# Efficient Cache Reconfiguration Using Machine Learning in NoC-Based Many-Core CMPs

SUBODHA CHARLES and ALIF AHMED, University of Florida, USA
UMIT Y. OGRAS, Arizona State University, USA
PRABHAT MISHRA, University of Florida, USA

Dynamic cache reconfiguration (DCR) is an effective technique to optimize energy consumption in many-core architectures. While early work on DCR has shown promising energy saving opportunities, prior techniques are not suitable for many-core architectures since they do not consider the interactions and tight coupling between memory, caches, and network-on-chip (NoC) traffic. In this article, we propose an efficient cache reconfiguration framework in NoC-based many-core architectures. The proposed work makes three major contributions. First, we model a distributed directory based many-core architecture similar to Intel Xeon Phi architecture. Next, we propose an efficient cache reconfiguration framework that considers all significant components, including NoC, caches, and main memory. Finally, we propose a machine learning–based framework that can reduce the exploration time by an order of magnitude with negligible loss in accuracy. Our experimental results demonstrate 18.5% energy savings on average compared to base cache configuration.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; **Reconfigurable computing**; **System on a chip**; • **Hardware** → **Network on chip**; *Metallic interconnect*; *Reconfigurable logic applications*; • **Theory of computation** → *Machine learning theory*;

Additional Key Words and Phrases: Cache reconfiguration, machine learning

**ACM Reference format:**
Subodha Charles, Alif Ahmed, Umit Y. Ogras, and Prabhat Mishra. 2019. Efficient Cache Reconfiguration Using Machine Learning in NoC-Based Many-Core CMPs. *ACM Trans. Des. Autom. Electron. Syst.* 24, 6, Article 60 (September 2019), 23 pages.
https://doi.org/10.1145/3350422

## 1 INTRODUCTION

Many-core architectures emerged as the most promising solution to satisfy growing performance demand under tight power and thermal constraints [6, 23, 41]. Multiple cores operating at a lower frequency than a single fast core enable better power and performance (PnP) tradeoff by running the workloads in parallel. The ITRS (International Technology Roadmap for Semiconductors) 2015 roadmap projects that the increased demand for information processing will drive a 30-fold increase in the number of cores by 2029 [17]. Indeed, one of the most recent many-core processor

This work was partially supported by the NSF grants CNS-1526687 and CNS-1526562.
Authors' addresses: S. Charles, 320 University Village, Apt. 3, Gainesville, FL 32603; email: subodha96@ufl.edu; A. Ahmed, 137 Yellowstone Dr Apt 106, Charlottesville, VA 22903; email: alifahmed@ufl.edu; U. Ogras, 650 E Tyler Mall, Tempe, AZ 85287; email: umit@asu.edu; P. Mishra, 432 Newell Drive, Gainesville, FL 32611-6120; email: prabhat@ufl.edu.

1084-4309/2019/09-ART60 $15.00
https://doi.org/10.1145/3350422

architectures—the Intel Xeon Phi, code named "Knights Landing" (KNL)—features 64–72 atom cores and 144 vector processing units [30].

The growing number of cores can be utilized effectively *only if* they have a fast and high throughput connection to memory. Thus, network-on-chip (NoC) has become the standard interconnect solution to address the communication requirements of many-core chip multi-processors (CMP) [25]. The NoC connects a wide variety of components including processor cores, co-processors, caches, and memory controllers to each other. In addition to an efficient NoC, optimal cache architecture is essential to reduce the memory access time as well as power consumption. The working set of the applications favors different cache sizes, while its spatial locality determines favored line size. Furthermore, applications with different temporal locality can benefit from different cache associativity. Studies have also shown that cache subsystem reconfiguration could lead to significant energy savings by meeting an application's diverse cache requirements [12, 14, 33, 34]. According to Amdahl's law, such a large contribution to the overall energy consumption makes these elements good candidates for PnP optimization. As the cache architecture affects the traffic being injected into the NoC, studying the memory hierarchy and NoC components at the same time is vital for on-chip PnP estimation of CMPs.

Dynamic cache reconfiguration (DCR) has been well studied as an effective cache energy optimization technique [34, 40]. DCR allows runtime tuning of the cache parameters (e.g., cache size, associativity, and line size) after deciding *when* and *how* to configure them using optimization algorithms. This enables the CMP to optimize energy consumption while maintaining the application's quality of service (QoS) standards. In a typical CMP architecture, level 1 (L1) caches are private for each core, whereas the level 2 (L2) cache is shared across all cores. Such an arrangement introduces dependencies between the L1 and L2 caches as the configuration of one can affect the cache accesses of the other, and vice versa [35]. Therefore, DCR techniques are commonly used to optimize L1 caches. Similarly, cache partitioning (CP) improves performance by eliminating the inter-task interference on a shared cache [15]. Hence, it is employed to judiciously divide portions of L2 cache to each core.

Dynamic tuning of multi-level caches is challenging since the exploration space is prohibitively large. Even with a small number of tunable cache parameters, the exploration space can grow exponentially making it impractical to do a simulation-based exhaustive exploration [32]. Several heuristics were proposed to reduce the exploration space by utilizing the independence between various cache parameters [12, 32]. Unfortunately, these approaches suffer from accuracy and inconsistency across different architectures.

Previous works on DCR and CP have not considered NoC traffic when calculating the overall system energy, and therefore are not suitable for making accurate architectural decisions. Our exploration framework is developed considering the complete memory hierarchy and NoC traffic. In order to explore the prohibitively large design space effectively, we propose and analyze a machine learning (ML) algorithm which predicts runtime and energy of applications with different cache configurations. This enables us to significantly reduce the exploration time, while maintaining the accuracy within an acceptable range.

Our major contributions can be summarized as follows:

(1) **DCR-CP-NoC co-optimization:** Since DCR in L1 and CP in shared L2 are closely coupled, we explore DCR and CP together in an NoC-based many-core architecture and compare it to previous studies on two-level cache configuration in bus-based architectures. This is the first attempt to consider NoC interactions during DCR.
(2) **Efficient static profiling:** We propose a machine learning algorithm to reduce the overall static profiling time compared to the exhaustive method by an order of magnitude

with minor impact on energy savings. Results are compared with exhaustive as well as heuristic-based approaches.

(3) **Dynamic programming–based optimization:** We propose a dynamic programming (DP)–based algorithm to find optimal L1 cache configurations for each application and L2 partition factors for each core. Effective utilization of DP allows this exploration to run with linear time complexity with respect to the number of cores.

(4) **Extensive evaluation:** We accurately model the NoC traffic flow and energy consumption. Then, we evaluate our approach by running 14 benchmarks on a realistic Intel Xeon Phi configuration with 32 tiles [30] using the gem5 full-system simulator [5].

The remainder of the article is organized as follows. Section 2 discusses the related work. Section 3 presents some background information required to understand our approach. Section 4 motivates the need for this work. Section 5 describes our exploration framework. Section 6 presents the experimental results. Finally, Section 7 concludes the article.

## 2 RELATED WORK

Reconfigurable cache architectures have been extensively studied utilizing techniques such as way shutdown, way management, cache partitioning, and resizing [2, 21, 24, 40]. Settle et al. [29] introduced a reconfigurable cache architecture specific for CMPs. These architectures were used to explore cache reconfiguration techniques in several orthogonal directions ranging from single core [32], multi core [13, 15], realtime [34], and embedded [35] systems. Cache partitioning (CP) techniques are mainly focused on improving the performance of many-core systems [18, 27]. Beckmann et al. proposed JIGSAW, a cache organization method that addresses scalability and interference issues of shared caches. However, their work only discussed reconfiguration in L2 cache. As shown in [35] and [15], reconfiguring L1 would change the L2 traffic and therefore, requires simultaneous L1 and L2 reconfiguration. The existing L1/L2 reconfiguration methods are primarily based on static and/or dynamic analysis [34, 35]. These methods explore various configurations and decide which one to use depending on application characteristics. If application characteristics are known *a priori*, static analysis is beneficial since dynamic analysis can pose significant overhead and lead to unpredictable performance impact. However, dynamic analysis of a limited set of configurations is the only option when static profiling is not feasible. The above approaches can lead to unrealistic results in NoC-based many-core architectures since they do not consider the energy impact of NoC traffic during exploration.

Studies on NoC traffic exploration proves the fact that NoC traffic largely affects the CMP PnP statistics [7]. Several studies were carried out in efficient distribution of memory traffic to provide QoS guarantees [19], optimum memory controller placement [37], and task scheduling [28]. Yet, the impact on NoC traffic behavior as a result of cache configuration has not been studied to date. Combining the effects of NoC communication to overall energy consumption, Chen et al. proposed to dynamically turn on and off L2 cache banks to save energy in optical NoCs which have silicon-photonic links [9]. This work is fundamentally different from ours as it works with optical NoCs that communicate using laser beams that shows completely different characteristics compared to an electrical on-chip network.

One of the major concerns in static and dynamic analysis of possible cache configurations is the exploration time. The exploration space grows significantly with the number of tunable cache parameters and levels of cache (L1, L2, etc.). In order to reduce the exploration complexity, heuristic-based approaches [12, 32] consider only a small set of potential configurations based on specific observations (such as independence or priority of specific parameters). While these approaches can reduce the exploration time, the quality of results can be far from optimal. Our proposed approach utilizes machine learning to drastically reduce the exploration time with minor (acceptable) loss

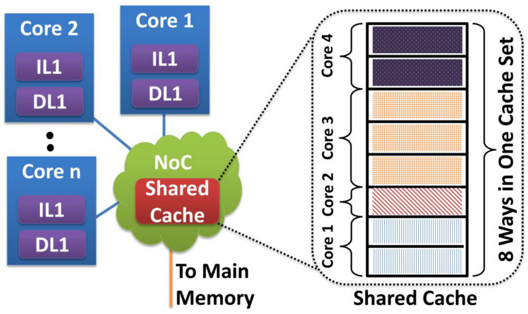


Fig. 1. Many-core architecture with private instruction (IL1) and data (DL1) caches as well as shared L2 cache.

in the quality of results. Our work provides a coherent framework that enables the exploration of optimum cache configurations in NoC-based many-core CMPs while addressing the exploration space complexity by a machine learning–based static profiling technique.

