructions counted from subroutine calls can take the majority of whole instruction amount and different calls on the same subroutine will behave in similar execution sequence and so as to energy consumption and performance. Thus, it is suitable to employ subroutine calls for allocating cache resources.

Moreover, those two appearances on locality can be reflected from a different perspective that the locality change or stable can be shown as the

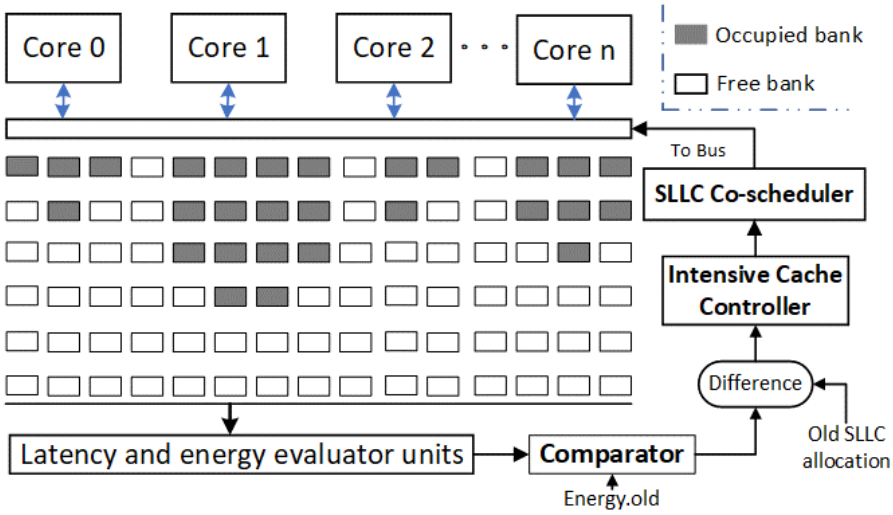


Figure 3-1. Controllable shared cache architecture. A feedback based control loop is formed into shared cache bank co-scheduler for dynamically allocating banks to private processing cores.

variation of energy consumptions. In other words, the energy consumptions will be reduced if allocated extra cache resources can satisfy the current resource demand. Thus, similar to the approximately optimal cache allocation on per-application granularity, each subroutine may have an approximately optimal cache allocation value, and once this value is explored, it can be shared with all the calls on this subroutine. Corresponding to the sampling intervals of control theory, each call of particular subroutine can act as a sampling interval felicitously.

Since the control loop and control intervals are realizable on cache resource allocation, the motivations of this thesis are to integrate control theory into cache resource allocation and to implement such design into SLLC for sufficient cache resource utilization.

3.3 Controllable Hierarchical Cache Design

3.3.1 Control Loop Based Architecture

As shown in Fig. 3-1, the shared cache banks serve to plenty of processing cores and are partitioned to many processing threads during the runtime. Note that static non-uniform cache architecture (SNUCA) supports bank-level allocation

with quite limited overhead. Thus, a traditional SNUCA model [38] is employed to propose cache architecture. To implement the control loop into the shared cache, several components should be designed to support the following three rules:

(1) Capture the energy consumption and hit rate information in the latency and energy evaluator units, and further to compare with the previous data stored in record table, and finally generate the difference values.

(2) Calculate the feedback value in the form of cache bank increment which is operated on the difference of energy consumptions. And combine the increment with previously allocated banks in the intensive cache controller.

(3) Re-allocate shared cache banks based on the output of intensive cache controller, and further implement it to target thread in the cache co-scheduler.

Note that each core conducts one control loop in asynchronous manner, in other words, the control loop in a core can be trigged only when a control interval of current thread appears. And free banks are allocated to cores based on physical distance evaluations, where co-scheduler will first allocate short-distance free banks or first retire long-distance occupied banks.

For each interval, the control loop will modify the bank allocation once until the locality is stable. Note that conventional intervals are designed with fixed number of execution time or counted dynamic instructions. For example, an interval holds in a period of ten million clock cycles or an interval contain ten million dynamic instructions. If there is an interval detected, the energy consumption of this interval can be compared with one of previous intervals. And then such difference value can be transformed from energy consumption difference into linked cache bank increment. Finally, the bank increment is delivered to shared cache co-scheduler for the purpose of allocating cache resource based on current demand. Along with instruction flow, most of intervals can work in optimal cache allocation, thereby saving plenty of on-chip energy consumption.

3.3.2 Energy Consumption and Latency Models

In order to trace energy consumption and latency values in each interval, minimal hardware scale is the first design principle to count runtime information because it is costly to directly measure those values. Alternatively, those values can be quantized by doing statistics on some basic information such as hit rate instead. And then, the energy consumption and latency values can be calculated with the aid of cache model in CACTIv6.5 [55], which has already counted the transistor-level energy consumption and latency together to form the precise energy and latency formulas as follows.

$$L_{Per\text{-}Access} = R_{hit} * (Max(L_{tag}, L_{data}) + L_{outdriver} + L_{Pre\text{-}Charge}) + R_{mis} * (L_{tag} + L_{off\text{-}chip\ memory}) \quad (3\text{-}1)$$

$$E_{Per\text{-}Access} = R_{hit} * (E_{dyn-hit} + E_{leakage-hit}) + R_{mis} * (E_{dyn-mis} + E_{leakage-mis}) \quad (3\text{-}2)$$

The cache model contains four key components including tag array, data array, output driver logic, and pre-charge logic. Each cache access will follow the standard access path uniformly. For example, an access miss follows the path of tag comparison, output miss signal and then access the off-chip memory. Hence, the energy consumption and latency values can be accumulated with the values of every cache accesses. As shown in Eq. 3-1 and 3-2, the latency and energy consumptions of single access can be achieved from the cache model named L with subscripts as tag, data, output-driver, pre-charge, and off-chip memory respectively, and energy named *E* with subscripts as *dyn-hit*, *leakage-hit*, *dyn-mis*, and *leakage-mis* respectively. And R_{hit} and R_{mis} stand for the probability of hit access and miss access, that is hit-rate and miss-rate, over all cache accesses within current interval, which can be calculated based on runtime statistic. As those parameters can be easily achieved from cache model and runtime counting, there are few calculation quantities needed during runtime, thereby causing few hardware and latency overhead.

3.3.3 Intensive Cache Controller Design

In this subchapter, the intensive cache controller is proposed to represent the flow of transforming control quantities. Assume that there is a dynamic instruction sequence named S, where S is equally divided by N to form an interval set as $\{1, 2, \ldots N\}$ and n represents the n-th interval ($n \in \{1, 2, \ldots N\}$). Assume that the energy consumption of a benchmark named β at the n-th interval is $E_\beta[n]$. Here, energy consumptions are the mean values that are normalized to per-access granularity in current interval. Hence, integrating with control theory, the discrete increment-Proportional-Integral-Derivative based intensive control formula can be written into the following form.

$$e_\beta[n] = E_\beta[n] - E_\beta[n-1] \qquad n \in \{2, 3, \ldots N\} \tag{3-3}$$

where $e_\beta[n]$ stands for the difference value of energy consumption between the n-th and the $(n-1)$-th intervals. Suppose that there is a disturbance happened at the n-th interval, and such disturbance leads to energy value change. For example, the n-th interval belongs to subroutine call of another selected subroutine. Thus, the locality change also happens and previous cache bank allocation is not optimal anymore. That is to say, the new cache allocation should be generated based on the difference value. Corresponding to sampling points and feedback quantity in control theory, the cache bank increment can be generated by calculating with recent three difference values to form the negative feedback-based control loop. And the cache allocation increment named $\triangle C$ can be written as the form of discrete increment-Proportional-Integral-Derivative (Increment PID) based control model as follows.

$$\triangle C = K_P * (e_\beta[n] - e_\beta[n-1]) + K_I * e_\beta[n] + K_D * \tag{3-4}$$

$$(e_\beta[n] - 2e_\beta[n-1] + e_\beta[n-2])$$

Where K_P, K_I and K_D represent setting parameters of proportional ratio, integral ratio, and derivative ratio, respectively.

In order to tune K_P, K_I and K_D properly, the characteristics of cache resource allocation should be considered as key constraints. Firstly, the defined intervals

are treated as control sampling points while the energy consumptions in those intervals are assumed as the control volumes similar to control theory. Moreover, each interval only generates one energy consumption, expanding to whole dynamic instruction sequence, N control volumes values can be achieved from N intervals. Secondly, as the locality change is accompanied with allocated bank scale in a concave sampling curve, difference values of intervals on energy consumption can be reflected with allocated cache bank increment. Furthermore, the output parameter $\triangle C$ is in the granularity of per cache bank, so that the cache bank increment should be large enough to trigger bank co-scheduling operation. And each control interval that may contain millions of dynamic instructions only generates control output once. As a result, the control process is inclined to work in low control frequency while minimal hardware and latency overheads are consumed by proposed extra control components. That is to say, there exists an alternative trade-off on control output accuracy and design overhead. Actually, the control output in per-bank based granularity acts as a rough control result, thus, minimal overhead is much more suitable rather than an accurate result. Consequently, the traditional control parameter tuning methods such as the self-adapting method and approximation method are not applicative to proposed control loop due to above considerations [3, 23].

Fortunately, each benchmark behaves similar runtime characteristics among repeated runtime sequences, in other words, a single benchmark can be simulated in the platform with similar results among many time simulations. Thus, such a repeatability feature can be employed to tune the control parameters with successive approximation method step by step.

In this first step, set proportional ratio K_P and derivative ratio K_D as zero, then do the simulation on current benchmark where integral ratio K_I is set in the range from zero to the experimental number (set as 5.0 in this thesis) with the cut and try increment value of 0.05. After running the benchmark with one hundred times, corresponding one hundred energy consumptions are compared to achieve the energy minimal one, and then set the relevant integral ratio as the approximate K_I.

In the second step, set K_I as the approximate integral ratio achieved in the first step and set derivative ratio K_D as zero, then simulate the benchmark with the proportional ratio K_P varying from zero to the experimental number (set as 5.0 in this thesis) in granularity of 0.01, then do the comparisons among five hundred energy consumptions to achieve the lowest one, and then determine the relevant integral ratio as the tuned proportional ratio K_P.

In the third step, set K_P as the tuned value and set derivative ratio K_D as zero, then repeat the simulations with integral ratio K_I ranging from zero to five in the granularity of 0.05, and finally, determine the tuned integral ratio K_P through energy comparisons.

In the final step, set K_P and K_I as the tuned values, then do the simulations with derivative ratio K_D ranging from zero to five in the granularity of 0.05, and finally determine the tuned derivative ratio K_D through energy comparisons.

Hence, three control parameters are achieved and further, the tuning method can be expanded to the rest benchmarks. As each simulation only is executed in several seconds and there are eight hundred simulations for one benchmark of total 29 benchmarks, the total work of tuning control parameters can be finished with ten hours. Thus, it is a practical proposal for tuning control parameters on all benchmarks.

As it is acceptable to employ minimal calculation logic on proposed control loop, the energy consumption and latency overhead of extra control components are ignorable, because that those control components only work once in an interval (contains millions of dynamic instructions). Moreover, the bank increment is only related to one current energy consumption and two previous energy values, thereby control loop has strong robustness on confronting steady-state error.

3.3.4 Cache Bank Allocating

In this subchapter, the processes of classifying benchmarks are represented by the proposed bank allocation method which takes take full advantage of hit rate

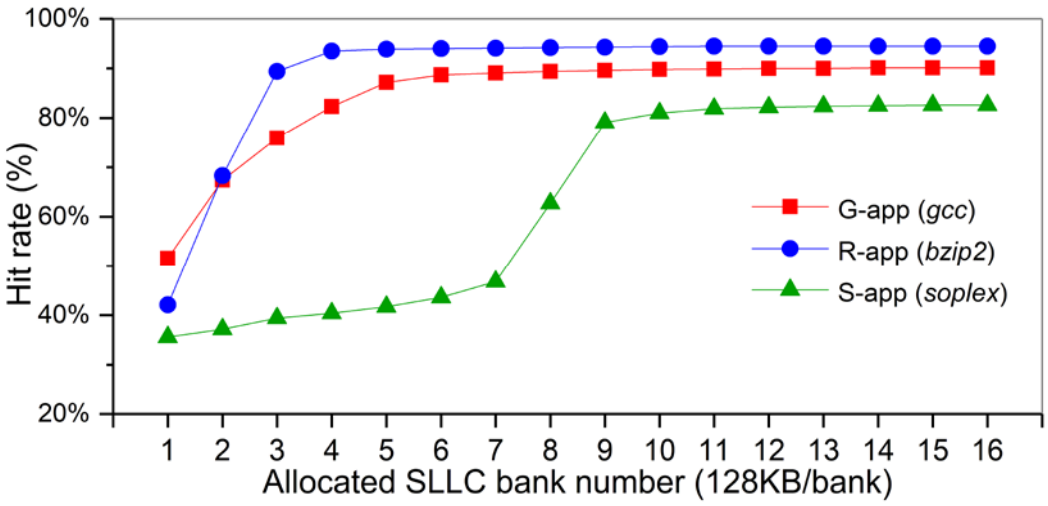


Figure 3-2. Tendency sampling curves on hit rate. Three benchmarks are selected as examples to stand for each access type. The allocated SLLC bank number range from one to sixteen.

features. As the locality or cache allocation is changed, the direct effect appears at hit rate variation. Thus, extra cache resources should be allocated to suitable threads for the purpose of achieving the most improvements on hit rate, and all benchmarks have their own characteristics so that they should be classified into several types in order to co-schedule the cache resources for many threads efficiently, thereby guaranteeing the fairness on each thread. Note that fairness in other researches [14, 62] is employed to mean cache resources for each thread to solve thread starving in case that busy threads will take all cache resources under least recently used (LRU) algorithm and rest threads will be starving without any cache resources allocated. But, even though cache resources are allocated fairly, some threads which need more cache resources may work in low efficiency and some threads which need fewer cache resources may lead to resource wasting. Thus, it is suitable to employ cache resource demand instead of fairness because the demands of each thread act as the true efficiency if those demands are supplied. As the demands of cache resources are highly related to allocated banks and further those demands are reflected by hit rate curves, hit rate tendencies also represent the tendencies of optimal cache resource allocation. As shown in Fig. 3-2, hit rate curves show quite different tendencies along with allocated cache bank increasing. In the first several banks, hit rates

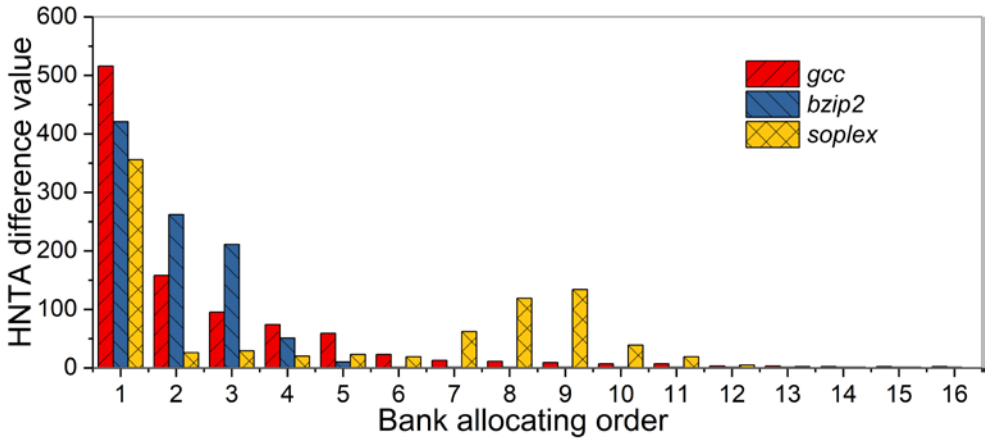


Figure 3-3. Sampling on HNTA difference values. The bank allocating order is set from allocating 1st bank to 16th bank in X-axis.

tend to be improved greatly, and then keep stable in the last several banks. And the amounts of variation on each hit rate curves show different features, and other benchmarks behave in the scope of those three tendencies. Hence, it is inspired that all benchmarks can be classified as several tendency types as follows.

(1) Rapid increasing benchmark (R-app).

(2) Gradual increasing benchmark (G-app).

(3) Sharp increasing benchmark (S-app).

Where those tendency types are determined by pre-experimental analysis as follows.

Define a benchmark β, then do the simulations under the condition of allocated banks arraying from one to N banks, repeatedly, and then get the hit rate curve as shown in Fig. 3-2. Here, set the hit rate curve to be represented with hit access numbers per thousand accesses (HNTA), named H_n where $n \in \{1, 2, \ldots N\}$ stands for the allocated bank number. Note that the only changed parameter in the platform is the allocated bank number and then hit rate variation is also only related to allocated bank number. Thus, the allocated one bank is changed and then leads to the hit rate variation, which can be represented by HNTA difference (h) as follows.

$$h_n = H_n - H_{n-1} \qquad \text{where } H_0 = 0, \quad n \in \{1, 2, \ldots N\} \tag{3-5}$$

Where h_n stands for the HNTA difference of hit rate variation with allocating *n*-th bank. That is to say, each h_n represents the access hit contributions of the *n*-th cache bank, and such value can act as the potential hit increment if one bank is allocated to *(n-1)*-th bank situation. Thus, all differences are counted into hit rate curves for the purpose of analyzing bank allocation features.

As shown in Fig. 3-3, the first several banks in the curve of *bzip2* benchmark supply the most access hits, and the rest banks are rarely contributed some access hits. Many banks in the curve of *gcc* benchmark can contribute some access hits although the amounts of hit rate variations are smaller than the ones in the first several banks. And the first several banks in the curve of *soplex* benchmark supply some access hits, and if allocated bank number is exceeded the reuse distance of some key instruction clusters, many access hits can be achieved [21]. As the middle side values of hit rate curves show different behaviors, a new evaluation metric M_β is proposed to classify all benchmarks, which can be defined as follows.

$$M_\beta = \frac{\sum_{n=1}^{N} h_n - \sum_{n=1}^{N/2} h_n}{\sum_{n=1}^{N} h_n - h_{min}} \tag{3-6}$$

Where h_{min}=h_1. Moreover, two thresholds named T_{S-BM} and T_{G-BM} are introduced to identify each benchmark type as follows ($T_{S-BM} > T_{G-BM}$).

(1) M_β satisfies Eq. 3-7, the benchmark β is classified as the sharp increasing benchmark (S-app).

$$M_\beta \geq T_{S-BM} \tag{3-7}$$

(2) M_β satisfies Eq. 3-8, the benchmark β is classified as the gradual increasing benchmark (G-app).

$$T_{S-BM} > M_\beta > T_{G-BM} \tag{3-8}$$

(3) M_β satisfies Eq. 3-9, the benchmark β is classified as the rapid increasing benchmark (R-app).

$$M_\beta \leq T_{G-BM} \tag{3-9}$$

Furthermore, exhaustive experiments are set based on benchmark analysis for the purpose of identifying two thresholds, and T_{S-BM} and T_{G-BM} are set as 0.27 and 0.06 to classify all SPEC benchmarks into S-app, G-app or R-app clearly.

Based on appearances of each benchmark type, the demands of each benchmark can be obviously identified that the first several cache banks in R-app benchmark should be ensured and then extra banks are useless, and the allocated cache banks for S-app benchmark should exceed the reuse distance, and allocated cache banks for G-app benchmark have the lowest priority. Thus, all threads with different benchmarks can be supplied with allocated cache banks just as their cache resource demand and the rest cache banks can be set into low-power mode, thereby saving plenty of energy consumption.

3.3.5 Locality-aware Control Interval Design

In the previous cache allocation methods [6, 9], intervals are achieved by partitioning on dynamic instruction flow or dividing execution time into many

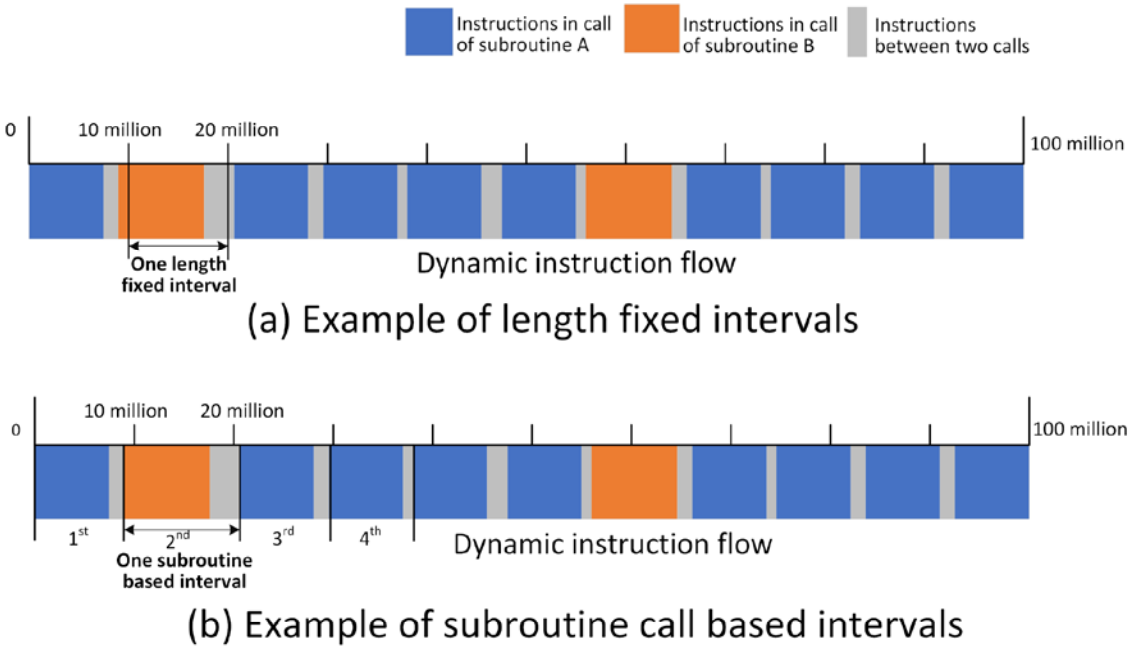


Figure 3-4. Example of tuning interval methods. (a) Length fixed intervals; (b) Subroutine call based intervals. Where the dynamic instruction flow is set as 100 million dynamic instructions in total, and one length fixed interval is set as 10 million dynamic instructions.

uniformed intervals. Although those methods are much simpler to implement into the platform, the arbitrary segmentation of those methods will lose the ability to tracing runtime locality change for the purpose of saving more energy consumption. Based on the distribution appearances of subroutines analyzed in subchapter 2.3, hot subroutine calls act as the prefect intervals, which are highly related to demand variation and calls on same subroutine behave similar demand also. As a result, subroutine-call based intervals are in accordance with runtime energy features rather than equally partitioned methods. As shown in Fig. 3-4(a), the fixed-length based intervals are generated by equally partitioning dynamic instruction flow into intervals with same amount of dynamic instructions. Obviously, this method cannot trace cache resource demand variation. As shown in Fig. 3-4(b), subroutine-call based intervals are used to partition the dynamic instruction flow by subroutine calls. As dynamic instructions in selected subroutine calls take the majority proportion of overall dynamic execution sequence, the dynamic instructions which are counted from the start instruction of a subroutine call to the start instruction of the next subroutine call can be treated as one subroutine-call based interval. Note that formed intervals will have different dynamic instruction numbers inside because each interval length is only related to the period between two start instructions. Thus, repeated calls on same subroutine can be treated as demand stable intervals and optimal cache resource allocation can be explored within those intervals corresponding to current subroutine, and further explored results can be directly employed by later calls on this subroutine. For example, interval 1 and 3 share a similar cache resource demand. Based on the statistic distributions of each subroutine in a benchmark shown in subchapter 2.3, all subroutines in a benchmark can be ordered into a queue based on the distribution of each subroutine, and then some hot subroutines are selected while some tiny subroutines or not commonly used subroutines are filtered out. As shown in Table 3-1, selected subroutine numbers of all benchmarks are experimentally analyzed to form control intervals.

