Dynamic Cache Pooling for Improving Energy Efficiency in 3D Stacked Multicore Processors

Jie Meng, Tiansheng Zhang, and Ayse K. Coskun
Electrical and Computer Engineering Department, Boston University, Boston, MA, USA
{jiemeng, tszhang, acoskun}@bu.edu

***Abstract*—Resource pooling, where multiple architectural components are shared among multiple cores, is a promising technique for improving the system energy efficiency and reducing the total chip area. 3D stacked multicore processors enable efficient pooling of cache resources owing to the short interconnect latency between vertically stacked layers. This paper introduces a 3D multicore architecture that provides poolable cache resources. We propose a runtime policy that improves energy efficiency in 3D stacked processors by providing *flexible heterogeneity* of the cache resources. Our policy dynamically allocates jobs to cores on the 3D stacked system in a way that pairs applications with contrasting cache use, while also partitioning the cache resources based on the cache hungriness of the applications. Experimental results demonstrate that the proposed policy improves system energy-delay product (EDP) and energy-delay-area product (EDAP) by up to 39.2% and 57.2%, respectively, compared to 3D processors with static cache sizes.**

I. Introduction

3D integration is a promising design technique for enabling heterogeneous integration of different technologies, increasing the transistor density per chip footprint, and improving system performance [1], [2]. Most of the prior work on multicore 3D systems exploits the performance or energy efficiency benefits of 3D systems by considering fixed, homogeneous computational and memory resources (e.g., [3]). Heterogeneous multicore design, however, can bring substantial benefits in reducing the energy consumption and cost. This is because applications have varying resource requirements (e.g., different cache uses), which can be addressed by including cores with different architectural resources in a single chip [4], [5].

Resource pooling, where architectural components of a core are shared by other cores, allows implementing *flexible heterogeneity* in a multicore system. In 2D multicore systems, resource pooling among the cores and accompanying scheduling techniques have been proposed (e.g., [6], [7]). However, the efficiency of resource pooling in 2D is limited by the large latency of accessing remote shared resources in the horizontal direction; thus, resource pooling in 2D is not scalable to a large number of cores. 3D stacked systems enable efficient resource pooling among different layers, owing to the short communication latency achieved by vertically stacking and connecting poolable resources using through-silicon-vias (TSVs). A recent technique proposes pooling performance-critical microarchitectural resources such as register files in a 3D system [8]. Their work, however, does not address the cache requirements of applications. The significance of the memory latency in determining application performance motivates investigating resource pooling of the caches. Cache pooling can provide additional low-cost heterogeneity of the resources available to the cores and bring substantial energy efficiency improvements.

This paper presents a runtime policy for improving energy efficiency of 3D stacked systems using cache resource pooling. Prior work on 3D system management mostly focuses on optimizing the energy efficiency or balancing the temperature (e.g., [9], [3]). In 3D systems with resource pooling, it is necessary to design policies that are aware of the application cache requirements. The runtime management policy should manage the poolable resources according to the characteristics of workloads that are running on the system, while considering the interplay between performance and energy. To address this need, we design an integrated cache management and job allocation policy that dynamically maximizes the energy efficiency of 3D systems. Our experimental results demonstrate that, using minimal architectural modifications complemented with an intelligent management policy, 3D stacked systems achieve higher energy efficiency through cache resource pooling. Our specific contributions are as follows:

- We introduce a 3D stacked architecture with cache resource pooling. The proposed architecture requires minimal additional circuitry and architectural modifications in comparison to using static cache resources. Leveraging this resource pooling design, we are able to achieve higher performance, lower power, and smaller chip area compared to 3D stacked systems with static cache resources.
- We propose a novel application-aware job allocation and cache pooling policy. Our policy predicts the resource requirements by collecting the performance characteristics of each application at runtime. It then allocates jobs with contrasting cache usage in adjacent layers and determines the most energy-efficient cache size for each application.
- We evaluate dynamic cache resource pooling for both high-performance and low-power 3D multicore systems. For a 4-core low-power system, 3D cache resource pooling reduces system EDP by up to 39.2% and system EDAP by 57.2% compared to using fixed cache sizes. For a larger 16-core 3D system, our technique provides 19.7% EDP reduction and 43.5% EDAP reduction in comparison to a 3D baseline system with 2MB L2 caches.

The rest of the paper starts with an overview of the related work. Section III introduces the proposed 3D architecture with cache resource pooling. Section IV presents our application-aware workload allocation and cache pooling policy. Sections V and VI provide the evaluation methodology and the experimental results. Section VII concludes the paper.

