Dynamic Cache Pooling in 3D Multicore Processors

TIANSHENG ZHANG, JIE MENG, and AYSE K. COSKUN, Boston University

Resource pooling, where multiple architectural components are shared among cores, is a promising technique for improving system energy efficiency and reducing total chip area. 3D stacked multicore processors enable efficient pooling of cache resources owing to the short interconnect latency between vertically stacked layers. This article first introduces a 3D multicore architecture that provides poolable cache resources. We then propose a runtime management policy to improve energy efficiency in 3D systems by utilizing the flexible heterogeneity of cache resources. Our policy dynamically allocates jobs to cores on the 3D system while partitioning cache resources based on cache hungriness of the jobs. We investigate the impact of the proposed cache resource pooling architecture and management policy in 3D systems, both with and without on-chip DRAM. We evaluate the performance, energy efficiency, and thermal behavior for a wide range of workloads running on 3D systems. Experimental results demonstrate that the proposed architecture and policy reduce system *energy-delay product* (EDP) and *energy-delay-area product* (EDAP) by 18.8% and 36.1% on average, respectively, in comparison to 3D processors with static cache sizes.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]

General Terms: Design

Additional Key Words and Phrases: Policy, energy efficiency, cache resource pooling, 3D stacking, runtime policy

ACM Reference Format:
Tiansheng Zhang, Jie Meng, and Ayse K. Coskun. 2015. Dynamic cache pooling in 3D multicore processors. ACM J. Emerg. Technol. Comput. Syst. 12, 2, Article 14 (August 2015), 21 pages.
DOI: http://dx.doi.org/10.1145/2700247

1. INTRODUCTION

3D integration technology is a promising design technique for integrating different technologies into a single system, increasing transistor density, and improving system performance [Black et al. 2006; Loh 2008]. Most prior work exploits the performance or energy efficiency benefits of 3D processors by considering fixed, homogeneous computational and memory resources (e.g., Black et al. [2006] and Loh [2008]). Fixed homogeneous resources, however, cannot always meet potentially diverse resource requirements (e.g., different cache and memory usage) for applications running on a system. Heterogeneous multicore design is proposed as a solution to this challenge by including cores with different architectural resources in one chip. Nevertheless, heterogeneous design is traditionally more challenging compared to homogeneous design, and also does not necessarily provide the desired flexibility to adapt to dynamic resource requirements.

Resource pooling, where components of a core are shared by other cores, enables *flexible heterogeneity* (each core can adjust its hardware resources flexibly) in a multicore processor, and thus is an alternative design technique. The flexible heterogeneity

This work has been funded by NSF grant CNS-1149703 and Sandia National Laboratories.
Authors' addresses: T. Zhang (corresponding author), J. Meng, A. K. Coskun, Department of Electrical and Computer Engineering, Boston University, Boston, MA 02215; email: tzhang@bu.edu.

DOI: http://dx.doi.org/10.1145/2700247

provided by resource pooling can address various resource requirements from applications by reconfiguring the resources among cores with the same architecture [Ipek et al. 2007; Ponomarev et al. 2006]. Due to substantial benefits in reducing total chip area and improving system energy efficiency, resource pooling has drawn attention in multicore processor design. Several resource pooling designs and scheduling strategies have been proposed for multicore processors (e.g., Zhuravlev et al. [2010] and Martinez and Ipek [2009]). However, in conventional 2D processors, resource sharing is limited by the long access latency of remote shared resources in the horizontal dimension. Sharing resources across the chip becomes particularly inefficient for a large number of cores and large chip sizes.

3D stacked processors include *through-silicon vias* (TSVs) to connect the layers. TSVs can be used for pooling resources among different layers, owing to their short length and low latency. A resource pooling technique for 3D systems with identical layers was recently proposed to allow vertically pooling performance-critical microarchitectural components such as register files and load/store buffers [Homayoun et al. 2012] using TSVs. This work, however, does not consider pooling cache or memory resources. Considering the significance of the memory resources to application performance, we believe that cache resource pooling can provide additional heterogeneity of resources among the cores at low cost and can bring substantial energy efficiency improvements.

In this article, we first propose a cache resource pooling architecture for 3D systems with identical layers. Then, we design a runtime policy for the proposed architecture to manage poolable cache resources according to the characteristics of workloads on the system (e.g., instructions retired and cache access behavior), while considering the trade-offs between performance and energy. Our contributions are as follows.

—We introduce a 3D cache resource pooling architecture where the 3D system consists of homogeneous logic layers. This architecture requires minimal additional circuitry and architectural modifications in comparison to using static cache resources.

—We propose a novel application-aware job allocation and cache pooling policy. Our policy predicts the cache resource requirements of different applications by collecting performance counter data at runtime and determines the most energy-efficient cache size for each application. The job allocation policy places jobs with complementary cache characteristics on adjacent layers in the stack to improve efficiency.

—We evaluate the proposed method on 3D systems both with and without DRAM stacking. We provide a memory latency computation model that uses an M/D/1 queuing model for accurate performance characterization of 3D systems with stacked DRAM.

—We evaluate the proposed dynamic cache resource pooling technique for both high-performance and low-power 3D multicore processors, explore its scalability to a 3D processor with a larger number of cores, and also investigate the corresponding thermal behavior. Our experimental results show that, for a 4-core 3D system, cache resource pooling reduces system *energy-delay product* (EDP) by 18.8% and system *energy-delay-area product* (EDAP) by 36.1% on average compared to using fixed cache sizes. For a 16-core 3D processor, our technique reduces EDP by up to 40.4%.

The rest of the article starts with an overview of related work. In Section 3, we discuss the motivation for runtime cache and DRAM management. Section 4 introduces the proposed 3D cache resource pooling architecture. Section 5 presents our application-aware workload allocation and cache resource pooling policy. Section 6 provides the experimental methodology. Section 7 quantifies the benefits of the proposed architecture and runtime policy, and Section 8 concludes.

2. RELATED WORK

Resource pooling and the corresponding runtime management in traditional 2D multicore systems have been studied extensively, while for 3D systems, resource pooling architectures and well-designed management policies remain as open problems.

2.1. Resource Pooling

Prior work on resource pooling has mainly focused on 2D multicore systems. Ipek et al. [2007] propose a reconfigurable architecture to combine the resources of simple cores into more powerful processors. Ponomarev et al. [2006] introduce a technique to dynamically adjust the sizes of performance-critical microarchitectural components, such as reorder buffer or instruction queue. However, as the number of on-chip cores increases, the long access latency between resources on 2D chips makes it difficult to get fast response from the pooled resources. In 3D architectures, stacking the layers vertically and using TSVs for communication enable short access latency among on-chip resources. Homayoun et al. [2012] were the first to explore microarchitectural resource pooling in 3D stacked processors for sharing resources at a fine granularity. Their work, however, does not investigate the potential of pooling cache resources. Since the cache is also a performance-critical component in computer systems, an architecture with cache resource pooling in 3D systems can bring performance improvements to the system. However, none of the prior work investigates cache resource pooling in 3D systems.

2.2. Cache Partitioning and Reconfiguration

Cache sharing and partitioning have been well studied in 2D multicore systems. Varadarajan et al. [2006] propose molecular caches that create dynamic heterogeneous cache regions. Qureshi and Patt [2006] introduce a low-overhead runtime mechanism to partition caches between multiple applications based on the cache miss rates. Chiou et al. [2000] propose a dynamic cache partitioning method via columnization. However, the benefits of cache sharing in 2D systems are highly limited by the on-chip interconnect latency. Kumar et al. [2005] 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 been investigated recently. Sun et al. [2009] explore the energy efficiency benefits of 3D stacked MRAM L2 caches. Prior work on 3D caches and memories either considers integrating heterogeneous SRAM or DRAM layers into 3D architectures (e.g., Meng et al. [2012] and Jung et al. [2011]) or involves major modifications to conventional cache design (e.g., Sun et al. [2009]). Compared to such heterogeneous 3D systems, a 3D system with homogeneous layers and resource pooling features is more scalable to a larger number of cores and a wider range of workloads.

