Toward On-chip Network Security Using Runtime Isolation Mapping

MOHAMMAD SADEGH SADEGHI, SIAVASH BAYAT SARMADI, and
SHAAHIN HESSABI, Sharif University of Technology, Iran

Many-cores execute a large number of diverse applications concurrently. Inter-application interference can lead to a security threat as timing channel attack in the on-chip network. A non-interference communication in the shared on-chip network is a dominant necessity for secure many-core platforms to leverage the concepts of the cloud and embedded system-on-chip. The current non-interference techniques are limited to static scheduling and need router modification at micro-architecture level. Mapping of applications can effectively determine the interference among applications in on-chip network. In this work, we explore non-interference approaches through run-time mapping at software and application level. We map the same group of applications in isolated domain(s) to meet non-interference flows. Through run-time mapping, we can maximize utilization of the system without leaking information. The proposed run-time mapping policy requires no router modification in contrast to the best known competing schemes, and the performance degradation is, on average, 16% compared to the state-of-the-art baselines.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; *Robotics*; • **Networks** → Network reliability;

Additional Key Words and Phrases: On-chip interconnect, runtime mapping, non-interference, timing side-channel

ACM Reference format:

Mohammad Sadegh Sadeghi, Siavash Bayat Sarmadi, and Shaahin Hessabi. 2019. Toward On-chip Network Security Using Runtime Isolation Mapping. *ACM Trans. Archit. Code Optim.* 16, 3, Article 28 (July 2019), 25 pages.

<https://doi.org/10.1145/3337770>

1 INTRODUCTION

In the current many-core era, it is a challenging task to eliminate on-chip interference, which is caused by contention among co-scheduled applications in the critical shared resources, such as on-chip network (NoC). Non-interference among applications in NoC is a method to eliminate the security threat of timing channel attack in many-core platforms [33, 41, 44]. This threat stems from the latencies that are experienced by the malicious applications, which can divulge information when the victim applications are accessing shared resources. The goal of non-interference is to isolate traffic flows, thereby preventing information leakage to malicious applications, intentionally

Authors' addresses: M. S. Sadeghi, S. B. Sarmadi, and S. Hessabi are with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran; emails: mssadeghi@ce.sharif.edu, {sbayat, hessabi}@sharif.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/07-ART28

<https://doi.org/10.1145/3337770>

Fig. 1. Structure of a two-dimensional 8x8 NoC-based many-core with eight memory controllers (MCs).

(covert channel) or unintentionally (side channel) [41]. Ristenpart et al. [34] show such possible cache-based side-channel attack on Amazon EC2 hardware to reveal users' passwords.

Where and how an application is mapped in a NoC-based many-core can significantly eliminate interference between applications. First, distance from memory controllers and also the location of an application relative to others affect interference among applications. Second, regular shape and contiguous mapping can eliminate contention among applications using minimal routing algorithms while irregular shape needs more complex routing algorithms and hardware mechanisms to confine flows among applications. Hence, runtime mapping policy can have a significant impact on timing channel, per-application performance and system performance as we will demonstrate in this work. While prior research (e.g., [17, 23]) tackled the problem of how to map an application, strict non-interference behavior between applications in the NoC is less well understood. Although operating systems have responsibility for mapping applications to cores, their methods are oblivious to inter-application interference in NoC-based many-cores [4, 7, 8, 45].

Prior work provided hardware support for non-interference in shared hardware resources, e.g., caches [28, 29, 42, 43], memory controllers [19, 20, 22, 37, 40], and on-chip networks [33, 41, 44]. These mechanisms can be used as building blocks by a many-core operating system. However, all proposals were evaluated without consideration for resource management at software level.

Current NoC-based many-cores are based on NUMA systems. Figure 1 shows the logical layout of a many-core processor with cores organized in a mesh-based structure such as, Intel SCC [25], Kalray MPPA manycore [2], Adapteva Epiphany [1], or the 64-core Tilera processor [35]. Based on these many-core platforms, we assume that the memory controllers are located in corners of our baseline tiled many-core architecture. Tiles consist of a core, L1 cache and L2 cache bank. Applications in dynamic workloads arrive and leave the system at runtime with unpredictable nature. Task graph model [17, 18, 23] of applications is dispatched on cores. Context-switching overhead in many-cores is reduced by operating one task per core. In this model, the task is a single function/portion of code that communicates with other tasks.

Trivino et al. [39] proposed a NoC virtualization technique that isolates the application through dynamic configuration bits of switches at the router. Yet, despite isolation mechanism, they confine only packets from boundaries of each application and do not propose any idea regarding the process of runtime mapping and resource allocation, relying on a resource manager operating under OS control.

In this article, we explore runtime mapping policies to provide non-interference. The goal is an efficient policy that uses mapping to place applications of each domain contiguously such that the

system utilization is improved. Our mapping implementation explores two approaches: L-shape isolated (Liso) mapping and Isolated mapping.

L-shape isolated (Liso) mapping. How should an isolated domain be progressed gradually to maintain security in an unsized rectangle? We demonstrate that the unsized rectangular shape can guarantee strict timing channel across isolated domain. In essence, unsized rectangular shape removes the dependency of requested allocation size on confidential data. In other words, it might be possible for an application to discover that there is another application, but there is no ability to observe runtime changes in other domains. We show this by comparing throughput of two domains under insecure and secure scenarios.

Isolated mapping. What shape should be considered during isolated runtime mapping? We show that the most appropriate shapes to gain higher system throughput are those that support rectangle shapes to significantly mitigate the complexity by means of minimal routing algorithm and no hardware overhead.

We make the following contributions in this article:

- We propose a new runtime mapping algorithm to eliminate interference between security domains. Non-interference runtime mapping is motivated by the combination of proposing a software level approach to eliminate timing channel attack and critical runtime resource management process in many-core systems.
- We demonstrate that runtime mapping algorithm can eliminate timing channel by separating applications into security domains. We show that regional progress of each security domain has a significant impact on non-interference performance.
- We extensively evaluate the proposed mapping policies on different network sizes using a suite of diverse applications. We evaluate the overhead of the Liso approach and show that Liso results in 8% and 16% performance overhead compared to two baseline approaches on average. Moreover, our approaches have no additional hardware overhead compared to the best known competing schemes.

The remainder of this article is organized as follows. In Section 2, we discuss related work. Section 3 describes the motivation and security model. In Section 4, we describe the target architecture and hardware assumptions. Our schemes are explained in Section 5. We present methodology and simulation results of our schemes in Sections 6 and 7, respectively. Finally, Section 8 concludes the article.

2 RELATED WORK

To discuss related work, we separate all non-interference techniques into two categories: software-level non-interference on-chip interconnect, which can be improved by runtime mapping, and hardware (micro-architecture)-level non-interference on-chip interconnect, which is not targeted by runtime mapping. Prior work has followed two main approaches to eliminate the interference observed by on-chip interconnect: static TDM scheduling at the VC level, and NoC virtualization.