Table 3-1. Statistics on selected hot subroutines.

Benchmark	Type	Hot Sub.	$C_{app(x)}[0]$	PID-K_P,K_I,K_D
400.perlbench	G-app	22	6	0.92, 0.15, 0.15
401.bzip2	R-app	6	4	1.28, 0.2, 0.3
403.gcc	G-app	19	5	0.89, 0.1, 0.2
410.bwaves	R-app	7	4	1.36, 0.2, 0.25
416.gamess	G-app	17	5	0.95, 0.25, 0.3
429.mcf	S-app	8	13	0.71, 0.1, 0.25
433.milc	G-app	19	6	0.88, 0.2, 0.15
434.zeusmp	R-app	9	5	1.07, 0.2, 0.3
435.gromacs	G-app	20	6	0.85, 0.25, 0.2
436.cactusADM	S-app	11	12	0.69, 0.15, 0.2
437.leslie3d	R-app	6	4	1.38, 0.1, 0.25
444.namd	G-app	13	7	0.82, 0.1, 0.3
445.gobmk	G-app	15	7	0.74, 0.1, 0.25
447.dealII	G-app	21	6	0.86, 0.15, 0.2
450.soplex	S-app	6	11	0.63, 0.1, 0.2
453.povray	G-app	19	6	0.91, 0.2, 0.25
454.calculix	G-app	16	7	0.84, 0.25, 0.25
456.hmmer	R-app	6	3	1.42, 0.2, 0.35
458.sjeng	R-app	8	4	1.28, 0.25, 0.25
459.GemsFDTD	S-app	13	11	0.64, 0.2, 0.15
462.libquantum	R-app	5	3	1.41, 0.25, 0.25
464.h264ref	R-app	8	4	1.37, 0.25, 0.2
465.tonto	S-app	16	12	0.61, 0.25, 0.2
470.lbm	R-app	4	3	1.47, 0.2, 0.15
471.omnetpp	S-app	18	13	0.63, 0.15, 0.25
473.astar	R-app	7	4	1.15, 0.2, 0.2
481.wrf	S-app	15	12	0.68, 0.15, 0.2
482.sphinx3	S-app	12	9	0.68, 0.1, 0.2
483.xalancbmk	G-app	21	7	0.95, 0.15, 0.2

3.3.6 Discrete PID-based Control Algorithm

As the traditional PID based control loop cannot be directly employed in our proposed design, several modifications should be done to integrate control theory into cache resource allocation method. Firstly, the difference values of energy consumption among intervals should mark with corresponding subroutine calls, so that calls on same subroutine are considered as a series of

ALGORITHM Ⅰ

Intensive SLLC increment Generating

Input

$E_\beta[n]$: energy per access of benchmark β in n-th interval;

Where $E_\beta[0] = E_\beta[-1] = 0$;

$C_{app(\beta)}[n]$: allocated banks of benchmark β in n-th interval; K_P, K_I, K_D and $C_{app(\beta)}[0]$ are listed in Table. 3-1;

$R_{hit}[n]$: hit rate in n-th interval;

r, ***g*** and ***s***: act as trigger marks of interval start for R-app, G-app and S-app;

Bank: free bank set;

Output

$C_{app(\beta)}[n]+\triangle C_{bank}$: required bank in next interval;

```
1:  x ← application loading;
2:  r, g, or s ← application type;
3:  for from n=1 to N−th interval appears do
4:      eβ[n] and Rhit[n] calculation;
5:      △Cbank =KP*{eβ[n]- eβ[n-1]}+ KI*eβ[n]+ KD*{eβ[n]-2eβ[n-1]+ eβ[n-2]};
6:      △Cbank= int[△Cbank];   //normalized in integer
7:      if Rhit[n] > Rhit[n-1] then
8:          △Cbank = -|△Cbank|; //banks decreased
9:      else
10:         continue;
11:     end if
12:     if △Cbank > Bank & one of r and s appears then
13:         retire banks of G-app for R- and S-app until G-app banks are
            reduced to Capp(β)[0];
14:         △Cbank=retired banks + Bank;
15:     else
16:         if △Cbank > Bank & g appears then
17:             △Cbank= Bank;
18:         end if
19:     end if
20:     return Capp(β)[n]+△Cbank;
21: end for
```

intervals, which have similar cache resource demand. Secondly, control outputs should be normalized into an integer number of banks as the cache co-scheduling operation is done in the granularity of per cache bank. Finally, the cache co-scheduling operation should comply with the allocation priority of each benchmark type. As described in Algorithm I, the modified PID-based cache resource allocation process is represented in the form of pseudo-code as follows.

In the beginning, the control interval in the n-th order is defined as counted dynamic instructions between the start instruction to the start instruction of the (n+1)-th interval. Thus, the energy consumption $E_\beta[n]$ and hit rate $R_{hit}[n]$ can be calculated based on runtime counting in the n-th interval. Moreover, the cache bank allocation value in the granularity of per application is named as $C_{app(\beta)}[0]$, which can act as the initial allocated bank number to current benchmark. Hence, for each sequential interval, energy consumption difference values $e_\beta[n]$ and $R_{hit}[n]$ are calculated as shown in line 4, then previous two difference values $e_\beta[n-1]$ and $e_\beta[n-2]$ are set into control formula for calculating the bank increment in the line 5, and such value is normalized into integer number for the purpose of per bank based allocation. In the line 7 and 8, the negative feedback-based control output is ensured for bank increasing in case of hit rate rising. From line 12 to line 19, the priority of bank co-scheduling for many threads is set as R-app type first, S-app type second and G-app last. And finally, return the target amount of bank number to co-scheduler.

To theoretically analyze the controllable cache allocation algorithm, three cases are assumed to explain the process of cache allocation based on the control output as follows.

(1) Case 1: control output $\triangle C_{bank}$ is equal to zero, which means that cache resource demands in current interval and previous two intervals are similar, or current cache resource demand variation is not large enough to trigger the integral number of bank allocation.

(2) Case 2: control output $\triangle C$ is a negative integral number, which means that the cache resource demand in current interval is changed and some of previous allocated banks are superfluous for current interval because that the locality of *n*-th interval has changed to desire fewer cache banks than previous intervals. Thus, the returned bank number is smaller than $C_{app(\ )}$[n].

(3) Case 3: control output is a positive integral number, which means that the cache resource demand in current interval is changed and some of previous allocated banks are insufficient for current interval because that the locality of the *n*-th interval has changed to desire more cache banks than previous intervals. Thus, the returned bank number is larger than $C_{app(\ )}$[n].

Corresponding to allocated bank positions in Fig. 2-2 as an example, case 1 will happen if the demands in the current interval and previous two intervals are five banks (assume five is the optimal bank number). If the allocated bank number in previous two intervals is six and locality is changed as five in current interval, case 2 happens to remove one superfluous bank. And if the allocated bank number in the previous two intervals is four, and locality demand is

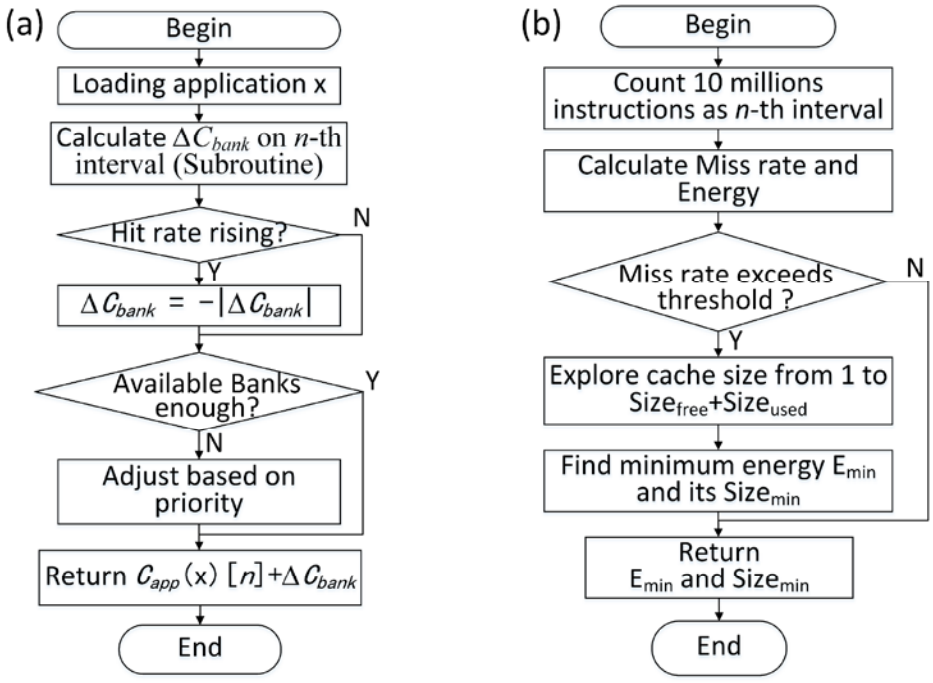


Figure 3-5. Flow charts of two tuning methods. (a) Proposed controllable tuning method with subroutine-based intervals; (b) Exploring method with length-fixed intervals.

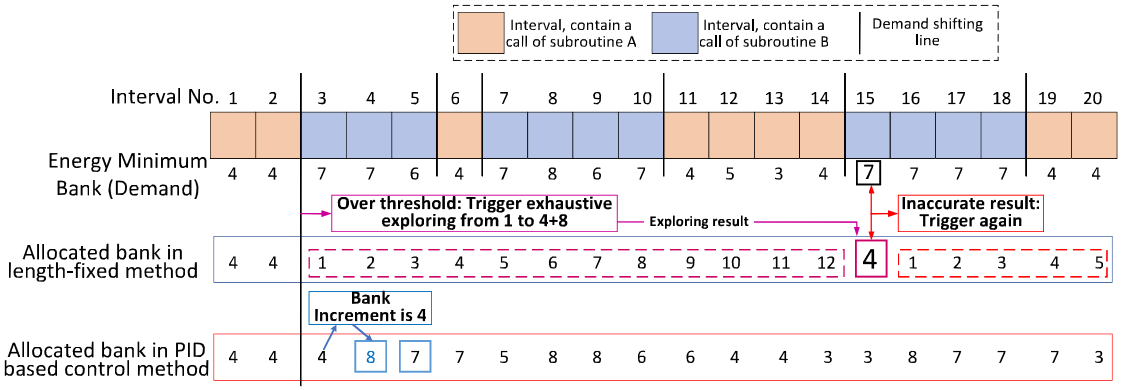


Figure 3-6. An example of runtime tuning effects on fixed-length based method and PID-based method. Assume (1) 20 equal-length intervals in total; (2) 8 free banks available; (3) Initial allocated bank number in call of subroutine A is 4, and in call of subroutine B is 7.

changed as five in the current interval, case 3 happens to add one more bank to cache co-scheduler. As a consequence, the locality change can be quickly traced and then generate the optimal cache resource allocation in the granularity of per interval dynamically, thereby cache resource demands are always self-controlled to supply with plenty of energy consumption saved.

Figure 3-5(a) shows the flow chart of proposed controllable tuning method with subroutine-based intervals. For each interval start, the controller can generate the bank increment quickly, and then determine the direction of such bank increment by using hit rate variation, and if desired banks are lacking, adjust allocation based on application priority. Proposed method only needs few intervals to achieve accurate tuning results, so that those results can be applied to the next interval in time. As shown in Figure 3-5(b), previous exploring method with length-fixed intervals needs to explore the energy-lowest cache size once miss rate exceeds the threshold. However, such exploring process costs expensive latency overhead, and its tuning results are usually outdated after long exploring time.

In order to represent the efficiency of PID based cache tuning method, example flow in Figure 3-6 shows tuning effects of fixed length method and proposed control method. In the end of second interval, locality change will trigger an exploring process for next 12 intervals (8 free + 4 occupied banks),

and then apply the explored result 4 banks to fifteenth interval. If it is not suitable, do exploring process again. In control method, locality change in third interval will generate a bank increment to apply for the fourth interval, and remain stable within few intervals. And locality change in sixth, eleventh, fifteenth and nineteenth interval can be handled rapidly. As a consequence, the PID based cache tuning method has advantages as follows.

(1) PID tuning results are adaptive to all calls on same selected subroutine. As tuning intervals are classified by calls of selected subroutines, locality can be easily traced based on jumps from a call of one subroutine to another. In other words, proposed tuning method can be naturally adaptive to locality changes.

(2) High tuning speed and Accurate tuning results. The bank increment is generated directly based on energy difference values of past three intervals, so that each tuning result represents current bank demand (approach to ideal), and remains steady in few intervals rather than long exploring time of fixed length method.

(3) Sensitive to disturbances. Any disturbance can be imported to controller by caused energy variation, however, tuning results of fixed length method can be affected only if energy variation caused by the disturbance is larger than the threshold. So that proposed method is much sensitive to disturbances.

As a consequence, PID control is suitable for cache tuning process with above advantages.

3.4 Evaluation Strategy

3.4.1 Experimental Platform Setups

To evaluate our proposed method, experimental setups are described including platform model, benchmarks, comparisons and evaluation metrics as follows.

Platform model: the instruction-tracing driven based full system simulator Gem5 [53] is employed for constituting a multi-level multi-core on-chip system model, which consists of private cache level, shared cache level, main memory, and multi-core processors, and each component of the simulation platform is set

as listed in Table 3-2. Note that the hierarchical cache modules are represented by extended CACTIv6.5 cache area, energy and integration analysis model, where the first cache level is set as the private cache and the second cache level is set as the shared cache.

Benchmarks: during verifying the cache allocation method in the granularity of both per-application and per-subroutine-call intervals, the SPEC benchmark suite is employed to fully evaluate the efficiency of proposed method. And Table 3-3 lists the setups on selected benchmark groups, which are mixed into co-executed runtime processes.

Comparisons: the proposed cache allocation method is compared with the other two allocation methods, one is to simulate in the granularity of per-application intervals and the other one is to simulate with approximately optimal bank allocated and the rest banks are set in sleep mode [10].

Evaluation Metrics: To unify the performance and energy consumption metrics, the total evaluated values are normalized by the cache request numbers to form the normalized energy consumption per access. And the performance is represented by IPC values which are counted from the IPC values in all threads simultaneously. In all evaluation tests, the counting period is uniformly set as the period of executing five-billion-dynamic-instructions.

Table 3-2. Test System Configurations.

Component	Configurations
Processor Unit	2.0 GHz, 4 cores, single thread per core, 1.1 V supply voltage, 128 IW entries, 32 nm technology library, 30 cycle TLB miss latency.
L1I/L1D Cache	32 KB instruction cache, 32 KB data cache for a core (private), 4-way, 64 B line size, 4 cycle latency.
Shared L2 Cache	4 MB total size, 256 KB for a core (private), 8-way, 64 B line size, 8 cycle latency,
Main Memory	4 GB Double Data Tate (DDR4 2133MHz, 1.2V), 8KB page size, 95 cycle latency.

3.4.2 Simulation Workload Setups

As there are four threads existing in four processing cores, each core is allocated with a single thread to execute a working set copy from a particular benchmark. Thus, those threads are isolated into each private processing unit and partition lines in shared cache averagely allocate shared cache banks to each processing thread. During runtime, free banks can be re-allocated to imperative threads

Table 3-3. Statistics on selected hot subroutines.

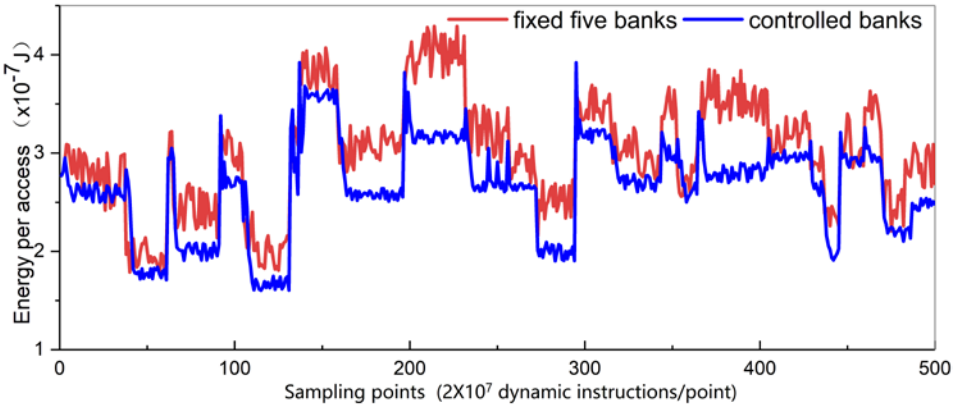
Selected Benchmarks	Group	Type	$\sum C_{app(x)}[0]$
gcc-gcc-gcc-gcc	all gcc	G-G-G-G-app	20
bzip2-bzip2-bzip2-bzip2-	all bzip2	R-R-R-R-app	16
gcc-namd-bzip2-zeusmp	gnbz	G-G-R-R-app	20
namd-gobmk-soplex-mcf	ngsm	G-G-S-S-app	38
astar-zeusmp-soplex-mcf	azsm	R-R-S-S-app	33
mcf-sphinx3-gobmk-zeusmp	msgz	S-S-G-G-app	34
astar-zeusmp-soplex-gobnk	azsg	R-R-S-G-app	27
namd-gobmk-bzip2-mcf	ngbm	G-G-R-S-app	31
soplex-mcf-sphinx3-gobmk	smsg	S-S-S-G-app	40
bzip2-astar-zeusmp-gcc	bazg	R-R-R-G-app	18

based on control outputs. Hence, eight groups are formed by mixing different four benchmarks, as shown in Table. 3-3, each benchmark is allocated to one thread for parallel simulation.

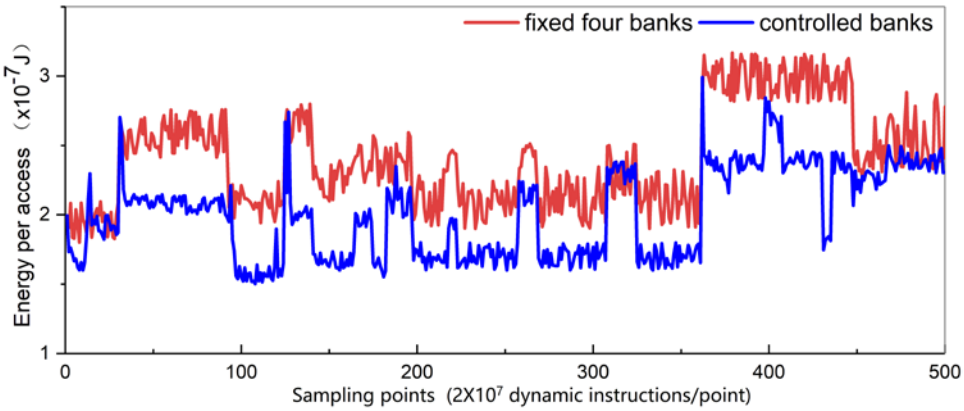
3.5 Controllable Cache Resource Optimizing Results

Firstly, three benchmarks where each one stands for a classified locality type (G-app, S-app or R-app) are selected as examples to represent the runtime allocated bank variations along with executing flow. The *gcc* benchmark is initially allocated with five banks as the approximately optimal bank allocation

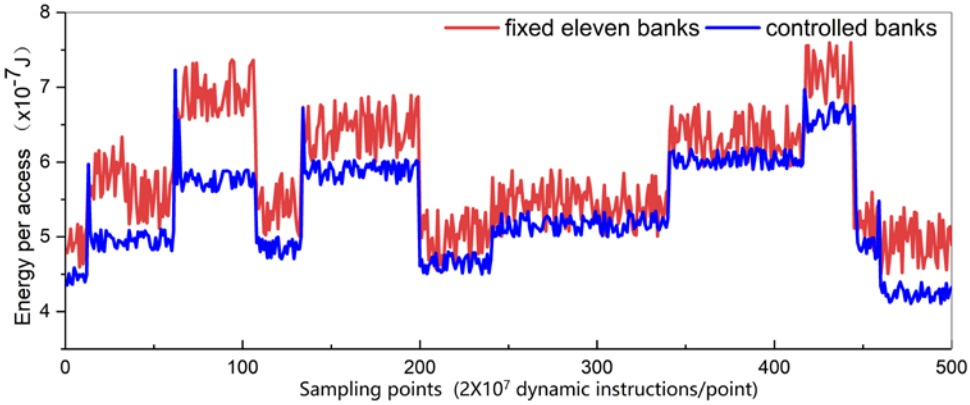
number in the granularity of per application, which shows the runtime features of a gradual increasing application. And then nineteen hot subroutines of *gcc* benchmark are selected to apply their calls as control intervals, and further, the



（a）Energy per access profiling with *gcc* benchmark.

