DR-SNUCA: An Energy-Scalable Dynamically Partitioned Cache

Abstract—Multicore processors continue to power the datacenters, which provide cheap, volatile, and limitless computing over the network. At the same time, these processors are becoming increasingly common in embedded systems, such as the smartphones. As the number of processing elements increases within these processors, so does the pressure to share the critical on-chip cache resources, but this must be done energy-efficiently and without sacrificing resource guarantees. We propose a scalable dynamic cache-partitioning scheme, DR-SNUCA, which provides an energy-efficient way to reduce resource interference over caches shared among many processing elements. Our results show that DR-SNUCA reduces overall system power consumption by 16.27% compared to associatively partitioned caches, such as DNUCA. We also describe a progress-based cache allocation algorithm that increases the total throughput for a 32-thread system by 36.52% over existing miss-based allocation policies.

I. INTRODUCTION

Multicore processors are ubiquitous today, with most commercial processors consisting of multiple processing elements. These processors have helped increase the computational density and energy-efficiency of not just the personal computers, but also datacenters powering the cloud computing ecosystem, which is increasingly gaining traction through both public [3] as well as private offerings. Multicore processors have also started encroaching embedded systems, and are in fact crowding some domains, such as the smart phones. In these, and other, domains utilizing multicore processors there is a trend toward greater concurrency [5] that will soon move us from an era of multicore designs into an era of scalable manycore designs, because they optimize energy per operation for high compute workloads [4].

In manycore processors, there are pressures to share cache resources [1]. Independently providing each processing element sufficient cache to meet its peak demand [18] would be prohibitively expensive; conversely, dividing an affordable amount of cache area into fixed cache partitions may lead to local resource shortages despite sufficient aggregate resources [29]. Sharing the last level of cache allows higher cache utilization than statically partitioned caches. However, sharing also leads to resource contention, or interference, which causes difficulties in maintaining performance guarantees, and in monitoring and managing resource usage. Furthermore, decisions made in the presence of interference can lead to substantial inaccuracies, as shown by Govindan et al. [13].

Finding the right balance in managing cache resources for manycore processors is challenging. The datacenter and embedded system domains pose multiple design requirements for the manycore processors, as shown in Table I. Power is now a first-order concern in both domains, so they both require high energy-efficiency. At the same time, they require high resource utilization to improve throughput and area-efficiency respectively. Finally, due to QoS requirements, resource guarantees are highly valuable when designing multicore processors for these domains.

Non-Uniform Cache Access (NUCA) architectures [20] come very close to embodying these three goals. These architectures spatially divide the cache into multiple cache-arrays, accessed through a network-on-chip, or NoC, to provide energy-efficiency compared to monolithic cache designs. In a NUCA cache, access energy depends heavily on the number of cache-arrays accessed for each request. Static Non-Uniform Cache Access (SNUCA) architectures use a fixed indexing function and accesses only one cache-array for every cache request, which keeps the energy consumption low, as shown in Figure 1. By configuring these index functions for each core, these SNUCA architectures can divide the cache into independent partitions specific to a given workload, thus providing resource guarantees; however, due to fixed hashing, the number of SNUCA cache sets cannot be dynamically changed. As a result, if we partition a shared cache into SNUCA components to keep the associativity fixed, cache allocations cannot be dynamically changed, which leads to reduced cache utilization due to changing application working sets; therefore, SNUCA fails to deliver on one of the three design goals.

The cache designs proposed by Nesbit et al. [27], Hsu et al. [16], and Guo et al. [14], use associative partitioning to create a dynamically partitioned (DNUCA) cache, which can provide better cache utilization than SNUCA. However, in DNUCA a cache line can be placed on any of the allocated

<table>
<thead>
<tr>
<th>Design Demands</th>
<th>Datacenters</th>
<th>Embedded Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy-efficiency</td>
<td>Limit operating costs</td>
<td>Limit power budgets</td>
</tr>
<tr>
<td>High utilization</td>
<td>Limit cost of ownership</td>
<td>Limit area budgets</td>
</tr>
<tr>
<td>Resource guarantees</td>
<td>Meet QoS agreements</td>
<td>Real-time tasks</td>
</tr>
</tbody>
</table>

TABLE I. SIMILAR DESIGN REQUIREMENTS FOR MANYCORE PROCESSORS IN DATACENTERS AND EMBEDDED SYSTEMS

Fig. 1. Associatively partitioned DNUCA caches are not energy scalable. The portion of energy spent in L2 caches increases with cache size when using DNUCA, even with XOR-based way-prediction [28], partial-tag match [19] and cache migration [17]. For SNUCA, the portion of energy spent in caches remains low, but SNUCA is not dynamically repartitionable.