## 3 BACKGROUND

### 3.1 Configurable Cache Architecture

Figure 1 shows a standard NoC-based many-core architecture with a shared L2 cache, private instruction (IL1), and data (DL1) caches. Both L1 and L2 caches are reconfigurable. The L1 cache configuration can be changed by changing its capacity, line size, and associativity. The L2 cache is partitioned among all the cores and the partitions are decided depending on the application requirements.

The cache architecture used in our work contains four separate banks operating as four separate ways. Figure 2 shows cache configurability of L1 caches with an illustrative example. If the base cache size is 4kB, we have four 1kB banks [39] (Figure 2(a)). Cache associativity can be reconfigured by concatenating neighboring ways (Figure 2(b)). To change cache sizes, gated $V_{dd}$ is used to shut down banks causing the effective cache size to shrink. A 4kB cache can have four-way, two-way, and one-way (direct mapped) associativity. However, if the cache size is reduced to 2kB, it can only have two-way and one-way associativity because a four-way associativity will mean shutting down or concatenating half of the bank/way and that is not supported in the architecture (Figure 2(c)). Line size can be changed by changing the number of unit length blocks fetched during each cache access (Figure 2(d)).

This reconfigurable cache architecture has very little overhead and requires simple hardware changes [40]. Runtime re-configuration of the L1 cache is done by using special configuration registers. The configuration registers inform the cache tuner, which is a lightweight process implemented on hardware, to concatenate ways to change associativity. Similarly, the configuration registers can be configured to shut down ways causing the cache size to change. It is important to note that our contribution is an efficient technique that determines which cache configuration should be used for a given application. As explained in related work and the following sections, runtime configuration of caches is a well studied problem and our architecture proposes to use existing mechanisms to tune the cache once an optimum configuration is found.

For the shared L2 cache, we use a way-based partitioning method that differs from traditional LRU replacement policy that implicitly partitions cache sets based on demand [29]. Figure 1 depicts a cache set with eight-way associativity which can be partitioned in the granularity of ways. Each core will access only the group of ways assigned to it in all the cache sets. LRU replacement is enforced in each individual group by maintaining a separate set of "age bits." Way-based partitioning

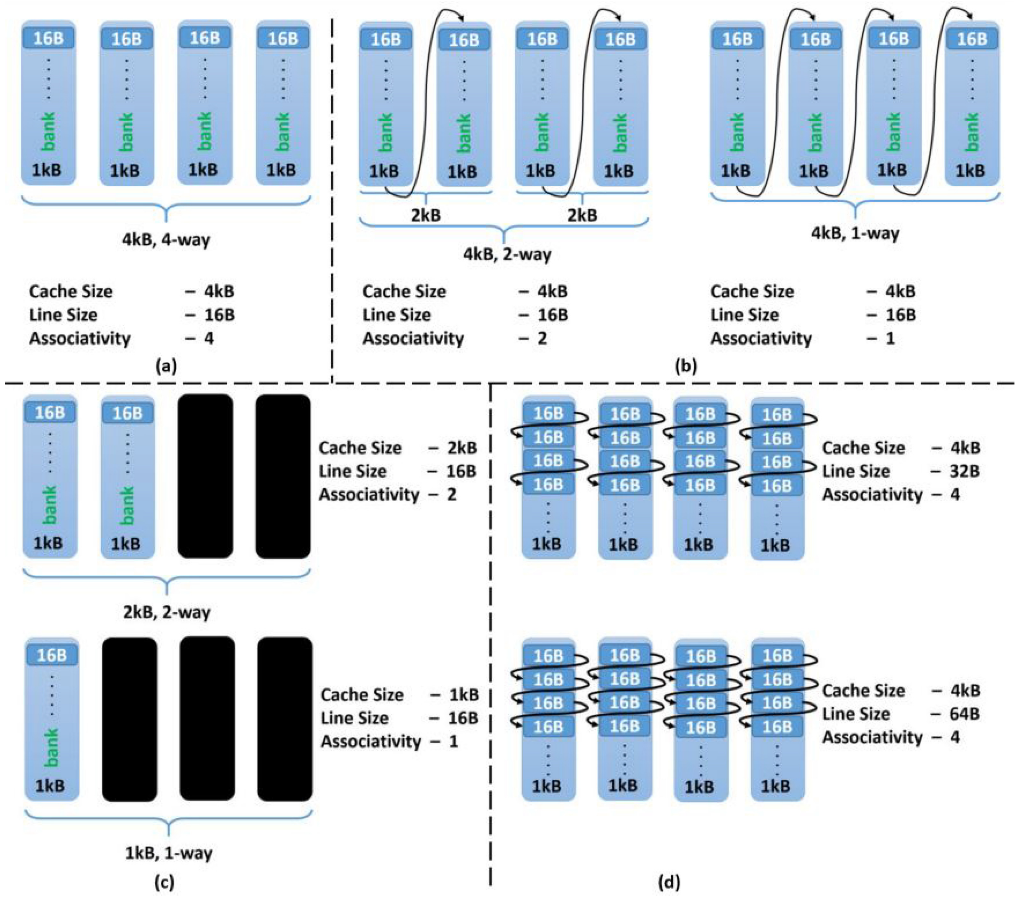


Fig. 2. Cache configurability of a 4kB cache arranged as four banks. (a) Base cache layout with four banks, (b) change of associativity by way concatenation, (c) change of cache size by shutting down ways, and (d) change of line size by fetching subsequent blocks and storing them in subsequent cache blocks.

is useful for exploiting energy efficiency. The number of ways assigned to a core is referred to as the core's *partition factor*. Core 1 in Figure 1, for example, has an L2 partition factor of two. In our study, we use static cache partitioning. In other words, each core's L2 partition factor is constant throughout system execution and is predetermined. Since L1 DCR has a major impact on L2 CP, the exploration framework should support tuning of all possible parameters simultaneously [32]. For example, the number of L2 accesses is dependent on the number of L1 misses. Also, the miss penalty of the L1 cache is dependent on the configuration of the L2 cache.

## 3.2 Memory Modes in KNL Architecture

Knights Landing (KNL) is the codename for the second generation Xeon-Phi processor introduced by Intel which targets highly parallel workloads [30]. An overview of the KNL architecture is shown in Figure 3. It has 36 tiles arranged in a Mesh interconnect. It supports two types of memory: (i) multi-channel DRAM (MCDRAM), which is a 16 gigabyte high-bandwidth memory, and (ii) double data rate (DDR) memory, which has a capacity of 384 gigabytes with less bandwidth. The architecture gives the option of configuring these two memories in several configurations, which are called *memory modes*.

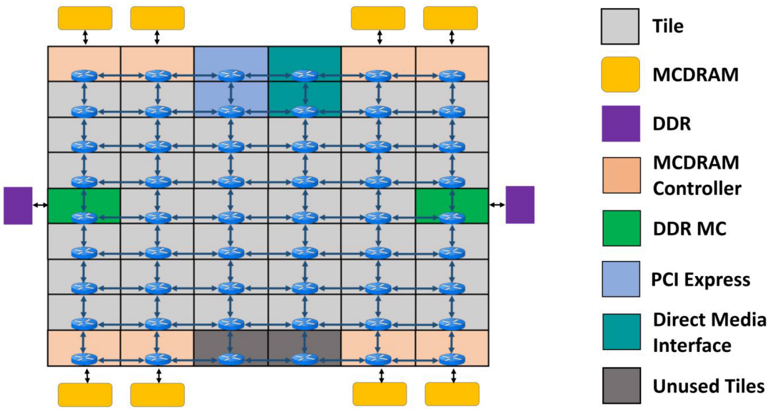


Fig. 3. Overview of the KNL architecture [30].

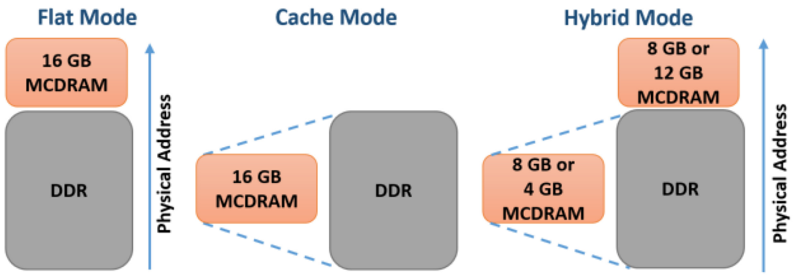


Fig. 4. Three memory modes available in KNL architecture: Flat, Cache, and Hybrid [30].

KNL introduces three memory modes which can be selected during boot time from the BIOS. These modes are *Flat, Cache*, and *Hybrid* (Figure 4). The introduction of the memory modes enables the exploitation of the different characteristics of applications running on the processor.

— **Flat Mode:** In Flat mode, the MCDRAM and the DDR memory are both mapped in the same system address space. This mode is ideal for applications with data that can be separated into categories of a larger, low-bandwidth region, and a smaller, high-bandwidth region.

— **Cache Mode:** In Cache mode, MCDRAM acts as a last level cache which sits in between the DDR memory and the L2 cache. The cache is direct mapped with a cache line size of 64 bytes. All memory requests first go to the MCDRAM for a cache memory lookup. If there is a cache miss, they are sent to the DDR memory.

— **Hybrid Mode:** In Hybrid mode either a quarter or a half of the MCDRAM is used in Cache mode and the rest is used as Flat mode memory. The DDR memory will be served by the cache portion. This works well for a variety of applications that take advantage of storing frequently accessed data in flat memory while also benefiting from regular caching.