2.3. Runtime Management of Multicore Systems

To manage on-chip resource pooling among cores, an intelligent runtime policy is required. Recent research on runtime policies in 2D systems generally focuses on improving performance and reducing the communication, power, or cooling cost through job allocation. For example, Snavely and Tullsen [2000] present a mechanism that allows the scheduler to exploit workload characteristics for improving processor performance. Das et al. [2012] propose an application-to-core mapping algorithm to maximize system performance. Bogdan et al. [2012] propose a novel dynamic power management approach based on the dynamics of queue utilization and fractional differential equations to avoid inefficient communication and high power density in networks-on-chip.

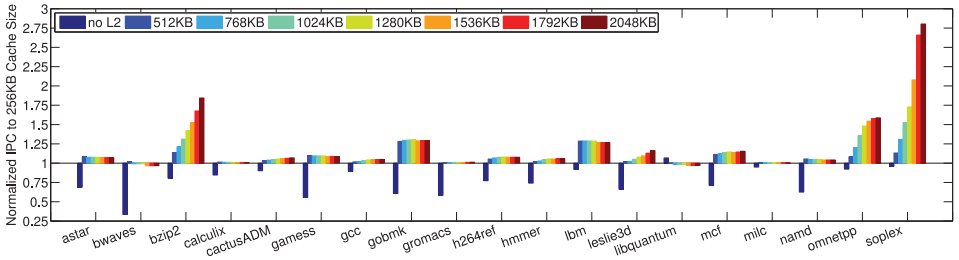


Fig. 1. IPC of SPEC CPU 2006 benchmarks for increasing L2 cache size. The IPC values are normalized with respect to using a 256KB L2 cache.

Dynamic job allocation on 3D systems mostly addresses the power density and thermal challenges induced by vertical stacking. For example, dynamic thermally aware job scheduling techniques use the thermal history of the cores to balance temperature and reduce hotspots [Coskun et al. 2009a; Zhu et al. 2008]. Hameed et al. [2011] propose a technique to dynamically adapt core resources based on application needs, and thermal behavior to boost performance while maintaining thermal safety. Prior work has not considered cache resource pooling among cores at runtime.

2.4. Distinguishing Aspects from Prior Work

To the best of our knowledge, our work is the first to propose a cache resource pooling architecture complemented with a novel dynamic job allocation and cache pooling policy in 3D multicore systems, which requires minimal hardware modifications. Our dynamic job allocation and cache pooling policy differentiates itself 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 maximum extent. Compared to our earlier work on cache resource pooling [Meng et al. 2013], we evaluate the performance, energy efficiency, and thermal behavior of multicore 3D systems both with and without DRAM stacking. We also compare our research with the most related previous work: selective way cache architecture [Albonesi 1999]. In addition, this article improves the performance model of the 3D system with stacked DRAM by introducing a detailed, accurate memory access latency model for on-chip memory controllers.

3. MOTIVATION

Modern processors get significant performance improvement from caches. Generally speaking, CPU performance is increasing along with cache size. However, larger caches consume higher power and bring varying performance improvements for different applications due to their varying cache usage. Thus, depending on the applications, the optimal cache size to achieve the best *energy-delay product* (EDP) may differ. In homogeneous 3D stacked systems, each core on each layer has a fixed-size L2 cache which potentially restrains the system's performance and energy efficiency. In this section, we investigate the impact of L2 cache size on performance and energy efficiency of various applications. Although we focus on the L2 cache in this article, our work can also be applied to other levels of cache in computer systems.

To quantify the impact of L2 cache, we simulate a single core with varying L2 cache size (from 0KB to 2048KB with a step of 256KB) and compare performance and power results for applications in the SPEC CPU 2006 benchmark suite. We use Gem5 [Binkert et al. 2006] for performance simulation, and McPAT [Li et al. 2009] and CACTI 5.3 [Thoziyoor et al. 2008] for computing the core and cache power, respectively (details of our simulation methodology are presented in Section 6). Figure 1 shows the

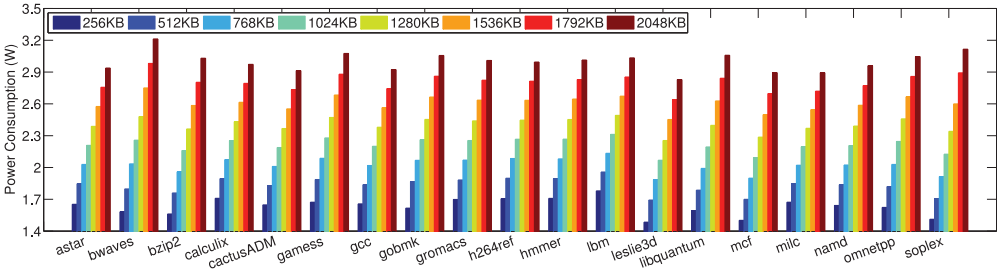


Fig. 2. Power consumption of SPEC CPU 2006 benchmarks under cache size from 256KB to 2048KB.

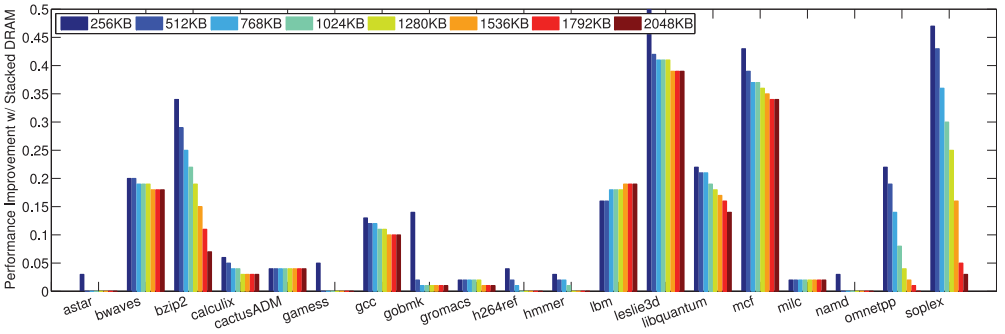


Fig. 3. IPC improvement of SPEC CPU 2006 benchmarks for changing to stacked DRAM.

normalized IPC of all applications under different L2 cache configurations. As shown in Figure 1, among all applications, *soplex, omnetpp*, and *bzip2* have significant performance improvement at large L2 cache size of up to 1.8x. We call such applications *cache-hungry* applications. On the other hand, applications such as *bwaves* barely benefit from an L2 cache larger than 256KB. Figure 2 shows the total power consumption for all applications under the same cache configurations. For all applications, power consumption increases as the L2 cache size goes up. Figure 1 and Figure 2 indicate that, while some applications' EDP strongly benefit from large caches, others have marginal or no benefits. Such variance of IPC and power motivates tuning the cache size used by applications to optimize system EDP.

Furthermore, the memory access behavior also differs among applications, thus the memory architecture is another factor affecting performance and energy efficiency. Due to the small area of TSVs, an on-chip 3D stacked DRAM architecture [Loh 2009] enables multiple on-chip memory controllers, which allows for parallel memory accesses. As a result, the average queuing latency in memory controllers is shortened and system performance is significantly improved. Figure 3 shows the performance improvement for applications when changing the memory access latency from off-chip memory to on-chip stacked memory, and Figure 4 shows the L2 *miss per kilo-instruction* (MPKI) of applications under different cache configurations with off-chip memory. The figures demonstrate that the performance improvement of using on-chip memory is highly dependent on the L2 MPKI of the application. Applications such as *astar, calculix*, and *hmmer* do not gain obvious performance improvement by changing from off-chip DRAM to stacked DRAM, while some other applications' performance is quite sensitive to the memory architecture (e.g., *bwaves, gcc, libquantum*). As for *bzip2, omnetpp*, and *soplex*, they benefit more from stacked DRAM when they have less cache resources. Memory access rate directly influences the queuing delay in memory controllers. Hence, if all

