Dynamic Cache Pooling in 3D Multicore Processors

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

Resource pooling, where multiple architectural components are shared among the 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 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 the jobs to cores on the 3D system, while partitioning the cache resources based on the cache hungriness of the jobs. We investigate the impact of the proposed cache resource pooling architecture and management policy in 3D systems with and without on-chip DRAM. We evaluate the performance, energy efficiency, and thermal behavior for a wide range of workloads running on the 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, Policy, Energy Efficiency

Additional Key Words and Phrases: Cache resource pooling, 3D stacking, Runtime policy

1. INTRODUCTION

3D integration technology is a promising design technique for integrating different technologies into a single system, increasing the transistor density, and improving system performance [Black et al. 2006; Loh 2008]. Most of the 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; 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, it does not necessarily provide the desired flexibility to adapt to dynamic resource requirements.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2015 ACM 1550-4832/2015-$15.00$ D01 : http://dx.doi.org/10.1145/0000000.0000000

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 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 the substantial benefits in reducing the total chip area and improving the 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; 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 system with identical layers is 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 bring substantial energy efficiency improvements.

In this paper, 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 the poolable cache resources according to the characteristics of workloads on the system (e.g., instructions retired and cache access behavior), while considering the tradeoffs 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 with and without DRAM stacking. We provide a memory latency computation model that uses 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 paper starts with an overview of the 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 pro-
vides the experimental methodology. Section 7 quantifies the benefits of the proposed architecture and runtime policy, and Section 8 concludes the paper.

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. propose a reconfigurable architecture to combine the resources of simple cores into more powerful processors [Ipek et al. 2007]. Ponomarev et al. introduce a technique to dynamically adjust the sizes of the performance-critical microarchitectural components, such as reorder buffer or instruction queue [Ponomarev et al. 2006]. 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. are the first to explore microarchitectural resource pooling in 3D stacked processors for sharing resources at a fine granularity [Homayoun et al. 2012]. Their work, however, does not investigate the potential of pooling cache resources. Since cache is also a performance-critical component in computer systems, an architecture with cache resource pooling in 3D system 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. propose the molecular caches that creates dynamic heterogeneous cache regions [Varadarajan et al. 2006]. Qureshi et al. introduce a low-overhead runtime mechanism to partition caches between multiple applications based on the cache miss rates [Qureshi and Patt 2006]. Chiou et al. propose a dynamic cache partitioning method via columnization [Chiou et al. 2000]. However, the benefits of cache sharing in 2D systems are highly limited by the on-chip interconnect latency. Kumar et al. demonstrate that sharing the L2 cache among multiple cores is significantly less attractive when the interconnect overheads are taken into account [Kumar et al. 2005].

Cache design and management in 3D stacked systems have been investigated recently. Sun et al. explore the energy efficiency benefits of 3D stacked MRAM L2 caches [Sun et al. 2009]. Prior work on 3D caches and memories either considers integrating heterogeneous SRAM or DRAM layers into 3D architectures (e.g., [Meng et al. 2012; 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. Run-time Management of Multicore Systems

To manage the 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 et al. present a mechanism that allows the scheduler to exploit the workload characteristics for improving processor performance [Snavely and Tullsen 2000]. Das et al. propose an application-to-core mapping algorithm to maximize system performance [Das et al. 2012]. Bogdan et al. propose a...
novel dynamic power management approach based on the dynamics of queue utilizations and fractional differential equations to avoid inefficient communication and high power density in Network-on-Chips [Bogdan et al. 2012].

Dynamic job allocation on 3D systems mostly addresses 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 the temperature and reduce hot spots [Coskun et al. 2009a; Zhu et al. 2008]. Hameed et al. propose a technique to dynamically adapt core resources based on application needs and thermal behavior to boost performance while maintaining thermal safety [Hameed et al. 2011]. 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 the 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 with and without DRAM stacking. We also compare our work with the most related previous work: selective way cache architecture [Albonesi 1999]. In addition, this paper 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 sizes on performance and energy efficiency of various applications. Although we focus on L2 cache in this paper, our work can also be applied to the other levels of caches in computer systems.