## 4 MOTIVATION

### 4.1 Impact of DCR on Power and Performance

As an illustrative example, we ran FFT benchmark from SPLASH-2 benchmark suite on a similar architecture and recorded energy and runtime values for two DL1 configurations: **32K_2W_32B**[1]

[1]In this article, we show cache configurations using three parameters. For example, **32K_2W_32B** represents a cache with 32kB cache capacity, two-way set associativity, and 32 byte line size.

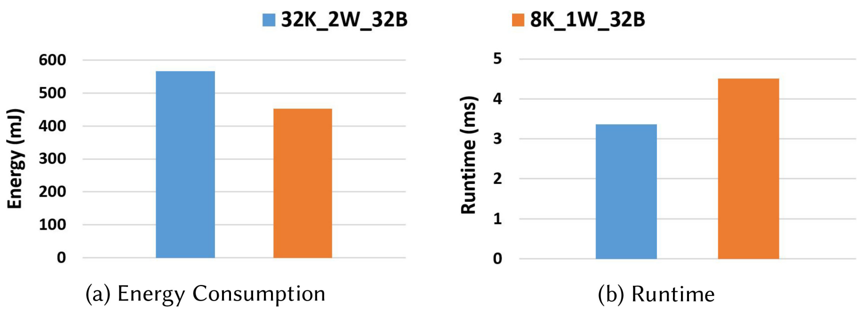


Fig. 5. Energy consumption and runtime of two cache configurations running FFT. The choice of a specific configuration presents a tradeoff between energy savings and performance overhead.

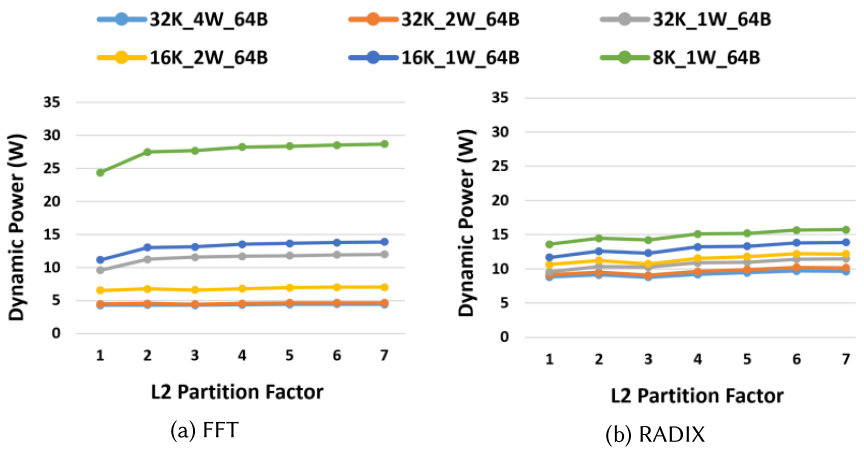


Fig. 6. NoC dynamic power consumption with different DL1 configurations and L2 partition factors. IL1 cache is fixed at 32K_2W_64B.

and **8K_1W_32B** [36]. The IL1 configuration is the same for both executions. As shown in Figure 5, they give different runtime and energy values. If the performance of the system is not critical, using the configuration that gives the least energy consumption (8K_1W_32B) should be selected. If the performance target is to execute the application in 4*ms*, we can observe in Figure 5(b) that 8K_1W_32B does not meet the requirement. In that case, 32K_2W_32B should be selected, which meets the desired performance even though it consumes more energy.

In reality, the number of possible cache configurations is much larger than two. For example, Section 5.3 shows that there are 504 valid cache configurations in our specific exploration framework. When NoC power consumption is considered, the energy optimization problem becomes even more complex.

With NoC being the most preferred interconnection technology in modern CMPs, it is imperative to account for the NoC while exploring the cache configurations. As shown in Figure 6, power consumption of the NoC portion of CMP is a function of the application executed, L1 and L2 configurations. We observe that with increased L1 cache size, NoC power consumption decreases. This is expected because increasing the L1 cache size causes less L1 misses and as a result, fewer packets being injected into the network. From the NoC power model illustrated in Section 5.2, we can see that decreasing the number of packets on the network decreases the NoC power. NoC power is shown in this figure instead of energy for comparisons across applications by eliminating

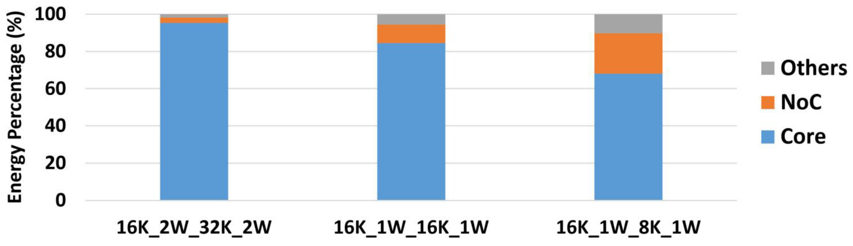


Fig. 7. Core and NoC energy as a percentage of total CMP energy consumption. Line size is fixed at 64B and a partitioning factor of 2 is used for every application. 16K_2W_32K_2W interpreted as 16kB IL1 with two-way associativity and 32kB DL1 with two-way associativity.

the performance factor. In this study, both the applications are running on exactly the same CMP model.

Figure 7 illustrates NoC energy consumption as a percentage of total energy. NoC energy, core energy, and energy contribution from other components (last level cache, memory, etc.) are shown for comparison. As expected, the NoC energy consumption decreases with increasing cache size and associativity. More precisely, the NoC energy consumption percentage reduces by more than half from 22% to 10% when the DL1 cache size is doubled. This is expected since the number of requests going to off-chip memory is reduced. It further reduces to 3% with further increase in DL1 size and IL1 associativity. A fixed line size of 64B and a fixed partition factor of 2 are used for all three experiments. Irrespective of the choice of partition factor, our results show that NoC is an important contributor to overall energy consumption.

Given the above observations, it is evident that ignoring the energy contribution from NoC when exploring L1 DCR and L2 CP tradeoffs in an NoC-based many-core architecture can lead to misleading conclusions. Therefore, it is important to model NoC traffic as well as its energy accurately.

### 4.2 Impact of Memory Modes in KNL Architecture

To further emphasize the importance of cache reconfiguration exploration in many-core architectures, we ran some experiments on an Intel Xeon Phi 7210 hardware platform which implements the KNL architecture [16]. The selection between different memory modes is essentially a cache reconfiguration as the total amount of memory is fixed and it is divided among shared cache and main memory to fit the application characteristics. The two extreme configurations out of all the possibilities are Flat and Cache modes since Cache mode allocates all 16GB of MCDRAM memory as cache and Flat mode allocates it as main memory. Since the Hybrid configurations fall in the middle, we ran tests using these extreme configurations to illustrate their effects on application runtime.

Figure 8 shows the execution time of six complex benchmarks running on the KNL Flat and Cache modes. All these applications show benefits in the Cache mode. Yet, the percentage speedup varies drastically between a minimum of 2.3% for *LINPACK* benchmark to a maximum of 77.1% for *LBS3D*. This behavior is expected when executed on a Xeon Phi hardware board since the Cache mode is able to exploit the memory access patterns in LBS3D benchmark, and as a result, provided significant performance improvement compared to Flat mode. On the other hand, Cache mode is slightly better than Flat mode in the case of LINPACK memory access patterns. It is important to note that the execution time for each application is normalized with respect to the execution time

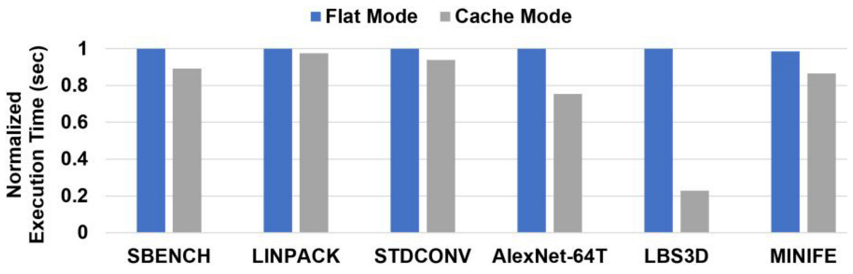


Fig. 8. Execution time variation in Cache vs Flat memory modes in an Intel Xeon Phi 7210 processor.

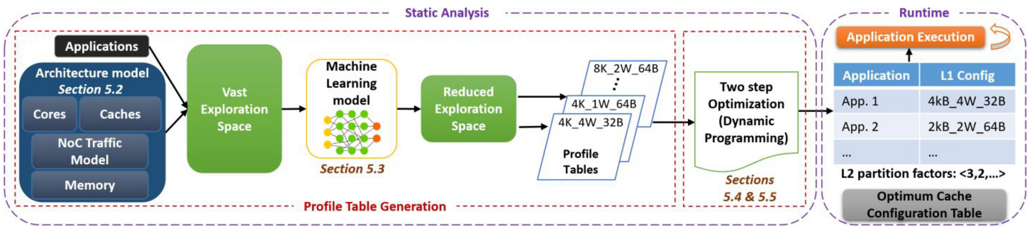


Fig. 9. Overview and main steps in our proposed approach. An optimum cache configuration table is generated through static analysis (machine learning reduces the static profiling effort, and dynamic programming finds the best possible configuration for an application). When an application starts execution (during runtime), the cache tuner selects the respective configuration from the optimum cache configuration table and tunes the cache parameters.

in Flat mode. Therefore, the comparison shows how much Cache mode will benefit over Flat mode for a given application.

This proves that state-of-the-art CMPs support different cache configurations and as a result, application power and performance can vary drastically. However, instead of the limited number of possible cache configurations introduced in the KNL architecture, it is beneficial to have more tunable cache parameters so that more optimization opportunities are available. This has not been possible due to the lack of simulation frameworks that capture all the significant components in a CMP including NoC, caches, and main memory. The simulation framework should be accompanied by an efficient cache reconfiguration framework which selects the best cache configuration for a given application.