II. Related Work

This section discusses the prior work on (1) runtime management, (2) cache design and reconfiguration, and (3) resource sharing and pooling for both 2D and 3D systems as well as how our research differentiates from them.

Recent research on job allocation in 2D systems generally focuses on improving performance and reducing the communication, power, or cooling cost. For example, Das et al. [10] propose an application-to-core mapping algorithm to maximize system performance by separating network-latency-sensitive applications from network-bandwidth-intensive applications. Dynamic job allocation on 3D systems mostly targets power density and thermal challenges induced by vertically stacking. For example, dynamic thermally-aware job scheduling techniques use the thermal history of the cores to balance the temperature and reduce hot spots [3], [9]. As resource pooling in 3D systems is a novel design technique, prior work has not considered the sharing of cache resources among the cores during runtime management.

Cache sharing and partitioning have been well studied in 2D systems. Varadarajan et al. propose a concept of molecular caches which creates dynamic heterogeneous cache regions [11]. Qureshi et al. introduce a low-overhead runtime mechanism to partition caches between multiple applications based on the cache miss rates [12]. Chiou et al. propose a dynamic cache partitioning policy that restricts the replacements into a particular subset of columns [13]. However, the benefits of cache sharing in 2D systems are highly limited by the on-chip interconnect latency. Kumar et al. [14] demonstrate that sharing the L2 cache among multiple cores is significantly less attractive when the interconnect overheads are taken into account. Cache design and management in 3D stacked systems have also emerged as research topics recently. Sun et al. study the architecture-level design of 3D stacked L2 caches [15]. Prior work on 3D caches and memories, however, either considers integrating heterogeneous SRAM or DRAM layers into 3D architectures (e.g., [16], [17]), or involves major modifications to conventional cache design (e.g., [15]).

Prior work on resource pooling has mainly focused on 2D systems. Ipek et al. propose a reconfigurable architecture to combine the resources of simple cores into more powerful processors [4]. Ponomarev et al. introduce a technique to dynamically adjust the sizes of the performance-critical microarchitectural components, such as the reorder buffer or the instruction queue [5]. Homayoun et al. are the first to explore microarchitectural resource pooling in 3D stacked processors for sharing resources at a fine granularity [8]. None of the prior work investigates cache resource pooling in 3D systems.

To the best of our knowledge, our work is the first to propose a 3D stacked architecture complemented with a novel dynamic job allocation policy to enable energy-efficient cache resource pooling in 3D multicore systems. In comparison to 3D systems with static cache resources, our design requires minimal hardware modifications. Our dynamic job allocation and cache pooling policy differentiates from prior work as it partitions the available cache resources from adjacent layers in the 3D stacked system in an application-aware manner and utilizes the existing cache resources to the maximum extent.

III. Proposed 3D Stacked Architecture with Cache Resource Pooling

In this section, we describe our 3D architecture that enables vertical cache resource pooling. As shown in Figure 1, we explain the architecture on a four-layer 3D system, which has one core on each layer with a private L2 cache. Figure 1 (a) and (b) are the baseline 3D systems with 1MB and 2MB static private L2 caches, respectively. In our 3D architecture design shown in Figure 1 (c), each core has a 1MB private L2 cache and the vertically adjacent caches are connected using TSVs for enabling cache resource pooling.

A. Design Overview

We next describe the modifications compared to the conventional cache architecture to enable cache resource pooling in 3D systems. The modified cache architecture allows cores in the 3D stacked system to increase their private L2 cache sizes by utilizing the cache resources from the other layers with negligible access latency penalty. The objectives of our design are: (1) to increase performance by increasing cache size when needed, and (2) to save power by turning off un-used cache partitions. We focus on pooling L2 caches as L2 cache usage varies significantly across applications. It is possible to extend the strategy to other levels of data caches.

Cache size is determined by the block size, the number of sets, and the level of associativity. We adjust the cache sizes in this design by changing the cache associativity. We leverage the selective way cache architecture introduced in prior work [18], which turns off unnecessary cache ways for saving power in 2D systems. We call each cache way a *cache partition*. Each partition is independently poolable to one of its adjacent layers. In order to maintain scalability of the design and provide equivalent access time to different partitions, we do not allow cores in non-adjacent layers to share caches. We also do not allow a core to pool partitions from both upper and lower layers at the same time to limit design complexity. In fact, we observe that for most of the applications in our experiments pooling cache resources from two adjacent layers at the same time does not bring considerable performance improvement.