To quantify the impact of L2 cache, we simulate a single core with varying L2 cache sizes (from 0 KB to 2048 KB with a step of 256KB) and compare the performance
and power results for the applications in SPEC CPU 2006 benchmark suite. We use
Gem5 [Binkert et al. 2006] for performance simulation, 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
normalized IPC of all applications under different L2 cache configurations. As shown
in Fig. 1, among all applications, soplex, omnetpp, and bzip2 have significant perfor-
mance improvement at large L2 cache sizes of up to 1.8x. We call such applications
cache-hungry applications. On the other hand, applications such as bwaves barely ben-
efit from an L2 cache larger than 256 KB. Figure 2 shows the total power consumption
for all the applications under the same cache configurations. For all applications, the
power consumption increases as the L2 cache size goes up. Figure 1 and Fig. 2 indi-
cate 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, 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 the system performance is significantly improved. Figure 3 shows the performance improvement for the applications when changing the memory access latency from off-chip memory to on-chip stacked memory and Fig. 4 shows the L2 miss per kilo-instruction (MPKI) of the 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. The memory access rate directly influences the queuing delay in the memory controllers. Hence, if all 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. The vertically adjacent caches are connected via TSVs for cache pooling, as shown in Fig. 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 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 non-cache-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 caches.

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 the 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 Fig. 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 the 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 local cache partitions. The status of the local cache partitions is used for deciding the output location of the output data and hit signals. Secondly, we introduce Remote Cache Status Registers (RCSR) 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 RCSR 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 denotes the lower layer. 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 Fig. 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 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 Fig. 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 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 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 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.

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 the cross-section of the 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 from 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 the performance. Therefore a policy for monitoring and adjusting the 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 2to1 multiplexer has 12 transistors. We need 10 1-bit transistors for LCSRs and RCSRs, one 64-bit 1to4 demux and one 64-bit 4-to-1 mux for data transfers, one 30-bit 1to4 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 1to2 demux for sending back hit signal to remote layers and 2 AND gates for generate L2 cache requests. Thus, the total number of transistors needed by the extra registers and logic in our design is limited to 5460 (10×1-bit register + 2×64-bit 1to4 demux + 2×30-bit 4-to-1 mux + 1×2-bit 4-to-1 mux + 1×1-bit 1to2 demux + 2×AND gate). We assume there are 128 TSVs for two-way data transfer between caches, 60 TSVs for the
Dynamic Cache Pooling in 3D Multicore Processors with DRAM Stacking

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 to be 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 $0.1\text{mm}^2$, which is negligible compared to the total chip area of $10.9\text{mm}^2$. Prior work [Homayoun et al. 2012] shows that the layer to layer delay caused by TSVs is $1.26\text{ps}$, which has no impact on the system performance as it is much smaller than the CPU clock period at 1GHz. If there is on-chip DRAM, the memory controller is also connected to DRAM through TSVs, which brings extra 512 TSVs for data transmission and 32 TSVs for sending commands to the memory module.

5. RUN-TIME 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 Fig. 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 Fig. 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 off-line linear regression model that takes runtime performance counter data as inputs. 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., $10\text{ms}$) using the default reserved cache partition (each core has a reserved L2 cache partition of $256\text{KB}$ that cannot be pooled). The performance counters that

![Diagram](image-url)

**Fig. 7.** (a) The cross-section view of large 3D-CRP system; (b) An example showing cache resource pooling within a column.
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 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 the job pair with higher average IPC to the available cores closest to heat sink as shown in Fig. 9 (a). The reason is that the 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 the high performance improvement. On the contrary, for non-cache-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:

\[
\frac{\text{Power}}{\text{IPC}^2} > \frac{\text{Power} + \Delta \text{Power}}{(\text{IPC} + \Delta \text{IPC})^2}
\]

Stage 1: Job Allocation

- assign a single cache partition to each job \(J_i\)
- predict perf. improvement \((p_i)\)
- sort all jobs based on \(p_i\)
- e.g., \(p_1 > p_3 > p_4\)
- allocate \(J_1\) & \(J_4\), \(J_2\) & \(J_3\) on adjacent layers (see Figure. 9)