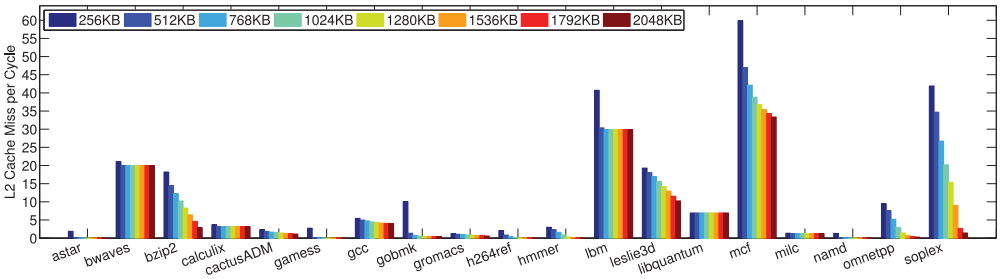
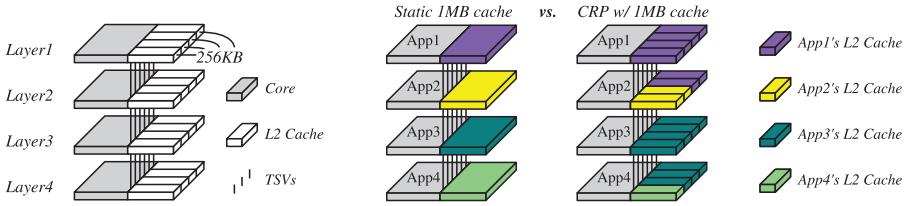


Fig. 4. L2 MPKI of SPEC CPU 2006 benchmarks under cache size from 256KB to 2048KB.



(a) 1MB cache pooling

(b) cache resource allocation using static cache and CRP

Fig. 5. Proposed 3D system with cache resource pooling versus 3D systems with static 1MB caches. In (a), cores are able to access caches on adjacent layers through the TSVs.

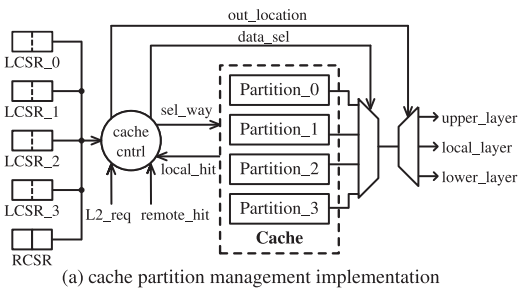
cores have a high memory access rate, the memory controller queuing delay increases dramatically. Therefore, in the 3D stacked systems we also need to consider the memory access intensity for each memory controller.

4. PROPOSED 3D STACKED ARCHITECTURE WITH CACHE RESOURCE POOLING

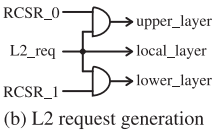
In this section, we propose a homogeneous 3D architecture that enables vertical *cache resource pooling* (CRP). The proposed architecture is demonstrated using a four-layer 3D system that has one core with a private 1MB L2 cache on each layer. Vertically adjacent caches are connected via TSVs for cache pooling, as shown in Figure 5(a). On-chip communication is performed through shared memory. Thus, all private L2 caches are connected to a shared memory controller. Figure 5(b) shows an example case of the differences in cache resource allocation between the systems with static L2 caches and CRP. In this case, Applications 1 and 3 require larger caches than the other two applications, and thus acquire extra cache resources from their adjacent layers in the CRP architecture. In contrast, in a system with static L2 caches, an application can only work with a fixed amount of cache.

4.1. 3D-CRP Design Overview

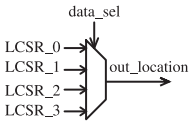
Enabling cache resource pooling in 3D systems requires some modifications to the conventional cache architecture. The modified cache architecture allows cores in the homogeneous 3D stacked system to increase their private L2 cache sizes by pooling the cache resources from the other layers at negligible access latency penalty. The objective of our design is to improve the system energy efficiency, which can be divided into two aspects: (1) to improve the performance by increasing cache size for cache-hungry applications, and (2) to save power by turning off unused cache partitions for noncache-hungry applications. We focus on pooling L2 caches because L2 cache usage varies significantly across applications, as shown in Section 3. It is possible to extend the strategy to other levels of data cache.



(a) cache partition management implementation



(b) L2 request generation



(c) output location generation

LCSR	Cache Partition Status
00	Turned off
01	Used by local layer
10	Used by lower layer
11	Used by upper layer

(d) Local cache status

Fig. 6. Cache resource pooling implementation.

Cache size is determined by cache-line size, number of sets, and level of associativity. In this design we adjust cache size by changing the cache associativity. We base our architecture on the selective way cache architecture proposed in prior work [Albonesi 1999], which aims at turning off unnecessary cache ways for saving power in 2D systems. We call each cache way a *cache partition* in our design. 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 cache resources. We also do not allow a core to pool cache 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 would not bring considerable performance improvement.

4.2. 3D Cache Partition Management Implementation

In order to implement cache resource pooling in 3D systems, we introduce additional hardware components to the conventional cache architecture. As shown in Figure 6, we make modifications to both cache status registers and cache control logic.

For 3D CRP, the cores need to be able to interact with cache partitions from the local layer and remote layers. First, 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 local cache partitions. There are four possible statuses for each local cache partition: *used by local layer, used by upper layer, used by lower layer*, and *turned off*. Each LCSR keeps two bits to indicate the current status of the corresponding partition as listed in the table in Figure 6(d). The status of local cache partitions is used for deciding the output location of the output data and hit signals. Second, we introduce *remote cache status registers* (RCSRs) for the L1 cache so that the L1 cache is aware of its remote L2 cache partitions when sending L2 requests. We maintain two 1-bit RCSRs in L1 caches for each core. L1 I- and D-caches can use the same RCSR bits, as both caches' misses are directed to the L2 cache. 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 the corresponding adjacent layer in addition to those in the local layer. RCSR_0 denotes the upper layer and RCSR_1 the lower. The values of the registers are set by the runtime management policy, which we discuss in Section 5.

Using the information from LCSRs and RCSRs, the cores are able to communicate with cache partitions from multiple layers. When there is an L1 miss, the core sends this request and the requested address based on the values of RCSRs, as shown in Figure 6(b). Once the requests and addresses arrive at the cache controller, the tag from the requested address is compared with the tag array. At the same time, the

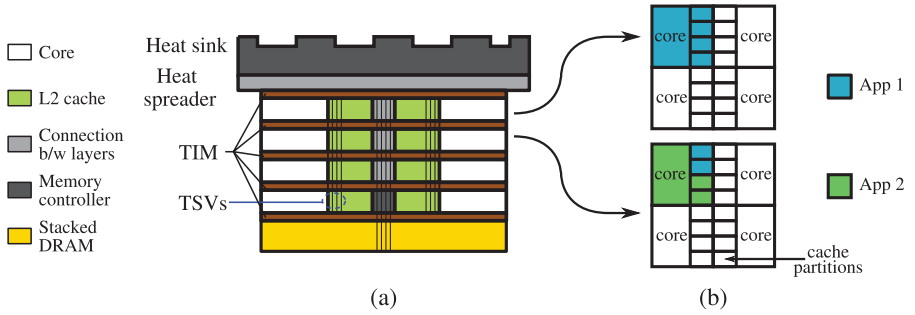


Fig. 7. (a) A cross-section view of large 3D-CRP system; (b) an example showing cache resource pooling within a column.

entries of each way are 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 6(c). When there is an 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 and remote hit signals are 0, this indicates an L2 miss.

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

4.3. Larger 3D-CRP Systems with On-Chip DRAM

We call all the cores vertically stacked in the 3D architecture a *column*. In the single-column system, we apply off-chip DRAM because the chip area is not big enough to hold sufficient memory (e.g., 1GB). For a larger system with more cores organized in columns, the memory access rate increases as the number of cores increases, which results in longer memory access latency. Since stacked DRAM enables multiple memory controllers in the system, it helps reduce the average memory access latency for larger 3D systems. Figure 7 shows a cross-section of a large 3D-CRP system and an example of cache resource pooling within a column, respectively, in (a) and (b), using a 16-core system as an example. Stacked DRAM layers are located at the bottom of the 3D-CRP system and there are four memory controllers on the bottom logic layer, one for each column. However, the workload of each column may have a different memory access rate depending on the applications running on it. In this situation, the memory access latency of different columns differs. The column with jobs that all have a high memory access rate suffers long memory access latency while the column with non-memory-intensive jobs does not. Through job allocation, we can migrate a number of memory-intensive jobs to the column with low memory access rate so as to decrease the memory access latency and thus improve performance. Therefore, a policy for monitoring and adjusting job allocation in the aspect of memory accesses is necessary for such designs.