Static Scheduling. Static TDM scheduling approaches [33, 41, 44] work on a predetermined static schedule and requires hardware overhead. These techniques get a static share of the network's bandwidth, irrespective of the traffic flow in other domains, and are consequently bandwidth inefficient. Wang et al. [41] proposed reversed priority with static limit (RPSL) approach, which provides one-way information leak protection. Their approach uses priority-based arbitration and static virtual circuit (VC) allocation. To prevent DoS attack, the RPSL approach needs an additional mechanism to statically limit the use of each port by particular domains. Wassel et al. [44] proposed SurfNoC, an on-chip network that reduces the latency occurred by temporal partitioning. SurfNoC modifies network router design to provide packet switching, and relies on static

virtual channel scheduling. SurfNoC with crossbar input acceleration can improve performance dramatically. However, this performance improvement is accompanied with a high hardware cost. High buffering requirements of TDM-based NoCs lead to an expensive approach in providing non-interference. Psarras et al. [33] consider application domains as separate virtual networks. Each domain uses a static share of the network's bandwidth. It needs buffers per application domain to sufficient performance.

NoC Virtualization. Prior work has also considered NoC virtualization. Trivino et al. [39] propose isolation to the NoC that includes reconfiguration mechanism of the Logic-Based Distributed Routing technique to allow dynamic configuration bits of switches at the router. This technique is required for irregular-shaped isolation that needs more complex routing algorithm. Also in this approach, no information is provided about resource management requirements such as location and shape of partitions, which are critical to eliminate inter-application interference created by dynamic multithreaded workloads. In this approach, a crucial role of runtime mapping, which is the main contribution of our work, is ignored.

Runtime Mapping. Similar to prior work, we focus on mapping application to core in many-core systems. This work focuses on many-cores in cloud and embedded systems, wherein dynamic workloads including multithreaded applications arrive, execute, and terminate continuously in unpredictable manner. We consider runtime application mapping as a crucial step in resource management to eliminate inter-application interference as opposed to reactive steps like task migration [31, 32]. Each application is modeled as a task graph, where tasks are individual computational blocks that communicate with each other [26]. In prior work [17, 23], runtime mapping is done as contiguous mapping without considering isolated rectangle mapping. Therefore, there will be some inter-application interference. Furthermore, these approaches cannot ensure complete non-interference in the system due to the lack of considerations for cache directories and DRAM controllers. Thus, contiguous mapping only improves performance of the system without ensuring non-interference. SHiC [17] is a contiguous mapping that uses smart stochastic hill climbing algorithm to map applications contiguously. MapPro [23] is a contiguous mapping by enforcing near convex shape for mapping applications to minimize dispersion and external congestion. Das et al. [12] propose an application-to-core mapping policy that reduces inter-application interference in the on-chip interconnect and memory allocations. However, they focus on multiprogrammed workloads and map each application on a core. Dey et al. [15] propose a dynamic thread-to-core mapping for bus systems to mitigate contention for the shared resources in the memory hierarchy.

In this article, we propose runtime mapping to provide non-interference and eliminate timing channel attack. In this work, the location and shape of the region for an application is automatically extracted by run-time mapping to place it closer to memory and isolated from others. This allows run-time mapping to reduce access latency and eliminate interference for all secure applications. Our proposed run-time mapping mechanism can be used in conjunction with mechanisms that eliminate other timing channels, such as memory and cache. Our approach can be used for both embedded and cloud systems with emphasis on performance and timing channel attack.

3 MOTIVATION AND SECURITY MODEL

In this section, we present our main idea through an example and some definitions, discuss the security model, and state the problem that we address in this article.

3.1 Motivation

We illustrate our proposed mapping process to eliminate a timing side channel protection using four applications. It should be noted that our main goal in this article is security achieved by

Fig. 2. Two mapping approaches.

non-interference; so we propose different approaches in this area. Before showing an example of our approaches, some terminologies need to be introduced.

Let APP (N, S, SD) characterize an application where N, S, and SD are application name, size, and security domain, respectively. Also, “P” and “CM” symbols in Figure 2 are defined as “penalty node” and “control manager,” respectively. Penalty node is an unused core in a security region during mapping process. Control manager is a master node. The numbers along with each task of an application show the pattern of allocating cores based on L-shape algorithm.

As shown in the isolated mapping in Figure 2(b), whenever an application enters the system, its objective is to find the smallest square on the platform such that all communication packets of the application are confined to their isolated region, and minimum fragmentation occurs after the mapping. In this way, as defined in Reference [10], minimum external network contention occurs. External contention happens when two flows from different applications contend for the same links. Lowering the external contention provides the benefit of decreasing the additional communication costs and execution time of the applications. However, the drawback of this approach is that with this contiguity constraint, the achievable throughput is degraded as a result of the increased turnaround time [18]. More details can be found in the example discussed in Figure 2. Figure 2(a) shows the characteristics of applications (Applications 1 through 4). More precisely, each application contains a number of tasks that will acquire their resources. Figures 2(b) and 2(c) show the result of application mapping after a specific time through different approaches. These two different approaches are as follows.

- Approach 1: Through rectangle mapping, this approach attempts to isolate each application to minimize external contention and communication cost. In other words, it provides security per individual applications.
- Approach 2: Through L-shape mapping, this approach attempts to make an isolated and rectangle region for a group of applications, which have the same security domain to

Fig. 3. Normalized average turnaround time of applications for different workload sizes.

Fig. 4. Privilege levels and protection rings [38].

maximize throughput. In other words, it provides security per domain (a set of applications that do not need to be isolated from each other).

In this example, we assume that the XY-routing algorithm is used for communication. For Approach 1, the external contention is minimized for the mapping of three applications. However the fourth application with size of 13 does not fit for mapping even though enough resources exist. In other words, throughput and utilization of the system have decreased.

Our motivation for applying Approach 2 is to gain more throughput. This approach fits more applications on the system, and subsequently, decreases the turnaround time (total time between submission of an application and its completion) of each application. We have analyzed the turnaround time of a set of workloads for a network size of 100 nodes. The result is shown as a histogram in Figure 3. It can be seen that non-isolated approach (SHiC) [17] has the lowest turnaround time; therefore, we use SHiC as a baseline. Approach 2 has an improvement of about 60% compared to Approach 1. The basic idea is that by considering different security domains, we can accumulate applications with the same security domain, and therefore improve the system utilization and throughput. In other words, utilization is improved by filling fragmented resources for each recently entered application for a specific security domain.

3.2 Security Model

We use hierarchical “ring” protection model [36] to describe our security model. Figure 4 shows hierarchical privilege levels based on ring model. CM is placed in ring -1 . Rings 0, 1, and 2 are used for subsequent privileged levels (operating system (OS) and middleware). Ring 3 is considered for the least privileged software applications. Customer applications of the mutually distrusting reside at the least privileged level (ring 3). At this level, we define domains for a group of applications that do not need to be isolated from each other, and we prevent information flow between domains. We can apply hierarchical trust model, such as lattice [14], at the least privileged level to permit one-way information flow from low-security to high-security domains.