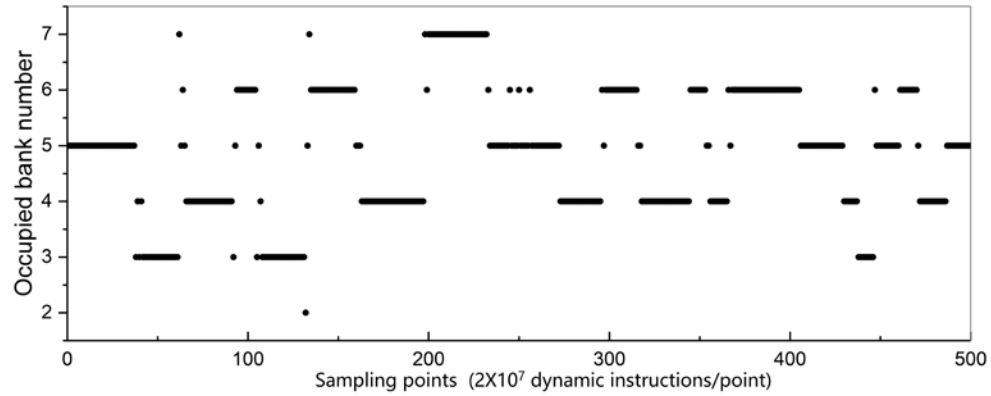


（b）Energy per access profiling with *bzip2* benchmark.

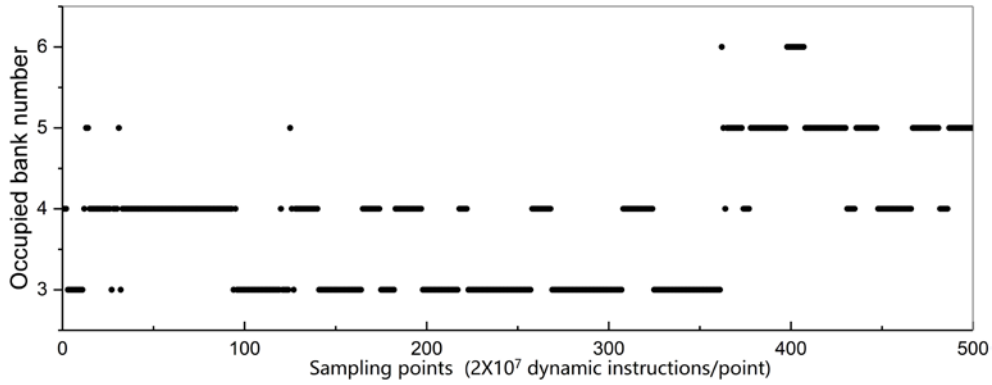


（c）Energy per access profiling with *soplex* benchmark.

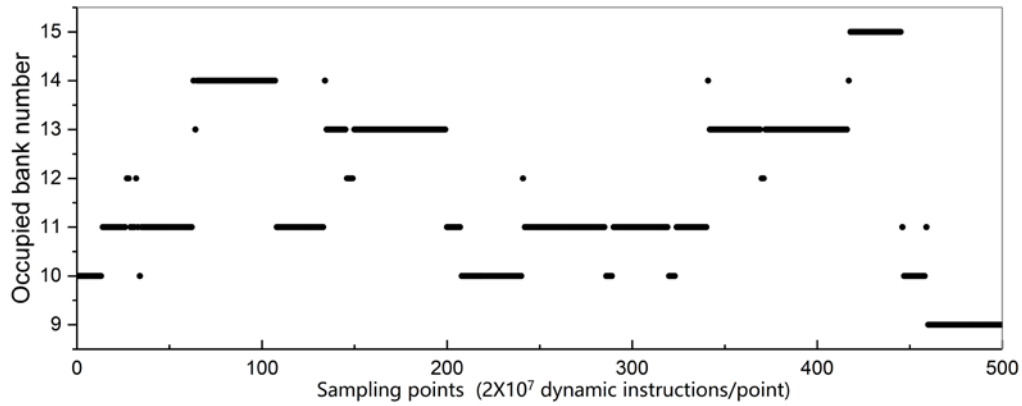
Figure 3-7. Sampling on energy consumptions. Where each sampling point in X-axis stands for a set of twenty million dynamic instructions, and fixed $C_{app(x)}[0]$ allocation method and proposed controllable method are profiled.



（a）Controlled bank number profiling with *gcc* benchmark.



（b）Controlled bank number profiling with *bzip2* benchmark.



（c）Controlled bank number profiling with *soplex* benchmark.

Figure 3-8. Runtime controlled bank allocation. Three benchmark *gcc*, *bzip2* and *soplex* are selected to profile runtime control outputs. Each sampling point in X-axis stands for a set of twenty million dynamic instructions.

cache is dynamically allocated with the outputs of cache allocation control. Similarly, the *bzip2* benchmark is initially allocated with four banks as the approximately optimal bank allocation number to shows the runtime features of the rapidly increasing application, and six hot subroutines of this benchmark are selected to apply their calls as control intervals and further the runtime cache is

dynamically allocated with the outputs of cache allocation control. And the *soplex* benchmark is initially allocated with eleven banks as the approximately optimal bank allocation number to shows the runtime features of the sharp

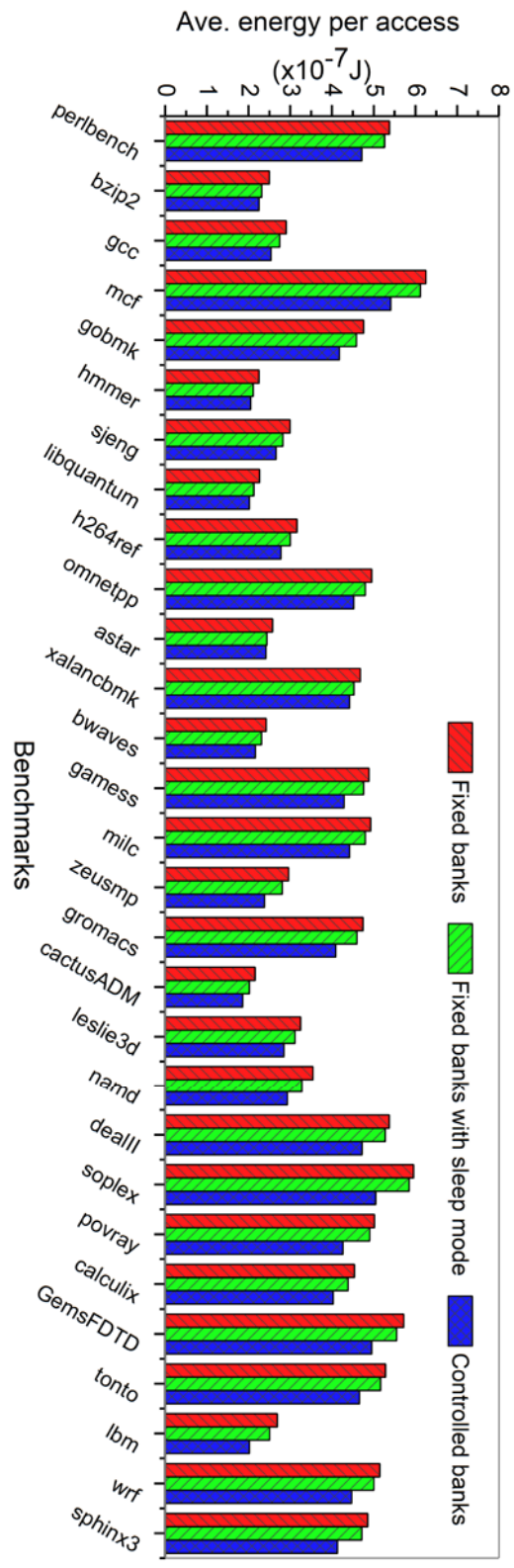


Figure 3-9. Energy comparisons of three methods. Where all benchmarks are simulated to profile average energy consumptions with different cache allocation methods.

increasing application, and eight hot subroutines of this benchmark are selected to apply their calls as control intervals and further, the runtime cache is dynamically allocated with the outputs of cache allocation control. As shown in Fig. 3-7, the energy consumptions in both fixed bank and controlled bank allocation methods are represented. The results in *gcc* benchmark are shown that energy consumptions with fixed bank allocated are averaged as 0.285μJ, and energy values in the period between 210th sampling point to 230th point reach the maximum energy consumption as 0.397μJ, but in the period between 110th sampling point to 125th point the energy consumptions are the minimum one as 1.936μJ. Correspondingly, the energy consumptions with controlled bank allocation are averaged as 0.249μJ, and energy values in the period between 210th sampling point to 230th point is about 0.291μJ, but in the period between 110th sampling point to 125th point the energy consumption is about 1.685μJ. The above particular appearances on curves indicate that those profiled energy consumptions in controlled bank allocation are always smaller than the ones of fixed bank allocation and its curve is also much more stable than the one of fixed bank scenario. Moreover, the energy values in *bzip2* benchmark are averaged as 0.247μJ with fixed banks allocated and with controlled banks allocated energy values are averaged as 0.215μJ. As to the *soplex* benchmark, energy values in the fixed bank and controlled bank allocation methods are about 0.596μJ and 0.489μJ, respectively.

Furthermore, the controlled bank allocation results during runtime are profiled as shown in Fig. 3-8. It can be affirmed that the runtime outputs of controlled bank processes are varying along with locality change, and the allocated bank number is controlled to change just as such locality and further meet the cache allocation demands dynamically. As a result, the energy consumptions with controlled bank allocation can be saved because the runtime cache bank demands are always supplied with controlled outputs.

As shown in Fig. 3-9, the energy consumptions of all SPEC benchmarks are represented with three cache allocation methods applied. In the results of fixed bank allocation scenario, the average energy consumption of all benchmarks is

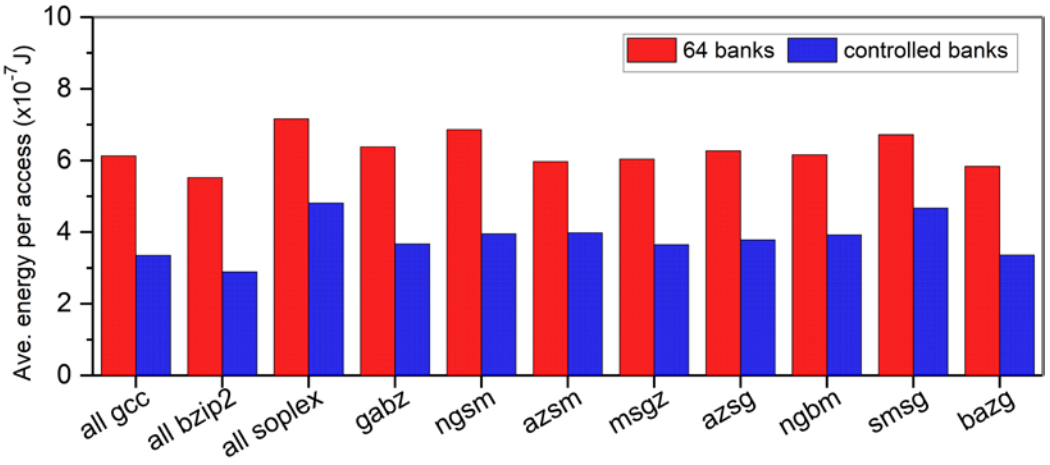


Figure 3-10. Energy sampling on mixed workloads. Where full bank allocation method is set as base ones.

about 0.407μJ. If t

.392μJ. With controlled bank allocation, the average energy consumption is about 0.348μJ. Clearly, the proposed method can save on average of 17.3% and 13.6% energy consumption over fixed bank allocation and fixed bank allocation with sleep mode, respectively.

In order to evaluate the efficiency of proposed method in parallel execution situation, eight benchmark groups where each group contains four benchmarks are employed to co-run in the four threads concurrently. As some integer

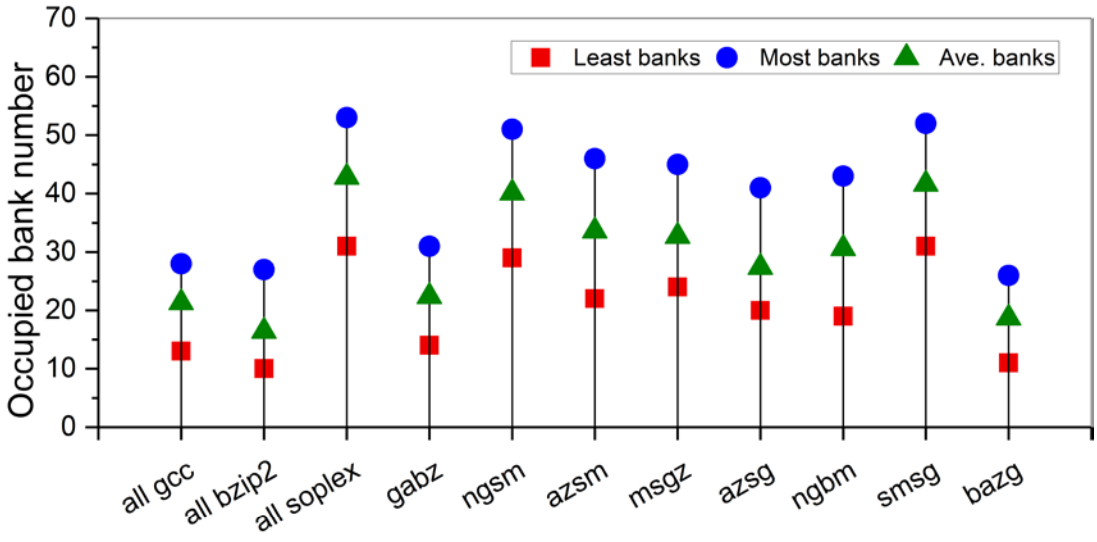


Figure 3-11. Controlled bank allocation statistics. All values are averaged by profiling runtime occupancy on all control intervals.

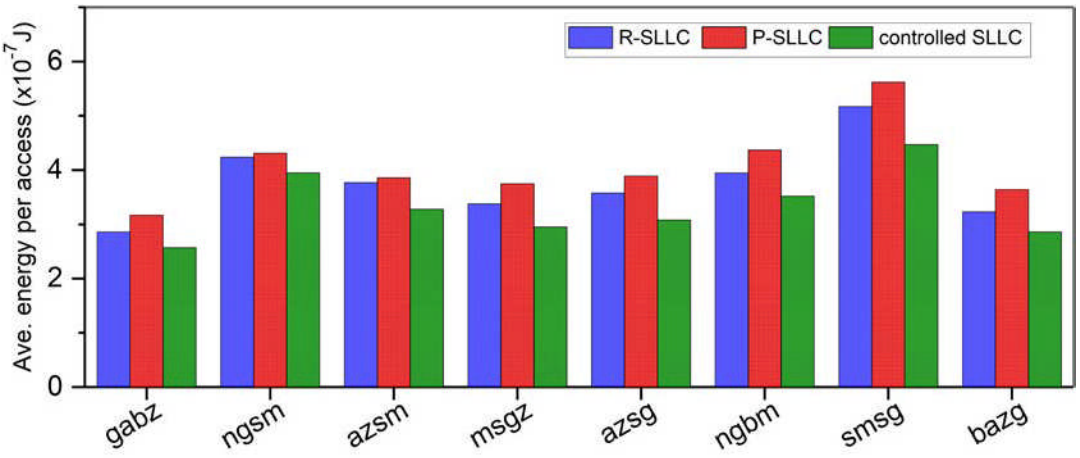


Figure 3-13. Energy comparisons on three methods. The controllable SLLC method is simulated with same simulation conditions of R-SLLC and P-SLLC methods.

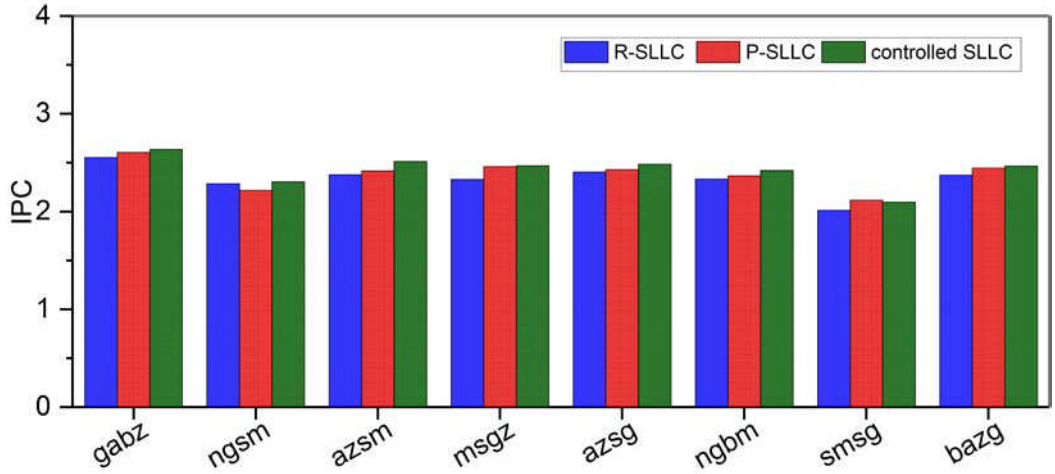


Figure 3-12. Performance comparisons on three methods. Where all methods are simulated under conditions.

benchmarks are not suitable in the multi-processor executing situation, each thread is allocated with the same working set copy from some benchmarks during runtime to form a single thread situation, and the simulation results are calculated as the real parallel executing situation.

As shown in Fig. 3-10, the energy consumptions with all 64 banks allocated are averaged as 0.627μJ, and if the cache banks are controlled by proposed method, the energy consumptions are averaged as 0.382μJ. Typically in the groups of *all bzip2* and *smsg*, the energy consumptions can be saved as the most

47.92% and the least of 31.86%, respectively. Meanwhile, the average energy saving can be 39.71%. Correspondingly, the occupied bank numbers are profiled into Fig. 3-11 during runtime. Clearly, the cache bank demands are varying along with execution sequence and the variation ranges tend to be very large. For example, the occupied bank numbers vary from 19 to 43 on *ngbm* benchmark group. As a consequence, at most 46 banks are redundant with no benefits but wasting energy consumption and delaying cache accesses.

Fig. 3-12 and Fig. 3-13 show energy consumptions and performance values in three design methods, respectively, where R-SLLC stands for the reconfigurable SLLC method [6, 9, 20], the P-SLLC stands for phase-based partitioning SLLC method [2, 11, 18]. The energy consumptions of R-SLLC, P-SLLC, and controlled SLLC are on average of 0.377μJ, 0.407μJ and 3.34μJ, respectively. And IPC values of R-SLLC, P-SLLC and controlled SLLC are on average of 2.33, 2.38 and 2.42, respectively. It can confirm that the proposed cache allocation method can save on average of 11.6% (value of R-SLLC as base value) and 18.2% (value of P-SLLC as base value) energy consumption over R-SLLC method and S-SLLC method, respectively. And the IPC of controlled SLLC has a few improvements over R-SLLC and P-SLLC on average of 3.9% and 1.7%, respectively.

3.6 Summary

In per-application granularity, an approximately optimal cache allocation can be easily achieved by exhausting all simulation results of the benchmark with different bank numbers allocated. Along with executing sequence, the profiled energy per access values in five hundred sampling periods show drastic variation in values. Similarly, runtime cache resource demands in per-interval granularity also behave diverse features, and fixed bank allocation method is verified that such a method cannot be adapted in all intervals. In other words, each interval has its own cache resource demand. Based on particular amplitude of improvement variation along with allocated bank increasing, all benchmarks can be classified into three patterns. As shown in three examples, *gcc*

benchmark behaves a little insensitivity with bank allocation, *bzip2* benchmark is sensitive with extra bank allocation and access pattern on *soplex* benchmark shows a phase-step based curve characteristic. As a result, both hit rate and energy consumptions are highly related on cache resource allocation, and two appearances are verified that there exists one optimal bank allocation point for each benchmark while at this point, energy consumption tends to be the lowest one while the hit rate tends to be remaining stable at the approximate maximum value. Furthermore, such optimal value can be evolved to applying for all intervals. The other appearance is that allocating bank number from one to the optimal number incrementally, hit rates are increasing rapidly and energy consumptions follow to be decreasing greatly, and in case of allocating banks from the optimal bank number to all bank allocated, the hit rates of all three patterns almost remain stable and energy consumptions are increasing as the hardware overhead scale. Obviously, those appearances confirm to the feedback-based control process.

Hence, the cache resource demands are described as the optimal cache allocation which represents the true requirement on achieving both low energy consumption and access latency in the granularity of per subroutine call. In other words, more banks are needed in case that allocated bank number is less than the resource demand, and in case of exceeding the resource demand, redundant banks should be removed off the current allocation dynamically during runtime, thereby optimal cache allocation is supplied for each subroutine call. And the output responses of controlled cache co-scheduling will be convergent to the steady-state value and any disturbance (locality change) can be handled in negative feedback control path. As a consequence, it is verified that plenty of energy consumption can be saved with controlled outputs over the fixed bank allocation, and also the energy consumptions behave much stable.

Moreover, proposed interval design actuates as the internal factor on the improvement of applying controllable cache allocation, because that subroutine calls have similar locality behaviors and also cache resource demands, thus, the optimal allocation on a subroutine call can be successive to other intervals,

thereby achieving precise cache allocation with minimum allocation overhead. As to previous intervals, the exploring process should be always running with plenty of energy overhead. However, the proposed control loop only generates one output corresponding to one selected subroutine and the rest thousands of calls on this subroutine are mostly allocated with the same cache allocation output. Hence, the energy consumptions of controlled cache allocation are far less than both reconfigurable allocation method and partitioned allocation method.

Chapter 4

STACKED 3D ON-CHIP CACHE NETWORK

4.1 Introduction on Cache Access Optimization

In modern CMP systems, shared caches serve concurrently with many cores, and they act as the key devices for bridging shared interconnecting traffic. However, those processing cores are concurrently accessing the only shared last level cache, and all those units are crowded at a two-dimensional chip area, which may cause serious congestions among so many devices. Moreover, the crossbars and interconnecting wires between many on-chip devices are forced to link each other with extreme long physical distance, thereby interconnecting latency is unbearable in efficient on-chip platform design. Furthermore, the complex on-chip cache hierarchies can take even more than half of both the on-chip area and energy consumption [46, 50].

To ease on-chip interconnecting traffic, recent researches try to reduce the physical distance of on-chip devices by extending the multi-layer architecture to the third dimension so that processing cores, cache hierarchies, and off-chip memory can be stacked together into three-dimensional architecture. Thus, each processing core can directly link to its private cache parts, shared cache parts, and off-chip memory to form a closely interconnected processing path, while the physical wires linking upper and lower layers are replaced with Through

Silicon Vias (TSVs). Hence, the interconnecting latency of stacked architecture is far less than that of traditional two-dimensional architecture.