4.4. Implementation Overhead Evaluation

To evaluate the area overhead of our proposed design, we assume each 1-bit register requires 12 transistors, each 1-bit 4-to-1 multiplexer requires 28 transistors, and each 1-bit 2-to-1 multiplexer has 12 transistors. We need 10 1-bit transistors for LCSRs and RCSRs, one 64-bit 1-to-4 demux and one 64-bit 4-to-1 mux for data transfers, one 30-bit

1-to-4 demux and one 30-bit 4-to-1 mux for address transfers (for 4GB memory, we need 32-bit demux and mux), one 2-bit 4-to-1 mux for output location selection, one 1-to-2 demux for sending back hit signals to remote layers, and 2 *AND* gates for generating L2 cache requests. Thus, the total number of transistors needed by the extra registers and logic in our design is limited to 5460 ($10\times$1-bit register+ $2\times$64-bit 1-to-4 demux + $2\times$30-bit 4-to-1 mux + $1\times$2-bit 4-to-1 mux + $1\times$1-bit 1-to-2 demux + $2\times$*AND* gate). We assume there are 128 TSVs for two-way data transfer between caches, 60 TSVs for the memory address bits, and 4 additional TSVs for transferring L2 requests and hit bits between the caches on adjacent layers. To connect to a memory controller, we assume there are 30 TSVs for memory address bits, and 512 TSVs for receiving data from the memory controller. TSV power has been reported much lower compared to the overall power consumption of the chip [Zhao et al. 2011], thus we do not take TSV power into account in our simulations. 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 [Homayoun et al. 2012] shows that the layer-to-layer delay caused by TSVs is $1.26ps$, which has no impact on system performance as it is much smaller than the CPU clock period at 1 GHz. If there is on-chip DRAM, the memory controller is also connected to DRAM through TSVs, which brings an extra 512 TSVs for data transmission and 32 TSVs for sending commands to the memory module.

5. RUNTIME CACHE RESOURCE POOLING POLICY

To effectively manage the cache resources and improve the energy efficiency of 3D systems with CRP architecture, we introduce a runtime job allocation and cache resource pooling policy. We first explain the details of the proposed policy for the 3D CRP system shown in Figure 5, where each layer has a single core and a 1MB L2 cache. Then we introduce the extended policy for larger 3D CRP systems.

5.1. Overview of Cache Pooling

Based on the fact that different applications have various requirements on cache resources to achieve their optimal energy efficiency, as stated in Section 3, we propose a two-stage runtime policy to allocate the cache resources within a single-column 3D CRP system. The flowchart is shown in Figure 8. The policy contains two stages: (1) *job allocation*, which decides the core each job should run on, and (2) *cache resource pooling*, which distributes the cache resources among a pair of jobs.

Stage 1: Job Allocation across the Stack

In this stage, we allocate the jobs to the 3D system with both 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 that takes runtime performance counter data as input. As a first step, we assign n jobs to n cores in the 3D system 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 reserved L2 cache partition of 256KB that cannot be pooled). The performance counters that we use in 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 applications and validate the model using another 4 applications. The prediction error is less than 5% of the actual performance improvement on average. When implemented in a real system, this

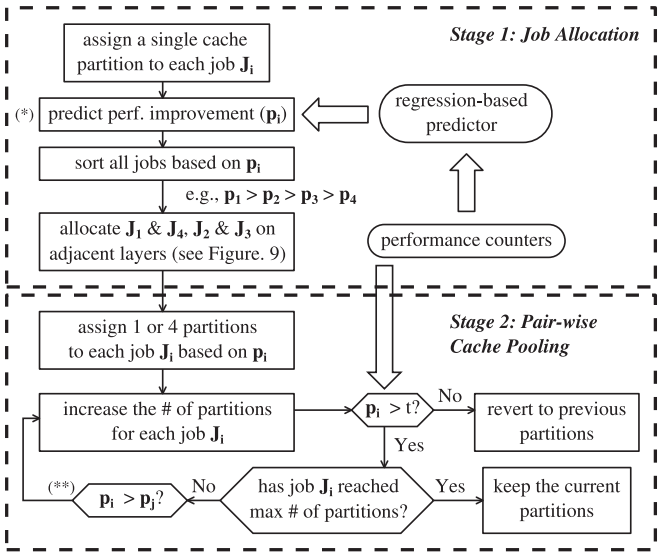
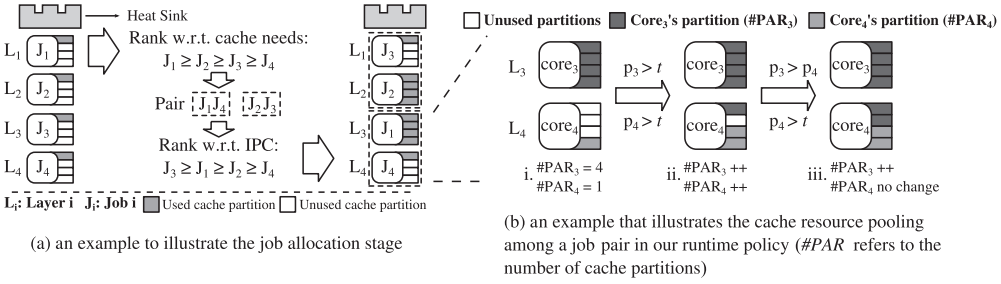


Fig. 8. A flowchart illustrating our runtime job allocation and cache resource pooling policy. (*) p_i represents the predicted IPC improvement for each job when running with 4 cache partitions compared to running with 1 partition. (**) Condition is checked only if J_i and J_j are competing for the same partition.



(a) an example to illustrate the job allocation stage

(b) an example that illustrates the cache resource pooling among a job pair in our runtime policy (*#PAR* refers to the number of cache partitions)

Fig. 9. The 2-stage intra-column runtime job allocation and cache resource pooling policy.

predictor can be integrated with the OS. The OS needs to periodically read the hardware performance counters to collect data and provide feedback to the predictor.

We then sort 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 as ($J_1 \geq J_2 \geq J_3 \geq J_4$) according to their p_i. In this case, we group four jobs into two pairs (J_1, J_4 and J_2, J_3). For the sake of temperature consideration, we allocate that job pair with higher average IPC to the available cores closest to the heat sink as shown in Figure 9(a). The reason is that those cores on layers closer to the heat sink can be cooled faster in comparison to cores farther from the heat sink [Coskun et al. 2009a].

Stage 2: Cache Resource Pooling among Job Pairs

In the second stage of the 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 IPC improvement a job should get from an extra cache partition to achieve a lower EDP. The key to derive t is based on the following assumption: The EDP of cache-hungry jobs decreases when the number of cache partitions of the job increases

due to great performance improvement. On the contrary, for noncache-hungry jobs, the EDP increases when the acquired cache partitions increase because the performance is only slightly improved while the energy consumption increases. To obtain a lower EDP, the following inequality should be satisfied.

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

Here *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

$$\frac{\triangle IPC}{IPC} > \sqrt{1 + \frac{\triangle Power}{Power}} - 1 = t. \tag{2}$$

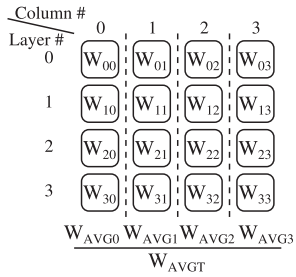
When performance improvement is larger than t, increasing the number of partitions reduces the EDP of the job. We compute t as 3% on average based on our experiments with 19 SPEC benchmarks. We compute the amount of cache partitions to assign to each job by utilizing the threshold and p_i. If p_i of one job is greater than 9.3% $((1 + 3\%)^3 - 1)$, 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 number of 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. We keep the job with current partitions once it reaches the maximum number of partitions. The last condition is only checked if jobs J_i and J_j are competing for the same cache 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 9(b). In step i, the performance improvements of both jobs are greater than the threshold, so we increase one cache partition for both $Core_3$ and $Core_4$, 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 ($Core_3$ in this case).