### 4.3 Design Space of Possible Cache Configurations

Even though having more tunable cache parameters gives more optimization opportunities, it can cause the number of possible cache configurations (design space) to grow exponentially. This makes it infeasible to run all possible cache configurations exhaustively and come to a conclusion on the best cache configuration for a given application. The existing solutions for this as explained in Section 2, suffer from loss of accuracy and inconsistency across architectures. Therefore, we propose a machine learning–based approach that achieves high accuracy and drastically reduces the exhaustive exploration time. We provide a detailed calculation for the motivation of our approach in Section 5.3.1 after we have defined the terminology used in our article in Section 5.1.

## 5 EFFICIENT CACHE-NoC-MEMORY CO-EXPLORATION

Figure 9 shows an overview of our cache reconfiguration framework. It consists of two parts: (i) static profiling of application programs, and (ii) runtime (dynamic) inter-application cache reconfiguration. The static profiling is used to determine the most profitable cache configuration for a given application under various design constraints. This is done by reducing the exploration space using a machine learning–based approach (Section 5.3) and running a dynamic programming–based optimization algorithm to find the optimum cache configurations (Section 5.4 and Section 5.5). An optimum cache configuration table is created at this stage; each row of the table contains the most profitable cache configuration for an application. When an application starts execution, the cache tuner changes the cache based on the configuration stored in the table at runtime. This cache tuning is possible according to the reconfigurable cache architecture described in Section 3.1. The goal of this article is to develop a framework that efficiently constructs the optimum cache configuration table. Using the configurations in the table to tune the cache during runtime is beyond the scope of this article. Runtime configuration is a well studied problem and existing mechanisms can be applied to deal with it.

This section is organized as follows. We first provide the problem formulation (Section 5.1) followed by the power/energy models (Section 5.2). The next three subsections describe the three important steps in our static profiling framework: (i) machine learning–based static profiling (Section 5.3), (ii) dynamic programming–based per-core optimization (Section 5.4), and (iii) optimization across all cores (Section 5.5).

## 5.1 Problem Formulation

We model the many-core system with the following parameters and convert the exploration into a minimization problem.

- The CMP consists of $n$ cores. The set of all cores is denoted by $\mathbb{P} : \{p_1, p_2, \ldots, p_n\}$.
- Each core consists of private IL1 and DL1 caches. Each private cache can be configured into $r$ different configurations as explained in Section 3. The set of all cache configurations is $\mathbb{C} : \{c_1, c_2, \ldots, c_r\}$.
- The L2 cache, which is shared among all $n$ cores, is $\alpha$-way associative with way-based partitioning support.
- $m$ independent applications $\mathbb{T} : \{\tau_1, \tau_2, \ldots, \tau_m\}$ are executed on the CMP model without violating QoS requirements quantified by a *Performance Target* $D$. In a given application mix, each application can finish at different times, but there is a common soft deadline by which all applications should be completed. Violations of this performance target deteriorate the QoS. The illustrative example in Section 4 introduces the usage of $D$. How to select $D$ for a given application set is described in more detail in Section 6.2.

The assumption of a common soft deadline for an application set, which is represented by the performance target, is made to generate results that are comparable with existing approaches. In many classes of embedded systems, the tasks can be divided into multiple sets of tasks, where the priority as well as the deadline is the same for the tasks in each set. In other words, the priority as well as the deadline will be different for tasks in two different sets. In the extreme, each set may contain only one task, thereby enabling individual deadlines for each task. Our approach is applicable for tasks with individual deadlines without any changes to the proposed algorithm.

The goal of our optimization algorithm is to find a reconfiguration scheme $\mathbf{R}$ for IL1 and DL1 and a partition scheme $\mathbf{\Pi}$ for the L2 cache such that the assigned applications run with minimal energy $E$ without violating QoS standards where

$$E = E_{cores} + E_{caches} + E_{noc} + E_{memory} + E_{directories}. \tag{1}$$

The inputs to the proposed algorithm are as follows:

—Set of all possible L1 cache configuration schemes **R** which assigns a cache configuration to each IL1 and DL1 cache for each application: $\mathbf{R} : \mathbb{T} \rightarrow C_I, C_D$.
—Set of all possible L2 cache partitioning schemes $\mathbf{\Pi} : \{w_1, w_2, \ldots, w_n\}$ which assigns $w_k$ ways to core $k$.
—An application mapping scheme $\mathbf{M}$: $\mathbb{T} \rightarrow \mathbb{P}$ which maps the $m$ applications to the available cores. The application mapping is beyond the scope of this article and **M** is assumed to be available. Here, $\delta_k$ denotes the number of applications mapped to core $k$.

Let $\tau_{k,i}$ denote the $i^{th}$ application running on core $k$. Similarly, $\epsilon_{k,i}(c_I, c_D, w_k)$ is the total CMP energy consumption for $\tau_{k,i}$ with $c_I$, $c_D$ as IL1 and DL1 cache configurations, respectively, and $w_k$ partition factor. In other words, $\epsilon_{k,i}(c_I, c_D, w_k)$ denotes the energy contribution to total energy $E$ from $\tau_i$. Here, $t_{k,i}(c_I, c_D, w_k)$ represents the time spent by $\tau_{k,i}$ with said cache configurations. Then, the optimization problem can be expressed as the minimization of

$$E = \sum_{k=1}^{n} \sum_{i=1}^{\delta_k} \epsilon_{k,i}(c_I, c_D, w_k) \tag{2}$$

subject to

$$\max_{k=1..n} \left( \sum_{i=1}^{\delta_k} t_{k,i}(c_I, c_D, w_k) \right) \leq D, \tag{3}$$

$$\sum_{k=1}^{n} w_k = \alpha; w_k \geq 1, \forall k \in [1, n]. \tag{4}$$

As shown in Equations (1) and (2), $E$ consists of energy consumption in cores, caches, NoC, off-chip memory, and directories. The constraint in Equation (3) guarantees that all applications will meet the required QoS standards quantified by the performance target $D$, whereas Equation (4) verifies that the partitioning scheme is valid.

## 5.2 Cache Coherent Traffic Flow and Energy Models

This section describes the traffic flow and energy model used in the NoC and Cache of our architecture model.

*5.2.1 NoC Traffic and Energy Model.* Since dynamic power consumption of an NoC is a function of the traffic flow, we need to model a realistic traffic flow and use an energy model that captures the changes in the traffic flow. For this purpose, we model the traffic flow of *Cache* memory mode in KNL architecture. An example is shown in Figure 10 to illustrate the traffic behavior of a memory request in Cache mode. When multiple cores are active and send many packets to the network, their contention for resources is captured by the model presented in [1]. We used the same model in our experiments that includes a credit-based flow control mechanism, buffers, arbiters, and so on to capture packet behavior accurately.

In a typical mesh network where a router is connected to each processing element, energy consumption for sending one bit of data from a source tile ($t_s$) to a destination tile ($t_d$) can be calculated using the Manhattan distance between them [4]. The *bit energy metric* defined by Ye et al. [38] defines the dynamic part of the communication energy as

$$E_{bit} = E_F + E_L + E_B, \tag{5}$$

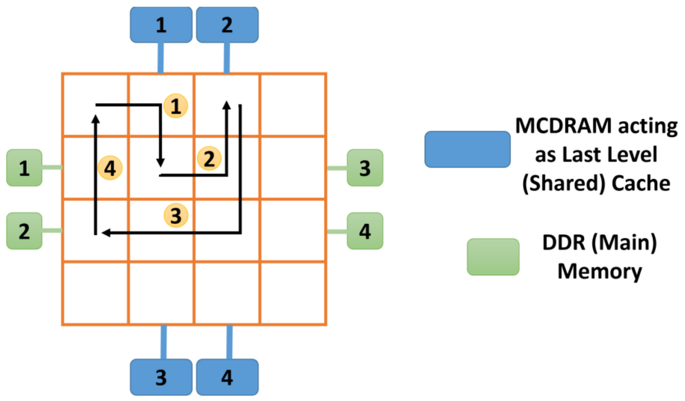


Fig. 10. Traffic model in Cache memory mode in KNL architecture in the case of an L1 and MCDRAM miss: (1) L1 cache miss. Memory request injected on the network to check the tag directory, (2) request forwarded to MCDRAM which acts as a cache after miss in tag directory, (3) request forwarded to memory after miss in MCDRAM, and (4) data read from memory and sent to the requester.

where $E_F$, $E_L$, and $E_B$ represent the energy consumption per bit by the switch fabric, link, and buffer, respectively. If $V_i$ represents the supply voltage for tile $i$ and bit energy values are measured at $V_{DD}$, the energy needed to transmit one bit through $P$ tiles can be abstracted by [26]:

$$E_{bit}^{t_s,t_d} = \sum_{i \in P} (E_F(i) + E_L(i) + E_B(i)) \cdot \frac{V_i^2}{V_{DD}^2}. \tag{6}$$

The links connecting the routers consume power because of their switching activity (toggle between logic 0 and logic 1). Power consumption in a traditional router with four pipeline stages (routing computation, virtual channel allocation, switch allocation, and switch traversal) is a combination of power consumed in buffers, arbiter, allocator, and crossbar switch. This is captured by $E_B$ and $E_F$.

Assuming that the architecture is fixed, the energy model in Equation (6) shows two main contributors to NoC power:

(1) Number of packets injected to the network: this is directly related to the number of cache misses in L1 and L2 caches.
(2) Average hops traversed by packets: depends on the placement of NoC components such as memory controllers, home directories, and last level cache.

This NoC energy model was adopted from the work done by Ogras and Marculescu [25] and it is validated with both simulations and real hardware data.