Since the stacked architecture is recently employed as the key technology to optimize on-chip integrating distribution, some concomitant issues arise to limit the efficiency of such architecture, typically in tracing locality behaviors and managing concurrent accesses, which are highly related with the output traffic among upper and lower layers. Moreover, the shared last-level cache needs to serve many threads with a vast amount of shared data flows, which may cause runtime congestions and data coherence problems as many shared data will be existing in multiple partitioned shared cache bank groups [37, 59]. Hence, some researches tend to solve the defects of stacked architecture by shared cache partitioning methods [2, 31], shared data management methods [13, 14, 52] or OS-based allocation methods [63], which can partly ease the on-chip shared data traffic among threads.

In this chapter, novel shared data management methods are proposed to form the dedicated cache routing networks on private cache level [71] or shared cache level [72] under another angles, which focus on the access paths in the granularity of the per-shared-data-access level. Firstly, detailed pre-experimental simulations are set to analyze the characteristics of shared cache accesses, and further all shared cache accesses are classified based on particular characteristics of processing path and access distributions in each access type. Thus, six kinds of access types are selected to cover up all shared access hits, and the potential improvements on the processing path of each type are analyzed for the purpose of filtering and further optimizing those processing paths. Secondly, conventional router structure is enhanced to support data interaction by a cache to cache network that bridges the vast amount of data interconnecting traffic in the manner of parallel transportation. The proposed router is enhanced with three functions as access type recognition, a consistency processing module and a data multi-direction delivering module corresponding to the steps of shared data processing path. And the enhanced modules work in a concurrent path, thereby causing very few latency overheads. Thus, many shared data

accesses can be filtered and further be handled in proposed concurrent paths, and the latency of handling each filtered access is much lower than that of a shared cache access. Moreover, this work uses IC compiler tool to actualize the router implementation for the purpose of generating the layout details. And further, cache network design is integrated within on-chip system model for verifying the efficiency of such a network on both energy consumption and performance improvement. Finally, simulation results show that the performance represented by IPC can be improved on an average of twenty-six percent and the energy consumption can be saved on average of ten percent over the baseline platform.

4.2 Access Issue Description and Motivation

As shown in Fig. 4-1, a stacked 3D on-chip architecture is represented as an example to motivate the shared cache network design. In this architecture, the first layer is stacked with processing cores and private caches, where each private cache is linking with one enhanced router, and further those routers are linking to their partitioned cache bank groups through TSVs. And those routers can consist of a data interaction network that transports shared data among private caches or among shared cache. Meanwhile, the shared cache is stacked in the middle layer and each partitioned bank group is linking to one conventional router, which stretches out to the bottom layer for linking to memory.

4.2.1 Access Issue Classifying

To classify cache accesses into different patterns, the PIN [17] analysis tool is employed for recognizing each access and further counting them into proportions. Based on the pre-experimental analysis of shared access features, the access patterns are identified as follows.

As shown in Fig. 4-1, assume that Core0 wishes to load *DataX* from Bank1 which has a copy of *DataX* from off-chip memory. Concurrently, another thread of *CoreN* is desiring to load *DataX* also. Based on the conventional processing

Figure 4-1. Proposed 3D on-chip architecture. Where each processing core unit and its linked private L1 cache are connected with a router in the first layer, and the shared cache including cache controller and directory controller is partitioned to each private unit in the middle layer, and further reach out to a router to connect with the off-chip memory in the bottom layer.

path on shared data, all shared cache banks are partitioned to each private processing core unit in the form of cache bank groups, and the data in current bank group cannot be accessed by other private processing core units. Hence,

there will be a miss access in *CoreN* if its cache bank group has not contained the *DataX* in its partitioned bank group, and then *DataX* should be loaded from off-chip memory to the partitioned bank group, thereby consuming polyploidy access latency. Moreover, after the cache bank group of *CoreN* has loaded the *DataX*, such data is shared by two threads and any write access on this data will cause data incoherence, and many clock cycles are needed to maintain those data. For example, a write to *DataX* should be updating to all its old copies (or write invalidate). However, conventional maintaining methods require complex processes and devices to monitor all shared data, and maintaining latency is unacceptably large. Hence, a new interconnecting repeater should be proposed for bridging among any two cache bank groups or any two private caches for routing shared data in parallel.

Based on observations from Fig. 4-1, the shared data can only be interconnected in the path which is routing among different layers. However, the shared data is not allowed to interconnect among partitioned cache bank groups. And the routers in the upper layer work much faster than the off-chip memory on the shared data which exists in other bank groups if those routers are enhanced to route shared data across bank groups in parallel. Hence, the novel router is proposed in this thesis to bridge the shared data among partitioned bank groups. For example, *CoreN* can load *DataX* in a processing path as the steps: lookup for *DataX* in Bank1, then deliver the *DataX* from Router0 to *RouterN*, and then return the *DataX* to the thread of *CoreN*, thereby avoiding the previous miss access. Furthermore, how many shared accesses can be filtered out is highly high correlated in the proposal efficiency, and the proportions of filtered shared cache accesses are represented in the following subchapter.

4.2.2 Access Issue Distribution

In order to classify each shared cache accesses into several types, the PARSEC benchmark suite [54] is employed in the simulation environment which is consisted of two processing core units, two eight-bank-partitioned bank groups,

and other interconnecting logics. Thus, each shared cache hit is compared with real-time previous contents of shared cache, and thereby each hit type is counted into its access distribution amount. Hence, the access distributions are counted into different access types, which can be defined based on three access path features as follows.

(1) If current shared cache access has objective data in both shared cache bank groups of two processing cores, such access is classified as shared accesses. As shown in Fig. 4-2(c), the copies of previous target data are existing in two executing threads. Thus, a write access to the target data will cost plenty of clock cycles to update all old data in the conventional path. However, the new data can bypass to private cache of sharer thread in the proposed path.

(2) If current shared cache access only has objective data in the cache bank group of the other processing core, such access is classified as crossed accesses. As shown in Fig. 4-2(b), the copies of previous target data are existing in other executing threads. Thus, the requester will encounter a private cache miss and further load target data from the next layer in case of the conventional executing path. Meanwhile, the new target data from the requester should be delivered to

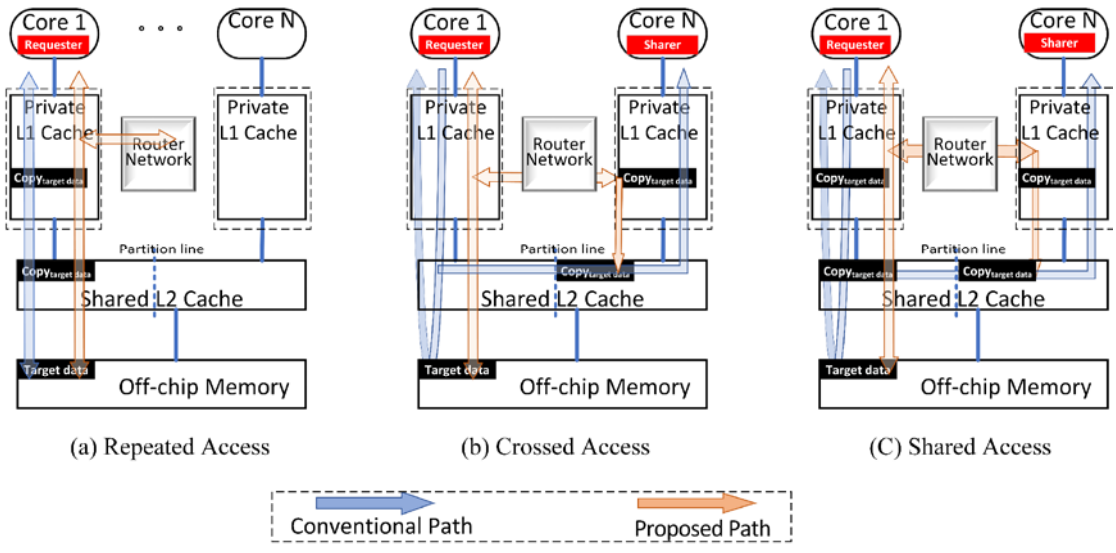


Figure 4-2. Schematic diagram of cache access classifications. (a) Repeated access patterns; (b) Crossed access patterns and (c) Shared access patterns.

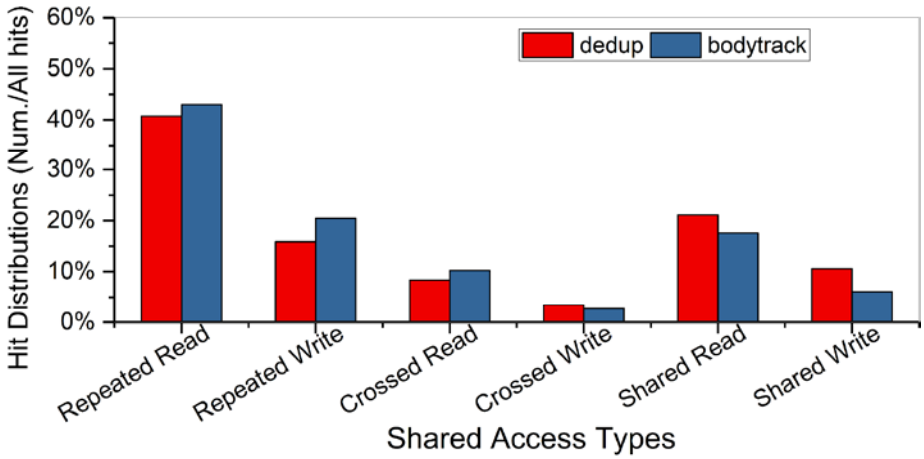


Figure 4-3. Distributions of hits on access patterns. Where *dedup* and *bodytrack* benchmarks are employed as examples.

update old copies in sharer threads. In the proposed path, the target data copies of other threads will be sent to requester in virtue of enhanced cache network, thereby saving many clock cycles.

(3) If current shared cache access only has objective data in the cache bank group of current processing core, such access is classified into repeated accesses, including repeated read accesses and repeated write accesses. As shown in Fig. 4-2(a), target data can be echoed by private cache (conventional path) or record entries of the router (proposed path).

As a result, there exists six kind of accesses, that is, write and read access for each type of three, and they are classified to further count access distributions except for pure miss accesses (target data are not existing in current cache). After integrating the PIN analysis tool into the simulation platform, sixteen record counters are employed to trace all shared cache accesses which are requested from two processing cores. All the benchmarks from PARSEC benchmark suite are simulated to account runtime accesses in the platform which has two processing cores, sixteen shared cache banks.

As shown in Fig. 4-3, two benchmarks, *dedup* and *bodytrack*, are used as examples to show the hit distributions of six access types. The shared write accesses take the proportion of 10.5% (*dedup*) and 6.1% (*bodytrack*), which

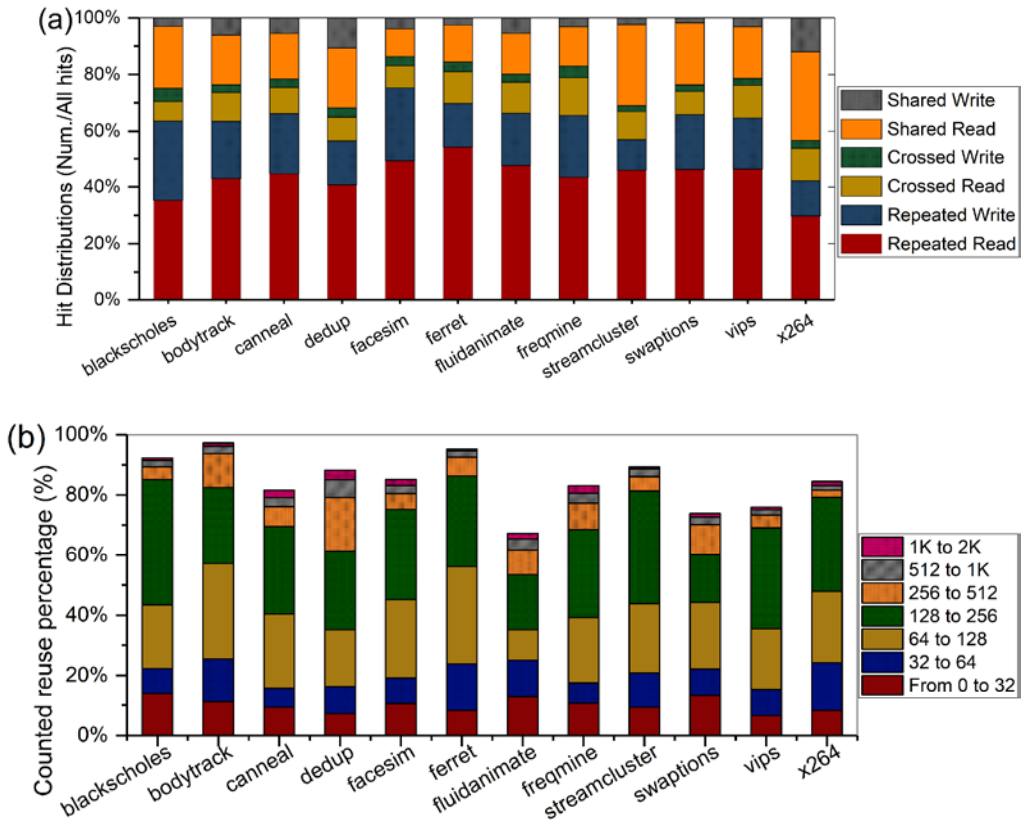


Figure 4-4. Hit distribution and reuse distance of each access patterns, where valid accesses are counted from the pattern set of hit numbers at each partitioned bank groups. (a) Statistics of hit distributions among all benchmarks; (b) Reuse distance along with reuse entries varying.

may cause coherence problem. And the hit distributions of shared read accesses take the proportion of 21.1% (*dedup*) and 17.6% (*bodytrack*), which can be filtered for short access latency. The hit distributions of crossed write accesses take the proportion of 3.4% (*dedup*) and 2.7% (*bodytrack*), which may be shifted to shared write access. The hit distributions of crossed read accesses take the proportion of 8.4% (*dedup*) and 10.2% (*bodytrack*), which can be passing across the bank group of other processing core unit rather than off-chip memory. The hit distributions of repeated write accesses take the proportion of 15.8% (*dedup*) and 20.4% (*bodytrack*), which may be a continuous write on the same data. The hit distributions of repeated read accesses take the proportion of 40.7% (*dedup*) and 42.9% (*bodytrack*), which can be reused with previous hit information.

Fig. 4-4(a) shows access distributions of all benchmarks, three appearances can be revealed as follows.

(1) Shared read and write accesses take a proportion of average 18.6% and 5.2%, respectively, which may cause serious access conflictions in shared data.

(2) Crossed read and write accesses take about 9.7% and 3.6%, respectively, and those accesses will encounter misses in their partitioned bank group but hit at the other group.

(3) Repeated read and write accesses take about 43.8% and 19.1%, respectively, and those accesses will quite frequently hit at the partitioned bank group.

Those access distribution values are counted by using large enough record entry set only for simulation purpose, in which all cache accesses can be captured. However, it is too costly to use large record entry set in real CMP systems due to large hardware overhead. Thus, reuse distance features along with record entry set varying should be explored to find the suitable record entry scale. As shown in Fig. 4-4(b), the record entry set with only 32 entries can count an average of 10.1% of all accesses as reuse accesses (accesses hit at record entry set). If cache accesses are counted by 256 record entries, about 72.6% (accumulated from 0 to 256) of all accesses can be classified as reuse accesses. Even adding record entry set from 256 to 2K, only an extra 4.73% of reuse accesses can be achieved over that of 256 record entry set. Thus, there is a tradeoff on selecting suitable record entry set.

As a consequence, above features on access distributions indicate that the potential improvement in reducing access latency and saving energy consumption can be achieved if shared accesses can be linked across bank groups, crossed accesses can get target data from the other bank group and repeated accesses can be filtered into a recording table for data reuse. Thus, the upper layer router network acts as a suitable operation object to bridge across bank groups, because all data and requests are interconnected by the router network. Meanwhile, shared cache accesses can be relayed in a concurrent

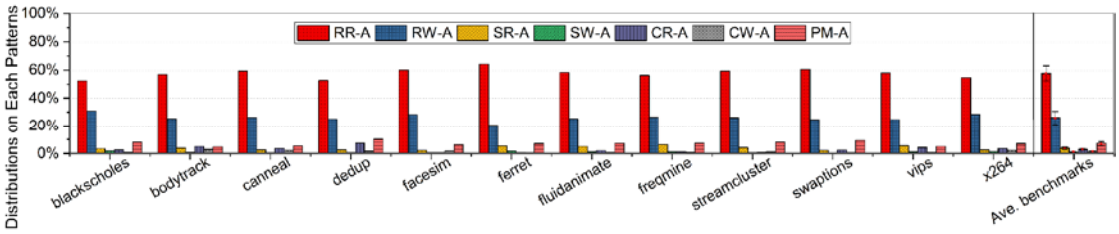


Figure 4-5. Distributions of hits on private cache access patterns, where valid accesses are counted from the pattern set of hit numbers at each private caches. RR-A and RW-A stand for repeated read and write accesses, SR-A and SW-A stand for shared read and write accesses, CR-A and CW-A stand for crossed read and write accesses, and PM-A stands for pure miss accesses, respectively.

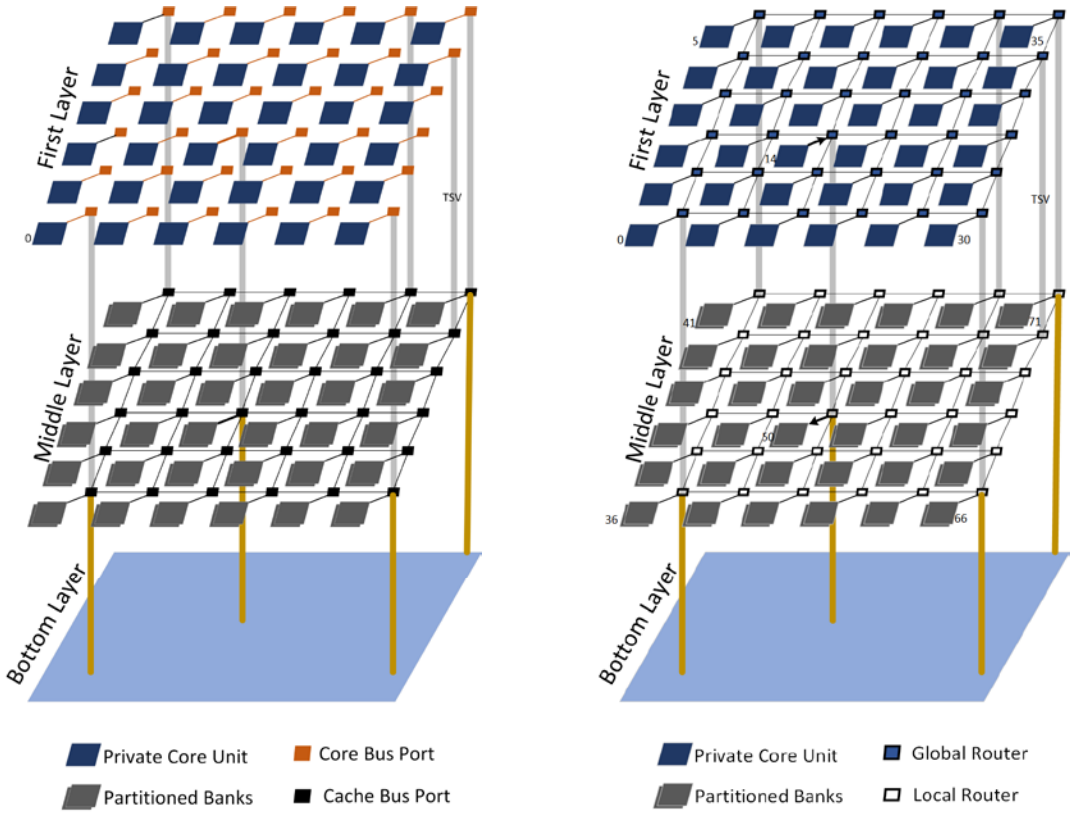
access path, and some selected cache accesses are recorded and further handled for efficient bypassing. In the next subchapter, the proposed router network is represented in detail.

As to access patterns in private cache level, hit distributions of private cache accesses are listed in Fig. 4-5, where seven access patterns are behaving various hit distribution values among benchmarks. Typically in crossed read (CR-A), crossed write (CW-A) and shared write (SW-A) accesses, those access patterns can hold the percentage of 2.98, 1.31 and 0.95, respectively. Although the proportions of the above three patterns are small over repeated accesses (RR-A, RW-A) and pure miss (PM-A) accesses, their coherence maintaining processes are so expensive that each access costs the access latency for dozens of times over common accesses, thereby causing serious performance degradation. Thus, a novel solution is proposed to ease those harmful accesses in this thesis.

4.3 Three-dimensional Cache Network on SLLC Level

4.3.1 Stacked Multi-layer System Architecture

Fig. 4-6(a) shows the three-layer baseline architecture, in which each private core unit only can link to partitioned banks in the middle layer by a core bus port and there is no inter-connection among private core units. As shown in Fig.



(a) Baseline architecture (b) Proposed architecture

Figure 4-6. Stacked 3D architecture design. (a) Baseline architecture; (b) Proposed architecture. Where all on-chip components and off-chip memory are stacked into three layers and each layer is linked by TSVs.