The cache designs proposed by Nesbit et al. [27], Hsu et al. [16], and Guo et al. [14], use associative partitioning to create a dynamically partitioned (DNUCA) cache, which can provide better cache utilization than SNUCA. However, in DNUCA a cache line can be placed on any of the allocated...
cache-arrays, so it may need to check multiple cache-arrays for every cache access. Mechanisms such as XOR-based way-prediction \[28\], partial-tag match \[19\], and cache-block migration \[17\] reduce the number of cache-arrays checked per cache access. However, Figure 1 shows that, even with these mechanisms\(^1\), cache accesses in an associatively partitioned DNUCA consume a progressively larger portion of processor energy as cache size increases. This makes DNUCA based last-level caches energy-inefficient as aggregate cache size grows.

**Insight:** While working sets change, necessitating dynamism in partitioning, they do not change rapidly. Thus, the frequency with which we can repartition does not need to be high and we should optimize performance and energy for the time between allocations.

We introduce **Dynamically Repartitionable Static NUCA**, or DR-SNUCA, a dynamically-repartitionable shared cache with static-mapping during steady-state. DR-SNUCA provides energy efficiency and high cache utilization as well as fixed resource guarantees, and it does not interrupt execution during reconfiguration. This paper shows a complete set of results for DR-SNUCA in a manycore processor, including architectural mechanisms that make the decisions as to how to reconfigure the caches at runtime without interrupting execution.

DR-SNUCA uses set partitioning i.e. growing or shrinking cache allocations by changing the number of sets allocated to an application while keeping associativity constant. We use indirect cache addressing to reduce reconfiguration overheads introduced during changes to cache allocations, to enable online reconfiguration. We also introduce **Tag-Duplication** to avoid execution stalls during the cache reconfiguration and keep DR-SNUCA’s performance comparable to DNUCA. Finally, we propose distributing cache partitions between applications based on the progress made by each application so far, and show that this provides higher throughput than algorithms that minimize total cache misses.

We evaluate DR-SNUCA against a DNUCA cache on a 32-core manycore processor where the cores and on-chip network, or OCN, are modeled along the lines of the cores and OCNs in commercial manycore processors (e.g. Tilera). Our evaluation shows that DR-SNUCA reduces overall system energy by 16.27% on average, while performing within 0.5% of DNUCA. Furthermore, we show that the area and energy overheads of reconfiguration in DR-SNUCA are small. Finally, we show that a progress-based allocation algorithm can improve overall processor throughput compared to existing miss-based algorithms. Overall, the results argue for using DR-SNUCA in future manycore processors.

**II. DYNAMICALLY REPARTITIONABLE STATIC NUCA (DR-SNUCA)**

DR-SNUCA provides a last-level shared cache for embedded manycores that can be dynamically partitioned and is also energy-scalable. DR-SNUCA dynamically partitions the shared cache between applications and each application’s

---

\(^1\)For every cache size, we chose the partial tag size that minimized the overall energy consumption.
in each tag using dataLoc bits. All these arrays are connected through a dynamic OCN. With this indirect addressing, cache reconfiguration moves only the tag and not the data, which significantly reduces the reconfiguration costs, since cache tags are significantly smaller than lines.

For accessing a cache line under indirect cache addressing, the location of the tag-array holding the cache line tag is determined based on the tagLoc bits in the physical address, but the data can be placed on any of the data-arrays, as shown in Figure 3. An application’s tagLoc width equals \( \log_2 \) of the number of cache-arrays allocated. The cache set for an address in both tag-arrays and data-arrays is determined based on the index bits. The associative way for both tag and data-arrays is also the same for every cache line present in the cache, as shown in Figure 3. Thus, we only need to store the location of the data-array in dataLoc bits. To find an address location, we use tagLoc bits from the address, located just above the index bits, to determine the location of the cache-array that can hold the data for the application. We then use the index bits to determine the cache set within the cache-array, as shown in Figure 3. DR-SNUCA maintains a one-to-one correspondence between all tag and data locations in the cache at all times. This correspondence is useful in avoiding dead dataLoc references, or unreachable data locations, as well as improving the indirect cache access and reconfiguration performance, and can change only during cache reconfiguration.