Information leakage is not possible for an application due to the unpredictable size of mapped region at runtime. This is because the region of mapping is not required to be in full size; i.e., all cores in region need not to be occupied. This way, it does not leak any information dependent on confidential data. We show security evaluation in Section 7.1. To protect against DoS attack, our approach considers a limit on the resources that are allocated to each security domain. Moreover, the limit can change over time, based on some feedback policies while it does not reflect sensitive

information or demands. For example, for two security domains (A and B), if security domain “A” demands many requests, then we can set a low value such as 20% of total cores to “B” security domain. In this way, security domain “A” can get at least 80% of the cores while preserving the security.

This security model includes timing channel protection but does not cover physical attacks such as EM emission, temperature, and so on. We assume that the NoC can be trusted. We also assume that all requests and responses through the NoC have been routed correctly. Moreover, we trust peripheral devices and memories. We also assume that CM is completely isolated and cannot be compromised.

4 TARGET ARCHITECTURE AND HARDWARE ASSUMPTIONS

In this section, we first describe supported NoC topologies with the aid of isolated security domain fact. In addition, we show that our isolated security domains can be secured against external attacks through LBDR approach. Then, we discuss memory configurations and cache hierarchy. Finally, we demonstrate an overall view of the isolated system.

4.1 Direct and Indirect Network Topologies

Direct network topologies with regular physical arrangement is well matched to packaging constraints [11]. In strictly orthogonal topologies, such as k-ary n-cube nodes can be numbered by using their coordinates in an n-dimensional space. Routing can be easily implemented by selecting a link that decrements the absolute value of the offset in some dimensions [11]. Physically minimal paths in torus and mesh networks allow these networks to exploit physical locality between communicating nodes. However, indirect network topologies (k-ary n-flies) such as butterfly, clos, and fat tree are unable to exploit such locality [11]. Tree is an informal and popular direct network topology. One of the most interesting properties of tree is that, for any connected graph, it is possible to define a tree that spans the complete graph. As a consequence, for any connected network, it is possible to build an acyclic network connecting all nodes by removing some links. This property can be used to define a routing algorithm for any irregular topology [16].

Our main idea for non-interference communication flows in direct network topologies [11] is based on the following fact. An isolated partition in an on-chip network is chosen based on the main topology such that the base aspects of a network (such as its routing, flow control and microarchitecture) are maintained. Moreover, design-time metrics (such as degree and diameter) in the subgraph are maintained for regular networks.

4.1.1 Argument. We assume a partition shape has the same topological shape as the network topology. For example, square and rectangle shapes can be used for a 2D mesh, or a sub-tree as a tree topology.

- **Definitions**

- *Definition1:* A partition covers a set of nodes in an n-dimensional network where packets can take any available path/channel inside the partition.
- *Definition2:* Two nodes have a bidirectional path, if both directions exist based on routing algorithm inside a partition.
- *Definition3:* A partition is defined as a *complete* partition if there is at least one bidirectional path between any two nodes inside the partition.

- **Fact:** A security domain (SD) is isolated if it includes at least one complete partition in a direct network topology.
- **Justification:** The network topology is usually a graph (i.e., either a mesh or a tree). An SD is a subgraph of such network and clearly inherits its aspects and design metrics. Since

Fig. 5. Isolated security domains for tree and 2D mesh topologies.

Fig. 6. Hardware specification.

there are bidirectional paths between all nodes in the original network, all nodes in the subgraph must also have such paths and therefore it is a complete partition. In Figure 5, some examples are shown for tree and 2D mesh topologies.

4.2 LBDR

Our proposed isolated security domain intrinsically eliminates traffic interference among applications. However, to confine all traffic internally and prohibit entering external traffics into the isolated region, we use logic-based switching mechanism to strictly prevent unwanted malicious behaviors. In all supported NoC topologies, there is the same property that the end nodes can communicate with the rest of nodes through a minimal path defined in the original mesh topology. Logic-based switching mechanisms such as LBDR [21] can be applied in all topologies that fulfill this property. Regarding the connectivity bits, each output port has a bit, referred to as C_x , indicating whether a switch is connected through the x port to its neighbor. Thus, connectivity bits are C_n , C_e , C_w , and C_s . In addition, LBDR eliminates the need for additional hardware to recover from pathological scenarios such as deadlock. This is a great advantage in terms of silicon cost (and power), and limits the impact of reconfiguration on performance. LBDR allows uninterrupted operation even during reconfiguration. This choice reduces the signaling between NoC switches and control manager, since configuration of the routing logic depends on just a few configuration bits. More details of this mechanism can be found in Reference [21]. Logic-based switching mechanism relies on the use of some connectivity bits. As shown in Figure 6(a), all security domains are configured such that all their border routers block incoming and outgoing packets through specific settings of the connectivity bits. To this end, when a security domain is mapped, connectivity bits of boundary router switches are set to zero.

4.3 Memory Configuration and Cache

While this work concentrates on timing channels in NoC systems, several other timing channels exist in real systems. We believe that the solution in this article must be combined with a set of side channel mitigation strategies (e.g., memory and cache partitioning) to address all conceivable system vulnerabilities.

Fig. 7. Different configurations of memory controllers (MCs). We assume a security domain per each MC.

4.3.1 Memory. Modern chip packaging has enough flexibility to place the memory controllers anywhere on the chip [3]. Hence, memory controller placement is complementary to our approaches, and we can improve our mechanism with different memory controller placement configurations as shown in Figure 7. This topic is not discussed in this article, and can be considered as a promising future direction. It is expected that the future, and even the current many cores need to have more memory controllers on chip; therefore, we assume that each security domain has access to a home memory controller that ensures non-interference memory access.

We assume our target many-core architecture has private L1 and L2 caches and distributed memory controllers. Therefore, we believe that our approach can eliminate inter-domain interference including traffic generated on-chip to/from memory controllers by proposing memory controller aware placement of security domains.

Our approach is based on the observation that many-core systems have multiple memory controllers, each of which can be accessed independently without any interference. This reveals an interesting trade-off. On the one hand, interference between applications could be completely eliminated if each application's data was mapped to a different memory controller, and was isolated from distrusting neighbor applications (such as Isolated approach). On the other hand, even if so many memory controllers were available, mapping each application to its own memory controller would underutilize memory capacity and would reduce the opportunity for memory-level parallelism within each application's memory access stream. Therefore, the main goal of our approach is to find a suitable point in this trade-off by mapping the applications with the same security domain in isolated regions and providing isolated off-chip accesses for applications to their own memory controllers.