4-6(b), proposed architecture is built based baseline architecture. The first layer of proposed architecture is stacked with private core unit including private cache and processing core, and for each private core unit, it is allocated with a router to connect with other routers of the corresponding private core unit. Thus, a router network is formed to ensure data interaction among threads. Moreover, each router interlinks with one bank group which is partitioned from shared last level cache, and each bank group is managed with cache controller and directory controller logics, and also is connected with a router and further interlink to the off-chip memory. Note that each private core unit contains one processing thread and is allocated with one shared bank group which is equally partitioned from the shared cache, meanwhile, each bank group owns one cache controller

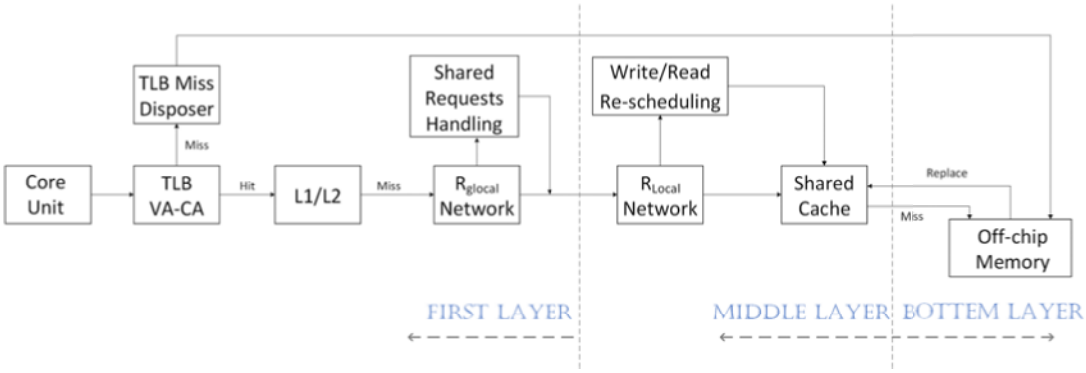


Figure 4-7. Access path of proposed stacked multi-layer system.

and one directory controller. As a consequence, the proposed router architecture is modified based on the baseline architecture which is enhanced with several functions and is implemented on the routers of the first layer.

4.3.2 Access Path Design in the Stacked System

To well describe the access path of dealing with access issues, Fig. 4-7 represents the usage of first-layer-set router network (global router) and middle-layer-set router network (local router) in the system [76]. In the first layer, the core sends out access requests and those requests in present virtual address (VA) mode can be converted as the physical address (PA) mode in virtue of a translation look-aside buffer (TLB) unit. If a request to private cache encounters a miss, this request will be sent to the linked global router, and then triggers two concomitant access processing mechanisms: 1) Send shared data package from top layer to middle layer. 2) Spread the request in a global router network for exploring parallel accessed requests which come from other private core units, and then reschedule the shared data request in the global router. Thus, latency overhead of proposed mechanism can be fully covered and save some cycles. Meanwhile, a hit in rescheduling mechanism will expend very few cycles, but also avoids queuing up at shared data accessing procedure.

4.3.3 Enhanced Router Hardware Design

To support data interaction among cache, proposed design consists of three important components over conventional router. Firstly, a new component is

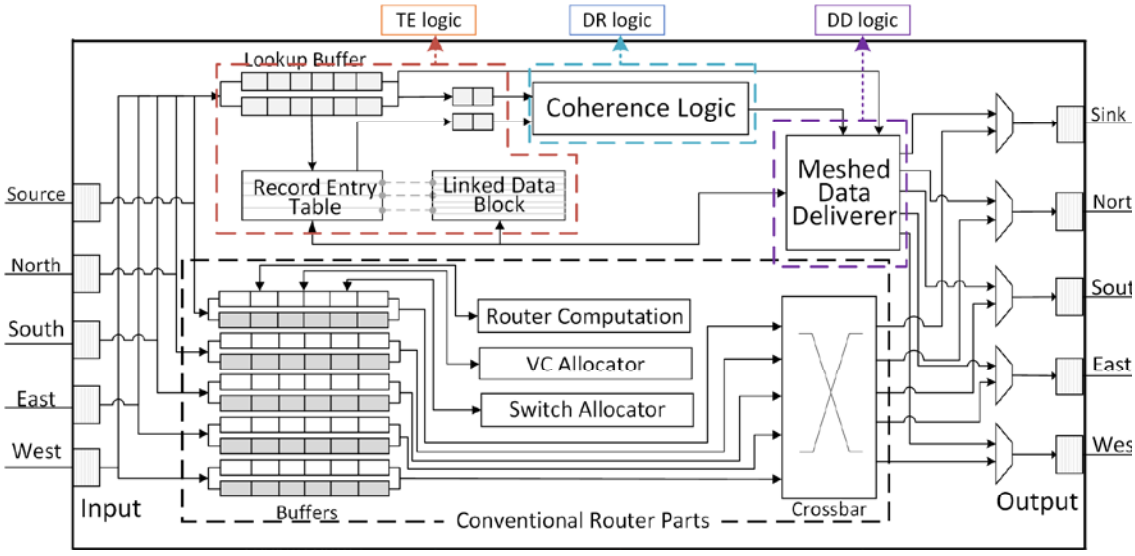


Figure 4-8. Proposed router architecture. Where both conventional routing pipeline and enhanced concurrent routing pipeline are concurrently dealing with five-port inputs.

designed to find shared data copies rapidly, and one component is needed to generate coherence state, and also one component is employed to interact shared data. Those components are described as follows.

1) Target accesses exploration (TE logic).

2) Data replacement (DR logic).

3) Data deliverer (DD logic).

Fig. 4-8 represents the structural diagram with above enhanced components [72]. In spite of the conventional router pipeline marked in the dotted box, the proposed components are designed to form a concurrent processing pipeline. Note that the concurrent pipeline works with conventional pipeline since each shared cache access arrives in the input ports, and once the target data has achieved in the concurrent pipeline, such shared cache access can be handled directly.

As shown in Fig. 4-9, the conventional router pipeline processes four states to deal with every package. If there is a package existing at the router ports, it will be loaded into write buffers first, and then be allocated with output ports in the switch allocation logic and an unoccupied virtual channel also. And then

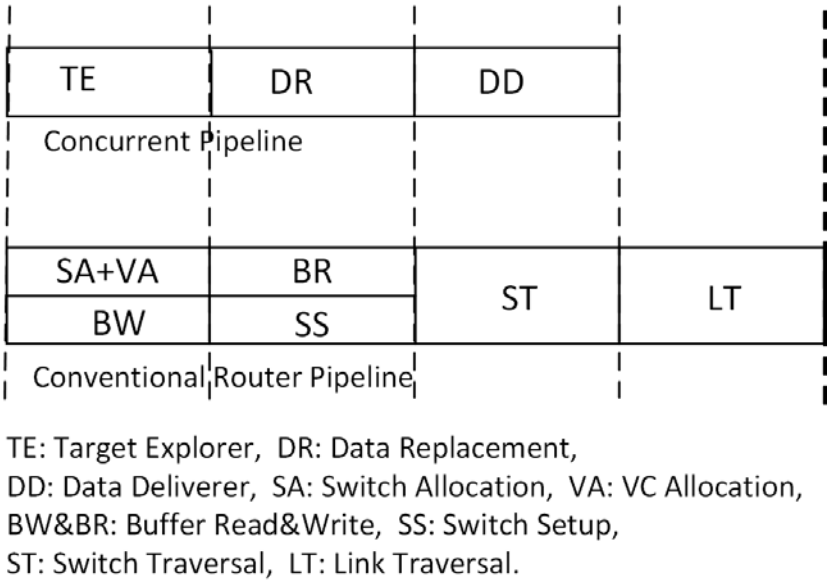


Figure 4-9. Routing pipeline stages on proposed concurrent pipeline (three states) and conventional routing pipeline (four states).

such package is loaded with traversal switch control and further is loaded by link traversal. Concurrently, the proposed pipeline is ordered in the sequence as follows. Firstly, the accesses are explored in the lookup buffer to identify which access type it belongs to, and further can be stored or loaded to the record logics in the state of target explorer. And next, this access is checked to ensure data coherence in the replacement coherence logic. At last, data deliverer can

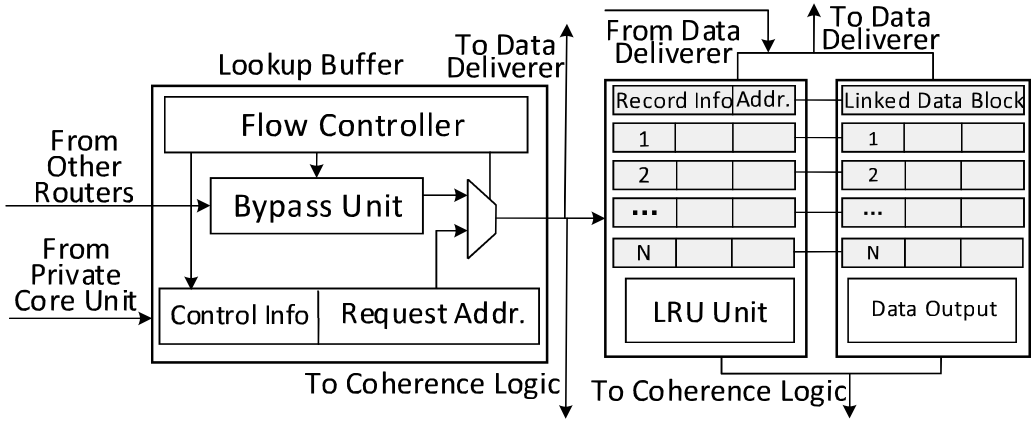


Figure 4-10. Proposed target explorer component (TE) including lookup buffers, record entries and table.

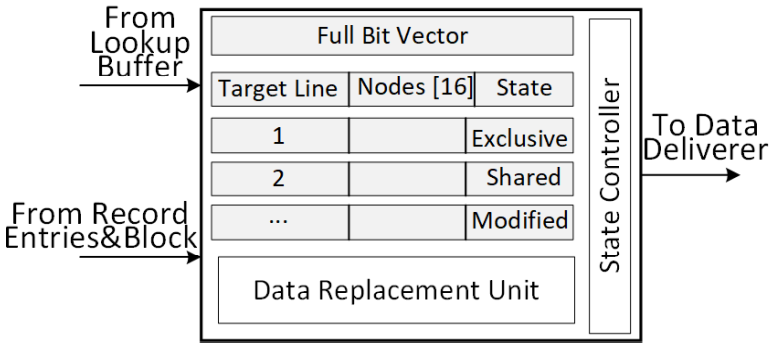


Figure 4-11. Proposed data replacement coherence component (DR) including a full bit vector based directory and data replacement unit.

transport target data to processing core unit in case of access hit on record logics, or such access is handled in conventional pipeline.

As shown in Fig. 4-10, the internal design details of target explorer are represented. For the accesses from other routers, a bypass unit is employed to quick explore among the record entries for the purpose of locating previous target data. If there is a hit, the target data can be checked in the next state, otherwise bypassing the current router. In case shared cache access is requested from the private core unit, the address of current request will trigger a lookup to identify whether there is a previous hit existing or not, and further such request is bypassing to the target explorers for checking data sharing. Moreover, all shared data should be checked data coherence first before such data is delivered to data deliverer logic. And the data record entries only hold some pervious hits while all entries are maintained with the LRU replacement policy. Meanwhile, the data record table will hold the target data of recorded previous hits corresponding to the contents of data record entries. Furthermore, the retired data from data record entries and tables should be checked the data coherence with the data stored in other routers, and then those data will be retired to shared cache bank group. In case that target data is delivered from other routers, such data will be stored into data record entries and tables.

Fig. 4-11 depicts the design details of DR component, which is designed for checking the consistency and marking directory state. It has two input ports, where one port receives data from lookup buffer into full bit vector based

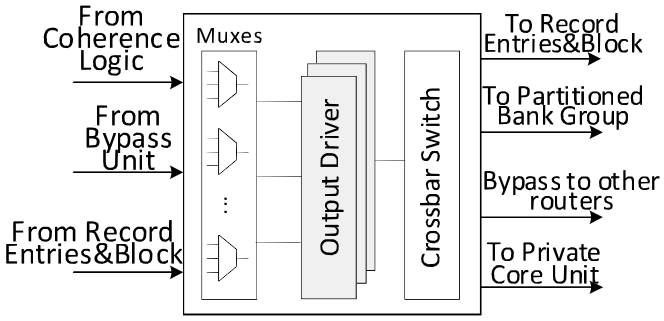


Figure 4-12. Proposed data deliverer component (DD) including selecting muxes, output driver and crossbar logics.

coherence algorithm and the other port is linked with data record entries and table. Moreover, the state controller dri0vers three kinds of operation as follows. Firstly, if target data will be stored into record logic, the coherence check process will be triggered to ensure that previous target data is same as the current target data. Secondly, each retired data from record logic to shared cache bank group will be checked first to ensure that the retired data is same as the one in the linked bank group. Finally, such data should be checked to ensure that such data is same as one of other routers.

As shown in Fig. 4-12, data deliverer (DD) is represented to receive three kinds of transporting requests from coherence logic, bypass unit, record entry table and linked data block. All those requests are selected in the mux array and further driven by output driver logic, and finally delivered to crossbar switch logic to four directions as record entry table and linked data block, linked partitioned bank group, bypassing to other routers, and private core unit.

4.4 Three-dimensional Cache Network on Private Cache Level

4.4.1 Modified Router Architecture

Data sharing in private cache level can cause much larger recovery latency because that those data are existing in high-speed private cache level and any data inconsistency will lead to expensive thread interruption. Thus, proposed router design is suitable to apply in private cache level for interconnecting threads. As shown in Fig. 4-13, a modified router is designed to work with

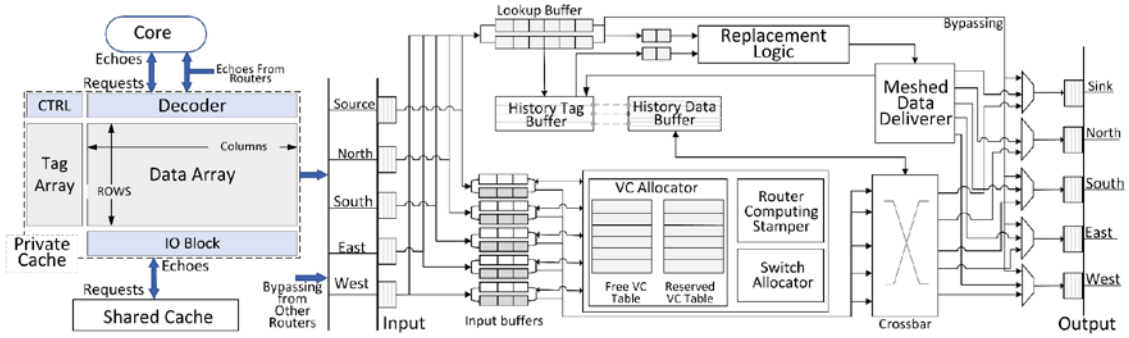
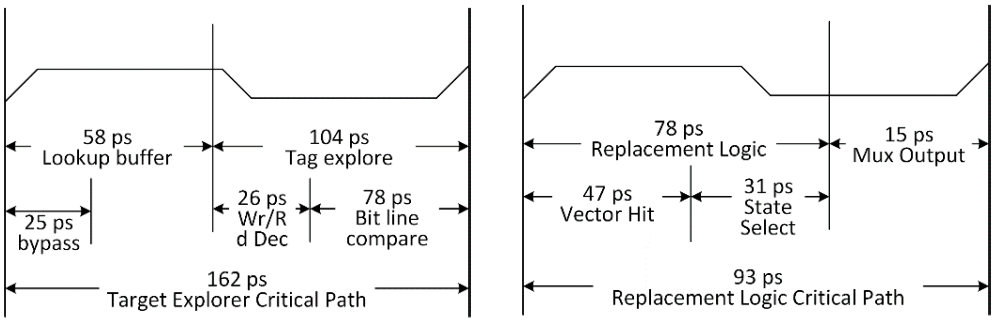


Figure 4-13. Modified router architecture, which is integrated into private cache level.

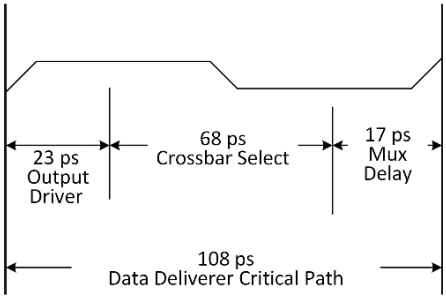
private cache in parallel [71]. If there is a hit at any thread, target data can be delivered to the requester or updated to sharer. And previous router structure is modified to meet routing requirements on private cache as follows.

(1) Each private access should be bypassed to the lookup buffers of all routers rapidly. Thus, a dedicated channel is employed to spread new requests in the router network.



(a) Latency evaluation on TE logic.

(b) Latency evaluation on DR logic.



(c) Latency evaluation on DD logic.

Figure 4-14. Latency evaluations on three components. (a) TE logic, (b) DR logic, and (c) DD logic.

(2) The virtual channel (VC) allocator is re-designed with two VC tables: free table and reserved table.

(3) Input and output buffers are greatly reduced for the dapper buffer scale, where the input buffered routing method is applied to this router.

(4) Both the history tag and data array are maintained with a new updating algorithm, which supports data replacement from other routers and shared cache.

(5) The replacement logic is integrated with MOESI_directory based coherence protocol, which is interlinking to the coherence logic of shared cache.

As a consequence, the modified router can support high-speed data interaction among executing threads in the proposed concurrent path. And routing paths of key access patterns are described in the next subsection.

4.4.2 Routing Paths of Four Patterns

Based on layout reports of router components in Table 4-1, access latency of each routing path can be evaluated by measuring the latency between source-CLK to the sink point. As shown in Fig. 4-14, access latency values in three key components are represented in detail. For access path in TE logic, tag bits can be bypassed with 25 ps and exploration in lookup buffer costs 58 ps. Data output process needs 104 ps including write or read decoding and bit line selecting. For access path in DR logic, the full bit vector needs 47 ps to decide data category and further endows data state within 31 ps, and finally, exports marked data in 15 ps. For access path in DD logic, output driver costs 23 ps to load target data from previous logic to crossbar unit, and crossbar selection costs 68 ps in latency, and finally exports data within 15 ps.

As shown in Fig. 4-15, latency savings of four access patterns are represented. Compared to access latency of conventional private cache path (assume 2000 ps), shared accesses and crossed accesses are flowing in proposed router network as follows.

(1) Shared read saving: If there is a request to target data of shared read access, the decoded tag information will be explored to find previous access

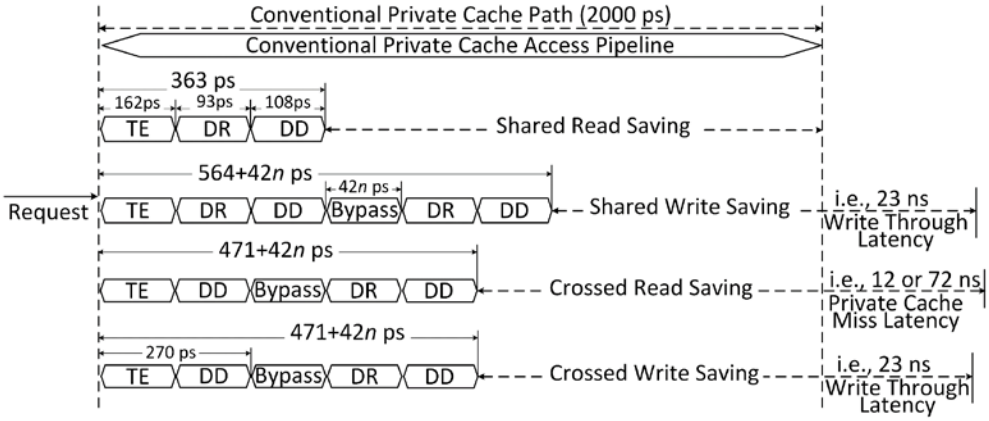


Figure 4-15. Latency savings on each access pattern. Where 42*n* stands for the latency of bypassing *n* routers and $n \leq 6$ for XY routing in a 4×4 router network.

within 162 ps. And this access is marked with coherence state, and at last, output data to requester with the aid of DD logic. Thus, the overall latency of shared read access is about 363 ps.

(2) Shared write saving: If there is a write request from the requester to previous target data, the decoded tag information will be bypassing to the rest routers at the same time. Then, target data copies in router network can be located for further updating. The new target data will replace old target data in TE logic, and then coherence state is marked. At last, the new shared data is bypassed to any related routers for updating their old data. Next, the coherence state will be marked and further write new data back to update all copies existing in shared cache. Thus, the overall latency of shared write access is about (564+42*n*) ps.

(3) Crossed read saving: If there is a request to target data of crossed read access, the decoded tag information will be bypassing to the rest routers at the same time. Then, target data copies in router network can be located for further exporting, and this access is marked with coherence state. At last, output new data from the router of any sharer to the current router of requester. However, there is a cache miss in conventional access path as the target data are not existing in private cache. Thus, the overall latency of a crossed read access is about (471+42*n*) ps.

(4) Crossed write saving: If there is a write request from the requester to previous target data, the decoded tag information will be bypassing to the rest routers at same time. Then, target data copies in router network can be located for further updating. The new target data will replace old target data, and at last, bypass new data to any related routers for updating their old data. Next, coherence state will be marked, and further write new data back to update copies existing in shared cache. Thus, the overall latency of a crossed write access is about (471+42n) ps.

To sum up, processing latency of conventional path in above cases can be 33.3~37.9, 5.5, 16.6~23.4 and 31.8~44.8 times (Calculate from n=6 to n=1) over the ones of proposed path, respectively. Moreover, those access patterns take up about 0.95%, 3.82%, 2.98% and 1.31% as shown in Fig. 4-5, respectively. Theoretically, overall performance employed with proposed router can be improved greatly.

4.5 VLSI layout verifications

4.5.1 Layout Setups

In order to evaluate the energy and latency of enhanced router architecture, the proposed router is implemented by the VLSI development process of hardware language description in Modisim tool, static timing analysis, synthesis and layout in Synopsys IC Compiler tool [77]. Cache model is built by CACTIv6.5 tool [55], where SLLC is set as 16MB size, 128 banks.

The pseudo-code of enhanced router logic is represented in Algorithm II. For each shared cache access, it will be checked in the lookup buffer to explore the previous target data (from line 5 to 8). If such access hits in the target explorer, the data can be transported to data deliverer and further return to private core unit, and if such access misses in current target explorer, a bypassing process is triggered to explore among other routers to locate same kind of accesses (from line 9 to 15). If such access hits at other routers, the target data is transported to current router expect that the access is detected as a

continuous write (from line 16 to 19). In case of a continuous write, this access should write to record entries and table first, and the previous write is retired to write at partitioned bank group, thereby continuous write is handled (from line 20 to 25).