**Reconfiguration.** Application cache allocations can dynamically change during application execution on interval boundaries. During this online reconfiguration we may have to shift some tags and evict some data while maintaining the one-to-one correspondence between all tag and data locations. During reallocation, we find the new tag locations for cache lines based on their tagLoc bits, and if it is not the same as their current locations, due to a possible change in the hashing scheme, we move the tags to their new blocks at the same index and way, as shown in Figure 4. The dataLoc bits in the existing tags at these new locations are also copied back to the old locations to maintain the tag and data location correspondence.

If the cache allocation increases, no change is required in the data blocks. However, if the cache allocation shrinks, we must select which cache lines to preserve and which to evict. To support this selection, in addition to the associative LRU within a set, we also maintain an LRU vector for every equivalent location (same index, way) across all cache-arrays currently owned by an application. We call this the Application-LRU, and it is stored on the application’s core. On allocation reductions, we evict the application-LRU entries for each equivalent location, as shown in Figure 4. We use the application-LRU to find the lines to evict, as this provides better cache locality, and hence performance, than a naive selection, such as preserving all cache lines from a single cache-array. We have to pro-actively evict these lines because if we fail to writeback all the dirty lines, the cache will become incoherent. We writeback only the dirty lines to save bandwidth. For the cache lines that are to be evicted, we still maintain the dataLoc bits in them in order to preserve the correspondence between tag and data locations.

During cache reconfiguration, it is difficult to handle memory requests for a cache line whose tag is in transit. There are three basic approaches to handling this scenario. First, we could have a protocol to track the tag during its transit and allow intermediate structures to respond, but this is complicated and can cost additional time and energy. Second, we could stall the application execution until the reconfiguration finishes, but this will reduce application performance. Third, Tag-Duplication, which maintains two tag block arrays and copies the tags from the arrays allocated for the current interval into the arrays allocated for the next interval, as shown in Figure 4. While the reconfiguration is going on, all tag lookups for an application are sent to the tag blocks allocated to the application for both the current and previous intervals, which guarantees that the tag will be matched if present in the cache. DR-SNUCA uses tag-duplication to prevent application stalling and handle memory requests during cache reconfiguration. Our experiments show that the reconfiguration period is relatively small compared to the interval length, which keeps the costs of dual-lookup during reconfiguration low.

There are many potential ways to allocate the shared resources of a manycore processor among its concurrent applications in order to facilitate both performance fairness and aggregate throughput. In this paper we use the scheme to minimize the cache miss rate, proposed by Qureshi et al. [29], as our baseline for comparing DNUCA and DR-SNUCA. This policy partitions the associative cache ways to minimize the overall cache misses and is an improvement over existing commercial manycore system resource allocation schemes, which do not use cache partitioning to minimize cache misses. Later, in Section IV we show how using application progress, rather than cache misses, to guide resource allocation can even further improve throughput.
III. DR-SNUCA EVALUATION

We model the structure of our evaluation prototype along the lines of commercial tiled manycores (e.g. Tile64 [4]) and embedded processors [1], and use a fixed reconfiguration interval of 25 million cycles \(^2\). We use PTLsim [36] and a memory-system emulator to simulate execution of multiple applications on a single many-core chip while sharing last level cache and off-chip memory accessed through memory controllers, as shown in Figure 2. The emulator internally uses DRAMSim2 [35] for modeling details of the DRAM memory system. Detailed specifications of our evaluation model are presented in Table II. We analytically model the area and power consumption using area and energy numbers, static as well as dynamic, obtained from RAW [24] and McPAT [22] scaled to 45nm, as specified in Table III. In order to reduce simulation run times, we extract application representative phases using SimPoint [32] and then concurrently run SimPoint combinations.