B. 3D Cache Data Way Management Implementation

In order to implement cache resource pooling in 3D systems, we propose minimal modifications to the cache architecture. As shown in Figure 2 (a), we make modifications to (1) cache status registers and (2) cache control logic.

For 3D cache resource pooling, the cores need to be able to interact with cache partitions from different layers. We introduce a Local Cache Status Register (LCSR) for each local L2 cache partition (e.g., there are four partitions in a 1MB cache in our design) to record the status of the cache partitions. We also introduce Remote Cache Status Registers (RCSR) for the L1 cache so that the L1 cache is aware of its remote cache partitions. RCSR and LCSR logics are illustrated in Figure 2 (a), (b), and (c). The values of these registers are set by the runtime management policy, which we discuss in Section IV.

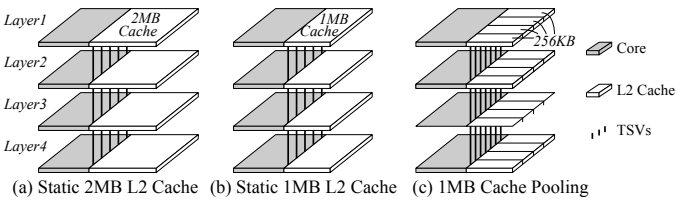


Fig. 1: Proposed 3D system with cache resource pooling versus 3D systems with static 1MB and 2MB caches. In (c), cores are able to access caches on the adjacent layers through the TSVs.

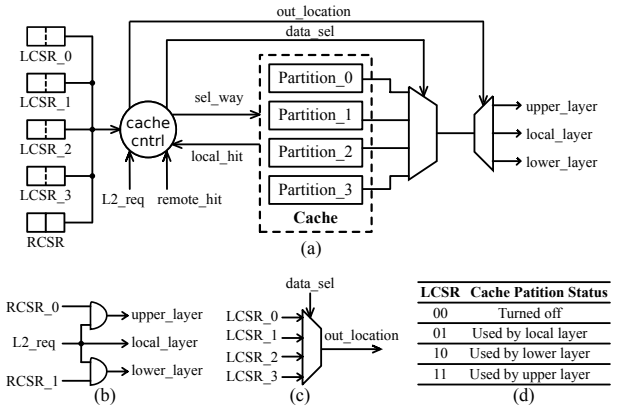


Fig. 2: Cache resource pooling implementation: (a) Logic for cache resource pooling (b) L2 request generation logic (b) Output location generation logic (c) Local cache status registers.

There are four possible situations for each local cache partition: used by local, used by upper or lower layer, or turned off. Each LCSR keeps two bits to indicate the current status of the corresponding partition as listed in the table in Figure 2 (d). In addition, we maintain two 1-bit RCSR in L1 caches for each core to be aware of its remote cache partitions. L1 I- and D-caches can use the same RCSR bits as both caches' misses are directed to the L2. If both RCSRs of an L1 cache are set to 0, it means there is no remote cache partition in use. In contrast, an RCSR bit is set to 1 if the core is using cache partitions from corresponding adjacent layers. RCSR_0 denotes the lower layer and RCSR_1 denotes the upper layer.

Using the information from LCSRs and RCSRs, the cores are able to communicate with cache partitions from multiple layers. When there is an L2 request, the core sends this request and the requested address based on the values of RCSRs. Once the requests and addresses arrive at the cache block, the tag from the requested address will be compared with the tag array. At the same time, the entries of each way will be chosen according to the index. The output destinations of data and hit signals are determined by the LCSR value of the corresponding cache partition after a cache hit. We add a multiplexer to select the output destination, as shown in Figure 2 (c). When there is a L2 cache hit, the hit signal is sent back to the cache at the output destination according to the value in LCSR. When both the local hit signal and the remote hit signal are 0, this indicates an L2 miss.

As the cache partitions can be dynamically re-assigned by the runtime policy, we need to maintain the data integrity of all the caches. In case of a cache partition re-allocation (e.g., a partition servicing a remote layer is selected to service the local core), we flush all blocks from a cache way before it is re-allocated. We use the same cache coherence protocol in our design as in the conventional 2D caches. When a cache line is invalidated, both LCSRs and RCSRs are reset to 0 to disable the access from the remote layers while data entries in the local caches are deleted.