ALGORITHM II

Router's Pseudo Verilog Code

Input CLK, Addr_Bus_In, Router_Mesh_in, Router_passby_In;

Output Request_data, Router_Mesh_Out, Router_Passby_Out;

```
1:  Wire define;
2:  Buffer define;
3:  Reg define;
4:  always @ (CLK)
5:  begin
6:  if (Request_addr arrives)
7:      lookup exploring;
8:      wave in global router network;
9:      if (a buffer hits)
10:         block deliver;
11:         return Request_data;
12:         data replacement in requested record entries and table;
13:     else
14:         access current router;
15:         wave in router network;
16:         if (hit at other local router&!continuous write)
17:             block deliver;
18:             return Request_data;
19:             replacement in requested record entries and table;
20:         else
21:             if (continuous write detected)
22:                 write to record entries and table;
23:             else
24:                 return Request_data from partitioned banks;
25:                 update lookup buffer;
26:     router bypassing;
27:     coherence check;
28:     update shared cache;
29: end
```

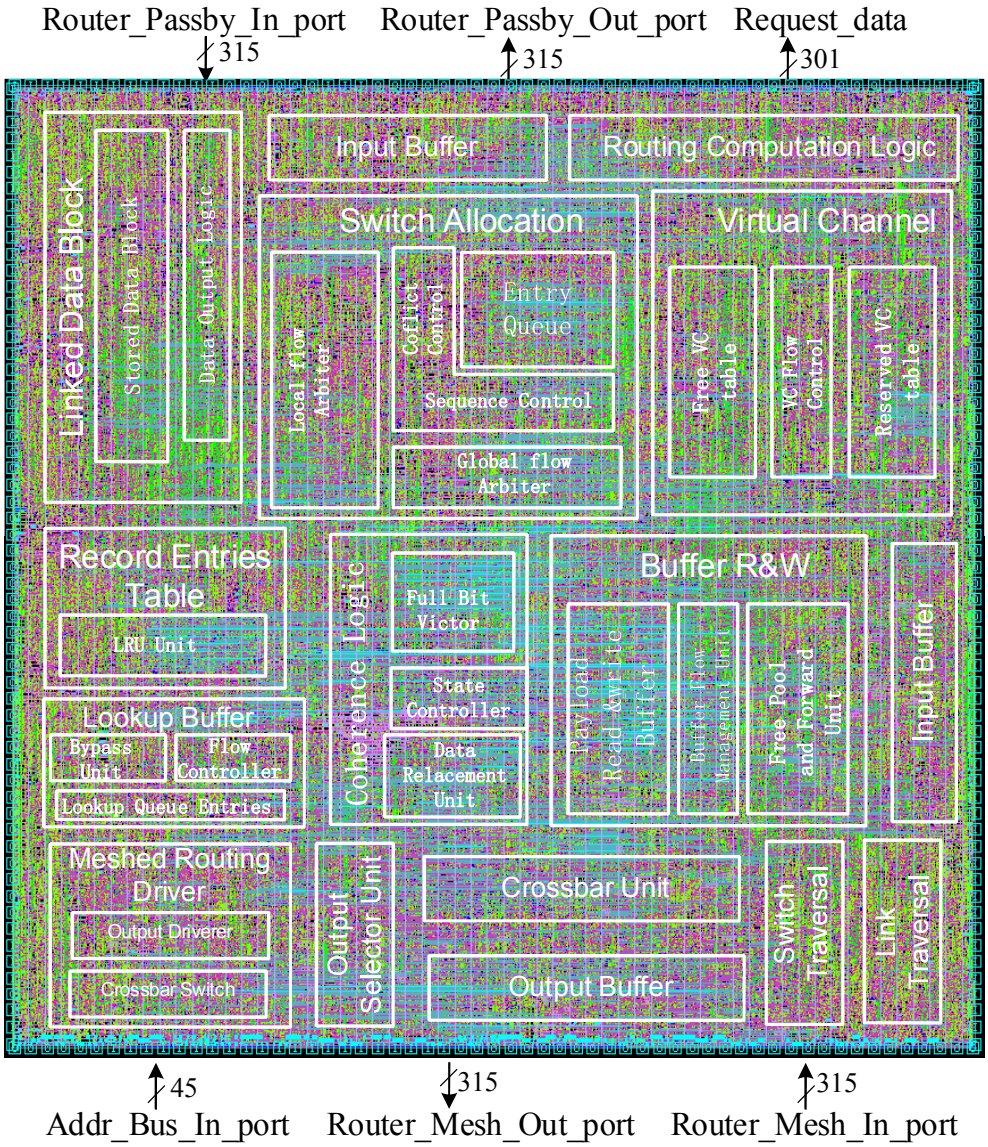


Figure 4-16. Router layout details. Ports to four transmission directions (N, S, E, W) are omitted.

Moreover, the router bypassing is employed to transmit target data of crossed access in the router network. And all shared data should be checked for data coherence, and the retired data will be checked and then should be delivered to current partitioned bank group (from line 26 to 28). Hence, some shared cache accesses can be filtered and further be handled in concurrent pipeline with very low latency.

4.5.2 Implementation Results and Overhead Analysis

As shown in Fig. 4-16, the layout diagram is represented on the design area

setup of 1.3*1.3 mm^2. Note that each component is highlighted to draw their probable occupied area, and details are marked with white frames. Moreover, the layout report is listed in Table 4-1, and the five-level logic hardware is counted with 165594 gates while the design area is about 1.5 mm^2 and the total dynamic power is about 56.9 mW. Thus, the total area of proposed router network which contains sixteen routers can be counted as 16*1.69=27.04 mm^2, and the total power is counted as 16*(total dynamic power + leakage power)=1.09 W. Moreover, layout reports for stacking private cache level are listed in Table 4-1. Single router hardware is counted with 113944 gates while the design area is about 0.68 mm^2 and the total dynamic power is about 48.6 mW. Thus, the total area of proposed router network which contains sixteen routers can be counted as 10.9 mm^2, and the total power is counted as 16*(total dynamic power + leakage power)=0.86 W.

Firstly, design differences of router structures between shared cache level and private cache level should be declared with three parts: (1) Path latency

Table 4-1. Layout Report Details [71, 72].

Name	Quantity	Values for Shared Cache Level	Values for Private Cache Level
Layout Setups	Clock frequency	533 MHz	2.0 GHz
	Supply voltage	0.9 V	1.1 V
	Technology library	45 nm	45 nm
Statistic Reports	Levels of Logic	5	5
	Number of ports	4852	2237
	Cell Count	24715	14189
	Gate Count	165594	113944
Area Reports	Non-combinational Area	0.367 mm^2	0.1182 mm^2
	Combinational Area	0.532 mm^2	0.2996 mm^2
	Cell Area	0.899 mm^2	0.4179 mm^2
	Net Area	0.625 mm^2	0.2637 mm^2
	Design Area	1.504 mm^2	0.6815 mm^2
Energy Reports	Cell Internal Power	31.5495 mW	31.1685 mW
	Net Switching Power	25.3889 mW	17.4368 mW
	Total Dynamic Power	56.9384 mW	48.6053 mW
	Cell Leakage Power	11.4590 mW	5.1749 mW
Clock Reports	Clock Sinks	4263	2865
	Longest Path	0.278 ns	0.262 ns
	CTBuffers	1165	729

requirements are very different as several nanosecond level in shared cache but several hundred picosecond level in private cache; (2) On-chip locations are very different as lying after private cache or in parallel with private cache; (3) Router architectures are very different as router in private cache level is redesigned including router model (input-buffered architecture), three additional components and data maintaining path for supporting several times fast access speed.

To compare additional hardware over conventional cache network, we take the router network and shared cache together into account to compare with the hardware scale of our proposed router. To compare with a standard 16 MB size shared cache (gate scale is counted in CACTIv6.5), the design area of proposed router only takes up 1.5 mm^2 area. To compare with Intel Xeon E7-4850 processor, which possesses 16 processing cores, 115 W power and 45 mm * 52.5 mm=2362.5 mm^2 package area, the hardware overhead of our proposed router network is about 1.14% of the Intel Xeon chip, and the power overhead only takes up 0.95%.

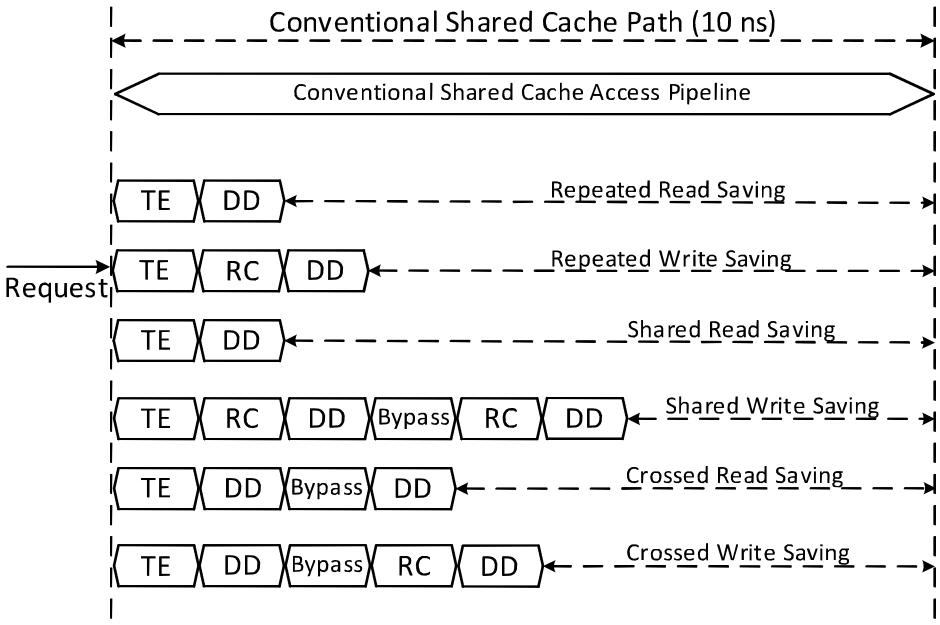


Figure 4-17. Potential latency saving of modified paths. Where the latency of each state is not represented in scale. TE: Target Explorer, RC: Replacement Coherence logic, DD: Data Deliverer.

As to the latency of each access type, the pipeline states of each type are represented in Fig. 4-17. For each hit on the proposed pipeline, the latency of such access is classified into six situations as follows. For repeated read access, it only needs to explore the target data in TE state and then the target data is delivered to processing thread, thereby saving plenty of latency. For repeated write access, it needs to explore the target data in TE state, and then the target data is checked with data coherence in RC state for saving some latency. For a share read access, it only needs to explore the target data in TE state and then the target data is delivered to private core unit in DD state, thereby saving plenty of latency, which has a similar process as the one of repeated read access. For a shared write access, it needs to explore the target data in TE state, and then the target data is checked with data coherence in RC state, and then the target data is delivered to private core unit in DD state, and further such access is bypassing to the routers which have the previous target data, and then the data will be checked with data coherence again, and further deliver the new target data to replace the ones of other routers, thereby saving plenty of latency rather than

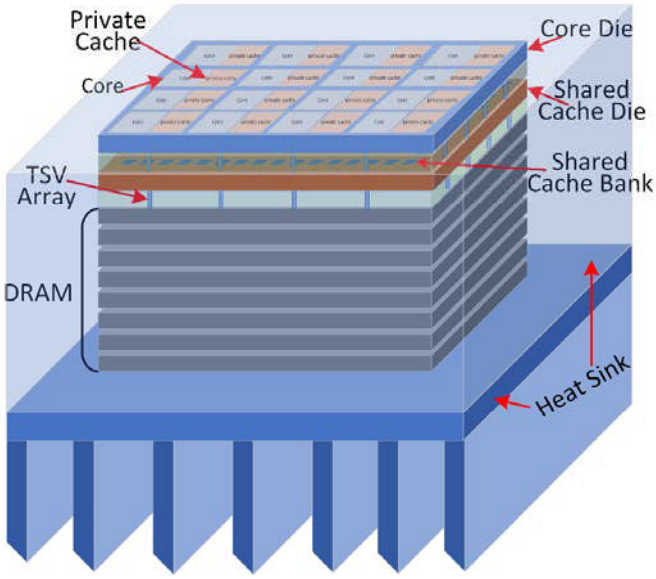


Figure 4-18. Chip design of a 16-core 3-layer stacked architecture, where the first layer is stacked with cores and private caches, the second layer is stacked with shared cache, and the third layer is stacked with multi-laminated memory.

the latency on both shared cache access and coherence check. Any crossed read needs to be explored with shared data existing in other threads, and then transport shared data to current router and further return the data to private core unit, thereby avoiding a shared cache miss. For a crossed write access, it will be explored with previous target data in other routers and then the data can be written to current record entries and table after this data is checked with the target data of other routers to ensure data coherence. And if data incoherence happens, the new target data is delivered to replace previous data in other routers, thereby saving plenty of latency rather than the latency of both a shared cache write access and data coherence checking.

4.5.3 Evaluations on 3D Stacked Chip

As shown in Fig. 4-18, the stacked chip diagram shows that three layers are embedded into a heat sink for purpose of heat dissipation, and 16 cores and private caches are stacked in the core die while each thread processes a TSV array stretching out to interlink the layer of shared cache die. Table 4-2 shows key parameter setups on the proposed 3D chip. The heat sink is far larger than chip thickness, as a result, heat generated from three layers can be dissipated greatly. And the heat sink has a thermal conductivity value of 350 W/(m-K). The memory layer is stacked with a multi-laminated structure, where the number of stacked data pages can reach to 128 layers or more. In this thesis, four data pages are stacked in the third dimension acting as a page group, and sixteen page groups are placed in a two-dimensional area. And each page group processes a TSV array for data transmission.

Table 4-2. Setups on proposed 3D chip.

Parameter	Value	Parameter	Value
Chip thickness	0.00026 m	Specific heat capacity	1.75 MJ/(m^3-K)
Thermal conductivity (Si)	110 W/(m-K)	Initial temperature	328.15 K
Stacked layer number	3	Ambient temperature	298.15 K
Heat sink thickness	0.0039 m	Temperature threshold	363.15 K
Thermal conductivity	350 W/(m-K)	Model grid	32×32

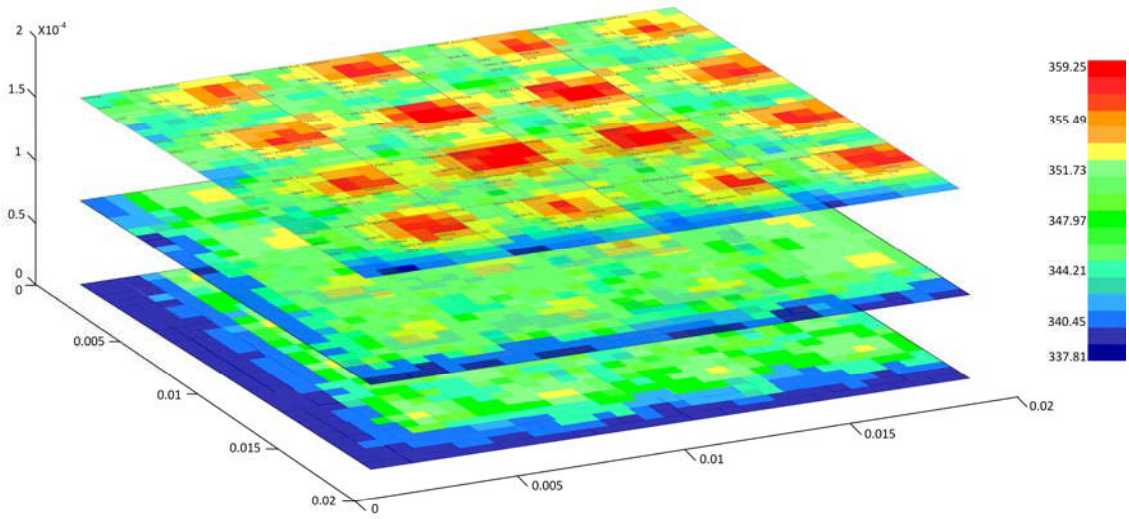


Figure 4-19. Steady thermal simulation on three stacked layers, where the units for the three-dimensional coordinates are Meters (m), and the unit of temperature is Kelvin (K).

The HotSpot tool (version 6.0) [78] is employed to conduct thermal verification in this thesis. As shown in Fig. 4-19, 3D stacked chip architecture is simulated with eleven hours to achieve a steady temperature state. And each layer is partitioned into 32×32 grids for modeling the average temperature in each grid. The temperature values range from 337.8 K to 359.3 K in three layers. Typically in the first layer, processing cores generate many heats to push up die temperature, which results in local overheating at core die. With the aid of heat sink, the maximal temperature of each grid is lower than 363.15 K (temperature threshold). Consequently, proposed 3D chip architecture is practicable under thermal constraint.

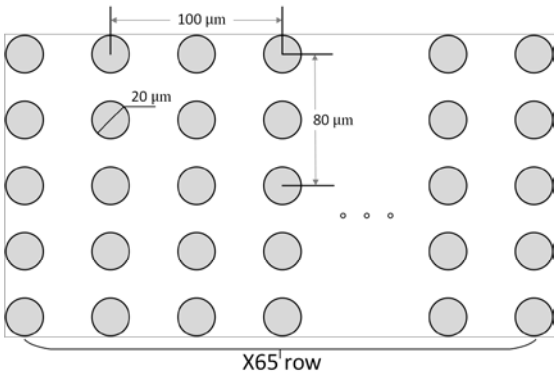


Figure 4-20. TSV placement example, where each TSV array contains 325 TSVs.

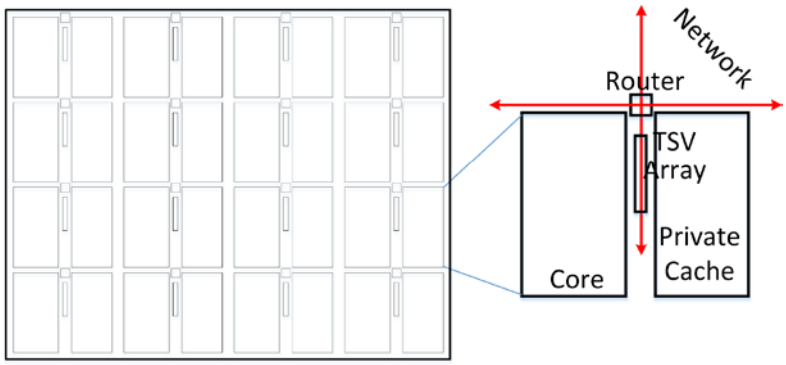


Figure 4-21. Placement example of the first layer, including a router, a processing core, a private cache and a TSV array.

As the data length spreading among routers is 325 bits, Each TSV array is designed to transport one data once in parallel. In this thesis, a 325-bit-long data address is packed as one routing package, and each package consists of five flits while the flit length is 65 bits. Thus, the TSV array is designed as a group of 5×65 TSVs. The placement of a TSV array is represented as shown in Fig. 4-20, where the overall area of a TSV array can be calculated as 0.59 mm^2. Fig. 4-21 shows a placement example of the first layer, where one router is interlinked with nearby routers to build a 4×4 network, and a TSV array is used to interact with the middle layer.

Table 4-3 summarizes area evaluations on three layers, where the area details on key components are listed to represent approximated integrating area values.

Table 4-3. Area Evaluation Details.

Layer	Components	Area (mm^2)
First layer	Core die	9.29
	L1I+ITB	3.47
	L1D+DTB+W/RQu	4.16
	Router+TSVs	1.27
Second layer	L2 tag+array+output	79.49
	Directory logic	35.17
	Cache controller	18.63
	Router+TSVs	1.27
Third layer	Memory (×4 stacked)	194.02
	Router+TSVs	1.27

In the first layer, key components consist of core die, L1 instruction cache and data cache, enhanced router and TSV array. Thus, the overall area of 16-core stacked layer can be calculated as $16 \times (9.29+3.47+4.16+1.27)=291.04$ mm^2. In the second layer, the overall area of shared cache including cache controller and directory logic can be calculated as 153.61 mm^2. And in the third layer, sixteen page groups and interconnection logic together can take up 214.34 mm^2. Thus, area values normalized to the third layer can be calculated as 1.36, 0.72 over 1. As a consequence, it is suitable for arranging those on-chip components stacked in the proposed area.

4.6 Evaluation Strategy

4.6.1 Experimental Platform Setups

Platform model: the instruction-tracing driven based full system simulator Gem5 [53] is employed for constituting a multi-level multi-core on-chip system model, which consists of private cache level, shared cache level, main memory, and multi-core processors, and each component of the simulation platform is set as listed in Table 4-4. Note that the hierarchical cache modules are represented by extended CACTIv6.5 cache area, energy and integration analysis model,

Table 4-4. Test System Configurations.

Processor Unit	2.0 GHz, 16 cores, single thread per core, 1.1 V supply voltage, 128 IW entries, 45 nm technology library, 30 cycle TLB miss latency.
L1I/L1D Cache	32 KB instruction cache, 32 KB data cache for a core (private), 4-way, 64 B line size, 4 cycle latency.
Shared L2 Cache	16 MB total size D-NUCA, 128 banks, 1MB partitioned for a core (shared), 8-way, 128 B line size, 20 cycle latency.
Main Memory	4 GB Double Data Tate (DDR4 2133MHz, 1.2V), 8KB page size, 120 cycle latency.
Router Network	Mesh_XY Topology, 5 ports per router, 3 VCs per port, 15 flits per port.

where the first cache level is set as the private cache and the second cache level is set as the shared cache.

Router model: based on conventional router architecture in [49], the router layout reports are integrated within the router estimation tool for the purpose of latency and power evaluation, and those evaluated results are sent into the platform to drive the routing module which is meshed with mesh_XY topology [42].

Benchmarks: During verifying the proposed cache network, the PARSEC benchmark suite is much more suitable as it can private the per-instruction level simulation on the characteristics of data sharing and traffic modeling.

Comparisons: The proposed cache allocation method is compared with four methods including [10].

As to the cache network based proposal, both baseline platform and the proactive resource allocation [36] methods are compared under the same platform setups.

4.6.2 Evaluation Metrics

To unify the performance and energy consumption metrics, the total evaluated values are normalized by the cache request numbers to form the normalized energy consumption per access. And the performance is represented by IPC (instructions per clock) values which are counted from the IPC values in all threads simultaneously. In all evaluation tests, the counting period is uniformly set as skip first 1 billion, collect next 5 billion instructions, availably.

4.7 Stacked 3D Cache Network Verification

To evaluate the improvement of applying the proposed cache network, eight setup combinations are employed into runtime experiments as follows. The base platform which contains a conventional cache network is simulated for comparisons, and the platform which contains the proposed cache network is

simulated while the record entry size ranges from 32 to 2K, thereby seven setup combinations are achieved.