In our experiments, we partition the DRAM bandwidth to provide equal slowdown between applications using a fair queuing arbiter [26] at the memory controllers, which does fair scheduling across applications while staying within their equal allocated bandwidth quotas. We split the memory banks statically among the applications along with the corresponding row buffers to provide resource guarantees. This static partitioning is maintained at the memory controllers, and the memory page allocator (OS) allocates memory pages to applications only on the memory banks assigned to them. DRAM bank partitioning should not significantly impact performance on manycore systems, as we experimentally observed that applications were bottlenecked on memory bandwidth and not DRAM row buffers.

**Benchmarks and their Classification** We run combinations drawn from 26 benchmarks that span SPEC2K, SPEC2K6, and an internally-developed I/O intensive benchmark suite, as shown in Table IV. This selection provides a rich spectrum of cache and memory characteristics.

The space of all possible benchmark combinations is very large. Moreover, it provides no intuition about the benchmarks that we have not included in our evaluation. In order to limit the evaluation space as well as incorporate a structure into our evaluation, we classify our benchmarks by memory characteristics \(^3\) into a three-type taxonomy, and then examine runs that include different ratios of the three types. The taxonomy is as follows: An application which sees no drop in miss rate with increasing cache size is a stream application, an application which sees a sudden drop in miss rate with cache size is a cliff application, and an application whose miss rate drops gradually with increasing cache size is a slope application, as described in Figure 5. We can then run representatives of these classes to refine our manycore evaluation space and we can estimate behavior of similar applications.

\(^2\)25 million cycles was found to be the interval duration with the highest performance in our experiments.

\(^3\)In the applications examined, cache sensitivity was a strong classifier that predicted other characteristics, such as stream applications having good prefetching behavior and high bandwidth requirements. For a workload with high variance within cache sensitivity categories, additional classification axes would be beneficial.
Benchmarks can be classified based on the sensitivity of their miss rate to L2 cache sizes. For some applications like bzip2 (a) cache size has a steady impact on miss rate, while for others like apsi (b) it has no effect, and some applications like mgrid (c) have a cliff-like profile.

Without tag-duplication (TD), DR-SNUCA performs 10.3% worse in comparison to the baseline due to reconfiguration stalls. However, after adding tag-duplication, DR-SNUCA is able to perform within 0.5% of the baseline.

Energy consumed by cache reconfiguration (0.06%), including tag migration and data eviction, is very small.

Area consumed for reconfiguration mechanisms in DR-SNUCA, such as 6.42% for tag-duplication, is low.

reduce the energy consumption of L2 accesses; as a result, the portion of total energy consumed in L2 access when using DR-SNUCA is low (0.50%), as shown in Figure 8. Most of the energy is consumed in core execution (47.45% including L1 access) and main memory operations (45.36% for access and writeback). Energy consumed for reconfiguration in DR-SNUCA is low: 0.06% of the total energy. The tag-duplication required to support continuous execution during reconfiguration in DR-SNUCA consume only 6.42% of the area, as shown in Figure 9. Overall, the mechanisms used to support reconfiguration in DR-SNUCA are energy and area efficient, making it as efficient as SNUCA while also providing the ability to dynamically repartition.

IV. EFFECT OF CACHE ALLOCATION POLICIES

We examine the potential for further performance improvements to DR-SNUCA at the allocation policy level by exploring the use of progress-based, rather than miss-based allocation policies. Our approach relies on per-interval estimation of alone execution time, a concept introduced by Eyerman et al. [11] which describes the time an application would have taken if it had been running in isolation. We implement a scheme similar to the one proposed by Eyerman et al. to provide alone execution time estimates, and we compare our allocation approach that uses these estimations against our baseline, in which we have partitioned the caches to minimize the cache misses [29] (miss-based cache allocation). Every interval, we estimate the alone execution time for all possible cache allocations between applications and choose the allocation that will maximize the geometric mean of the application performances till that point, which can be approximated to the term shown in Equation 1. We call this progress-based cache allocation. This maximizes the forward progress of every application, while reducing unfair slowdowns for applications with lower performance.

\[
\text{Mean Progress}_{j,t} = \frac{\sum \text{Interval Alone-Execution-Time}_{j,i}}{\text{Total Alone-Execution-Time}_{i}} (1)
\]

During context switches, the incoming application is provided a cache allocation based on its existing progress. If it is a new application, it is started off with a default cache allocation. In our experiments, we divide the cache equally between all applications in the beginning.