Stage 2: Pair-wise Cache Pooling

- assign 1 or 4 partitions to each job \(J_i\) based on \(p_i\)
- increase the # of partitions for each job \(J_i\)
- \((**)\) \(p_i > p_j\) ? No
- has job \(J_i\) reached max # of partitions? Yes
- revert to previous partitions
- keep the current partitions

\(\text{Stage 1: Job Allocation}\)

\(\text{Stage 2: Pair-wise Cache Pooling}\)

Fig. 8. A flow chart 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.
IPC and Power refer to performance and power values before we increase the number of cache partitions, while $\Delta$IPC and $\Delta$Power are the variations in IPC and power when the job uses an additional partition. From this inequality, we obtain:

$$\frac{\Delta \text{IPC}}{\text{IPC}} > \sqrt{1 + \frac{\Delta \text{Power}}{\text{Power}}} - 1 = t$$

(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 Fig. 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 the cache-hungrienss among the columns, we perform inter-column job allocation after sorting the jobs as a load balancing policy in such systems.

We first assign the 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-hungrienss. 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 Fig. 10. After the inter-column job allocation, the system pairs the jobs and decides the cache resource allocation inside each column as stated in the previous subsections. Furthermore, from the memory access perspective, since the jobs within one column could have higher average memory access than the jobs in the other columns, the memory access latency of this particular column may be potentially 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 hungri
ness 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 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, $W_{AVG_1} > W_{AVG_4} > W_{AVG_3} > W_{AVG_2}$. The job with the highest weight in $C_1$ is swapped with the job with the lowest weight in $C_2$ and we get $W_{AVG_1} > W_{AVG_4} > W_{AVG_2} > W_{AVG_3}$. 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 $W_{AVG_i}$ and $W_{AVG_T}$ under the threshold. After the inter-column job reallocation, the cache-hungri
ness 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\text{ms} [Coskun et al. 2009b]. We also evaluate the cold start effect overhead by comparing the performance of the benchmark suite with and without cache warmup, as shown in Fig. 12. We can see from this figure that almost all applications suffer from 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\text{\mu s} from mcf. For most of the applications, the overhead is lower than 150\text{\mu s}. For multicore systems, the memory access rate is higher than 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 1\text{ms}. Other than cache cold start effect, when job migration happens, context switch also introduces performance degradation. However, the context switch overhead is no more than 10\text{\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 of 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 cores 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 could 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 Fig. 7. The stacked DRAM layers are placed at the bottom of the 3D system, as described in Section 4. Due to the area
Table I. Core Architecture Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>High-Perf</th>
<th>Low-Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Clock</td>
<td>2.1GHz</td>
<td>1.0 GHz</td>
</tr>
<tr>
<td>Issue Width</td>
<td>out-of-order 3-way</td>
<td>out-of-order 2-way</td>
</tr>
<tr>
<td>Reorder Buffer</td>
<td>84 entries</td>
<td>40 entries</td>
</tr>
<tr>
<td>BTB/RAS size</td>
<td>2048/24 entries</td>
<td>512/16 entries</td>
</tr>
<tr>
<td>Integer/FP ALU</td>
<td>3/3</td>
<td>2/1</td>
</tr>
<tr>
<td>Integer/FP MultDiv</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>Load/Store Queue</td>
<td>32/32 entries</td>
<td>16/12 entries</td>
</tr>
<tr>
<td>L1 I/DCache</td>
<td>64KB, 2-way, 2ns</td>
<td>16KB, 2-way, 2ns</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1MB, 4-way, 5ns</td>
<td>1MB, 4-way, 5ns</td>
</tr>
<tr>
<td>Core Area</td>
<td>15.75 mm²</td>
<td>3.88 mm²</td>
</tr>
</tbody>
</table>

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 single-core simulations shown in Section 3, we fast-forward 2 billion instructions and then execute 100 million instructions in detailed mode for all applications under L2 cache sizes from 0 to 2MB. For 4-core 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 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 simulations.