C. Area and Performance Overhead

Each 1-bit register requires up to 12 transistors and each 1-bit multiplexer requires up to 24 transistors. Thus, the total number of transistors needed by the extra registers and logic in our design is limited to 2568 (10×1-bit registers + 2×64-bit demux + 1×30-bit mux + 1×2-bit mux + 1×1-bit demux). We assume there are 128 TSVs for two-way data transfer between caches, 64 TSVs for the memory address bits, and additional 4 TSVs for transferring L2 requests and hit bits between the caches on vertically adjacent layers. TSV power has been reported to be low compared to the overall power consumption of the chip; thus, we do not take TSV power into account in our simulations [19]. We assume that TSVs have $10\mu m$ diameters and a center-to-center pitch of $20\mu m$. The total area overhead of TSVs is less than $0.1mm^2$, which is negligible compared to the total chip area of $10.9mm^2$. Prior work shows that the layer-to-layer delay caused by TSVs is $1.26ps$ [8], which has no impact on the system performance as it is much smaller than the CPU clock period at 1GHz.

IV. Runtime Application-aware Job Allocation and Cache Pooling Policy

In this section, we introduce our runtime job allocation and cache resource pooling policy for improving the energy efficiency of target 3D systems. We explain the details of our policy for the 3D system shown in Figure 1 (c), where each layer has a single core and a 1MB L2 cache; however, the policy is applicable to larger 3D systems as well.

Overview and Motivation

The fundamental motivation of our policy is that different workloads require different amounts of cache resources to achieve their highest performance. Figure 3 shows the instructions per cycle (IPC) of the SPEC benchmarks when running on systems with various L2 cache sizes (from 512KB to 2MB). Among all the workloads, $soplex$ has the largest throughput improvement at larger L2 cache sizes. We call such benchmarks *cache-hungry* workloads. On the other hand, benchmarks such as $libquantum$ barely have any performance improvement at larger L2 cache size. This observation motivates allocating the cache-hungry jobs in adjacent layers in the 3D stack with less cache-hungry jobs so that they can share a pool of cache resources efficiently (i.e., the cache-hungry job would use a larger number of cache partitions). We present the

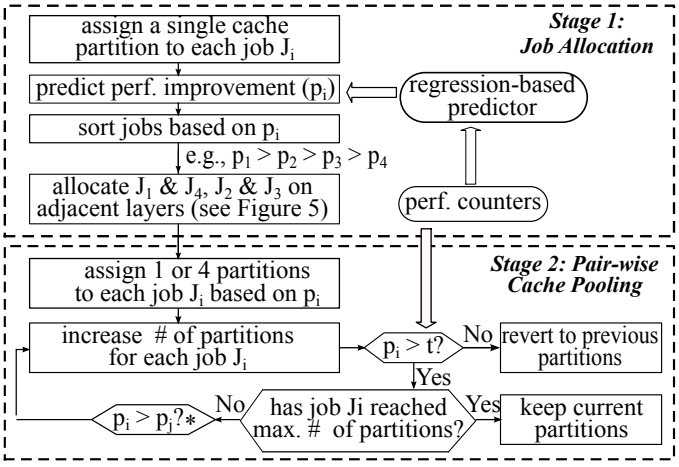


Fig. 4: A flow chart illustrating our runtime job allocation and cache resource pooling policy. * condition checked only if J_i and J_j are competing for the same partition.

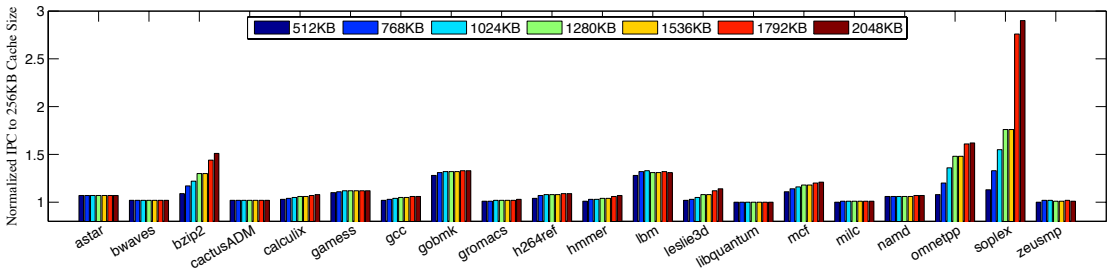


Fig. 3: IPC of SPEC benchmarks for increasing L2 cache size. The IPC values are normalized w.r.t using a 256KB L2 cache.

flow of our runtime job allocation and cache resource pooling policy in Figure 4. The policy contains two stages: (1) job allocation, which decides on which core each job should run on, and (2) cache resource pooling, which distributes a pool of cache partitions among a pair of applications.