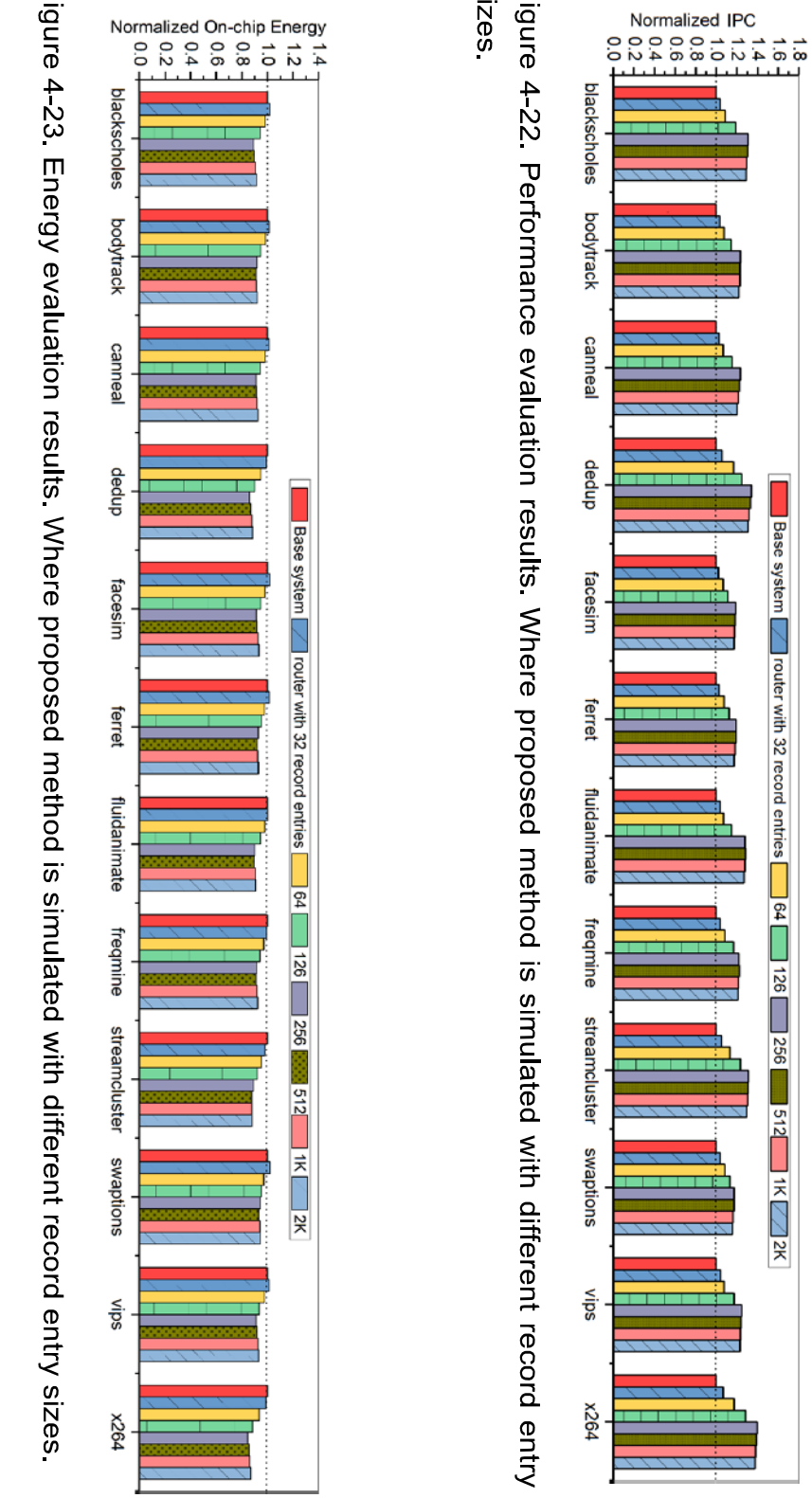


Figure 4-22. Performance evaluation results. Where proposed method is simulated with different record entry sizes.

Figure 4-23. Energy evaluation results. Where proposed method is simulated with different record entry sizes.

4.7.1 Performance and Energy Improvements

As shown in Fig. 4-22, the IPC values of proposed method with different record entry sizes are normalized to the value of base combination. In the combination of 32 record entries scenario, the IPC can be improved by on average of 4.2%, and in the combination of 256 record entries scenario, the value can be improved by on average of 26.1%. However, the IPC will be improved by on average of 24.3% in the combination of 2K record entries scenario. It can be observed that there is an optimal record entry number existing among all benchmarks. Typically, the lowest value of maximum improvement of all benchmarks is about 17.4 with *swaptions* benchmark in the combination of 512 record entries, and the highest value of maximum improvement of all benchmarks is about 39.5% with *x264* benchmark in the combination of 256 record entries.

As to energy consumption, Fig. 4-23 represents the normalized energy values of all benchmarks. Although there are some extra hardware added into cache network, the proposed architecture employed by simulation platform can improve the IPC greatly, so that the energy consumptions with proposed cache

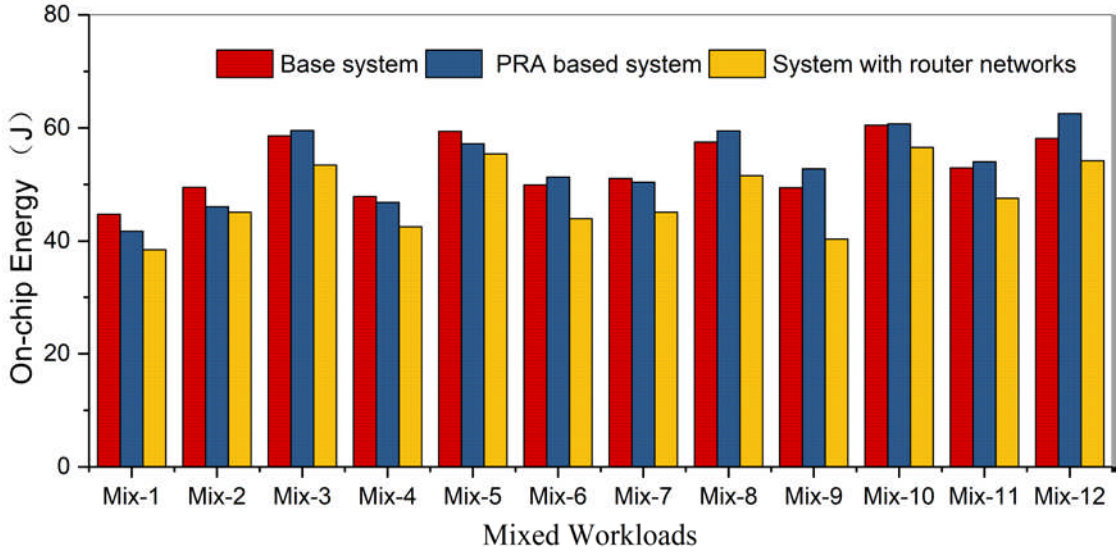


Figure 4-24. On-chip energy comparisons on mixed workloads. Proposed system is set as allocating 256 record entries.

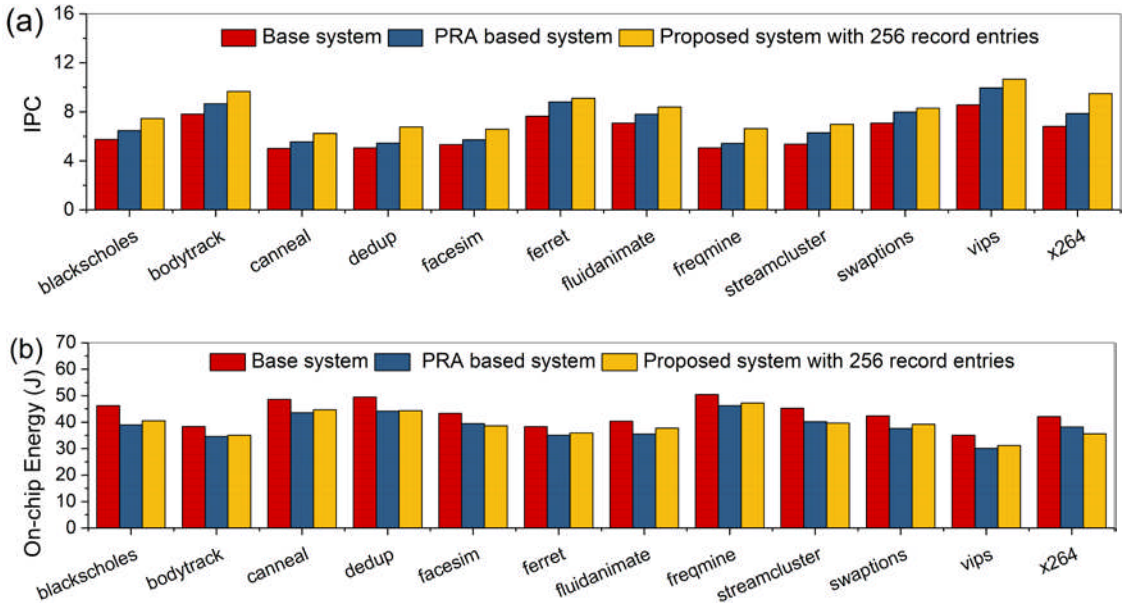


Figure 4-25. Energy and performance comparisons. Proposed system is set as allocating 256 record entries.

network in the combination of 256 record entries can be improved by 9.7% on average over the base combination. However, in the case of 32 record entries combinations, the energy consumptions of all benchmarks are a little larger than the ones of base combinations, in which there is 0.6% energy overhead on average.

Moreover, the system applied with proposed cache network (256 record entries uniformly) is compared with base system and proactive resource allocation (PRA) based system [36]. As shown in Fig. 4-24, the IPC values of all benchmarks are averaged as 6.82, 7.45 and 7.83, respectively on three designs. Clearly, proposed design outperforms the PRA based system by 5.1% on IPC, but in some benchmarks, energy consumption of PRA based design is less than that of proposed system, where the average energy consumption of PRA based system is less than that of proposed system by 1.2%.

In the setup combinations of executing mixed benchmark groups, as shown in Fig. 4-25 and Fig. 4-26, the IPC values of three designs for mixed benchmark groups are averaged as 6.51, 7.64 and 8.02, respectively on three designs. And the on-chip energy consumptions of them for mixed benchmark groups are averaged as 52.5, 44.9 and 45.1, respectively.

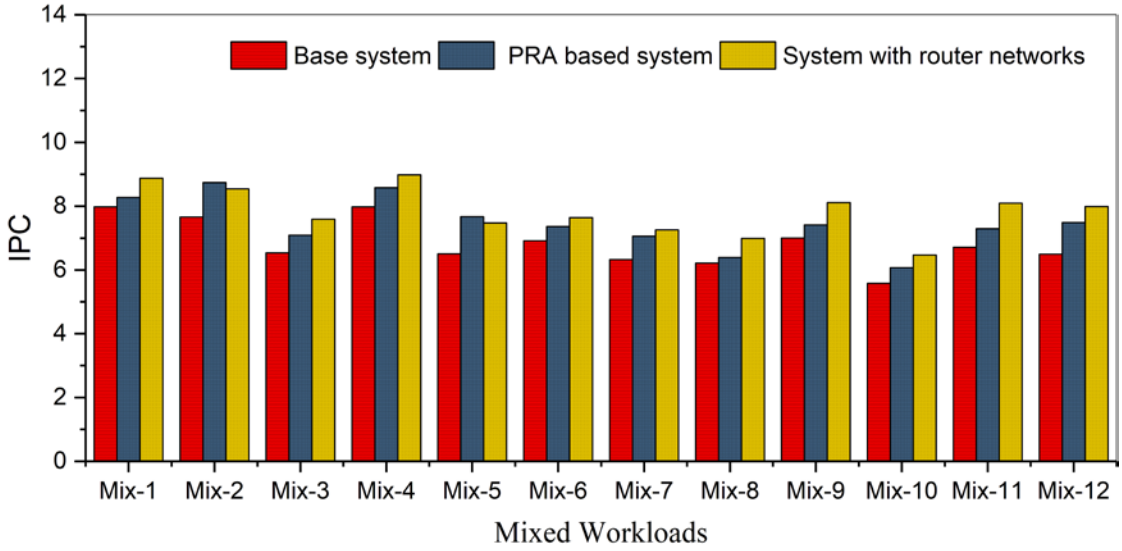


Figure 4-26. IPC comparisons on mixed workloads. Proposed system is set as allocating 256 record entries.

4.7.2 Result Analysis

Based on experimental results in the minimum granularity of each shared cache access, all access patterns which are classified for the purpose of latency reduction can be modified with a novel pipelined path, and plenty of clock cycles are reduced in this path. There are two features observed which lead to such latency improvement as follows. The first feature is that shared accesses, crossed accesses and repeated accesses together can take the majority proportion of all shared cache accesses, thus operating space of latency reduction is large enough for great improvement. The other feature is that each filtered access can be handled with minimum latency rather than that of previous access process, which can be confirmed as follows.

(1) Repeated accesses: For repeated read access, it quite possibly hits on the previous access result stored in record entries, and then the latency of modified path only sustains a period of locating previous access results. Otherwise, such an access request will be routed into the middle layer for exploring target data in the shared cache. Obviously, the previous access path is far longer than the modified path, in which a redundancy of shared cache access can be saved by the proposed method. For repeated write access, it may cause a

continuous write operation in the previous path, which will lead to coverage on current data. However, in the modified path, the write access is directly written into the record entries and then the previous target data is retired into shared cache. As a consequence, both repeated write and read can be fast handled in the first layer rather than access to shared cache in the second layer.

(2) Crossed accesses: It will miss on both record entries in the first layer and the partitioned bank group in the middle layer, and then off-chip memory access is needed to handle such miss, and costs hundreds of clock cycles. In the modified path, it allows to send shared data from the crossed routers to the current router within several clock cycles through the router network. Similarly, the crossed write access can be delivered from the current router to the crossed routers for renewing the target data in crossed routers. Thus, data incoherence is directly handled in modified path rather than the previous directory-based repairing path with many clock cycles.

(3) Shared accesses: For shared read access, it may be handled in the first layer with the modified path or hits on partitioned bank group as conventional manner, where some improvement on latency can be achieved. For shared write access, each modification on target data will trigger a coherence check process in the conventional path, which will waste plenty of latency. However, with the aid of router network, such modification on target data can be easily spread to routers that contain target data, thereby data coherence can be ensured with few clock cycles.

As a consequence, the proposed path can deal with those accesses much faster than the conventional path, which should access shared cache in the second layer or even to off-chip memory in the bottom layer with multiple scales of access latency. Corresponding to simulation results, the proposed path can improve performance by 26% over the conventional path.

4.8 Results on Applying Modified Router for Private Cache

The modified router is employed for interconnecting data accesses in the private cache level, and the router supports runtime access recognition on crossed and shared accesses and further matched accesses can be fast executed for the proposed processing path. Firstly, the throughput bandwidths of data flow between routers are evaluated with the history buffer scale changing. And the optimal buffer scale is selected as 32 entries, where throughput can reach to 117.9 Gbps.

As shown in Fig. 4-27, performance are compared among different record entries. The average values of 16 entries, 32 entries, 64 entries, and 1K entries are improved about 23.91%, 31.85%, 30.56% and 12.61% compared to the base system, respectively. Fig. 4-28 shows normalized energy values under different entry scales. The average values of 16 entries, 32 entries, 64 entries, and 1K entries are improved about 15.52%, 17.61%, 15.93% and 7.85% compared to base system, respectively. Fig. 4-29 shows the comparisons on normalized IPC values among two base system methods, PRA method and proposed method. The base method with large private cache is inferior to the base method by 0.82% on IPC. PRA based method can outperform the base method by 11.62%, and the proposed method behaves the best IPC about 31.85% compared to the base system. Fig. 4-30 shows the comparisons on normalized energy consumptions among four methods. The base method with large private cache is inferior to base method by 0.95% on energy consumption. Compared to base system, proposed design and PRA design show 17.61% and 10.93% energy saving over base system, respectively.

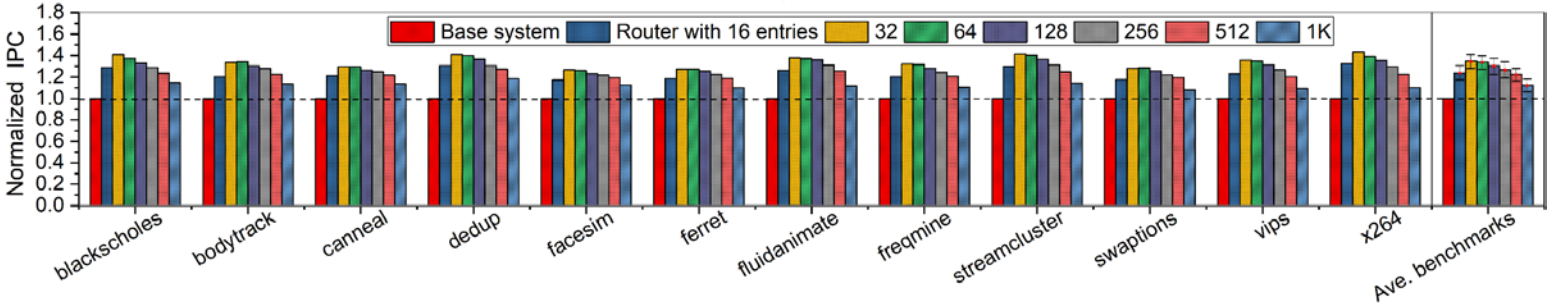


Figure 4-27. Performance evaluation results. Where proposed method is simulated with different record entry sizes.

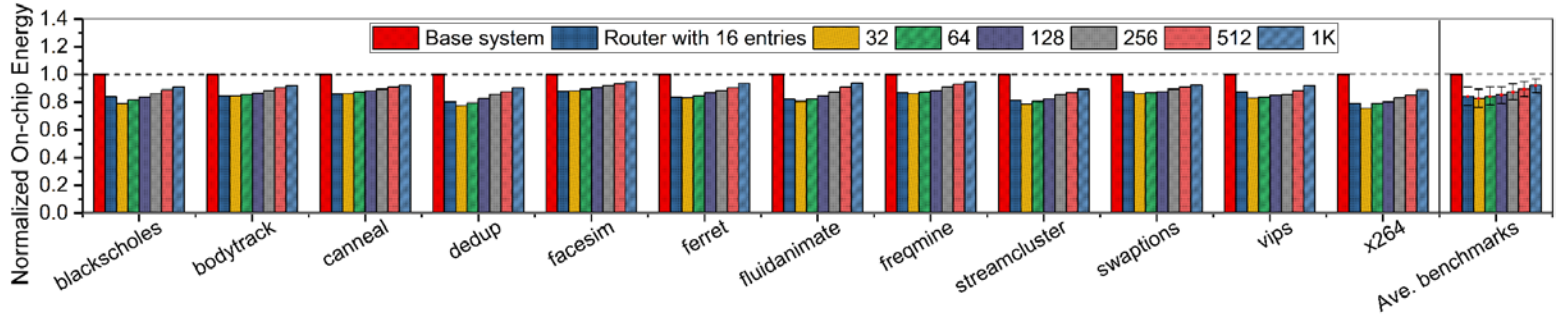


Figure 4-28. Energy evaluation results. Where proposed method is simulated with different record entry sizes.

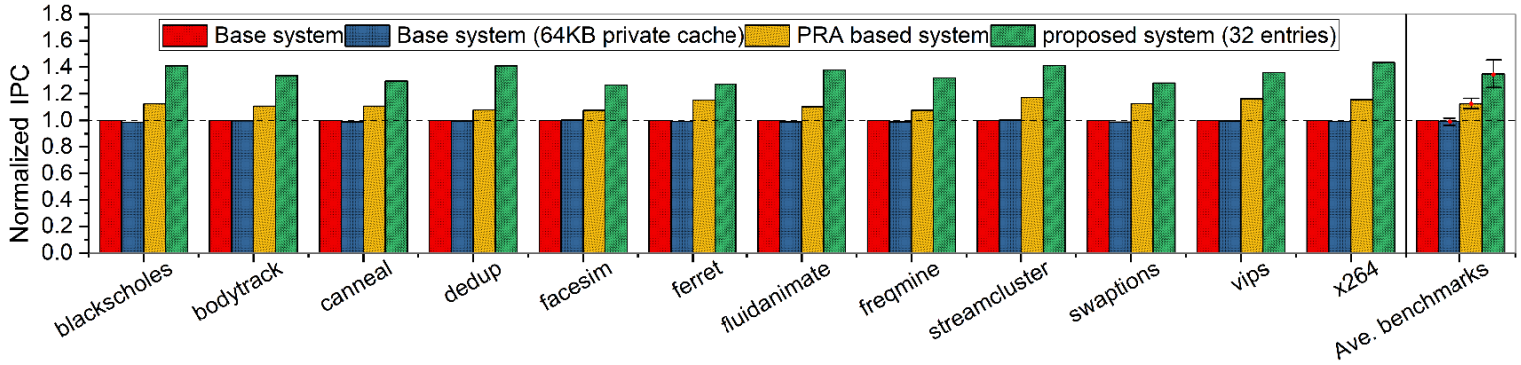


Figure 4-29. Normalized IPC on all benchmarks. Proposed system is set as allocating 32 record entries.

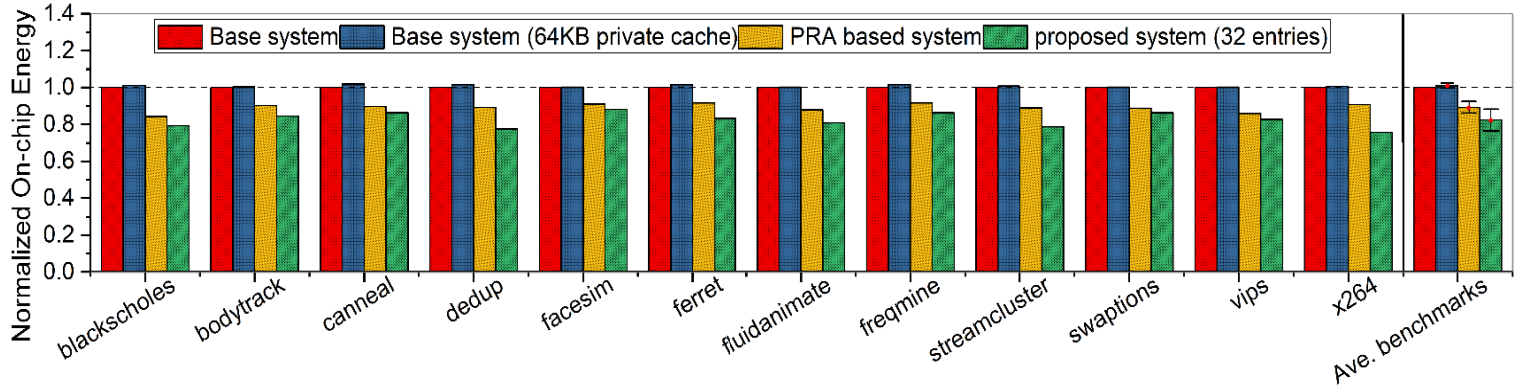


Figure 4-30. On-chip energy comparisons on all benchmarks. Proposed system is set as allocating 32 record entries.