How to enforce security domains for memory accesses? Isolating can be achieved by managing the memory accesses through cores in an appropriate manner. To enable isolating, memory access policies must be modified such that data requested by a core is opportunistically mapped to the home memory controller (HMC) of the core. To achieve this, we consider a modified version of the commonly used CLOCK [27] algorithm.

We need to map data requested by a core to the specified memory controller of the security domain, which we call HMC, or to the guest memory controller (GMC). Appertain to our memory-aware security domain placement approach (in Section 5.2), it is ensured that each security domain has access to an HMC. Moreover, when a security domain expands, it is possible that some memory controllers be covered as GMC, as shown in Figure 6(b).

To enable this policy, according to Reference [27], when a page fault occurs and free pages exist, preference is given to the free pages belonging to the HMC of a requesting core for allocating a new page. If no free pages belonging to the HMC exists, then a free page from the available GMC is allocated. When a page fault occurs and no free pages exists, preference is given to a page belonging to the HMC, while finding the replacement page candidate. It needs to look N pages beyond the default candidate found by the algorithm presented in Reference [27], to find a page that belongs to HMC. If finding a replacement candidate belonging to HMC is unsuccessful when searching N

pages beyond the default candidate, then the algorithm simply selects the default candidate for replacement. Through this biasing page allocation and replacement decisions, pages are allocated to a core's HMC or GMCs with non-interference approach and opportunistically achieve the effect of our proposed regional rectangle as a security domain.

The above modifications ensure that the new page replacement policy does not significantly perturb the existing replacement order, and at the same time opportunistically achieves the effect of isolating. Note that these modifications to virtual memory management (for both page allocation and page replacement) do not enforce a static partitioning of DRAM memory capacity; they only bias the page replacement policy such that it most likely allocates pages to a core from the core's home memory controller.

4.3.2 Cache. Another important security issue can be caused by interference among cores during runtime mapping, which we refer to it as timing interference via core mapping. This may happen through private resources of each core. The resources that are dedicated to each core are only used by one domain at a time. However, multiple domains can use these resources through dynamic allocation. Therefore, timing channels exist if the state is kept across context switches. Hence, to eliminate this timing channel, domains flush the per-core state when a core leaves a domain. To prevent information leakage, the time that these flushing operations take cannot depend on the domain's state. For example, cache flushing should not take longer when there are more dirty blocks. Therefore, after flushing, each core is blocked until the worst case writeback time is passed. To prevent writeback requests from interfering with the incoming process, the core must be stalled until all writebacks are complete. The time required to block the pipeline depends on the worst-case time that is required to drain all writebacks. When a region including occupied and unoccupied cores releases, the worst-case writeback time can be determined by assuming that every cache block in each part of the cache hierarchy is dirty, and needs to be written back. A rough approximation can be determined by multiplying the size of each private cache or shared cache of the region by the latency of the cache one step up in the hierarchy. For the last-level cache, we use the worst-case memory latency. In our evaluation, we found that the impact of flushing and writeback blocking is negligible.

4.4 Putting It Altogether

Our idea to support non-interference NoC, and eliminate timing channels is demonstrated through isolated security domain fact in which isolated partitions deploy all security domain requirements. Then, LBDR mechanism is used to confine internal and external interference. Furthermore, the process of accessing memory controllers through applications across the NoC is provided by memory aware placement of security domains. Security placement ensures that each security domain has an HMC or some GMCs. We have also described that private caches in our architecture are protected from leakage of core mapping by flushing and writeback blocking mechanism that is evaluated in Section 7.1.

To finalize our proposed methodology as a security defense and non-interference NoC to eliminate timing channel attacks, we need a mechanism for efficiently managing and allocating the isolated security domains. Now, we need to have a run-time mapping mechanism to map entering applications into isolated regions. In the next section, we describe our proposed run-time mapping problem.

5 ISOLATED RUNTIME MAPPING

In this section, we introduce Liso (Lshape-isolated) runtime mapping approach. We describe the problem of region finding and region selection to find an appropriate region for each security domain, followed by description of runtime mapping policies.

Fig. 8. Security domain configuration and definitions.

5.1 Overview of the Proposed Approach

In this approach, we consider each available corner as one security domain. We can create corners by splitting/dividing the NoC horizontally or vertically into some regions. Therefore, we can have n security domains. In this work, we assume that our NoC is a 2D mesh; so by each division in the NoC, we can add some corners. Therefore, we can split NoC iteratively to create security domains as shown in Figure 8(a). It is obvious that by splitting the NoC into symmetric parts (as shown in Figure 8(a)), we can create security domains that are balanced. The coordinates of each security domain can be identified by two points, which we call them as start point (SP) and corner point (CP) as shown in Figure 8(b). At first, both points have the same value. When security domain grows, the corner point changes to the new coordinates.

As shown in Figure 8(b), by considering a security domain at corner, it can expand at most in three directions (horizontal, vertical, diagonal). In other words, each security domain has one expandable region towards each direction. When an application enters the system, it has to fit into a security domain. Security domains in the system are in one of the two following situations, empty or occupied (not necessarily full). The expandable region finding algorithm (in Section 5.2.4) needs to find the available space in the empty or occupied security domain with the same security level. For finding an appropriate security domain, expandable region selection policy (discussed in Section 5.3) searches among the security domains that have enough space to fit the entering application.

After choosing an SD as a candidate for mapping of the current application, mapping process (in Section 5.4) maps SD's region accordingly. To discuss the rest of the section, we need to describe some definitions (in Figure 8(b)) as follows:

- **Region:** one area that contains some security domains (at most 4 security domains for 2D-mesh).
- **Security Domain (SD):** an isolated partition. Hereafter, we refer to number of SDs as nSD .
- **Start Point (SP):** indicates the coordinates of start point of a security domain partition.
- **Corner Point (CP):** indicates the coordinates of endpoint of a security domain partition.
- **Expansion side (ES):** the side of an SD that can fill horizontally or vertically. Each SD has only two expansion sides (horizontal side and vertical side).
- **Expandable Region:** available space towards each direction of an SD (three directions for each SD type in Table 1).
- **Coverage Area:** total area that a security domain can cover when expanding toward its directions.
- **Collisional SD (CSD):** a security domain that can be collided when an SD expands in its coverage area. Hereafter, we refer to number of CSDs as $nCSD$.

Table 1. Directions and Region Models for Each Type of Security Domain

SD Type Number	Direction	Region/Mapping Type
0	N-NE-E	Vertical (Top)-Diagonal-Horizontal (Right)
1	N-NW-W	Vertical (Top)-Diagonal-Horizontal (Left)
2	S-SE-E	Vertical (Bottom)-Diagonal-Horizontal (Right)
3	S-SW-W	Vertical (Bottom)-Diagonal-Horizontal (Left)

Fig. 9. A, B, C, and D show the coverage area for SD0, SD1, SD2, and SD3, respectively. All CSDs are within each coverage area.