Stage 1: Job Allocation across the Stack

In the first stage, we allocate the jobs to the 3D system with energy efficiency and thermal considerations. The allocation is based on an estimation of the jobs' IPC improvement (p_i) when running with 4 partitions compared to running with 1 partition. The estimation of p_i is conducted using an offline linear regression model which takes runtime performance counter data as inputs. We first assign n jobs to n cores in the 3D systems in a random manner, and start running the jobs for an interval (e.g., $10ms$) using the default reserved cache partition (each core has a single reserved L2 cache partition of 256KB that cannot be pooled). The performance counters that we use in our estimation are L2 cache replacements, L2 cache write accesses, L2 cache read misses, L2 cache instruction misses, and number of cycles. The linear regression model is constructed by their linear and cross items. We train the regression model with performance statistics from simulations across 15 of our benchmarks and validate the model using another 5 benchmarks. The prediction error is less than 5% of the actual performance improvement on average.

We then sort all the jobs with respect to their predicted performance improvements and group them in pairs by selecting the highest and lowest ones from the remaining sorted list. For example, in Figure 5, there are four jobs sorted as ($J_1 \geq J_2 \geq J_3 \geq J_4$) according to their p_i. We group these jobs into two pairs (J_1, J_4, and J_2, J_3). It is possible to integrate our allocation strategy with thermally aware heuristics (e.g., [3]) by placing job pairs with higher IPC closer to heat sink. In this case, we allocate the job pair with higher average IPC to the available cores closest to heat sink as shown in Figure 5. The reason for this decision is that the cores on layers closer to the heat sink can be cooled faster in comparison to cores farther from the heat sink [3].

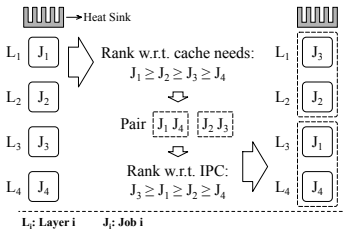


Fig. 5: An example to illustrate the job allocation stage.

Stage 2: Cache Resource Pooling Among Application Pairs

In the second stage of our policy, we propose a method to manage the cache resources within each job pair. In order to determine whether a job needs more cache partitions, we first introduce a performance improvement threshold (t). This threshold represents the minimum improvement that results in a lower EDP when the job uses an additional partition.

The key to derive t is based on the following assumption: The EDP of cache-hungry workloads decreases when the number of cache partitions available to the job increases. Thus, the following inequality should be satisfied:

$$\frac{Power}{IPC^2} > \frac{Power + \triangle Power}{(IPC + \triangle IPC)^2} \quad (1)$$

IPC and $Power$ refer to performance and power values before we increase the number of cache partitions, while $\triangle IPC$ and $\triangle Power$ are the variations in IPC and power when the job uses an additional partition. From this inequality, we obtain:

$$\frac{\triangle IPC}{IPC} > t = \sqrt{1 + \frac{\triangle Power}{Power}} - 1 \quad (2)$$

When performance improvement is larger than t, increasing the number of partitions reduces the EDP of the job. We compute t as 3% on average based on our experiments with 20 SPEC benchmarks.

We compute how many cache partitions to assign to each job by utilizing the threshold and p_i. If p_i of one job is greater than 9%, we assign 4 cache partitions to it; otherwise, we keep 1 partition for the job. The 9% is obtained from the threshold of increasing the partition from 1 to 4. Then, we iteratively increase the cache partitions for each job if three conditions are satisfied: (1) $p_i > t$, (2) the job has not reached the maximum number of partitions, and (3) $p_i > p_j$. The maximum number of partitions is 7 for jobs that are assigned with 4 partitions, while 4 for jobs that are assigned with 1 partition. If $p_i < t$, we revert the job to previous partitions. And we keep the job with current partition once it reaches the maximum number of partitions. The last condition only checked if jobs J_i and J_j are competing for the same partition.

We illustrate an example cache assignment where one job in a job-pair is assigned 1 partition and the other job is assigned 4 partitions in Figure 6. In step i, the performance improvements of both jobs are greater than the threshold, we increase one cache partition for both $Core_1$ and $Core_2$ as shown in step ii. Then, since they are completing the last available cache partition, we assign the cache partition to the job with higher performance improvement ($Core_1$).