5.2. Inter-Column Job Allocation on Larger 3D CRP Systems

For larger 3D CRP systems with multiple columns, the cache requirements and performance of cores might be different across the columns. To balance cache hungriness among the columns, we perform inter-column job allocation after sorting the jobs as a load balancing policy in such systems.

We first assign weights to each core according to the corresponding cache requirements. Then we average the weights for each column (W_{AVGi}) and the whole 3D system (W_{AVGT}) and compare each W_{AVGi} with W_{AVGT} to see the difference. A threshold is set up to check whether the difference between W_{AVGi} and W_{AVGT} is large. If the threshold is exceeded, the highest-weight task in the column with the largest W_{AVGi} and the lowest-weight task in the column with the smallest W_{AVGi} are swapped to balance the cache hungriness. This process is iterated until the difference between each W_{AVGi} and W_{AVGT} is under the threshold. We conduct job migration if needed after the iteration converges. The algorithm is shown in Figure 10. After the inter-column job allocation, the system pairs the jobs and decides the cache resource allocation inside each column, as stated in previous sections. Furthermore, from the memory access perspective, since jobs within one column could have higher average memory access than those in the other columns, the memory access latency of this particular column may be potentially



Inter-Column Reallocation:

while (**if** $|\mathbf{W_{AVGi}}-\mathbf{W_{AVGT}}| \geq$ *threshold* for any i = 0, 1, 2, 3)
do
1. Sort $\mathbf{W_{AVGi}}$;
2. Take the task with the largest weight in the column which has the largest $\mathbf{W_{AVGi}}$ as task 1;
3. Take the task with the smallest weight in the column which has the smallest $\mathbf{W_{AVGi}}$ as task 2;
4. Swap task 1 and task 2;
5. Compute the new $\mathbf{W_{AVGi}}$ for i = 0, 1, 2, 3;

endwhile

Fig. 10. Illustration of inter-column job allocation algorithm.

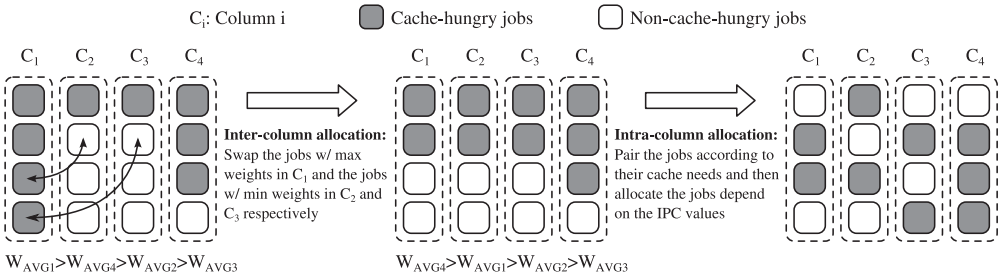


Fig. 11. An example of job allocation in a 16-core 3D system.

higher than the latency of the other columns. Therefore, when doing the inter-column job allocation, we also take the L2 *miss per cycle* (MPC) values into consideration. We balance the L2 MPC values as well as the cache hungriness so as to balance the memory accesses among all columns.

Figure 11 shows a simple example of job allocation in a 16-core 3D system with a cache resource pooling architecture. In this system, we have 4 columns and each column has 4 cores; the columns are named C_1, C_2, C_3, and C_4. Columns C_1 and C_4 initially have 4 and 3 cache-hungry jobs, respectively, while columns C_2 and C_3 only have 1 cache-hungry job each. We assume that, in this case, $W_{AVG1} > W_{AVG4} > W_{AVG3} > W_{AVG2}$. The job with the highest weight in C_1 is swapped with the job with the lowest weight in C_2 and we get $W_{AVG1} > W_{AVG4} > W_{AVG2} > W_{AVG3}$. Similarly, we swap the new highest-weight job in C_1 with the lowest-weight job in C_3 this time, and get the difference between each W_{AVGi} and W_{AVGT} under the threshold. After the inter-column job reallocation, the cache hungriness is balanced across the columns. Then we perform the proposed intra-column job pairing and allocation to finalize the location of each job. In a larger 3D system, using this inter-column job allocation, the cache needs are balanced and the cache resources can be utilized more efficiently.

5.3. Performance Overhead Evaluation

In order to improve the energy efficiency of the 3D system in presence of workload changes, we repeat our runtime policy every 100*ms*. We re-allocate the cache partitions among job 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, or increase the cache partitions from 4 to 7, which both result in the cache partitions flushed 3 times. Following a new cache configuration, there is no relevant data in the L2 cache. Thus the applications begin to execute with cold caches. The performance is degraded due to the cold-start effect in caches. Prior work estimates the cold-start effect of a similar SPEC benchmark suite as less than 1*ms* [Coskun et al. 2009b].

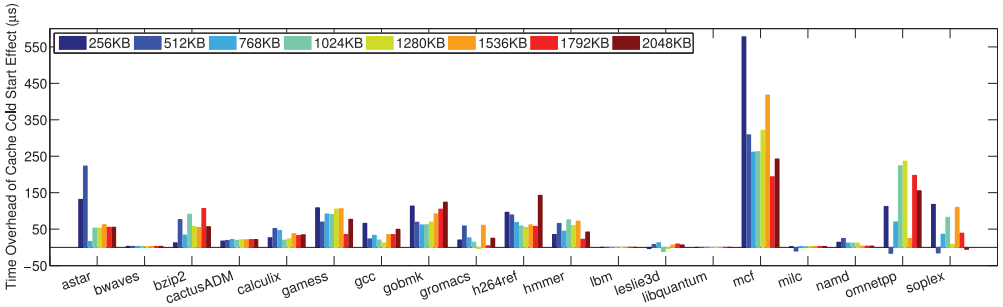


Fig. 12. Time overhead (μs) of cache cold-start effect for all applications under cache size from 256KB to 2048KB.

We also evaluate the cold-start effect overhead by comparing the performance of the benchmark suite both with and without cache warmup, as shown in Figure 12. We can see from this figure that almost all applications suffer cache cold-start effects, but in different amounts. For example, *mcf* suffers most from cold caches because it is much more cache intensive compared to the other applications. On the contrary, *lbm* almost has no cold-start overhead because it does not use much cache. The highest overhead is around $500\mu s$ from *mcf*. For most of the applications, the overhead is lower than $150\mu s$. For multicore systems, the memory access rate is higher than for single-core systems, which increases memory access latency. Thus, we also perform similar experiments for various memory access latencies and the results demonstrate that the cache warmup overhead is still under $1ms$. Other than the cache cold-start effect, when job migration happens, context switch also introduces performance degradation. However, the context switch overhead is no more than $10\mu s$ [Constantinou et al. 2005; Kamruzzaman et al. 2011], and techniques such as *fast trap* can further reduce time spent on it [Gomaa et al. 2004]. Thus the performance overhead of our policy is negligible for SPEC-type workloads.

6. EXPERIMENTAL METHODOLOGY

6.1. Target System

We apply the proposed cache resource pooling technique on both low-power and high-performance 3D multicore systems with 4 and 16 cores, respectively. The core architecture for the low-power system is based on the core in Intel SCC [Howard et al. 2010]. As for the high-performance system, we use the core architecture applied in the AMD Magny-Cours processor [Conway et al. 2009]. The architecture parameters for both systems are listed in Table I. The core areas are from the published data. For the 4-core 3D-CRP system, all 4 cores are stacked in one column using off-chip DRAM. In the 16-core system, there are 4 layers and each layer has 4 cores. Thus, there are 4 columns in the system and cores can pool cache resources within each column. Each column in the 3D system has a memory controller which is located on the layer farthest from the heat sink, as shown in Figure 7. The stacked DRAM layers are placed at the bottom of the 3D system, as described in Section 4. Due to the area restriction, the low-power 3D system needs 2 DRAM layers to have 1GB DRAM while the high-performance system only needs one.