- **Collisional SP (CSP):** SP of the Collisional SD.
- **Collisional CP (CCP):** CP of the Collisional SD.
- **Xsize:** number of nodes on X direction of the NoC.
- **Ysize:** number of nodes on Y direction of the NoC.

The rest of the section discusses how this overall approach can be efficiently realized in our proposed architecture.

5.2 Expandable Region Finding Problem

5.2.1 Security Domain Configuration. Based on the proposed security domain isolation fact, we use Guillotine split approach [30] as a procedure of placing an isolated SD to a corner of a free region. The actual process of placing isolated SDs is then modeled as an iterative function of the Guillotine split placement operation. In other words, this procedure can proceed to make at most n (size of network) isolated SDs. By default, we have four corners in a 2D mesh network. To apply the Guillotine procedure and insert more security domains, we can define more regions inside each region iteratively and continue the placement of security domains at corners, as shown in Figure 8(a). It should be noted that each security domain placement is aware of the location of the memory controllers on chip.

5.2.2 Security Domain Direction Types. According to placement of security domains, four types of security domains can be made in each region, which are indicated by their type number (Table 1). As shown in Figure 8(a) (for region 0), each type can cover three directions (the second column of Table 1) as well as three types for region and mapping (the third column of Table 1).

5.2.3 Security Domain Coverage Areas and Collisional Security Domains. By expansion of a security domain, it can grow up to the maximum area, which we call coverage area (in Figure 9). According to each security domain type, the coverage area can be calculated as shown in Table 2. When an SD expands, it is possible to collide with some security domains. Therefore, we can calculate all security domains that are located in coverage area of the SD. As shown in Figure 9, we call these security domains as collisional security domains (CSDs). Note that an SD can collide

Table 2. Coverage Areas for Each Type of Security Domain

SD Type Number	Coverage Area	Example (Figure 9)
0	$ X_{SP} - Xsize \times Y_{SP} - Ysize $	$SP(x, y) = SD_0(x, y) \Rightarrow 6 - 8 \times 4 - 8 $
1	$ X_{SP} + 1 \times Y_{SP} - Ysize $	$SP(x, y) = SD_1(x, y) \Rightarrow 5 + 1 \times 6 - 8 $
2	$ X_{SP} - Xsize \times Y_{SP} + 1 $	$SP(x, y) = SD_2(x, y) \Rightarrow 6 - 8 \times 3 + 1 $
3	$ X_{SP} + 1 \times Y_{SP} + 1 $	$SP(x, y) = SD_3(x, y) \Rightarrow 5 + 1 \times 5 + 1 $

Fig. 10. Different types of regions.

with an occupied CSD. We note that the process of SD configuration and indicating their CSDs are already known.

5.2.4 Region Finding Problem. For each security domain, four states can be defined as follows:

- (1) Security domain (SD) is empty ($CP = SP$); all collisional SDs (CSD) are also empty (Figure 10(a)).
- (2) SD is empty ($CP = SP$); at least one CSD is occupied (Figure 10(b)).
- (3) SD is occupied ($CP \neq SP$); all CSDs are empty (Figure 10(d)).
- (4) SD is occupied ($CP \neq SP$); at least one CSD is occupied (Figure 10(e)).

In each state, different types of regions according to Table 1 can be considered as follows:

- **State 1:** only one type of region (diagonal) can be constructed (Figure 10(a)).
- **States 2, 3, and 4:** as shown in Figures 10(b), 10(d), and 10(e), three types of regions (Horizontal, Diagonal, Vertical) can be constructed as follows:
 - **Horizontal region:** towards the left (for SD0 and SD3)/right (for SD0 and SD2), based on the direction type (W/E) of each security domain.
 - **Vertical region:** towards the top (for SD0 and SD1)/bottom (for SD2 and SD3), based on the direction type (N/S) of each security domain.
 - **Diagonal region:** towards the SW (for SD3), NW (for SD1), NE (for SD0), and SE (for SD2) directions.

To find the horizontal region, we need to find the $X_{Nearest}$ among all CSDs that are located in front of the current security domain horizontally. Also, to find the vertical region, we need to find the $Y_{Nearest}$ among all CSDs that are located vertically in front of the current security domain. Accordingly, to find all the diagonal regions, we need to scan diagonal regions (in Figure 10(b)) among all CSDs that are located diagonally in front of the current security domain. Note that for State 2 and State 4, the existence of diagonal regions is possible if both the horizontal and vertical regions exist. In other words, $X_{Nearest}$ and $Y_{Nearest}$ are the necessary values for finding the diagonal regions. As it is shown in Figure 10(c), a situation may occur where a diagonal region does not exist while $X_{Nearest}$ and $Y_{Nearest}$ are available.

Horizontal and vertical regions finding. The calculations of the $X_{Nearest}$ and $Y_{Nearest}$ variables in each type of security domain depend on the order of the SP and CP in two directions

Fig. 11. Different types of vertical and horizontal regions.

Table 3. Horizontal and Vertical Conditions

Group	State	Conditions
VERT(0, 2)	① $X_{CSP} \leq X_{CCP}$	$X_{CSP} \leq X_{CP}$ AND $X_{CCP} \geq X_{SP}$
	② $X_{CCP} < X_{CSP}$	$X_{CCP} \leq X_{CP}$ AND $X_{CSP} \geq X_{SP}$
VERT(1, 3)	① $X_{CSP} \leq X_{CCP}$	$X_{CSP} \leq X_{SP}$ AND $X_{CCP} \geq X_{CP}$
	② $X_{CCP} < X_{CSP}$	$X_{CCP} \leq X_{SP}$ AND $X_{CSP} \geq X_{CP}$
HORZ(0, 1)	① $Y_{CSP} \leq Y_{CCP}$	$Y_{CSP} \leq Y_{CP}$ AND $Y_{CCP} \geq Y_{SP}$
	② $Y_{CCP} < Y_{CSP}$	$Y_{CCP} \leq Y_{CP}$ AND $Y_{CSP} \geq Y_{SP}$
HORZ(2, 3)	① $Y_{CSP} \leq Y_{CCP}$	$Y_{CSP} \leq Y_{SP}$ AND $Y_{CCP} \geq Y_{CP}$
	② $Y_{CCP} < Y_{CSP}$	$Y_{CCP} \leq Y_{SP}$ AND $Y_{CSP} \geq Y_{CP}$

(horizontal and vertical). It means that in the horizontal direction, either CP or SP can be located before the other one, and also in the vertical direction CP can be located on top of SP, and vice versa. Based on these orders, the security domains are categorized in two groups for each direction as follows:

- **VERT(0, 2):** Security domains 0 and 2 are grouped for vertical region, because SP is located before CP in horizontal direction as shown ③ in Figure 11(a). Now, we can scan all CSDs that have the conditions in states ① or ② (in Figure 11(a) and Table 3) to find the $Y_{Nearest}$ to CP. Then, using the $Y_{Nearest}$, we can calculate the vertical region.
- **VERT(1, 3):** Security domains 1 and 3 are grouped for another vertical region, because CP is located before SP in horizontal direction as shown ③ in Figure 11(b). Now, we can scan all CSDs that have conditions in states ① or ② (in Figure 11(b) and Table 3) to find the $Y_{Nearest}$ to CP. Then, the vertical region can be calculated using the $Y_{Nearest}$.
- **HORZ(0, 1):** Security domains 0 and 1 are grouped for horizontal region, because CP is located on top of SP in vertical direction as shown ③ in Figure 11(c). Now, we can scan all CSDs that have conditions in states ① or ② (in Figure 11(c) and Table 3) to find the $X_{Nearest}$ to CP. Then, using the $X_{Nearest}$ the horizontal region can be calculated.
- **HORZ(2, 3):** Security domains 2 and 3 are grouped for another horizontal region, because SP is located on top of CP in horizontal direction as shown ③ in Figure 11(d). Now, we can scan all CSDs that have conditions in states ① or ② (in Figure 11(d) and Table 3) to find the $X_{Nearest}$ to CP. Then, the horizontal region can be calculated using the $X_{Nearest}$.

Diagonal region finding. As mentioned above, to find the diagonal region, we need to use the $X_{Nearest}$ and $Y_{Nearest}$ values. Therefore, based on these values, we scan the diagonal region in front of the current security domain to find all possible diagonal regions. First, we sort all CSDs based on the X value (X_{CCP} or X_{CSP}), which is closer to X_{CP} . Then, for each CSD, the values of

Fig. 12. Four possible conditions can be occur in finding diagonal region.

X and Y are compared, respectively, with X_{CP} and Y_{CP} to make a possible diagonal region. Four conditions can occur in this procedure, as are shown in Figure 12. This procedure is illustrated in Algorithm 1. The complete pseudo code for finding regions is presented in Algorithm 2.

ALGORITHM 1: Diagonal region procedure

Input: $X_{Nearest}$, $Y_{Nearest}$

Output: Diagonal region

```

1:  $X_{NR} = X_{Nearest}$ ,  $Y_{NR} = Y_{Nearest}$ 
2: Sort all occupied CSDs out in non-decreasing order of their  $X_{Nearest}$  ( $X_{CCP}$  or  $X_{CSP}$ ) to  $X_{CP}$ 
3: for all sorted occupied CSDs do
4:   if ( $X_{CCP} == X_{CP} \pm 1$  or  $X_{CSP} == X_{CP} \pm 1$ ) and ( $Y_{CCP} == Y_{CP} \pm 1$  or  $Y_{CSP} == Y_{CP} \pm 1$ ) then
5:     Break
        condition 2
6:   else if ( $Y_{CCP} == Y_{CP} \pm 1$  or  $Y_{CSP} == Y_{CP} \pm 1$ ) then
7:     Calculate the Region
        Break
        condition 3
8:   else if ( $X_{CCP} == X_{CP} \pm 1$  or  $X_{CSP} == X_{CP} \pm 1$ ) then
9:      $Y_{NR}$  update to  $Y_{CCP}$  or  $Y_{CSP}$ 
        condition 4
10:  else
11:    Calculate the Region
12:     $Y_{NR}$  update to  $Y_{CCP}$  or  $Y_{CSP}$ 
13:  end if
14: end for

```

5.2.5 Complexity of the Region Finding Algorithm. In States 1 and 3 of Algorithm 2, calculating the horizontal, vertical, and diagonal regions takes linear time. States 2 and 4 need $O(nCSD)$, since we need to scan all CSDs for the current SD, and the run time for sorting them is $O(nCSD \log nCSD)$ if using QUICKSORT. Thus, the run time of region finding algorithm is $O(nCSD + nCSD \log nCSD)$, i.e., $O(nCSD \log nCSD)$, since $nCSD < nCSD \log nCSD$. Therefore, finding regions for the current SD needs $O(nCSD \log nCSD)$.

Furthermore, we need to find regions for all SDs in the system. Thus, to determine the total time complexity of the region finding algorithm for all SDs, we assume that the system contains n SDs, where $n < \text{size of the NoC}$ and the $nCSD_i$ is the number of CSD for SD_i , where

$nCSD < n - 1$. Therefore, $nCSD_1 + nCSD_2 + \dots + nCSD_n = O(nSD \times nCSD \log nCSD)$. Therefore, finding regions for all SDs needs $O(nSD \times nCSD \log nCSD)$. However, we can use RANGE QUERY data structures and find the best areas of each SD in $O(n)$. Therefore, it will be $O(n^3)$ per entering application.

ALGORITHM 2: Region Finding

Input: SD, CSDs

Output: Find available horizontal, vertical and all diagonal regions for SD.

```

  State 1:
  1: if (SD is empty and CSDs are empty) then
  2:   Calculate diagonal region equal to total NoC size.
  3: end if
  State 2:
  4: if (SD is empty and CSDs are Occupied) then
  5:   for all occupied CSDs do
  6:     if (horizontal conditions) then
  7:       Calculate  $X_{Nearest}$  and horizontal region.
  8:     end if
  9:     if (vertical conditions) then
  10:      Calculate  $Y_{Nearest}$  and vertical region.
  11:    end if
  12:   end for
  13:   Sort all occupied CSDs out in non-decreasing order of their  $X_{Nearest}$  ( $X_{CCP}$  or  $X_{CSP}$ ) to  $X_{CP}$ 
  14:   for all sorted occupied CSDs do
  15:     if (diagonal conditions) then
  16:       Calculate diagonal region.
  17:     end if
  18:   end for
  19: end if
  State 3:
  20: if (SD occupied and CSD empty) then
  21:   Calculate horizontal, vertical and diagonal regions.
  22: end if
  State 4:
  23: if (SD occupied and CSD occupied) then
  24:   The same as state 2.
  25: end if

```

5.3 Region Selection Policy

When we want to select an SD with the smallest region to fit the entering application size, we have two groups of SDs, empty and occupied. Region selection policy first searches among the occupied SDs, which have the same security level number with the entering application to select a region; otherwise, it searches among empty SDs.

Complexity of the Region Selection Algorithm. The run time of the region selection algorithm has a complexity of $O(nSD)$. This is because the algorithm needs to execute nSD times to select one of its region. Note that the complexity of the region selection is added to the complexity of region finding algorithm, i.e., $O(nSD + nSD \times nCSD \log nCSD)$. Thus, the total time complexity of region finding algorithm is $O(nSD \times nCSD \log nCSD)$, since $nSD < nSD \times nCSD \log nCSD$.

Fig. 13. Region mapping conditions.