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 multi-program workload sets. In the M/D/1 model, arrival rate (λ) and service rate (μ) are required to compute the queuing delay, \( t_{\text{queuing}} \), as shown below:

\[
 t_{\text{queuing}} = \frac{\lambda}{2\mu(\mu - \lambda)} 
\]

In the proposed 3D-CRP system, there is one memory controller in each column, thus for multi-program 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 \( t_{\text{RAS}}+t_{\text{RP}} \) as the memory system service rate. For each multi-program workload, we first assign a sufficiently large value as the memory access latency to ensure that the arrival rate would not exceed the memory system service rate. Then we run the performance simulations and collect the memory access rate of the workload. Based on this arrival rate (λ₁) we compute the queuing delay (\( t_{\text{1}} \)) of the memory con-

![Fig. 13](image-url) 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 II. Main Memory Access Latency for the 3D CRP System

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC-to-MC</td>
<td>Latency due to short latency provided by TSVs</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>Queuing delay, computed by M/D/1 queuing model</td>
</tr>
<tr>
<td>Main Memory</td>
<td>On-chip 1 GB DRAM: $t_{\text{RAS}} = 36\text{ns}, t_{\text{RP}} = 15\text{ns}$</td>
</tr>
<tr>
<td>Total Delay</td>
<td>Queuing delay + $t_{\text{RAS}} + t_{\text{RP}}$</td>
</tr>
<tr>
<td>Memory Bus</td>
<td>On-chip memory bus, 2GHz, 64-byte bus width</td>
</tr>
</tbody>
</table>

troller. The new memory access latency is the sum of LLC-to-MC delay, DRAM module access time and the queuing delay ($t_1 + t_{\text{RAS}} + t_{\text{RP}}$), as shown in Table II. Then we feedback this new latency to Gem5 and collect the arrival rate ($\lambda_2$) from the second round simulations. If $\lambda_1$ and $\lambda_2$ converges (e.g., within 10% difference), the new queuing delay ($t_2$) is similar to $t_1$ and $t_1 + t_{\text{RAS}} + t_{\text{RP}}$ 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 Fig. 13 we show the relationship between the memory access arrival rate and queuing delay as computed by Equation (3). Here we take bzip2 as an example. When running bzip2 with 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. Multi-program 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 Fig. 1. According to the applications’ memory-intensiveness and cache-hungrierness, 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 multi-program workload sets with 4 threads each, by combining cache-hungry and non-cache-hungry applications as shown in Table IV. We use nch# to represent non-cache-hungry# workload composition. Similarly, we apply lch#, mch#, hch#, and ach# to represent the other workload compositions. Among these workloads, nch# contains only non-cache-hungry applications, lch#, mch# and hch# includes 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 the 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.

7.2. Performance & 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
Table III. Benchmark classification according to memory-intensiveness and cache-hungriness

<table>
<thead>
<tr>
<th>Cache-hungry</th>
<th>Memory-intensive</th>
<th>Non-memory-intensive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bzip2, omnetpp (1-3), soplex (1-6)</td>
<td>omnetpp (4-7), soplex (7)</td>
</tr>
<tr>
<td>Non-cache-hungry</td>
<td>bwaves, gcc, gobmk(1), mcf, libquantum, lhm, leslie3d, astart, calculix, cactusADM, milc, namd, gobmk (2-7), gromacs, h264ref, hmmer</td>
<td></td>
</tr>
</tbody>
</table>

L2 cache; (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, EDAP comparison between 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 1MB baseline, SCW baseline and 3D-CRP, SCW’s performance is always slightly lower than 1MB baseline while 3D-CRP outperforms 1MB baseline as long as there are cache-hungry jobs in the workloads. The performance improvement of 3D-CRP over 1MB baseline is up to 11.2% among all workloads. The reason is that 3D-CRP always turns off the unnecessary cache partitions and pools them to cache-hungry jobs, which boosts the system performance. In Fig. 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 non-sharing 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-hungry jobs and non-cache hungry jobs, 3D-CRP improves the energy efficiency by up to 38.9% compared to 2MB baseline. Although the 2MB baseline always provides the best performance, it can not offer the optimal system energy efficiency. For EDAP, as shown in Fig. 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 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 Fig. 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 comparing to 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.