We compare these two allocation policies against the aggregate throughput of applications when ran alone on the chip. There is a slowdown when applications are sharing the resources (0.58 and 0.43 for progress-based and miss-based allocation schemes), as opposed to running alone. However, the implementation details of online estimation of application progress is beyond the scope of this paper.
we observed that throughput improves by 36.52% on average for our progress-based allocation, compared to the miss-based allocation, as shown in Figure 10(a). These gains are made possible due to allocating resources based on first-order performance metrics, i.e. application, rather than indirect metrics, such as cache misses. Moreover, this also allows us to simultaneously allocate differently sized resources (cache and bandwidth) with a shared objective i.e. increasing throughput, leading to increased overall resource utilization, as opposed to the miss-based allocation in which different resources are allocated with conflicting objectives (throughput and fairness for bandwidth and cache allocation respectively). This higher resource allocation also leads to an improvement in the performance of each application, by 19.21% on average due to better resource utilization.

![Progress-based cache allocation](image)

**Fig. 10.** Progress-based cache allocation. Our progress-based cache allocation scheme leads to higher overall throughput (a) compared to miss-based allocation (36.52% on average), and higher performance (b) for each application (19.21% on average) due to better resource utilization.

V. RELATED WORK

There exists a rich literature of techniques for improving the cache performance as well as energy-efficiency. These techniques have proposed improving cache utilization, such as V-way [30] and Hy-way [10] caches; reducing cache pollution, such as EAF [31]; and caching specialized regions, such as stacks and heaps [6]. While some of these techniques discussed the harmful effects of cache interference and proposed algorithmic approaches, such as set-pinning [34] or migration [15], dynamic cache partitioning has been proposed as a more transparent alternative for cache management, such as fair-caching [21] for last-level caches and MorphCache [33] for multi-level cache hierarchies in mid-size cores. Nesbit et al. [27] improved upon previous techniques by partitioning cache access bandwidth as well. We spatially distribute the shared cache (NUCA), as proposed by Kim et al. [17], to reduce access energy and time. However, the existing NUCA techniques proposed, such as SNUCA and DNUCA, do not satisfy our requirements, since SNUCA is not dynamically reconfigurable and DNUCA is not energy-efficient for large cache sizes required for manycore architectures, even when we use optimization techniques, such as way-prediction [28], partial-tag matching [19] and data migration [17]. Thus, we extend SNUCA with cache-indirection [2] to create DR-SNUCA, which is both dynamically reconfigurable as well as energy-efficient for large cache sizes. Cache partitioning techniques at different spatial granularities have been proposed, such as the page-level scheme by Cho et al. [8] and the memory-address-map based scheme proposed by Lin et al. [23]; however, we partition the cache at a finer spatial granularity without incurring heavy reconfiguration and access costs. Compositional caches [25] have been proposed which demonstrate the performance benefits of set-partitioning for embedded architectures. We further improve the energy-efficiency of such designs through indirect addressing, which reduces data movement.

Previous techniques allocate cache partitions based on various metrics, such as marginal utilities by Qureshi et al. [29]. Hsu et al. [16] tune their cache allocation algorithm to maximize different metrics such as fairness and throughput. Guo et al. [14] allocate cache partitions based on QoS provided by choosing between strict, elastic, and opportunistic schemes, while Cong et al. [9] dynamically adapt a software managed cache to balance cache-set utilization. Our online cache allocation scheme, in contrast to the offline profiling-based allocation schemes [7], on the other hand proposes to repartition cache based on application progress, a direct performance metric. We have used a fixed interval duration in our evaluation; however, DR-SNUCA can be reconfigured at variable interval durations using online tuning techniques [12].

VI. CONCLUSION

DR-SNUCA is an energy-scalable dynamically partitioned cache, which reduces the energy consumption for a 32-core system by 16.27% compared to DNUCA caches, while performing within 0.5%. The area and energy overhead of the reconfiguration mechanisms are low, and thus DR-SNUCA can be used to provide interference-free on-chip caches for manycore processors. We also show that a progress-based cache allocation algorithm leads to higher resource utilization and throughput, when used in conjunction with other dynamically partitioned resources, compared to cache allocation algorithms minimizing total cache misses.

REFERENCES


C. Liu et al., “Organizing the last line of defense before hitting the memory wall for CMPs,” in *HPCA*, 2004.


A. Jaleel, “Memory characterization of workloads using instrumentation-driven simulation.”