5.4 Region Mapping Problem

Based on the region selection policies, three types of regions can be chosen for mapping. For horizontal and vertical regions, the process is straightforward and the nodes on horizontal or vertical sides/directions will be allocated iteratively for entering application. For diagonal region, as far as the application size is larger than the sum of the two expansion sides, it maps diagonally (for example, APP[12] in SD0 in Figure 13, the size of application is 12 nodes). When the size of the application becomes less than the sum of two sides, it maps based on the following conditions:

- (1) **when two expansion sides have unequal sizes:**
 - (a) If the application size is greater than the size of the larger expansion side, then it maps diagonally (for example, APP[7] in SD0 in Figure 13).
 - (b) If the application size is less than or equal to the size of the larger expansion side and greater than the size of the smaller expansion side, then it maps towards the direction of the larger expansion side (for example, APP[5] and APP[4] map vertically in SD0 in Figure 13).
 - (c) If the application size is less than or equal to the size of the smaller expansion side, then it maps towards the direction of the smaller expansion side (for example, APP[3] maps horizontally in SD0 in Figure 13).
- (2) **When two expansion sides have equal sizes:**
 - (a) If the application size is greater than the expansion side, then it maps diagonally (for example, APP[5] in SD1 in Figure 13).
 - (b) If the application size is less than or equal to the expansion side, then it maps towards either direction (for example, APP[3] in SD1 in Figure 13).

For example, all applications in Figure 13 can be mapped as follows (APP[size]):

- **Map diagonally:** APP[12] in SD0 and SD1, APP[7] in SD0 and SD1, APP[5] in SD1, APP[4] in SD1
- **Map vertically:** APP[5] in SD0, APP[4] in SD0, APP[3] in SD1
- **Map horizontally:** APP[3] in SD0, APP[3] in SD1

Complexity of the Region Mapping Algorithm. The total run time of our algorithm has a complexity of $O(n_{APP})$, where n_{APP} is the application size. This is because the algorithm executes n_{APP} times while all tasks map into the region, based on the above conditions. Note that this complexity should be added to the complexity of the region finding algorithm, i.e., $O(n_{APP} + (n_{SD} \times n_{CSD} \log n_{CSD}))$. Thus, the total time complexity of region finding algorithm is $O(n_{SD} \times n_{CSD} \log n_{CSD})$, since $n_{APP} < n_{SD} \times n_{CSD} \log n_{CSD}$.

5.5 Isolated Approach

State-of-the-art run-time mapping approaches have proposed a solution for mapping applications in contiguous area. Their solutions concentrate on improving performance and latency of the network, while our focus in this work is on providing non-interference communication across the NoC, and eliminating the timing channels. Therefore, we need to isolate applications while the generated traffic can access the memory controllers. For comparing our proposed approach with others [17, 23], we have exploited them with LBDR mechanism and modified them to map in an isolated region for each application. We refer to these modified methods as *Isolated approach* (SHiC-iso and MapPro-iso). The best found contiguous rectangle from region selection method is passed to the mapping algorithm. Afterwards, application tasks are mapped inside the contiguous and completely isolated rectangle area. The main shortcoming of the *Isolated approach* for providing non-interference between applications is that it requires each application's data to be mapped to a different memory controller and also be isolated from distrusting neighbor applications. This would underutilize memory capacity and would reduce the opportunity for memory-level parallelism within each application's memory access stream. However, we assume that each isolated application has access to an independent memory controller. To overcome this shortcoming, our proposed mechanism maps the applications with the same security domain in isolated regions and provides isolated off-chip accesses for applications to their own memory controllers.

6 METHODOLOGY

6.1 Experimental Setup

We evaluate our techniques using Noxim, a cycle-accurate SystemC many-core platform [9]. To study the scalability effects, we also apply our techniques to NoCs with different sizes, from 8×8 to 20×20 nodes. We use the deterministic X-Y routing algorithm, finite input buffering, wormhole switching, and virtual-channel flow control. The mapping techniques are performed at the control manager (CM), residing in $n_{x,y} = (0, 0)$.

6.2 Workloads and Simulation Methodology

We evaluate traces from a diverse set of multithreaded applications including PARSEC [5] and SPLASH-2 [46] benchmark suites. Totally, we study 22 applications, including 11 SPLASH-2 and 13 PARSEC traces (Netrace [24]). We experiment with obtaining from full-system simulation using gem5 [6] in the *syscall* emulation mode. Table 4 summarizes our gem5 model configuration. The traces are collected for 500 million cycles after spawning threads and initializing caches. The traffic weights were calculated by counting the number of flits per each source-destination pair. Each application runs for 8 to 64 threads, with one thread pinned on every core. All our results are across 1368 generated application traces mixes as one workload combination.

We run all applications for 10 billion cycles over different network sizes from 64 to 400 cores. Each task is mapped to one core in the NoC using the proposed mapping techniques. A random sequence of applications is entered into the scheduler FIFO according to the desired rate. The sequence is kept fixed in all experiments for the sake of fair comparison.

6.3 Evaluation Metrics

We use latency, in clock cycles, of the network and impact on throughput of the system. We consider the performance metrics at the application-level or system-level to gauge performance of an open system, such as system utilization [13]. We evaluate utilization of the system through running each workload, where thousands of applications arrive and depart the system. We also use average weighted manhattan distance (AWMD) [17], as a metric to evaluate the power consumption of a

Table 4. gem5 Configuration for Traces

Core	2GHz processor, x86 ISA, out-of-order
L1 cache	32KB per-core (private), 4-way set, 64B block size, 3-cycle latency, MESI
L2 cache	256KB per core (private), 16-way set associative, 6-cycle latency, 64B block size, MESI
Memory	1GB DRAM, 150 cycle access, 4 on-chip Memory Controllers (one in each corner node)
Network	8x8 mesh, each node has a router, processor, private L1 cache, private L2 cache bank (all nodes), 128-bit bi-directional links, wormhole switched, virtual channel flow control, 2 VC's per Port, 5 flit buffer per VC

mapped application as defined by Equation (1):

$$AWMD_{map(A_p)} = \frac{\sum_{e_{i,j} \in E} w_{i,j} \times MD(nt_i, nt_j)}{\sum w_{i,j}}. \quad (1)$$

We use mapped region dispersion (NMRD) [17] value as a metric for evaluating the squareness of the mapped application. In fact, this factor is highly related to the performance of an application and network, because contiguous mapping of an application can impact internal and external congestions. NMRD is defined by Equation (2):

$$NMRD_{map(A_p)} = \frac{|MRD_{map(A_p)} - MRD_{SQ(|T|)}|}{MRD_{SQ(|T|)}}. \quad (2)$$

7 RESULTS

We evaluate the effect of our proposed method, Liso, on system and application performance (utilization and throughput), network latency and energy consumption. This evaluation has been performed in two modes with, and without security considerations. The first mode evaluates the comparison of dynamic mapping approaches without security concern (indicated by Liso, MapPro, and SHiC). In this case, we consider all applications in workload with the same security level. The second mode compares the proposed approach (indicated by Liso-sec) with isolated versions of two state-of-the-art approaches (indicated by SHiC-iso and MapPro-iso). In this case, we consider different security levels for applications in workload.