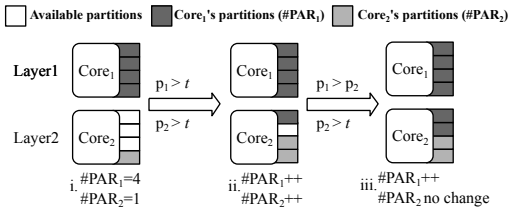


Fig. 6: An example that illustrates the cache resource pooling among a job-pair in our runtime policy. $\#PAR$ demonstrates the number of partitions.

Job Allocation on Larger 3D Systems

When there are multiple cores in one layer in the 3D system, we call all the cores vertically stacked in the 3D architecture a *column*. For such cases, we add an extra step after sorting the jobs to balance the cache-hungriness among the columns using a load balancing policy. For example, in a 16-core 3D system with 4 layers, we have 4 columns namely C_1, C_2, C_3, and C_4. Columns C_1 and C_4 initially have 4 and 3 cache-hungry jobs, while columns C_2 and C_3 only have 1 cache-hungry job each. After the inter-column job reallocation, two jobs in C_1 are swapped with two jobs in C_2 and C_3, thus balancing the cache-hungriness. Using this inter-column job allocation, the cache needs are balanced and the cache resources can be utilized more efficiently.

In order to improve the energy efficiency of the 3D system in presence of workload changes, we repeat our runtime policy every $100ms$. We re-allocate the cache partitions among application pairs and flush the cache partitions whenever there is a re-allocation. In the worst case, we decrease the number of cache partitions for a job from 4 to 1, which results in the cache partitions flushed 3 times. The performance overhead is from both the job migrations in the job allocation stage and the cache partitions in the cache resource pooling stage, which is dominated by the cold start effect in caches. Prior work estimates the cold start effect of a similar SPEC benchmark suite as less than $1ms$ [20]. Thus, the performance overhead of our policy is negligible for SPEC type of workloads.

V. Experimental Methodology

We evaluate our results on both high-performance and low-power multicore 3D systems that have 4 to 16 cores. The core architecture for low-power system is based on the cores in Intel SCC [21]. For the high-performance system, we model the core architecture based on the AMD Family 10h microarchitecture used in the AMD Magny-Cours processor. The architectural parameters for cores and caches are listed in Table I.

We use the Gem5 simulator [22] to build the performance simulation infrastructure for our target systems, and use the system-call emulation mode in Gem5 with X86 instruction set architecture. We conduct single-core simulations in Gem5 with various L2 cache sizes. We estimate the performance results of the 4-core 3D system with cache resource pooling

TABLE I: Core Architecture Parameters

Parameter	High-Perf	Low-Power
CPU Clock	2.1GHz	1.0 GHz
Issue Width	out-of-order 3-way	out-of-order 2-way
Reorder Buffer	84 entries	40 entries
BTB/RAS size	2048/24 entries	512/16 entries
Integer/FP ALU	3/3	2/1
Integer/FP MultDiv	1/1	1/1
Load/Store Queue	32/32 entries	16/12 entries
L1 I/DCache	64KB, 2way	16KB, 2way
L2 Cache	1MB, 4way	1MB, 4way

TABLE II: Workload compositions

Workload	Benchmarks
non-cache-hungry1	bwaves gamess libquantum zeusmp
non-cache-hungry2	calculix milc namd leslie3d
low-cache-hungry1	leslie3d libquantum gamess omnetpp
low-cache-hungry2	zeusmp hmmer namd bzip2
med-cache-hungry1	astar h264ref soplex mcf
med-cache-hungry2	bzip2 cactusADM hmmer omnetpp
high-cache-hungry1	gromacs bzip2 omnetpp soplex
high-cache-hungry2	h264ref bzip2 omnetpp soplex
all-cache-hungry1	soplex soplex omnetpp bzip2
all-cache-hungry2	soplex bzip2 soplex bzip2

by configuring the memory bus width in the single-core case as one-fourth of the 4-core system's bus width. We compare our estimation of using single-core simulation results against running the 4-core simulation in Gem5 and observe that the average IPC error is limited to 1.7%. We fast-forward each benchmark for 2 billion instructions for warm up and execute with the detailed out-of-order CPUs for 100 million instructions. We use McPAT 0.7 [23] for 45nm process to obtain the dynamic power of the cores. The L2 cache power is calculated using CACTI 5.3 [24], and the dynamic power is scaled using L2 cache access rate. We calibrate the McPAT dynamic core power using the published power value for Intel SCC and AMD. We also model the temperature impact on leakage power using an exponential formula [25].