6.2. Simulation Framework

For our performance simulation infrastructure, we use the system-call emulation mode in the Gem5 simulator [Binkert et al. 2006] with X86 instruction set architecture. For the single-core simulations shown in Section 3, we fast-forward 2 billion instructions

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, 2-way, 2ns	16KB, 2-way, 2ns
L2 Cache	1MB, 4-way, 5ns	1MB, 4-way, 5ns
Core Area	15.75 mm^2	3.88 mm^2

Table II. Main Memory Access Latency for the 3D CRP System

LLC-to-MC	0ns (due to the short latency provided by TSVs)
Memory Controller	Queuing delay, computed by M/D/1 queuing model
Main Memory	On-chip 1 GB DRAM: $t_{RAS} = 36ns, t_{RP} = 15ns$
Total Delay	Queuing delay + t_{RAS} + t_{RP}
Memory Bus	On-chip memory bus, 2GHz, 64-byte bus width

and then execute 100 million instructions in detailed mode for all applications under L2 cache sizes from 0 to 2MB. For 4- and 16-core simulations with the proposed CRP technique, we also collect performance metrics from the same segment of instructions. We run McPAT 0.7 [Li et al. 2009] under 45nm process for cores' dynamic power consumption and then calibrate the results using the published power values. We use CACTI 5.3 [Thoziyoor et al. 2008] to compute the L2 cache's power and area, and scale the dynamic L2 cache power based on L2 cache access rate. We use HotSpot 5.02 [Skadron et al. 2003] for thermal simulation.

In this work we apply the M/D/1 queuing model for each memory controller to model the queuing delay rather than using a unified memory latency for all multiprogram workload sets. In the M/D/1 model, arrival rate (λ) and service rate (μ) are required to compute the queuing delay $t_{queuing}$, as shown next.

$$t_{queuing} = \frac{\lambda}{2\mu(\mu - \lambda)} \tag{3}$$

In the proposed 3D-CRP system, there is one memory controller in each column, thus for multiprogram workloads we sum up the memory access rate of each core in the column as the arrival rate to the corresponding memory controller. We use the DRAM response time ($t_{RAS} + t_{RP}$) as the memory system service rate. For each multiprogram workload, we first assign a sufficiently large value as the memory access latency to ensure that the arrival rate will not exceed the memory system service rate. Then we run performance simulations and collect the memory access rate of the workload. Based on this arrival rate (λ_1) we compute the queuing delay (t_1) of the memory controller. The new memory access latency is the sum of LLC-to-MC delay, DRAM module access time, and the queuing delay ($t_1 + t_{RAS} + t_{RP}$), as shown in Table II. Then we feed back this new latency to Gem5 and collect the arrival rate (λ_2) from the second-round simulations. If λ_1 and λ_2 converge (e.g., within 10% difference), the new queuing delay (t_2) is similar to t_1, and $t_1 + t_{RAS} + t_{RP}$ is the correct memory access latency in turn. Otherwise, we need to keep doing the iteration until two consecutive arrival rates converge. Based on our experience, the arrival rates always converge to a small range after 3 iterations. By doing this we assign a memory access latency value according to the various workloads' memory intensiveness, which improves the accuracy of the results. In Figure 13 we show the relationship between the memory access arrival rate

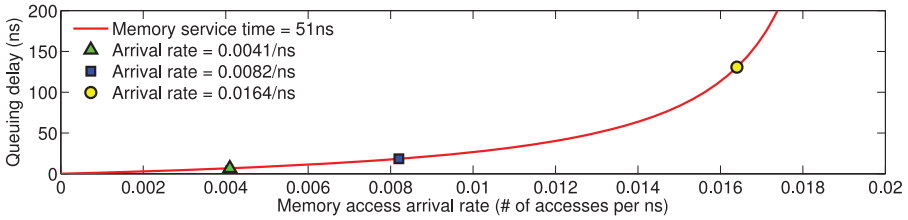


Fig. 13. The relationship between memory access arrival rate and memory controller queuing delay. The data points from left to right represent the memory access arrival rate of one *bzip2*, two instances of *bzip2*, and four instances of *bzip2*, respectively.

Table III. Benchmark Classification according to Memory Intensiveness and Cache Hungriness

	Memory-intensive	Non-memory-intensive
Cache-hungry	bzip2, omnetpp (1–3), soplex (1–6)	omnetpp (4–7), soplex (7)
Non-cache-hungry	bwaves, gcc, gobmk(1), mcf, libquantum, lbm, leslie3d,	astar, calculix, cactusADM, milc, namd, gobmk (2–7), gromacs, h264ref, hmmer

and queuing delay as computed by Eq. (3). Here we take *bzip2* as an example. When running *bzip2* with a 4-way 1MB L2 cache, the memory access rate is 0.0041/ns and the corresponding queuing latency is 6.75ns. If there are two instances of *bzip2* in the system, the memory access rate doubles and the queuing delay becomes 18.3ns. When there are four of them, the queuing delay increases to 130.7ns. Figure 13 shows this relationship. As the memory access rate increases, the queuing delay increases exponentially. When there are multiple memory controllers in the system, the memory accesses get distributed, thus the memory access latency is lower.

7. EXPERIMENTAL RESULTS

7.1. Multiprogram Workload Sets

To test the proposed technique and cache resource pooling policy, we select 19 applications from the SPEC CPU 2006 benchmark suite as listed in Figure 1. According to the applications' memory intensiveness and cache hungriness, we categorize the applications into four classes as shown in Table III. The numbers following the application refer to the corresponding cache configurations. For example, *omnetpp (1–3)* means that, when running with 1 to 3 cache partitions, *omnetpp* is memory intensive. For 4-core 3D systems, we compose 10 multiprogram workload sets with 4 threads each, by combining cache- and noncache-hungry applications as shown in Table IV. We use *nch#* to represent noncache-hungry# workload composition. Similarly, we apply *lch#*, *mch#*, *hch#*, and *ach#* to represent the other workload compositions. Among these workloads, *nch#* contains only noncache-hungry applications, *lch#*, *mch#*, and *hch#* include 1, 2, and 3 cache-hungry applications, respectively, while *ach#* includes only cache-hungry applications. As for the 16-core 3D system, we group four 4-core workload sets for each 16-core workload set based on their cache needs, as shown in Table V. From top to bottom, the number of cache-hungry applications in the workload set increases. When presenting the results, we compare IPC and EDP for each workload set under different 3D systems. Since area is a very important metric for evaluating 3D systems because die costs are proportional to the 4^{th} power of the area [Rabaey et al. 2003], we also use *energy-delay-area product* (EDAP) as a metric to evaluate the cumulative energy and area efficiency [Li et al. 2009] for the 3D systems.

Table IV. 4-Core System Workload Sets

Workload	Benchmarks
non-cache-hungry1	bwaves, gromacs, gobmk, milc
non-cache-hungry2	calculix, leslie3d, milc, namd
low-cache-hungry1	gamess, leslie3d, libquantum, omnetpp
low-cache-hungry2	bwaves, hmmer, namd, bzip2
med-cache-hungry1	astar, bzip2, 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

Table V. 16-Core System Workload Sets

Workload	Single column workload sets			
nch + nch	nch1	nch2	nch1	nch2
nch + lch	nch1	nch2	lch1	lch2
nch + mch	nch1	nch2	mch1	mch2
nch + hch	nch1	nch2	hch1	hch2
nch + ach	nch1	nch2	ach1	ach2
lch + ach	lch1	lch2	ach1	ach2
mch + ach	mch1	mch2	ach1	ach2
hch + ach	hch1	hch2	ach1	ach2
ach + ach	ach1	ach2	ach1	ach2

nch, lch, mch, hch, ach represent non-cache hungry, low-cache hungry, medium-cache hungry, high-cache hungry and all-cache hungry, respectively.