4.9 Summary

In the proposed system design, there are two trade-offs to choose buffer entry and private cache scales. For private cache, applying small cache size leads to a high miss rate, but the latency of each access hit is small. On the contrary, applying a large cache size can achieve a low miss rate, while access latency on each hit is large. Thus, the private cache size is experimentally explored as 32 KB for proposed system under the principle of performance first. Similarly, there is a trade-off on selecting record entry scale. Applying a small record entry scale leads to few detected accesses, but the latency of detecting process is small. On the contrary, applying a large record entry scale can achieve many detected accesses, while processing latency on each access is large. And both IPC and on-chip energy values under different record entries are according to such trade-off analysis.

Similar to analysis on applying router for shared cache level, the modified router network can detect crossed read, crossed write, shared read, and shared write, and each pattern takes up about 2.98%, 1.31%, 3.82% and 0.95% compared to overall accesses. And processing latency values of handling each access in the proposed path will cost about (471+42n), (471+42n), 363 and (564+42n) ps, respectively. Thus, IPC values of proposed system should be far larger than that of base system, and simulation results are according to this inference. As the proposed system can work fast even with some hardware overhead (total power is slightly increased), overall energy consumptions can be reduced with the same amount of workload, and simulation results on energy consumption are according to this inference.

Chapter 5

CONCLUSION AND FUTURE WORK

In this thesis, the shared cache resource demands during runtime are analyzed in detail in the granularity of per application based intervals, per subroutine call based intervals and per shared cache access based intervals. The pre-experimental results show that there is an approximate-optimal share cache allocation existing for each benchmark in per application based intervals, and the subroutine calls on same subroutine behave in similar locality features while total calls on several hot subroutines can take a majority of entire dynamic instruction scale, thereby those subroutine calls act as the perfect control intervals on shared cache allocation. Based on runtime curves on both hit rate and energy consumptions, a control loop is proposed to integrate self-control theory into a dynamic shared cache allocation method. And such design is integrated into a multi-processor based evaluation model, which can dynamically allocate shared cache resources to supply runtime cache resource demand in the granularity of per subroutine call. As a consequence, the proposed method applied to the platform can save plenty of energy consumption over the baseline system, reconfigurable design, and partitioned design methods.

Moreover, the access features are analyzed in per share cache access granularity, where the shared cache accesses can be classified into several types including repeated accesses, crossed accesses and shared accesses. For each access type, the executing processes of both read access and write access are

taken fully considerations on the potential improvements of modified pipelines. Consequently, executing processes of all six access types are optimized by means of enhanced router network, where the repeated accesses can be fast handled after matching previous hits, crossed accesses can be fast handled with the aid of target data in other routers rather than the data in off-chip memory and the shared accesses can be easily maintained in virtue of routing network rather than complex maintenance logics. Hence, such cache to cache network can interact many shared data accesses with far smaller access latency than the latency of shared data access, while the enhanced router is implemented with layout reports on IC compiler tool and the layout results show that the proposed router network only has about one percent of power and area overhead over an Intel chip. But the experimental results with proposed router network show an average of twenty-six percent performance improvement compared to base system.

In the future, controllable cache allocation design should be improved to fit the control situation of many concurrent threads. Furthermore, on-line PID parameter tuning design and cache allocation design under unknown workloads and common CMP system scenarios are needed to do explorations. And, current stacked multi-layer proposals are needed to extend both order of stacked layer magnitudes and integrated core scales for the purpose of high-level parallel computing. Also, more efficient routing algorithms are desired to handle large cache network.

Bibliography

[1] F. Hameed, A.A. Khan and J. Castrillon, “Performance and Energy Efficient Design of STT-RAM Last-Level Cache,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol.27, No.99, pp.1-14, Jan., 2018.

[2] W. Wei, et al., “HAP: Hybrid-Memory-Aware Partition in Shared Last-Level Cache,” ACM Transactions on Architecture and Code Optimization, Vol.14, No.3, pp.1-25, Sept., 2017.

[3] Z. Bubnicki, “Modern Control Theory,” Springer.2005925392, 2005.

[4] N. Muralimanohar, R. Balasubramonian and N.P. Jouppi, “CAC-TI 6.0: A Tool to Model Large Caches,” Bragantia HPL-2009-85, Apr., 2009.

[5] A. Phansalkar, A. Joshi and L.K. John, “Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite,” International Symposium on Computer Architecture (ISCA), Vol.35, No.2, pp.412-423, Jun., 2007.

[6] M. Rawlins and A. Gordon-Ross, “A Cache Tuning Heuristic for Multicore Architectures,” IEEE Transactions on Computers, Vol.62, No.8, pp.1570-1583, Mar., 2013.

[7] C. Zhang, F. Vahid and R. Lysecky, “A Self-Tuning Cache Architecture for Embedded Systems,” ACM Transactions on Embedded Computing Systems Vol.3, No.1, pp.407-425, May, 2004.

[8] S.M. Khan, Y.Y. Tian, and D.A. Jimenez, “Sampling Dead Block Prediction for Last-level Caches,” In Proc. 43rd Annual IEEE/ACM Int Microarchitecture (MICRO) Symp, pp.175-186, Dec., 2010.

[9] W. Wang and P. Mishra, “Dynamic Reconfiguration of Two-Level Cache Hierarchy in Real-Time Embedded Systems,” Journal of Low Power Electronics, Vol.7, No.1, pp.17-28, 2011.

[10] H. Homayoun, et al., “MZZ-HVS: Multiple Sleep Modes Zig-Zag Horizontal and Vertical Sleep Transistor Sharing to Reduce Leakage Power in On-Chip SRAM Peripheral Circuits,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol.19, No.12, pp.1-14, Dec., 2011.

[11] T. Adegbija and A. Gordon-Ross, “PhLock: A Cache Energy Saving Technique Using Phase-Based Cache Locking,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol.26, No.1, pp.1-10, Jan., 2018.

[12] W. Wang, P. Mishra and S. Ranka, “Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-time Multi-core Systems,” Design Automation Conference, pp.948-953, Jun., 2011.

[13] H. Kim, A. Kandhalu and R. Rajkumar, “A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems,” Real-time Systems, No.8114, pp.80-89, 2013.

[14] X. Zhou, W. Chen and W. Zheng, “Cache Sharing Management for Performance Fairness in Chip Multiprocessors,” International Conference on PACT, pp.384-393, Nov., 2014.

[15] C. Kim, D. Burger and S.W. Keckler, “An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches,” Acm Sigops Operating Systems Review, Vol.36, No.5, pp.211-222, 2002.

[16] J.H. Henning, “SPEC CPU2006 benchmark descriptions,” Acm Sigarch Computer Architecture News, Vol.34, No.4, pp.1-17, 2006.

[17] C.K. Luk, et al., “PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation,” Proceedings of 2005 ACM SIPLAN Notices, Vol.40, No.6, pp.190-200, 2005.

[18] X. Liao, et al., “A Phase Behavior Aware Dynamic Cache Partitioning Scheme for CMPs,” International Journal of Parallel Programming, Vol.44, No.1, pp.1-19, 2014.

[19] J. Cheng, et al., “An I/O Scheduling Strategy for Embedded Flash Storage Devices with Mapping Cache,” IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, Vol.37, No.4, pp.1-14, Jan., 2018.

[20] G. Chen, et al., “Reconfigurable Cache for Real-time MPSoCs: Scheduling and Implementation,” Microprocessors & Microsys-tems, Vol.42, pp.200-214, May, 2016.

[21] Y. Zhong, et al., “Program Locality Analysis Using Reuse Distance,” ACM Transactions on Programming Languages and Systems, Vol.31, No.9, pp.1-39, Aug., 2011.

[22] D.L. Schuff, et al., “Multicore-aware Reuse Distance Analysis,” IEEE International Symposium on Parallel & Distributed Processing, pp.1- 8, May., 2010.

[23] A.G. Aleksandrov and M.V. Palenov, “Self-tuning PID/I controller,” Automation & Remote Control, Vol.72, No.10, pp.2010-2022, Sept., 2011.

[24] A. Gordon-Ross, et al., “A One-Shot Configurable-Cache Tuner for Improved Energy and Performance,” Design, Automation & Test in Europe Conference, Apr., 2007.

[25] A. Gordon-Ross, et al., “Phase-based Cache Reconfiguration for a Highly-configurable Two-level Cache Hierarchy,” GLSVLSI '08, May, 2008.

[26] M. K. Qureshi and Y. N. Patt, “Utility-based Cache Partitioning: A Low-overhead, High-performance, Runtime Mechanism to Partition Shared Caches,” IEEE International Symposium on MICRO, pp.423–432, Dec., 2006.

[27] T. Oh, K. Lee and S. Cho, “An Analytical Performance Model for Co-management of Last-Level Cache and Bandwidth Sharing,” IEEE Symposium on MASCOTS, pp.150-158, July, 2011.

[28] H. Cook, et al., “A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness,” IEEE International Symposium on Computer Architecture, pp.308–319, Jun., 2013.

[29] Intel Corporation, “Intel 64 and IA-32 Architectures Developer’s Manual,” Vol. 3B, System Programming Guide, Part 2, Retrieved from http://goo.gl/sw24WL, 2016.

[30] N. Beckmann and D. Sanchez, “Modeling Cache Performance Beyond LRU,” IEEE International Symposium on High Performance Computer Architecture, pp.225-236, Mar., 2016.

[31] K. Huang, et al., “Curve Fitting Based Shared Cache Partitioning Scheme for Energy Saving,” IEICE Electron. Express, Vol.15, No.22, pp.1-12, Oct., 2018.

[32] P. Guo, et al., “A Bypass-Based Low Latency Network-on-Chip Router,” IEICE Electron. Express, Vol.16, pp.1-12, 2019.

[33] N. Kim, et al., “Benzene: An Energy-Efficient Distributed Hybrid Cache Architecture for Many core Systems,” ACM Trans. on TACO, Vol.15, No.1, pp.1-23, Mar., 2018.

[34] Z. C. Wang, et al., “VP-Router: On Balancing the Traffic Load in On-Chip Networks,” IEICE Electron. Express, Vol.15, pp.1-12, Sept., 2018.

[35] T. T. Lu, et al., “TSV-Based 3-D ICs: Design Methods and Tools,” IEEE Trans. on TCAD, Vol.36, No.10, pp.1-27, Oct., 2017.

[36] L. K. Pejman, et al., “Near-Ideal Networks-on-Chip for Servers,” IEEE International Symposium on High Performance, May, 2017.

[37] G. Aupy, et al., “Co-scheduling HPC Workloads on Cache Partitioned CMP Platforms,” CLUSTER 2018, pp.348-358, 2018.

[38] P. Safayenikoo, et al., “An Energy Efficient Non-uniform Last Level Cache Architecture in 3D Chip-multi Processors,” International Symposium on Quality Electronic, pp.373-382, May, 2018.

[39] V. Srinivasan, et al., “H3 (Heterogeneity in 3D): A Logic-on-logic 3D-stacked Heterogeneous Multi-core Processor,” IEEE International Conference on Computer Design, pp.145-155, Nov., 2017.

[40] A. K. Mishra, et al., “Architecting On-Chip Interconnects for Stacked 3D STT-RAM Caches in CMPs,” International Symposium on Computer Architecture, pp.69-80, Jun., 2011.

[41] G. H. Zheng, et al., “Design and Implementation of a NoC Router Supporting Multicast,” IEICE Electron. Express, Vol.11, pp.1-12, 2013.

[42] Y. S. Jeong and S. E. Lee, “Deadlock-free XY-YX Router for On-chip Interconnection Network,” IEICE Electron. Express, Vol.10, pp.1-12, 2013.

[43] X. Han, et al., “A Deadlock-free Subnetting Mechanism for High Performance Broadcasting in NoC,” IEICE Electron. Express, Vol.12, pp.1-12, 2015.

[44] Y. Yang, et al., “New Coaxial Through Silicon Via Applied for Three Dimensional Integrated Circuits,” IEICE Electron. Express, Vol.13, pp.1-12, 2016.

[45] C. Bienia, “Benchmarking Modern Multiprocessors,” Ph.D. Thesis, Princeton University, Princeton (2011).

[46] K. Aingaran, et al., “M7: Oracle's Next Generation Sparc Processor,” IEEE Micro, Vol.35, No.36, pp.1-9, 2015.

[47] M. R. Marty, “Cache Coherence Techniques for Multicore processors,” Ph.D.

Thesis, University of Wisconsin-Madison, Madison (2008).

[48] F. Wang, et al., “A Novel Guard Method of Through-Silicon-Via (TSV),” IEICE Electron. Express, Vol.15, pp.1-12, 2015.

[49] A. B. Kahng, B. Lin and S. Nath, “ORION3.0: A Comprehensive NoC Router Estimation Tool,” IEEE Embedded Systems Letters, Vol.7, No.2, pp.41-49, Jun., 2015.

[50] S. K. Sadasivam, et al., “IBM Power9 Processor Architecture,” IEEE Micro, Vol.37, No.2, pp.40-52, May, 2017.

[51] G. S. Sun, et al., “A Novel Architecture of the 3D Stacked MRAM L2 Cache for CMPs,” IEEE International Symposium on High Performance Computer Architecture, pp.239-250, Mar., 2009.

[52] I. Psaras, et al., “In-Network Cache Management and Resource Allocation for Information-Centric Networks,” IEEE Trans. on TPDS, Vol.25, No.304, pp.2920-2934, 2014.

[53] N. Binkert, et al., “The Gem5 Simulator,” ACM SIGARCH, Vol.39, May, 2011.

[54] C. Bienia, et al., “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” IEEE International Conference on PACT, pp.72-81, Oct., 2008.

[55] N. Muralimanohar, R. Balasubramonian and N.P. Jouppi, “CACTI 6.0: A Tool to Model Large Caches,” Bragantia HPL-2009-85, Apr. 2009.

Webpage source: https://www.hpl.hp.com/research/cacti/cacti65.tgz

[56] J. Kong, et al., “An Energy-Efficient Last-Level Cache Architecture for Process Variation-Tolerant 3D Microprocessors,” IEEE Trans. on Computers, Vol.64, No.9, pp.2460-2473, Sept., 2015.

[57] J. H. Li, et al., “Thread Criticality Assisted Replication and Migration for Chip Multiprocessor Caches,” IEEE Trans. on Computers, Vol.66, No.10, pp.1747-1762, Oct., 2017.

[58] A. Sodani, et al., “Knights Landing: Second-Generation Intel Xeon Phi Product,” IEEE Micro, Vol.36, No.2, pp.34-46, Apr., 2016.

[59] S. Gupta and H. Zhou, “Spatial Locality-Aware Cache Partitioning for Effective Cache Sharing,” International Conference on Parallel Processing, pp.150-162, Dec.,

2015.

[60] W. Shu and N. F. Tzeng, “NUDA: Non-Uniform Directory Architecture for Scalable Chip Multiprocessors,” IEEE Trans. on Computers, Vol.67, No.5, pp.740-747, May, 2018.

[61] X. Wang, K. Ma, and Y. Wang, “Cache Latency Control for Application Fairness or Differentiation in Power-constrained Chip Multiprocessors,” IEEE Transactions on Computers, Vol.61, No.10, pp. 1371–1385, Oct, 2012.

[62] S. Kim, et al., “Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture,” IEEE International Symposium on Parallel Architecture and Compilation Techniques, pp. 657–668, Oct., 2004.

[63] A. Herdrich, et al., “Cache QoS: From Concept to Reality in the Intel Xeon Processor E5-2600 v3 Product Family,” IEEE International Symposium on High Performance Computer Architecture, pp. 657–668, 2016.

[64] D. Lo, et al., “Heracles: Improving Resource Efficiency at Scale,” IEEE International Symposium on Computer Architecture, pp.450-462, Oct., 2015.

[65] S. Mittal, et al., “MASTER: A Multicore Cache Energy Saving Technique Using Dynamic Cache Reconfiguration,” IEEE Transactions on VLSI Syst., Vol.22, No.8, pp.1653–1665, Aug., 2014.

[66] T. Ishihara and F. Fallah, “A Non-uniform Cache Architecture for Low Power System Design,” International Symposium on Low Power Electronics and Design, pp.363-368, Aug., 2005.

[67] B, Zhao, et al., “Process Variation-Aware Non-uniform Cache Management in a 3D Die-Stacked Multicore Processor,” IEEE Transactions on Computers, Vol.62, No.11, pp.2252-2265, Nov., 2013.

[68] J. J. Valls, et al., “PS Directory: A Scalable Multilevel Directory Cache for CMPs,” The Journal of Supercomputing, Vol.71, No.8, pp.2847-2876, Aug., 2015.

[69] S. Demetriades and S. Cho, “Predicting Coherence Communication by Tracking Synchronization Points at Run Time,” International Symposium on Microarchitecture, pp.351-362, Dec., 2012.

[70] H. Zhao, et al., “SPACE: Sharing Pattern-based Directory Coherence for Multicore

Scalability," IEEE International Conference on PACT, pp.135-146, Sept., 2010.

[71] H. T. Zhao, X. Jia and T. Watanabe, "Router-integrated Cache Hierarchy Design for Highly Parallel Computing in Efficient CMP Systems," Electronics 2019, 8, 1363, 21 pages, Nov., 2019.

[72] H. T. Zhao, X. Jia and T. Watanabe, "Filter Router: An Enhanced Router Design for Efficient Stacked Shared Cache Network," IEICE Electronics Express, Vol.16, No.14, pp.1-6, Jun., 2019.

[73] H. T. Zhao, J. Y. Ye and T. Watanabe, "A Low-power Shared Cache Design with Modified PID Controller for Efficient Multicore Embedded Systems," Journal of Information Processing, Vol.27, pp.1-10, Feb., 2019.

[74] H. T. Zhao, et al., "Application-specific Shared Last-level Cache Optimization for Low-power Embedded Systems," IEEE International Conference on NEWCAS, pp.1-4, Jun., 2015.

[75] H. T. Zhao, et al., "Flexible L1 Cache Optimization for a Low Power Embedded System," IEEE International Conference on MEC, pp.1-5, Dec., 2013.

[76] H. T. Zhao, et al., "Pseudo Dual Path Processing to Reduce the Branch Misprediction Penalty in Embedded Processors," IEEE International Conference on ASIC, pp.1-4, Oct., 2013.

[77] Synopsys, Inc., "IC Compiler 1 Workshop Student Guide," Document Order Number: 20-I-071-SSG-008, Sept., 2008.

[78] R. Zhang, M. R. Stan and K. Skadron, "HotSpot 6.0: Validation, Acceleration and Extension," University of Virginia, Tech. Report, CS-2015-04, Aug., 2015.

Publications

Journal Papers

○ 1．Huatao Zhao, Xu Jia and Takahiro Watanabe, "Router-integrated Cache Hierarchy Design for Highly Parallel Computing in Efficient CMP Systems", Electronics 2019, 8, 1363, 21 pages, November 2019.

○ 2．Huatao Zhao, Xu Jia and Takahiro Watanabe, "Filter Router: An Enhanced Router Design for Efficient Stacked Shared Cache Network", IEICE Electronics Express, Vol.16, No.14, pp.1-6, Jun. 2019.

○ 3．Huatao Zhao, Jiongyao Ye and Takahiro Watanabe, "A Low-power Shared Cache Design with Modified PID Controller for Efficient Multicore Embedded Systems", Journal of Information Processing, Vol.27, pp.1-10, Feb. 2019.

4．Huatao Zhao, Xiao Luo, Chen Zhu, Tianbo Zhu and Takahiro Watanabe, "Behavior-aware Cache Hierarchy Optimization for Low-power Multi-core Embedded Systems", Modern Physics Letters B, Vol.31, No.19, pp.1-7, April. 2017.

International Conference

○ 1. Huatao Zhao, Jiongyao Ye, Xian Su and Takahiro Watanabe, "Application-specific Shared Last-level Cache Optimization for Low-power Embedded Systems", IEEE 13th International New Circuits and Systems Conference, pp.1-4, June 2015．

○ 2. Huatao Zhao, Sijie Yin, Yuxin Sun and Takahiro Watanabe, "Flexible L1 Cache Optimization for a Low Power Embedded System", Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer, pp.1-5, Dec. 2013.

○ 3. Huatao Zhao, Jiongyao Ye, Yuxin Sun and Takahiro Watanabe, "Pseudo Dual Path Processing to Reduce the Branch Misprediction Penalty in Embedded Processors", IEEE 10th International Conference on ASIC, pp.1-4, Oct. 2013.

Acknowledgements

With the utmost sincerity, I wish to appreciate my Sensei Professor Takahiro Watanabe of WASEDA University for his always encouragement and research support. During so many year's research living, he guided me expertly to stride across all challenges and difficulties, and emboldened me never give up. His affability in life and his commitment in academic attitude have great influence on my forming character, which will benefit greatly in future research. Appreciate sincerely for those invaluable impartments Sensei taught.

I would like to thank Professor Shinji Kimura and Professor Takashi Ohsawa of WASEDA University for their expert and significant suggestions, which do great help on my research.

Also, I wish to appreciate Mr. Zhang Xiaoyang of Synopsys Shanghai and his team for their technical supports to my research. They provided me professional knowledge in industrial view, which helped a lot to hit coming design topics.

Finally, I feel indebted to my forever Ms. Qiu Shaiqiu, from lover to wife in those years, for her pure affection and endless endeavor. She always encourages me to finish this work, and gives the most valuable gift - my coming baby. And wish to thank my parents for their dedicated support.