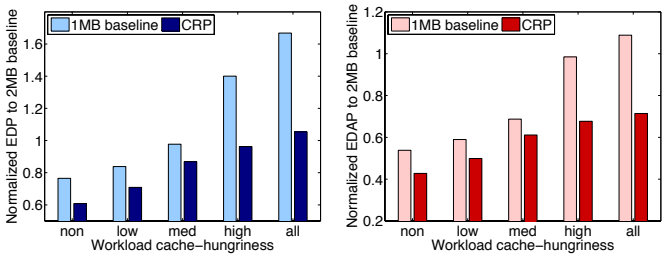
We select 20 applications from the SPEC 2006 benchmark suite as listed in Figure 3. We further compose 10 groups of multi-program workload sets with 4 threads, by combining cache-hungry applications with other applications that are not cache-hungry as shown in Table II.

VI. Results

This section presents the experimental results for our cache resource pooling design and runtime job allocation policy. In our 3D architecture with cache resource pooling (3D-CRP), each core has a 1MB private L2 cache. We use two 3D systems with homogeneous layers as baselines, on which each core has static 1MB and 2MB private L2 cache respectively.

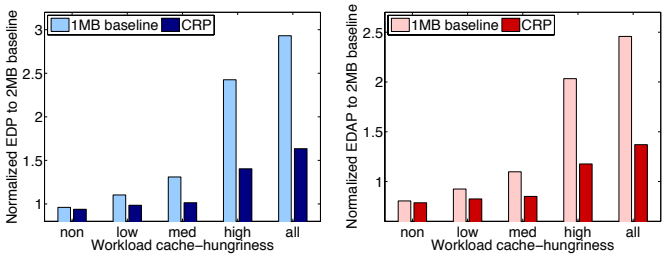
Figure 7 (a) presents the energy efficiency benefits of the 3D-CRP for the low-power system. We see that for all the workloads, 3D-CRP provides lower EDP in comparison to the 1MB baseline. For all-cache-hungry workload, 2MB baseline provides the best EDP because of the larger cache size. Our results show that 3D-CRP reduces EDP by up to 36.9% and 39.2% compared to 1MB and 2MB baselines, respectively.

Area is a very important metric for evaluating the 3D systems because die costs are proportional to the 4^{th} power of the area [26]. We use EDAP as a metric to evaluate the



(a) Normalized EDP (b) Normalized EDAP

Fig. 7: EDP and EDAP of low-power 3D system with cache resource pooling and its 3D baseline with 1MB static caches, normalized to its 2MB baseline.



(a) Normalized EDP (b) Normalized EDAP

Fig. 8: EDP and EDAP of high-performance 3D system with cache resource pooling and its 3D baseline with 1MB static caches, normalized to its 2MB baseline.

energy area efficiency [23]. As shown in Figure 7 (b), 3D-CRP outperforms both baseline systems for all workload sets, reducing EDAP by up to 57.2% compared to the 2MB baseline.

We also evaluate our 3D-CRP design and runtime policy on a high-performance 3D system. Figure 8 presents the EDP and EDAP results for the high-performance 3D systems with cache resource pooling and its 3D baseline with 1MB static caches, in comparison to the 2MB baseline. We observe that the EDP and EDAP reduction when applying cache resource pooling and our runtime policy are lower than that in the low-power system. 3D-CRP achieves up to 6.1% EDP and 21.3% EDAP reduction in comparison to its 2MB baseline.

In order to investigate the scalability of our policy, we evaluate our runtime policy on the 16-core low-power 3D-CRP system. The 16-core 3D system has 4 layers with 4 cores on each layer, and each core has 1MB private L2 cache. The core and cache architectures are the same as in the 4-core 3D-CRP systems. We combine 16 SPEC benchmarks as a low-cache-hungry workload and use it to evaluate the EDP and EDAP of the 16-core lower-power 3D-CRP system. We observe that, the low-power 3D-CRP system running our runtime policy provides 19.7% EDP reduction and 43.5% EDAP reduction in comparison to the 3D baseline system with 2MB L2 caches.