*5.2.2 Cache Energy Model.* The total energy consumption of cache ($E_{cache}$) is not only the energy consumed by the cache memory ($E_{array}$), but also the energy consumed by the memory addressing path ($E_{address\_path}$) and the I/O path ($E_{I/O\_path}$) [31]. Therefore, the total cache energy consumption can be calculated as

$$E_{cache} = E_{array} + E_{address\_path} + E_{I/O\_path}. \tag{7}$$

$E_{address\_path}$ is determined by the switching activity of the address bus. The tag and data memory arrays usually dominate the total cache energy consumption $E_{array}$. The energy model used in our approach is based on dynamic logic where bit lines are pre-charged on every access. Therefore, the energy consumed by the tag and data arrays will be determined by the number of accesses. The

I/O path includes I/O pads as well as address and data buses connected to it. Out of these components, the switching activity of the I/O pads usually dominate that energy component ($E_{I/O\_path}$). Therefore, the three main components of $E_{cache}$ can be computed as follows:

$$E_{array} = \alpha \cdot tag\_access + \beta \cdot blk\_access, \tag{8}$$

$$E_{address\_path} = \gamma \cdot bsr\_addr\_bus, \tag{9}$$

$$E_{I/O\_path} = \delta \cdot bsr\_addr\_pad + \epsilon \cdot bsr\_data\_pad, \tag{10}$$

where

$tag\_access$: access rates of bit-lines in cache tag arrays;
$blk\_access$: access rates of bit-lines in cache block arrays;
$bsr\_addr\_bus$: bit switching rates of address bus;
$bsr\_addr\_pad$: bit switching rates of address pads;
$bsr\_data\_pad$: bit switching rates of data pads;
$\alpha, \beta, \delta, \gamma, \epsilon$: constants depending on VLSI implementation.

This model is similar to the cache energy model implemented in the widely used McPAT simulator and has been verified with hardware data [20]. We used the default energy models available in the McPAT simulator for other components in the SoC.

## 5.3 Efficient Static Profiling Using Machine Learning

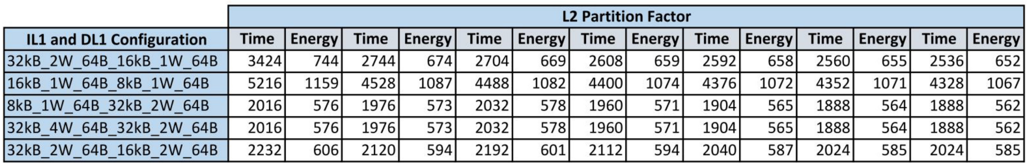
We need a profile table including the runtime and energy consumption of each application when running with all valid cache configurations to give as input to the optimization algorithm (second and third steps of Algorithm 2).

*5.3.1 Design Space of Possible Cache Configurations.* According to the reconfigurable cache architecture described in Section 3, both IL1 and DL1 have $6(= 3 + 2 + 1)$ possible configurations each. When two possible line sizes are used (64B and 32B), changing IL1 and DL1 at the same time, it gives $72(= 6 \times 6 \times 2)$ candidates for IL1 and DL1. It is infeasible to profile application with all possible L1 reconfiguration schemes $\mathbf{R}$, all possible L2 partition schemes $\mathbf{\Pi}$, for the whole application set $\mathbb{T}$ and all possible application mapping schemes $\mathbf{M}$ [35].

As a solution for this, Wang et al. [32] proposed to reduce the exploration time significantly based on the following observations:

- All the applications in Table 2 are independent with no inter-application data sharing. An application can always start and complete on the assigned core without any migrations happening during runtime.
- The L1 cache is private for each core and the configuration of L1 in one core does not have any effect on the other core's configuration as there are no multi-threaded applications that map to two cores in our application set.
- With L2 partitioning, each application uses an independent portion of the shared cache which makes it a logical private cache.

With the applications and all the caches being independent, we can profile each application as it was running on a uni-processor with a $w_i$-way associative L2 cache with the capacity equal to $\frac{w_i}{w} \times \textit{original cache size}$. In this case, the total number of simulations required would be $|\mathbf{R}| \times (\alpha - 1) \cdot m$. Thus, it takes $72 \times 7 \times 14 = 7{,}056$ simulations for 14 applications. Even after this reduction, the simulations take approximately 1 month to complete. Since the number of simulations can grow exponentially with the number of tunable cache parameters and applications, this exhaustive exploration becomes infeasible, when the design space becomes arbitrarily large. As a solution,



| | L2 Partition Factor | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IL1 and DL1 Configuration | Time | Energy | Time | Energy | Time | Energy | Time | Energy | Time | Energy | Time | Energy | Time | Energy |
| 32kB_2W_64B_16kB_1W_64B | 3424 | 744 | 2744 | 674 | 2704 | 669 | 2608 | 659 | 2592 | 658 | 2560 | 655 | 2536 | 652 |
| 16kB_1W_64B_8kB_1W_64B | 5216 | 1159 | 4528 | 1087 | 4488 | 1082 | 4400 | 1074 | 4376 | 1072 | 4352 | 1071 | 4328 | 1067 |
| 8kB_1W_64B_32kB_2W_64B | 2016 | 576 | 1976 | 573 | 2032 | 578 | 1960 | 571 | 1904 | 565 | 1888 | 564 | 1888 | 562 |
| 32kB_4W_64B_32kB_2W_64B | 2016 | 576 | 1976 | 573 | 2032 | 578 | 1960 | 571 | 1904 | 565 | 1888 | 564 | 1888 | 562 |
| 32kB_2W_64B_16kB_2W_64B | 2232 | 606 | 2120 | 594 | 2192 | 601 | 2112 | 594 | 2040 | 587 | 2024 | 585 | 2024 | 585 |

Fig. 11. A portion of the profile table generated from the machine learning algorithm for *stringsearch* benchmark. 32kB_2W_64B_16kB_1W_64B interpreted as 32kB IL1 with two-way associativity and 64B line size and 16kB DL1 with one-way associativity and 64B line size. Time and Energy are reported in **ms** and **mJ**, respectively.

we propose a machine learning–based approach which runs only few simulations and uses that as training data to tune a neural network model which then predicts the rest of the profile table. With this method, the time required to build the full profile table would be much lower. Figure 11 shows profile table entries for five L1 cache configurations (out of 72 possible configurations) created by machine learning.

*5.3.2 Algorithm.* First, a portion of the profile table entries are filled up by simulating an application on different configurations. Next, several models are trained with the collected data, and the model with least error is selected. If the error is within a threshold, then we predict the rest of the profile table entries using that model. Otherwise, we collect additional training data by running more simulations, and repeat the procedure until the error threshold criteria is satisfied. Algorithm 1 describes our machine learning–based approach. Here, $X_{all}$ denotes the set of all possible configurations for an application. $X_{sim}$ is the set of configurations on which we already simulated the application and have the corresponding table entries $Y_{sim}$. The remaining configurations are in $X_{pred}$ and needto be predicted. It is evident that $X_{all} = X_{sim} + X_{pred}$. Initially, all the table entries are empty. Thus, $X_{pred} = X_{all}$ and $X_{sim} = \emptyset$ [lines 2–3]. In the subsequent iterations, some of the configurations ($X_{sel}$) are randomly selected from $X_{pred}$ for simulation [line 6]. After simulating with $X_{sel}$ configurations, $Y_{sel}$ entries are put into the profile table. Consequently, $X_{sel}$ configurations are removed from $X_{pred}$ and added to $X_{sim}$ [lines 7–9]. The size of $X_{sel}$ determines the number of simulations carried out in each iteration. Next, we train multiple models with the filled table entries $Y_{sim}$ and their configurations $X_{sim}$. For model building purposes, these entries and configurations are divided into three groups: training ($X_{train}$, $Y_{train}$), cross-validation ($X_{cv}$, $Y_{cv}$), and testing ($X_{test}$, $Y_{test}$) [lines 10–11]. This essentially means that $X_{sim} = X_{train} + X_{cv} + X_{test}$ and $Y_{sim} = Y_{train} + Y_{cv} + Y_{test}$. The training set is used for training the model. Cross-validation is used for hyper-parameter tuning such as learning rate or regularization parameter. Test set is used for determining the prediction accuracy of models. A standard split is used in our experiments : 70% for training, 15% for cross-validation, and 15% for testing. These models are further tuned by parameter sweeping [lines 13–19]. In our experiments, we trained a shallow neural network, and changed the number of nodes in the hidden layer during the parameter sweep. If the prediction error is larger than the error threshold $\epsilon$, we collect more simulation data and repeat the model building procedure. Alternatively, if error is within the threshold, then the model is used for predicting the rest of the profile table. Input to the model will be the configuration set $X_{pred}$ and output will be the predicted energy and runtime values ($\hat{Y}_{pred}$) [line 24]. Full profile table is built by combining predicted data $\hat{Y}_{pred}$ and simulated data $Y_{sim}$ [line 25].

To calculate the error threshold, Normalized Root Mean Square Error (NRMSE) is used (Equation (11)). Using NRMSE is advantageous, since it is a relative measurement. So the same threshold can be used for all applications.

$$NRMSE = 100\% * \sqrt{MSE}/\bar{Y}, \tag{11}$$

where MSE is the mean squared error and $\bar{Y}$ is the average value. NRMSE is measured over the test dataset, $Y_{test}$ and $\hat{Y}_{test}$, while calculating the threshold. As experimental results demonstrate, our machine learning framework can give a speedup of 7.76 times when the error threshold is set at 5%.