7.2. Performance and Energy Efficiency Evaluation

For both low-power and high-performance 3D systems, we provide three baseline systems where each core has a: (1) static 1MB private L2 cache; (2) static 2MB private L2 cache; and (3) 1MB private cache with *selective cache ways* (SCW) [Albonesi 1999]. For the SCW baseline system, we also use the proposed policy to decide the best cache partitions for each job, but jobs can only require a maximum of 4 cache partitions since SCW does not allow pooling cache partitions from the other layers.

4-core 3D System. Figure 14 shows the IPC, EDP, and EDAP comparison between the 3D-CRP system and the other 3 baselines for the 4-core low-power system. All of the values for each metric are normalized to 2MB baseline. As expected, the 2MB baseline always has the best performance among all systems. Among the 1MB baseline, SCW baseline, and 3D-CRP, SCW's performance is always slightly lower than the 1MB baseline while 3D-CRP outperforms the 1MB baseline as long as there are cache-hungry jobs in the workloads. The performance improvement of 3D-CRP over the 1MB baseline is up to 11.2% among all workloads. The reason is that 3D-CRP always turns off unnecessary cache partitions and pools them to cache-hungry jobs, which boosts the system performance. In Figure 14(b) it is obvious that, for all workloads, 3D-CRP provides lower EDP than 1MB and SCW baselines. On average, 3D-CRP reduces EDP by 18.8% and 8.9% compared to 1MB and SCW baselines, respectively. For SCW, the nonsharing feature limits its improvement in EDP for workload sets with high cache hungriness (e.g., *hch* and *ach* workloads). For the workloads that contain both cache- and noncache-hungry jobs, 3D-CRP improves energy efficiency by up to 38.9% compared to the 2MB baseline. Although the 2MB baseline always provides the best

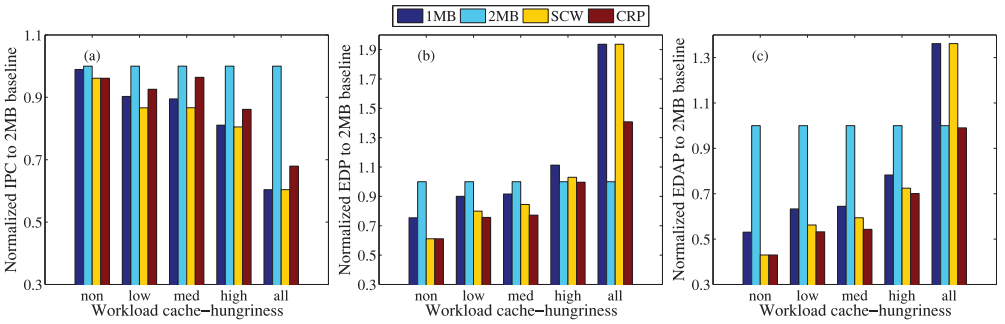


Fig. 14. Normalized IPC, EDP, and EDAP of low-power 3D-CRP system and 3D baseline systems with 1MB static caches, 2MB static caches, and 1MB caches with selective cache way.

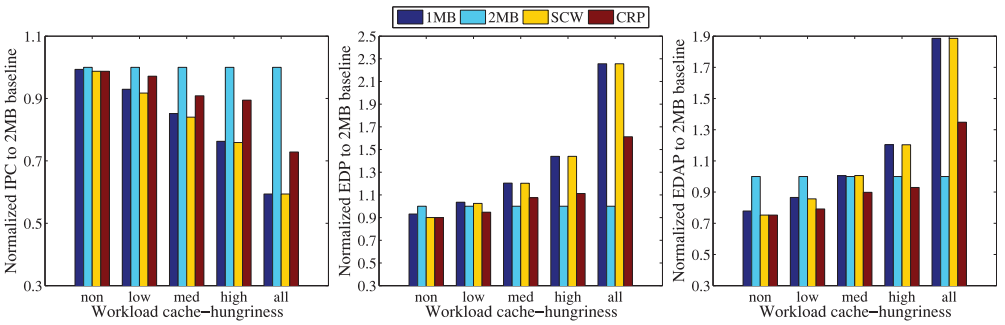


Fig. 15. Normalized IPC, EDP, and EDAP of high-performance 3D-CRP system and 3D baseline systems with 1MB static caches, 2MB static caches, and 1MB caches with selective cache way.

performance, it cannot offer optimal system energy efficiency. For EDAP, as shown in Figure 14(c), 3D-CRP outperforms all baselines for all workloads. The improvements brought by CRP over 1MB and SCW cases are the same as that of EDP because they have the same area, and the average EDAP improvement of CRP over the 2MB baseline is 36.1%.

Moreover, we also evaluate performance and energy efficiency for the high-performance system using the proposed 3D-CRP design and runtime policy, as shown in Figure 15. The same as the low-power 3D system, 3D-CRP achieves lower energy efficiency than 1MB and SCW baselines, and the average EDP reduction is 14.8% and 13.9%, respectively. When compared to the 2MB baseline, the EDP results are different because the high-performance core consumes much more power than the low-power core and 1MB L2 cache. Thus, the percentage of extra power consumption introduced by larger L2 caches is smaller than the percentage of extra performance improvement when it comes to cache-hungry jobs, which can be expressed as

$$\frac{\triangle IPC}{IPC} > \frac{\triangle Power}{Power} \Rightarrow \frac{Power}{IPC^2} < \frac{Power + \triangle Power}{(IPC + \triangle IPC)^2}. \tag{4}$$

Thus it ends up with lower energy efficiency. Nevertheless, for the workloads mixed with cache-hungry and noncache-hungry jobs, 3D-CRP performs similarly to the 2MB baseline on EDP while, from an EDAP perspective, 3D-CRP still improves over the 2MB baseline by up to 24.7% for *nch* and by 12.7% on average for mixed workloads.

We also integrate 3D systems with *microarchitectural resource pooling* (MRP) as proposed in Homayoun et al. [2012]. To evaluate the performance improvement with

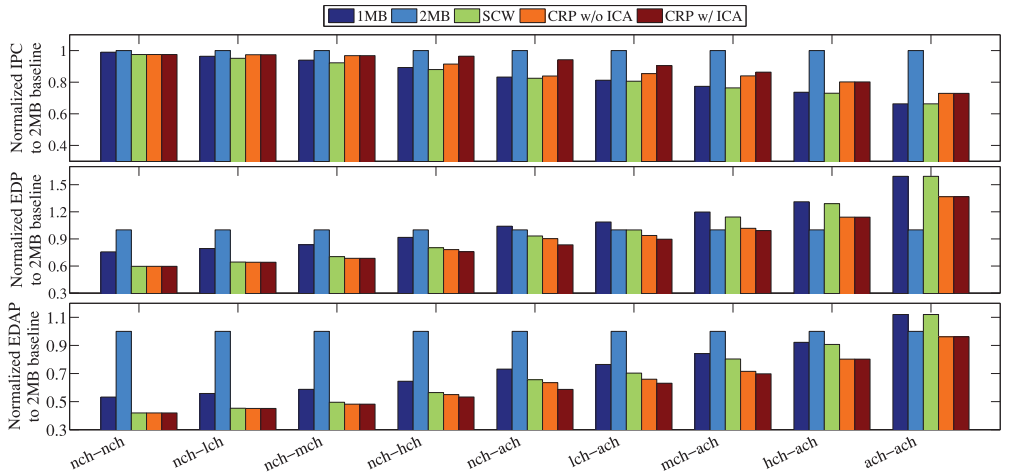


Fig. 16. Normalized IPC, EDP, and EDAP of 16-core low-power 3D-CRP system and 3D baseline systems with 1MB static caches, 2MB static caches, and 1MB caches with selective cache way. ICA refers to *inter-column job allocation*.

MRP, we run applications with 4× of the default sizes of the performance-critical components (reorder buffer, instruction queue, register file, and load/store queue), and compare the IPC results with those of the default settings. For applications running on a single low-power core, our experiments show that MRP improves system performance by 10.4% on average, and combining MRP and CRP provides an extra performance improvement of 8.7% on average in comparison to applying MRP alone.