Table IV. 4-core system workload sets

<table>
<thead>
<tr>
<th>Workload</th>
<th>Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-cache-hungry1</td>
<td>bwaves, gromacs, gobmk, milc</td>
</tr>
<tr>
<td>non-cache-hungry2</td>
<td>calculix, leslie3d, mife, namd</td>
</tr>
<tr>
<td>low-cache-hungry1</td>
<td>gomess, leslie3d, libquantum, omnetpp</td>
</tr>
<tr>
<td>low-cache-hungry2</td>
<td>bwaves, hmmer, namd, bzip2</td>
</tr>
<tr>
<td>med-cache-hungry1</td>
<td>astart, bzip2, soplex, mcf</td>
</tr>
<tr>
<td>med-cache-hungry2</td>
<td>bzip2, cactusADM, hmmer, omnetpp</td>
</tr>
<tr>
<td>high-cache-hungry1</td>
<td>gromacs, bzip2, omnetpp, soplex</td>
</tr>
<tr>
<td>high-cache-hungry2</td>
<td>h264ref, bzip2, omnetpp, soplex</td>
</tr>
<tr>
<td>all-cache-hungry1</td>
<td>soplex, soplex, omnetpp, bzip2</td>
</tr>
<tr>
<td>all-cache-hungry2</td>
<td>soplex, bzip2, soplex, bzip2</td>
</tr>
</tbody>
</table>
Table V. 16-core system workload sets. nch, lch, mch, hch, ach represent non-cache-hungry, low-cache-hungry, med-cache-hungry, high-cache-hungry and all-cache-hungry, respectively.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Single column workload sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>nch + nch</td>
<td>nch1 nch2 nch1 nch2</td>
</tr>
<tr>
<td>nch + lch</td>
<td>nch1 nch2 lch1 lch2</td>
</tr>
<tr>
<td>nch + mch</td>
<td>nch1 nch2 mch1 mch2</td>
</tr>
<tr>
<td>nch + hch</td>
<td>nch1 nch2 hch1 hch2</td>
</tr>
<tr>
<td>nch + ach</td>
<td>nch1 nch2 ach1 ach2</td>
</tr>
<tr>
<td>lch + ach</td>
<td>lch1 lch2 ach1 ach2</td>
</tr>
<tr>
<td>mch + ach</td>
<td>mch1 mch2 ach1 ach2</td>
</tr>
<tr>
<td>hch + ach</td>
<td>hch1 hch2 ach1 ach2</td>
</tr>
<tr>
<td>ach + ach</td>
<td>ach1 ach2 ach1 ach2</td>
</tr>
</tbody>
</table>

when it comes to cache-hungry jobs, which can be expressed as:

\[
\frac{\Delta IPC}{IPC} > \frac{\Delta Power}{Power} \Rightarrow \frac{Power}{IPC^2} < \frac{Power + \Delta Power}{(IPC + \Delta IPC)^2}
\]  

Thus it ends up with lower energy efficiency. Nevertheless, for the workloads mixed with cache-hungry and non-cache-hungry jobs, 3D-CRP performs similarly with 2MB baseline on EDP, while from 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 MRP, we run applications with four times 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 the results with 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 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 improvement.
for 3D-CRP. For example, for 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 the 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. While for a 4-core high-performance 3D system, we observe up to a 6.2°C 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°C 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 the temperature-aware job allocation only mch-ach and ach-ach operate under the system temperature threshold 85°C, while applying temperature-aware job allocation keeps all 16-core workloads operating under 85°C.

8. CONCLUSION

This paper 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 sizes and memory access latency on the application performance. We have then presented an application-aware job allocation and cache pooling policy to improve the energy efficiency and the 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 sizes. 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°C.

REFERENCES


Received December 2013; revised June 2014; accepted September 2014