**ALGORITHM 1:** Profile table generation using Machine Learning

```
/* Model building                                                    */
foreach application do
    $X_{pred} = X_{all}$
    $X_{sim} = \emptyset$
    $min\_error = \infty$
    while $X_{pred} != \emptyset$ and $min\_error > \epsilon$ do
        $X_{sel} = randomSample(X_{pred})$
        $X_{pred} = X_{pred} - X_{sel}$
        $X_{sim} = X_{sim} + X_{sel}$
        $Y_{sel} = simulate(X_{sel})$
        $[X_{train}, X_{cv}, X_{test}] = distribute(X_{sel})$
        $[Y_{train}, Y_{cv}, Y_{test}] = distribute(Y_{sel})$
        foreach regression algorithm do
            /* Parameter sweep                                       */
            foreach param do
                $model = train(X_{train}, Y_{train}, X_{cv}, Y_{cv}, param)$
                $\hat{Y}_{test} = predict(X_{test}, model)$
                $error = nrmse(Y_{test}, \hat{Y}_{test})$
                if $error < min\_error$ then
                    $min\_error = error$
                    $sel\_model = model$
                end
            end
        end
    end
    /* Profile table entry prediction                                */
    $\hat{Y}_{pred} = predict(X_{pred}, sel\_model)$
    $Y_{all} = Y_{sim} + \hat{Y}_{pred}$
end
```

## 5.4 Per-Core Optimization

We tackle the optimization problem in two steps. First we find the optimum L1 cache configuration for each core and then optimize across all cores to find the best L2 partition scheme. This subsection illustrates optimizing each core with best L1 cache configuration. Since static partitioning is used, applications running on one core will have the same partition factor, $w_k$. Thus, we find best **R** under different L2 partition factors. Mathematically, the goal is to find L1 configuration **R** to minimize $E_k(w_k) = \sum_{i=1}^{\delta_k} \epsilon_{k,i}(c_I, c_D, w_k)$ constrained by $\sum_{i=1}^{\delta_k} t_{k,i}(c_I, c_D, w_k) \leq D$, with $k$ and $w_k$ fixed $\forall k \in [1, n]$ and $\forall w_k \in [1, \alpha - 1]$.

To find minimum energy consumption, this can be discretized to simplify the problem and we use a DP algorithm on that. Let $\epsilon_k^{min}(w_k) = \sum_{i=1}^{\delta_k} min\{\epsilon_{k,i}(c_I, c_D, w_k)\}$ and $\epsilon_k^{max}(w_k) = \sum_{i=1}^{\delta_k} max\{\epsilon_{k,i}(c_I, c_D, w_k)\}$ denote minimum and maximum possible energy on core $k$. Thus, the energy consumption $E_k(w_k)$ of core $k$ is bounded by these min and max values. Let $\Phi_i^E$ be the current solution for the first $i$ applications where $E$ is the cumulative energy consumption achieving best runtime. Runtime $T[i][E]$ for $\Phi_i^E$ is stored in a two-dimensional table $T$ and the solution

$$\begin{array}{|c|}\hline \textbf{If } (T[i][E] > T[i-1][E - \epsilon_{k,i}(c_I, c_D, w_k)] + t_{k,i}(c_I, c_D, w_k))\ \{ \\ T[i][E] = T[i-1][E - \epsilon_{k,i}(c_I, c_D, w_k)] + t_{k,i}(c_I, c_D, w_k)\} \\ \hline \end{array}$$

Fig. 12. Recursive formula for dynamic programming.

for $\Phi_i^E$ is updated whenever the runtime can be improved. The recursive formula used in our DP approach is shown in Figure 12.

The final optimal energy consumption $E_k^*(w_k)$ is found by

$$E_k^*(w_k) = \min\{E_k \mid T[\delta_k][E_k] \leq D\}. \tag{12}$$

At the end of the DP algorithm, Equation (12) provides the solution for core $k$ with partition factor $w_k$, which has minimum energy consumption with QoS constraints satisfied.

### 5.5 Optimizing the Entire CMP

Now that we have the optimal energy $E_k^*(w_k)$ calculated for a given partition factor $w_k$ on core k, we combine the solutions across all cores to find the minimum total CMP energy consumption $E^*$ within all partition schemes $\mathbf{\Pi}$ as

$$E^* = min\left\{\sum_{k=1}^{n} E_k^*(w_k)\right\}, \quad \forall\left\{w_1, w_2, \ldots, w_n\right\} \in \mathbf{\Pi}. \tag{13}$$

Our approach across all the steps starting from building the profile table is summarized in Algorithm 2. Optimizing for each core to find the best L1 configuration is shown in step 2. At each iteration (lines 2–26), all discretized energy values ($\epsilon$) and L1 cache configurations for current application $\tau_{k,i}$ are examined. The recursive DP algorithm given in Figure 12 is included in lines 4–23 in two parts: initially for the first application and then from application 2 to $\delta_k$.

The time complexity for step 2 is $O(n \cdot \alpha \cdot \delta_k \cdot |\mathbf{R}| \cdot (\epsilon^{max} - \epsilon^{min})/q_t)$, where $\epsilon^{max} - \epsilon^{min}$ is the energy range and $q_t$ is the discretization interval. We have used a constant $q_t$ in the DP algorithm throughout our exploration. Step 3 iterates through all valid $\mathbf{\Pi}$ to find the final solution with time complexity $O(n \cdot |\mathbf{\Pi}|)$. Since cache parameters are constant for the given architecture model, the time complexity for both step 2 and step 3 is linear with respect to $n$.

When calculating the exact runtime without neglecting the constant factors, it is important to note that $|\mathbf{\Pi}|$ depends on the constraint in Equation (4). Therefore, this becomes the classic "stars and bars" problem in combinatorics. A theorem in combinatorics states that *for any pair of positive integers n and k, the number of k-tuples of positive integers whose sum is n, is equal to the number of* $(k-1)$*-element subsets of a set with n* − 1 *elements*. Therefore, according to our notation, $|\mathbf{\Pi}| = \binom{\alpha-1}{n-1}$. However, this calculation is not required since in reality, the number of ways ($\alpha$) is as small as 8 or 16 (8-way or 16-way). Given that $n$ represents the number of active cores, If $n$ is greater than $\alpha$, the cache partitions will have to be shared among cores. If $n$ is less than or equal to $\alpha$, $|\mathbf{\Pi}|$ would be small (e.g., if $\alpha = 16$ and $n = 12$, $|\mathbf{\Pi}| = 1{,}365$). Therefore, this calculation can be done in constant or linear time for most of the scenarios.

## 6 EXPERIMENTS

### 6.1 Experimental Setup

We used a cycle-accurate full-system simulator, gem5 [5], and a "GARNET2.0" [1] NoC model that is integrated with gem5, to model the multi-core architecture. Our goal was to model a realistic architecture that included reconfigurable L1, L2 caches, and an accurate NoC traffic model. Our previous work modeled the KNL architecture on the gem5 simulator by modifying the default

**ALGORITHM 2:** Selection of optimal cache configurations

```
/* 1st step: Building profile table (Section 5.3) */
Run Algorithm 1
/* 2nd step: Optimize on each core (Section 5.4) */
for $k = 1$ to $n$ do
    for $w_k = 1$ to $\alpha - 1$ do
        for $\epsilon = \epsilon_k^{min}(w_k)$ to $\epsilon_k^{max}(w_k)$ do
            for $c_I, c_D \in \mathbb{C}$ do
                if $\epsilon_{k,1}(c_I, c_D, w_k) == \epsilon$ then
                    if $t_{k,1}(c_I, c_D, w_k) < T[1][\epsilon]$ then
                        $T[1][\epsilon] = t_{k,1}(c_I, c_D, w_k)$
                    end
                end
            end
        end
        for $i = 2$ to $\delta_k$ do
            for $\epsilon = \epsilon_k^{min}(w_k)$ to $\epsilon_k^{max}(w_k)$ do
                for $c_I, c_D \in \mathbb{C}$ do
                    $\epsilon' = \epsilon - \epsilon_{k,i}(c_I, c_D, w_k)$
                    if $T[i-1][\epsilon'] + t_{k,i}(c_I, c_D, w_k) < T[i][\epsilon]$
                    then
                        $T[i][\epsilon] = T[i-1][\epsilon'] + t_{k,i}(c_I, c_D, w_k)$
                    end
                end
            end
        end
        $E_k^*(w_k) = \min\{\epsilon_k \mid T[\delta_k][\epsilon_k] \leq D\}$
    end
end
/* 3rd step: Optimize across all cores (Section 5.5) */
for all $\Pi_j = \{w_1, w_2, \ldots, w_n\} \in \Pi$ do
    $E_j^* = \sum_{k=1}^{n} E_k^*(w_k)$
    $E^* = min(E^*, E_j^*)$
end
return $E^*$
```

gem5 source to capture the behavior of Memory and Cluster modes in KNL [8]. There are other multi-core architectures such as Tilera TILE64 [3] and Kalray's MPPA-256 [10], which allow for predictable data transfers and composition of memory accesses. However, we did our experiments on the gem5 KNL model since we had validated the gem5 simulator model with results from real hardware (Intel Xeon Phi 7210 hardware board) in our previous work [8].

However, the full KNL implementation is quite complex to be modeled on gem5. For example, gem5 does not support tiles with two cores. Hence, there is one core in each tile in our experiments. This mimics the scenario where one core in each tile is switched off in the hardware board. Furthermore, KNL runs AVX512 instructions, whereas our gem5 KNL model runs ×86 instructions. Cache sizes were chosen such that the applications we used get a realistic hit percentage of around 95% in the L1 cache. If we used a larger cache size, the L1 hit rate would be 100%, and any discussion about cache reconfiguration will be meaningless. Modeling the entire KNL architecture in a simulator is beyond the scope of this article. Our goal was to model a realistic NoC traffic model. Even though the absolute values are not the same, the relative advantages/disadvantages of reconfiguration are accurately captured as shown in our previous work as well [8]. The core contributions of the article—cache reconfiguration mechanism and machine learning–based exploration space reduction—remain intact irrespective of the architecture.

The complete set of simulation parameters is summarized in Table 1. Power results were obtained by feeding the gem5 output statistics to the McPAT power modeling framework [20].