16-core 3D System. We also evaluate our runtime policy on the 16-core low-power 3D-CRP system with stacked DRAM to investigate the policy's scalability. As introduced in Section 6, the 16-core system has 4 layers with 4 cores on each layer, and each core has a private L2 cache. The DRAM layers are located at the bottom of the chip. The workloads for the 16-core low-power system are listed in Table V. Figure 16 shows the performance and energy efficiency results of 3D systems with different cache architectures. In this figure, from left to right, the cache hungriness of the workloads increases. For all 9 workloads, 3D-CRP outperforms 1MB and SCW baselines in IPC, EDP, and EDAP. 3D-CRP is always better than the 2MB baseline on EDAP. Moreover, with *inter-column job allocation* (ICA), there are further IPC and EDP improvements for 3D-CRP. For example, for the *nch-ach* workload, *3D CRP + ICA* further improves performance by 12.3% and energy efficiency by 7.8% compared to 3D CRP only. When considering memory accesses among the columns and adjusting the workload accordingly, the system performance improvement is around 5% for all workload sets.

7.3. Thermal Evaluation

We conduct steady-state temperature simulations to evaluate the impact of the proposed runtime policy on the on-chip temperature in 3D-CRP systems. For a 4-core low-power 3D system, since the cores have quite low power consumption, the temperature benefits are slight. By contrast, for a 4-core high-performance 3D system, we observe up to a 6.2^oC reduction in peak on-chip temperature compared to the temperature-wise worst possible job allocation for 4-core workloads. Across all workloads there is an average of 3.4^oC reduction in peak on-chip temperature. Such observations prove that our policy can effectively decrease the system peak temperature and prevent cores from exceeding the temperature threshold. In the 16-core 3D system, our results show that,

without temperature-aware job allocation, only *mch-ach* and *ach-ach* operate under the system temperature threshold 85^oC, while applying temperature-aware job allocation keeps all 16-core workloads operating under 85^oC.

8. CONCLUSION

This article has proposed a novel design for 3D cache resource pooling that requires minimal additional circuitry and architectural modification. We have first quantified the impact of cache size and memory access latency on application performance. We have then presented an application-aware job allocation and cache pooling policy to improve the energy efficiency and thermal behavior 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. Moreover, we have designed a memory controller delay model to adjust the memory access latency for different workloads and leveraged this model for all the 3D multicore system evaluations. Experimental results show that, by utilizing cache resource pooling, we are able to improve system EDP and EDAP by 18.8% and 36.1% on average compared to 3D systems with static cache size. The proposed inter-column job allocation manages to additionally improve performance by up to 12.3% and energy efficiency by up to 7.8% for larger 3D systems with on-chip DRAM. On the thermal side, our policy reduces the peak on-chip temperature of high-performance systems by up to 6.2^oC.

REFERENCES

David H. Albonesi. 1999. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the International Symposium on Microarchitecture (MICRO'99)*. 248–259.

Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. 2006. The M5 simulator: Modeling networked systems. *IEEE Micro* 26, 4, 52–60.

Bryan Black, Murali Annavaram, Ned Brekelbaum, John Devale, and Lei Jiang, et al. 2006. Die stacking (3D) microarchitecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO'06)*. 469–479.

Paul Bogdan, Radu Marculescu, Siddharth Jain, and Rafael T. Gavila. 2012. An optimal control approach to power management for multi-voltage and frequency islands multiprocessor platforms under highly variable workloads. In *Proceedings of the IEEE/ACM International Symposium on Networks on Chip (NoCS'12)*. 35–42.

Derek Chiou, Srinivas Devadas, Larry Rudolph, and Boon S. Ang. 2000. Dynamic cache partitioning via columnization. Tech rep., Massachusetts Institute of Technology. http://csg.csail.mit.edu/pubs/memos/Memo-430/memo-430.pdf.

Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. 2005. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News* 33, 4, 80–91.

Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. 2009. Blade computing with the AMD Opteron processor (Magny-Cours). http://www.hotchips.org/wp-content/uploads/hc_archives/hc21/2_mon/HC21.24.100.ServerSystemsI-Epub/HC21.24.110Conway-AMD-Magny-Cours.pdf.

Ayse K. Coskun, Jose L. Ayala, David Atienza, Tajana S. Rosing, and Yusuf Leblebici. 2009a. Dynamic thermal management in 3D multicore architectures. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'09)*. 1410–1415.

Ayse K. Coskun, Richard Strong, Dean M. Tullsen, and Tajana S. Rosing. 2009b. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *Proceedings of the SIGMET-RICS/Performance – Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09)*. 169–180.

Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2012. Application-to-core mapping policies to reduce memory interference in multi-core systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. 455–456.

Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. 2004. Heat-and-run: Leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*. 260–270.

Fazal Hameed, Mohammad A. A. Faruque, and Jorg Henkel. 2011. Dynamic thermal management in 3D multi-core architecture through run-time adaptation. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'11)*. 1–6.

Houman Homayoun, Vasileios Kontorinis, Amirali Shayan, Ta-Wei Lin, and Dean M. Tullsen. 2012. Dynamically heterogeneous cores through 3D resource pooling. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'12)*. 1–12.

John Howard, Saurabh Dighe, Sriram Vangal, G. Ruhl, Shekhar Borkar, et al. 2010. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the International Solid-State Circuits Conference (ISSCC'10)*. 108–109.

Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. 2007. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'07)*. 186–197.

Jongpil Jung, Kyungsu Kang, and Chong-Min Kyung. 2011. Design and management of 3D-stacked NUCA cache for chip multiprocessors. In *Proceedings of the ACM/IEEE Great Lakes Symposium on VLSI (GLSVLSI'11)*. 91–96.

Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2011. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 393–404.

Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. 2005. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA'05)*. 408–419.

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO'09)*. 469–480.

Gabriel H. Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'08)*. 453–464.

Gabriel H. Loh. 2009. Extending the effectiveness of 3D-stacked DRAM caches with an adaptive multi-queue policy. In *Proceedings of the International Symposium on Microarchitecture (MICRO'09)*. 201–212.

Jose F. Martinez and Engin Ipek. 2009. Dynamic multicore resource management: A machine learning approach. *IEEE Micro* 29, 5, 8–17.

Jie Meng, Katsutoshi Kawakami, and Ayse K. Coskun. 2012. Optimizing energy efficiency of 3-D multi-core systems with stacked DRAM under power and thermal constraints. In *Proceedings of the Design Automation Conference (DAC'12)*. 648–655.

Jie Meng, Tiansheng Zhang, and Ayse K. Coskun. 2013. Dynamic cache pooling for improving energy efficiency in 3D stacked multicore processors. In *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC'13)*. 210–215.

Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. 2006. Dynamic resizing of superscalar datapath components for energy efficiency. *IEEE Trans. Comput.* 55, 2, 199–213.

Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO'06)*. 423–432.

Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. 2003. *Digital Integrated Circuits: A Design Perspective*, 2nd ed. Prentice Hall.

Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. 2003. Temperature-aware microarchitecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA'03)*. 2–13.

Allan Snavely and Dean M. Tullsen. 2000. Symbiotic job scheduling for a simultaneous multithreaded processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. 234–244.

Guangyu Sun, Xiangyu Dong, Yuan Xie, Jian Li, and Yiran Chen. 2009. A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'09)*. 239–249.

Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. 2008. CACTI 5.1. http://www.hpl.hp.com/techreports/2008/HPL-2008-20.pdf?jumpid=reg_R1002_USEN.

Keshavan Varadarajan, S. K. Nandy, Vishal Sharda, Amrutur Bharadwaj, Ravi Iyer, Srihari Makineni, and Donald Newell. 2006. Molecular caches: A caching structure for dynamic creation of application specific heterogeneous cache regions. In *Proceedings of the International Symposium on Microarchitecture (MICRO'06)*. 433–442.

Xin Zhao, Jacob Minz, and Sung-Kyu Lim. 2011. Low-power and reliable clock network design for through-silicon via (TSV) based 3D ICs. *IEEE Trans. Components Packag. Manufact. Technol.* 1, 2, 247–259.

Changyun Zhu, Zhenyu Gu, Li Shang, Robert P. Dick, and Russ Joseph. 2008. Three-dimensional chip-multiprocessor run-time thermal management. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 27, 8.

Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10).* 129–142.

Received December 2013; revised June 2014; accepted September 2014