We also investigate 3D systems with microarchitectural resource pooling (MRP) as proposed in [8]. In order to evaluate the performance improvement with MRP, we run applications with four times of the default sizes of the performance-critical components (ROB, instruction queue, register file, and load/store queue), and compare the IPC results with the results with default settings. For applications running on a single core, our experiments show that MRP improves system performance by 10.4% on average, and combining MRP and CRP provides additional performance improvement of 8.7% on average in comparison to applying MRP alone.

VII. Conclusion

In this paper, we have introduced a novel design for 3D cache resource pooling that requires minimal circuit and architectural modification. We have then proposed an application-aware job allocation and cache pooling policy to improve the energy efficiency of 3D systems. Our policy dynamically allocates the jobs to cores on the 3D stacked system and distributes the cache resources based on the cache hungriness of the applications. Experimental results show that by utilizing cache resource pooling we are able to improve system EDP and EDAP by up to 39.2% and 57.2% in comparison to 3D systems with static cache sizes.

Acknowledgements

This work has been partially funded by NSF grant CNS-1149703 and Sandia National Laboratories.

References

[1] B. Black *et al.*, "Die stacking (3D) microarchitecture," in *International Symposium on Microarchitecture (MICRO)*, pp. 469–479, 2006.

[2] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *ISCA*, pp. 453–464, 2008.

[3] A. K. Coskun, J. L. Ayala, D. Atienza, T. S. Rosing, and Y. Leblebici, "Dynamic thermal management in 3D multicore architectures," in *DATE*, pp. 1410–1415, 2009.

[4] E. Ipek *et al.*, "Core fusion: accommodating software diversity in chip multiprocessors," in *ISCA*, pp. 186–197, 2007.

[5] D. Ponomarev, G. Kucuk, and K. Ghose, "Dynamic resizing of superscalar datapath components for energy efficiency," *IEEE Transactions on Computers*, vol. 55, pp. 199–213, Feb. 2006.

[6] S. Zhuravlev *et al.*, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS*, pp. 129–142, 2010.

[7] J. Martinez and E. Ipek, "Dynamic multicore resource management: A machine learning approach," in *IEEE Micro*, vol. 29, Sept 2009.

[8] H. Homayoun *et al.*, "Dynamically heterogeneous cores through 3D resource pooling," in *HPCA*, pp. 1–12, 2012.

[9] C. Zhu *et al.*, "Three-dimensional chip-multiprocessor run-time thermal management," *TCAD*, vol. 27, August 2008.

[10] R. Das *et al.*, "Application-to-core mapping policies to reduce memory interference in multi-core systems," in *PACT*, pp. 455–456, 2012.

[11] K. Varadarajan *et al.*, "Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions," in *MICRO*, pp. 433–442, 2006.

[12] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, pp. 423–432, 2006.

[13] D. Chiou and et al., "Dynamic cache partitioning via columnization," in *TechReport, Massachusetts Institute of Technology*, 2000.

[14] R. Kumar, V. Zyuban, and D. Tullsen, "Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling," in *ISCA*, pp. 408–419, June 2005.

[15] G. Sun *et al.*, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *HPCA*, pp. 239–249, 2009.

[16] J. Meng, K. Kawakami, and A. Coskun, "Optimizing energy efficiency of 3-d multicore systems with stacked dram under power and thermal constraints," in *DAC*, pp. 648–655, 2012.

[17] J. Jung, K. Kang, and C.-M. Kyung, "Design and management of 3D-stacked NUCA cache for chip multiprocessors," in *GLSVLSI*, 2011.

[18] D. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *MICRO*, pp. 248–259, 1999.

[19] X. Zhao, J. Minz, and S.-K. Lim, "Low-power and reliable clock network design for through-silicon via (TSV) based 3D ICs," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 1, no. 2, pp. 247–259, 2011.

[20] A. K. Coskun *et al.*, "Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors," in *SIGMETRICS*, pp. 169–180, 2009.

[21] J. Howard *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *ISSCC*, pp. 108–109, 2010.

[22] N. L. Binkert *et al.*, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, pp. 52–60, July 2006.

[23] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, pp. 469–480, 2009.

[24] S. Thoziyoor *et al.*, "CACTI 5.1," tech. rep., April 2008.

[25] J. Srinivasan *et al.*, "The case for lifetime reliability-aware microprocessors," in *ISCA*, pp. 276–287, 2004.

[26] J. Rabaey, A. Chandrakasan, and B. Nikolic., *Digital Integrated Circuits: A Design Perspective, 2nd edition*. 2003.