Table 1. System Configuration Parameters

| Processor Configuration | |
|---|---|
| Number of cores | 32 |
| Core frequency | 1.4GHz |
| Instruction set architecture | ×86 |
| **Memory System Configuration** | |
| L1 Cache | Private, reconfigurable, separate instruction, and data cache. Each 32kB in size. |
| L2 Cache | Reconfigurable, shared cache. 512kB in size. |
| Cache coherence | MESI two-level directory-based cache coherence protocol |
| Memory size | 4GB DDR |
| **Interconnection Network Configuration** | |
| Topology | $8 \times 4$ Mesh |
| Routing scheme | X-Y deterministic |
| Router | 4 port, 4 input buffer router with 3-cycle pipeline delay |
| Link latency | 1 cycle |

Table 2. Application Sets from the MiBench [11] and SPLASH-2 [36] Benchmarks

| **Application set** | **Cores 1, 2, 3, 4** | **Cores 5, 6, 7, 8** | **Core 9, 10, 11, 12** | **Core 13, 14, 15, 16** |
|---|---|---|---|---|
| **Set 1** | stringsearch, sha | FFT, Barnes | stringsearch, Lu | sha, FFT |
| **Set 2** | crc, Barnes | Radix, Lu | Cholesky, sha | qsort, FMM |
| **Set 3** | bitcnts, stringsearch | FMM, Water-Nsquared | Barnes, Lu | Cholesky, sha |
| **Set 4** | Radix, Lu, FFT | crc,sha,stringsearch | qsort, Barnes, Water-Nsquared | bitcnts, FMM, stringsearch |
| **Set 5** | patricia, Water-Nsquared, Barnes | dijkstra, bitcnts, qsort | Cholesky, Radix, crc | patricia, sha, Lu |
| **Set 6** | Lu, stringsearch, FFT, patricia | Water-Nsquared, FMM, qsort, dijkstra | FFT, Cholesky, dijkstra, sha | crc, Radix, qsort, bitcnts |

To evaluate the effectiveness of our approach, we use 14 benchmarks selected from MiBench [11]—*bitcnts, crc, dijkstra, patricia, qsort, sha, stringsearch*—and SPLASH-2 [36]—*FFT, Radix, Lu, FMM, Cholesky, Water-Nsquared, Barnes* benchmark suites. In order to make the size of MiBench benchmarks comparable with SPLASH-2, we use reduced (but well verified) input sets. Table 2 lists the application sets which are combinations of the selected benchmarks used in our experiments. We choose three application sets where each core contains two benchmarks, two application sets where each core contains three benchmarks, and one application set where each core contains four benchmarks.

Out of the 32 cores in the $8 \times 4$ mesh interconnect, we chose 16 cores just for experimental purposes. Simulation time can be prohibitive with a larger number of cores. This is evident from the results in Figure 17. Besides being practical, this core configuration enables us to mimic that about 50% of the cores from a chip will be typically utilized at a given time. As mentioned in Section 5.1, mapping of applications to cores is beyond the scope of this article and the application mapping is assumed to be given as an input. The task mapping problem in a soft real-time system has been studied before [22]. For our experimental results, to get the application mapping, we ran each application with the base cache configuration and grouped them so that the total execution

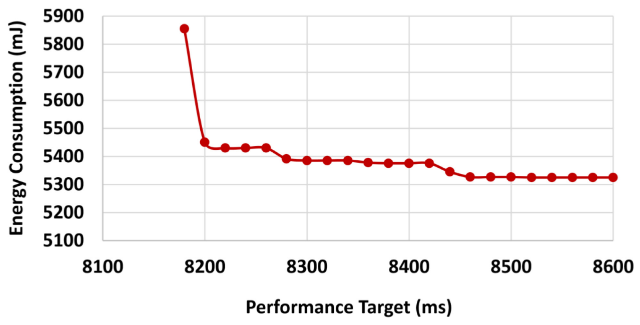


Fig. 13. Energy consumption variation with performance target. A relaxed target leads to more energy savings.

time for each set of applications is comparable. We cannot get the execution time to be exactly the same since it depends on cache configuration. However, the intent is to have a fair comparison and therefore not pair up a task with quick execution time with a task with very long execution time. This was done to make sure behaviors such as NoC congestion will be captured throughout the application runtime. Otherwise, if most cores finish their tasks and only some are running, the experiments will give results that do not capture traffic congestion in NoC. If a task is parallelized, it can be viewed as multiple tasks and mapping can be performed (a smart mapping algorithm is likely to map them to the cores in a cluster). The performance target $D$ is set in a way that there is a feasible L1 cache assignment for every partition factor in every core. In other words, all possible L2 partition schemes can be used.

## 6.2 Performance Target Selection

Equation (3) gives the performance target as a measure of providing the expected QoS for the application sets. Figure 13 shows the optimal energy consumption variation as a function of the performance target $D$ for application set 1 in Table 2. We swept the target from 8,100ms to 8,600ms. When the target is shorter than 8,180ms, there was no feasible solution. As the target was increased, it converged to the optimal which was 5,324mJ. Therefore, it is clear that the performance target will affect the best possible energy calculated by our approach. In our experiments, we selected the target such that each core can converge to an optimal energy under the base configuration. In this example, the selected target was 8,400ms.

## 6.3 Accuracy of Profile Tables Built with Machine Learning

Section 5.3 describes our machine learning algorithm for building the profile table with less number of simulations. Figure 14 shows the total amount of training data required for all benchmarks over different error thresholds. Here, training data is expressed as a percentage value. This is the ratio of profile table entries filled using simulations and total number of profile table entries, for all 14 benchmarks. Error threshold is expressed using NRMSE. As expected, the more training data we use, the lower the error threshold is. Error is less than 6% even with only 10% of training data. This essentially means approximately an order-of-magnitude speedup in profile table generation time compared to exhaustive simulations. Figure 15 provides the training data requirement for each benchmark. We can see that some benchmarks require a lot of training data for accurate predictions (e.g., Barnes, Lu, dijkstra), while some benchmarks need much less (e.g., bitcnts, sha). This observation forms the basis of using error threshold instead of fixing training data percentage.

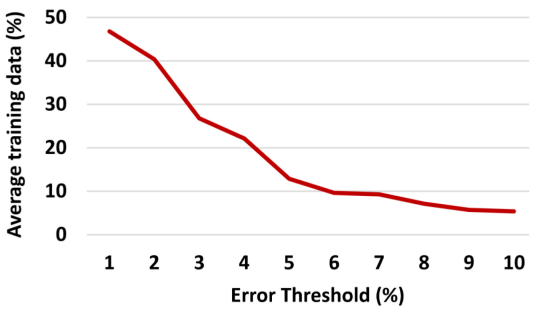


Fig. 14. Average training data required with varying error threshold for all benchmarks. For example, by using 12.86% training data, we can achieve results within 5% error margin.

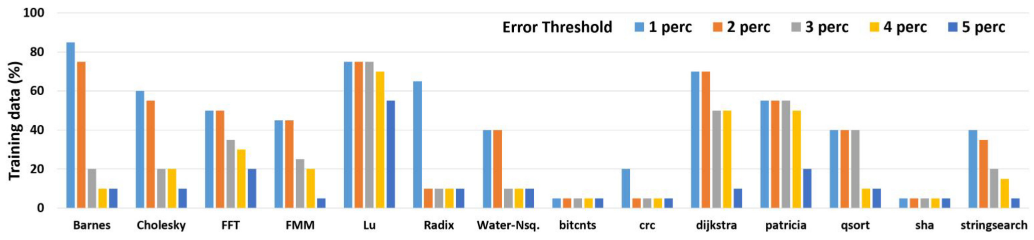


Fig. 15. Required training data for different error thresholds. Less than 6% error is observed with as little as 10% data.

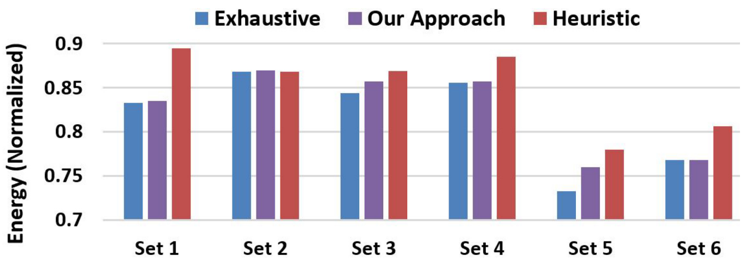


Fig. 16. Energy consumption observed compared to the Base cache configuration across all profile table generation techniques. The goal is to get results similar to the exhaustive approach.

The error threshold–based approach allows more simulation time for benchmarks that require more training data to be accurate.

To evaluate our approach, we compare results of the predicted profile tables with profile tables obtained from exhaustive and heuristic-based approaches. Figure 16 compares the accuracy of the following three strategies that can be used to build the profile table by showing the optimal energy consumption:

— **Exhaustive:** All cache configurations explored exhaustively. This acts as the reference which gives the global optimal solution.
— **Our approach (ML):** Configurations selected keeping the error threshold at 5% as training data and Algorithm 1 used for predicting.
— **Heuristic [32]:** Cache configurations selected using ICT heuristic (configurations where DL1 and IL1 are the same) proposed by Wang et al.

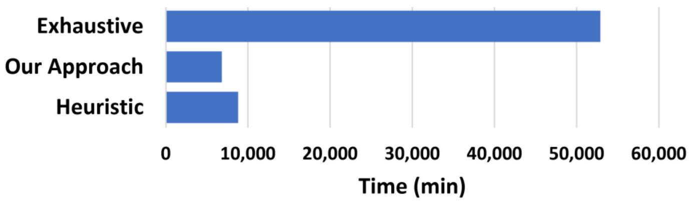


Fig. 17. Times taken for different static profiling approaches. Our approach takes the least time.

Algorithm 2 uses these profile tables to get the optimum cache configurations while maintaining QoS standards. Those values are then normalized to the energy consumption of the base cache configuration. As discussed in Section 6.1, our reconfigurable L1 cache has a base size of 32kB and a shared 512kB L2 cache. We observe in Figure 16 that compared to the base configuration, the profile table built with the exhaustive approach can achieve 18.49% energy savings on average across all application sets. The energy savings provided by our approach (ML) is very accurate (less than 1% error). It achieves the highest performance in set 1 with an error of only 0.2%, whereas heuristic gives an error of 6.9%. The heuristic-based approach gives relatively worse results as it populates only a portion of the profile table based on ICT. Therefore, it is likely that the optimum configuration found by exhaustive exploration is not in the profile table at all. In contrast, our approach uses the training data to predict the whole table and therefore, depending on the accuracy of the prediction, converges closer to the optimal.

The runtimes of these approaches are shown in Figure 17. It takes 36.73 days to complete the exhaustive exploration using the modified gem5 simulator on an unparallelized setup. Heuristic selects 16.67% of the total exploration space and hence, shows a speedup of six times. In contrast, our approach selects only 12.86% of the total exploration space as training data and takes 5 minutes to tune the neural network model, which results in an effective speedup of 7.76 times. The time taken to train the ML algorithm is negligible compared to the profiling time. Therefore, we obtain higher accuracy while consuming less time compared to the heuristic approach.

Our machine learning approach populates the whole profile table unlike the heuristic method which populates less than 10% of it. This allows us to estimate energy consumption for all possible cache configurations, and as a result, leads to better energy savings. Specifically, our approach provides up to 7% (3% on average) improvement in energy efficiency while being 1.29 times faster compared to the heuristic method. Most importantly, our approach is 7.76 times faster compared to the exhaustive method and still provides close to optimal energy savings (on average deviation of 0.05%).

## 7 CONCLUSIONS

This article explores cache configuration optimization in an NoC-based many-core architecture and addresses the issue of large exploration space in DCR. We significantly reduce the exhaustive exploration time by employing a machine learning–based approach. This approach selects only a few configurations and trains a neural network model which then predicts the rest of the profile table. The proposed DCR algorithm then finds the optimum L1 configuration for each core and optimum partition factor for the shared L2 cache. The results show an average energy savings of 18.49% compared to the base configuration across all application sets. The ML predictions gave accurate (less than 1% error) energy savings compared to the exhaustive method with a 7.76 times speedup which proves to be better than previously proposed heuristic-based methods both in terms of accuracy and speedup. Overall, our approach provides an order-of-magnitude reduction in exploration effort with negligible impact on accuracy.

## ACKNOWLEDGMENT

We would like to acknowledge the contribution of Nikhil Venkatesh for his help in generating results.

## REFERENCES

[1] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)* IEEE, 33–42.

[2] Alif Ahmed, Yuanwen Huang, and Prabhat Mishra. 2019. Cache reconfiguration using machine learning for vulnerability-aware energy optimization. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 2 (2019), 15.

[3] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, et al. 2008. Tile64-processor: A 64-core SoC with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference-Digest of Technical Papers*. IEEE, 88–598.

[4] Shubha Bhat. 2005. Energy Models for Network-on-Chip Components. *Master of Science, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, Eindhoven.*

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[6] Subodha Charles, Hadi Hajimiri, and Prabhat Mishra. 2018. Proactive thermal management using memory-based computing in multicore architectures. In *2018 9th International Green and Sustainable Computing Conference (IGSC'18)*. IEEE, 1–8.

[7] Subodha Charles, Yangdi Lyu, and Prabhat Mishra. 2019. Real-time detection and localization of DoS attacks in NoC based SoCs. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE'19)*. IEEE, 1160–1165.

[8] Subodha Charles, Chetan Arvind Patil, Umit Y. Ogras, and Prabhat Mishra. 2018. Exploration of memory and cluster modes in directory-based many-core CMPs. In *2018 12th IEEE/ACM International Symposium on Networks-on-Chip (NOCS'18)*. IEEE, 1–8.

[9] Chao Chen, José L. Abellán, and Ajay Joshi. 2015. Managing laser power in silicon-photonic NoC through cache and NoC reconfiguration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 34, 6 (2015), 972–985.

[10] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. 2013. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science* 18 (2013), 1654–1663.

[11] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization (WWC-4'01)*. IEEE, 3–14.

[12] Hadi Hajimiri, Kamran Rahmani, and Prabhat Mishra. 2012. Compression-aware dynamic cache reconfiguration for embedded systems. *Sustainable Computing: Informatics and Systems* 2, 2 (2012), 71–80.

[13] Po-Yang Hsu and TingTing Hwang. 2013. Thread-criticality aware dynamic cache reconfiguration in multi-core system. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'13)*. IEEE, 413–420.

[14] Yuanwen Huang and Prabhat Mishra. 2016. Reliability and energy-aware cache reconfiguration for embedded systems. In *2016 17th International Symposium on Quality Electronic Design (ISQED'16)*. IEEE, 313–318.

[15] Yuanwen Huang and Prabhat Mishra. 2017. Vulnerability-aware energy optimization using reconfigurable caches in multicore systems. In *IEEE International Conference on Computer Design (ICCD'17)*. IEEE, 241–248.

[16] Intel. 2016. Intel Xeon Phi Processor 7210. Retrieved on February 2, 2017 from https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors/7210.html.

[17] ITRS. 2015. International Technology Roadmap for Semiconductors (ITRS). Retrieved on June 23, 2018 from http://www.semiconductors.org/main/2015_international_technology_roadmap_for_semiconductors_itrs.

[18] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy K. John. 2009. Bank-aware dynamic cache partitioning for multicore architectures. In *International Conference on Parallel Processing (ICPP'09)*. IEEE, 18–25.

[19] Jae W. Lee, Man Cheuk Ng, and Krste Asanovic. 2008. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *ACM SIGARCH Computer Architecture News*, Vol. 36. IEEE Computer Society, 89–100.

[20] 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 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 469–480.

[21] Afzal Malik, Bill Moyer, and Dan Cermak. 2000. A low power unified cache architecture providing power and performance flexibility (poster session). In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED'00)*. ACM, 241–243.

[22] Sorin Manolache, Petru Eles, and Zebo Peng. 2008. Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 2 (2008), 19.

[23] Asit K. Mishra. 2011. *Design and Analysis of Heterogeneous Networks for Chip-multiprocessors*. The Pennsylvania State University.

[24] Mehdi Modarressi, Shaahin Hessabi, and Maziar Goudarzi. 2006. A reconfigurable cache architecture for object-oriented embedded systems. In *Canadian Conference on Electrical and Computer Engineering (CCECE'06)*. IEEE, 959–962.

[25] Umit Y. Ogras and Radu Marculescu. 2013. *Modeling, Analysis and Optimization of Network-on-Chip Communication Architectures*. Vol. 184. Springer Science & Business Media.

[26] Umit Y. Ogras, Radu Marculescu, Diana Marculescu, and Eun Gu Jung. 2009. Design and management of voltage-frequency island partitioned networks-on-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 3 (2009), 330–341.

[27] Rakesh Reddy and Peter Petrov. 2010. Cache partitioning for energy-efficient and interference-free embedded multitasking. *ACM Transactions on Embedded Computing Systems (TECS)* 9, 3 (2010), 16.

[28] Matthew Schuchhardt, Abhishek Das, Nikos Hardavellas, Gokhan Memik, and Alok Choudhary. 2013. The impact of dynamic directories on multicore interconnects. *Computer* 46, 10 (2013), 32–39.

[29] Alex Settle, Dan Connors, Enric Gibert, and Antonio González. 2006. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing* 2, 2 (2006), 221–233.

[30] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights landing: Second-generation intel xeon phi product. *IEEE MICRO* 36, 2 (2016), 34–46.

[31] Ching-Long Su and Alvin M. Despain. 1995. Cache designs for energy efficiency. In *Proceedings of the 28th Hawaii International Conference on System Sciences, 1995*, Vol. 1. IEEE, 306–315.

[32] Weixun Wang and Prabhat Mishra. 2009. Dynamic reconfiguration of two-level caches in soft real-time embedded systems. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI'09)*. IEEE, 145–150.

[33] Weixun Wang and Prabhat Mishra. 2011. System-wide leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in multitasking systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20, 5 (2011), 902–910.

[34] Weixun Wang, Prabhat Mishra, and Ann Gordon-Ross. 2012. Dynamic cache reconfiguration for soft real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)* 11, 2 (2012), 28.

[35] Weixun Wang, Prabhat Mishra, and Sanjay Ranka. 2011. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *48th ACM/EDAC/IEEE Design Automation Conference (DAC'11)*. IEEE, 948–953.

[36] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, Vol. 23. ACM, 24–36.

[37] Thomas Canhao Xu, Pasi Liljeberg, and Hannu Tenhunen. 2011. Optimal memory controller placement for chip multiprocessor. In *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 217–226.

[38] Terry Tao Ye, Giovanni De Micheli, and Luca Benini. 2002. Analysis of power consumption on switch fabrics in network routers. In *Proceedings of the 39th Annual Design Automation Conference (DAC'02)*. ACM, 524–529.

[39] Chuanjun Zhang, Frank Vahid, and Roman Lysecky. 2004. A self-tuning cache architecture for embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)* 3, 2 (2004), 407–425.

[40] Chuanjun Zhang, Frank Vahid, and Walid Najjar. 2005. A highly configurable cache for low energy embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)* 4, 2 (2005), 363–387.

[41] Ujjwal Gupta, Chetan Arvind Patil, Ganapati Bhat, Prabhat Mishra, and Umit Y. Ogras. 2017. Dypo: Dynamic pareto-optimal configuration selection for heterogeneous mpsocs. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 123.

Received January 2019; revised July 2019; accepted July 2019