7.1 Effect on Average Network Latency

Non-secure mode. NMRD, as the dispersion metric in Figure 14(a), shows how each mapping approach leads to more congestion. This congestion is the result of more dispersed mapping. SHiC and MapPro schemes map each application in a near square and contiguous region; however, their NMRD values increase by 8% and 44%, respectively with increasing network size due to filling the neighbour cores out of square shape. The Liso approach maps applications in a contiguous region referred to as a domain. Overall, Figure 14(b) shows the average network latency increment of Liso approach due to a dispersed mapped area by 27% and 23% compared to MapPro and SHiC, respectively.

Secure mode. Figure 14(d) shows the average network latencies of the approaches under security mode. We observe that latencies of all approaches are almost independent of the network size. For Liso scheme, this is the result of mapping applications with the same security level in a

(a) Non-secure mode NMRD

(b) Non-secure mode average network latency

(c) Secure mode NMRD

(d) Secure mode average network latency

Fig. 14. The effect of dispersal vs. contiguous mapping on latency for different network sizes.

(a) Non-secure mode power consumption

(b) Secure mode power consumption

Fig. 15. Power consumption for different number of nodes.

contiguous region, also referred to as domain; therefore, the average network latency of running applications are preminent to network size. However, the network latency overhead of Liso approaches are 48% and 29% on average compared with MapPro-iso and SHiC-iso, respectively. In Figure 14(c), the dispersion level of Liso mapping increases by 13% on average for both SHiC-iso and MapPro-iso approaches. We can conclude that the Liso scheme is scalable with network size when applied to security workloads.

7.2 Effect on Power Consumption

Non-secure mode. The power consumptions are extracted from AWMD metric as a cost of total packet delivery that is related to the number of hops that all packets traverse. The extracted power values follow the same trend as AWMD. Figure 15(a) clearly shows that Liso-4 performs poorly, since there are many non-contiguous regions in the system. We show that the Liso approach results in 20% and 45% energy consumption overhead on average, compared to MapPro and SHiC, respectively.

Secure mode. Figure 15(b) shows the effect of our Liso's approaches on energy consumption. We observe that Liso-4-sec, Liso-8-sec, Liso-16-sec, and Liso-32-sec increase energy consumption on average by 7% (10%), 10% (13%), 9% (12%), and 9% (12%) in comparison with MapPro-iso (SHiC-iso), respectively. The Liso-sec reduces energy consumption by 31% on average compared to the

Fig. 16. Performance impact of different isolation mapping approaches for different number of nodes.

Liso with no security requirement. This is mainly because the Liso scheme provides better mapping results when applications in workload need security requirement. We conclude that the mechanism we proposed in Section 5 toward security on NoC architecture is effective under workloads with security requirements.

7.3 Effect on Performance

Throughput and utilization are the first-order concerns for concurrent many-core systems. Figure 16(a) shows the utilization of the system for all approaches. We observe that Liso approaches suffer from utilization degradation by 13% and 26% on average over MapPro and SHiC, respectively, due to the overhead of region selection for mapping in Liso approaches. In the **non-secure mode**, throughput degradations of Liso approaches are 5% and 22% on average, compared with MapPro and SHiC, respectively, as shown in Figure 16(b). In the **secure mode**, all Liso approaches cause a decrease in average system utilization by 8% and 16% compared to MapPro-iso and SHiC-iso, respectively, as shown in Figure 16(c). Similarly, as Figure 16(d) shows, the throughput decreases by 19% and 14% on average for Liso approaches in comparison with MapPro-iso and SHiC-iso, respectively. Utilization degradation may happen once an application enters into the system and there is enough space to map it, but its security level does not allow such mapping. However, in *Isolated approach* the applications do not experience such waiting time.

(a) Memory hop count

(b) static isolation vs. dynamic isolation

Fig. 17. Memory hop count and performance impact of static isolation vs. dynamic isolation.

Figures 16(e) and 16(f) show the number of penalty nodes that each mapping causes. In both modes (secure and non-secure), SHiC and MapPro leads to a large number of penalty nodes. This indicates that their fragmentation and utilization features result in a large impact on system performance. Overall, the Liso (Liso-sec) approaches generate about 99% (90%) and 99% (96%) less penalty on average, over MapPro (MapPro-iso) and SHiC (SHiC-iso), respectively. We conclude that our performance results come from the trade-off between contention and penalty nodes (fragmentation). Using Liso-sec (Liso) approach, we see an average penalty node improvement of 93% (99%) with a throughput overhead of 17% (14%).

7.4 Sensitivity to the Number of Security Domains

Figure 16(c) shows the performance benefit of Liso mapping schemes using **the number of security domains**. Under secure mode for Liso-8-sec, Liso-16-sec, and Liso-32-sec approaches, average utilization improves by 20%, 33%, and 34% over the Liso-4, respectively. Similarly, Liso-8-sec, Liso-16-sec, and Liso-32-sec gain throughput by 47%, 57%, and 79% on average over the Liso-4-sec as shown in Figure 16(d). The performance improvement of Liso using more number of security domains is correlated to the number of penalty nodes. Figure 16(f) shows the percentage of the total penalty nodes caused by Liso approaches. The Liso-8-sec, Liso-16-sec, and Liso-32-sec cause about 42%, 43%, and 52%, on average, less penalty nodes compared to Liso-4-sec.

7.5 Memory Hop Count

We show the effect of different memory-aware security configurations on off-chip access pattern. To examine this, in Figure 17(a), we show the total number of memory hop counts that an application needs to travel for off-chip accesses, i.e., the average distance that the memory requests need to travel. We observe that memory hop count for Liso-8, Liso-16, and Liso-32 improve by 2%, 7% and 10% on average over Liso-4. We conclude that by increasing the size of network, we need to configure security domains and memory controllers according to system requirement. As the future work, a more sophisticated mechanism can be exploited, which considers the memory level parallelism and memory configuration.

7.6 Static vs. Dynamic

To show the superiority of dynamic isolation over static isolation, we compare utilization of dynamic isolation approaches against static ones. We set up identical regions for 4, 8, 16, and 32 security domains for Liso-4, Liso-8, Liso-16, and Liso-32 schemes. It means that based on network size, each security domain is confined evenly. For example, for a network size of 64, we assume that each security domain has 16 nodes. It should be noted that for a network of size 64, we consider applications with at most 16 tasks. It can be seen that utilization improves by 20% on average for dynamic approaches compared to the static approaches, as shown in Figure 17(b).

Fig. 18. Performance overhead of context switching for non-interference via core mapping.

7.7 Security Evaluation

We show non-interference and isolation properties of Liso by examining that the throughput of a security domain is independent of the applications that exist in other domains. We consider three workloads: (i) *W1*: 16 applications with security level 1, (ii) *W2*: 16 applications with security level 2, and (iii) *W3*: mix of both *W1* and *W2*. We run Liso-2-sec with size of 64 nodes for each workload, separately. We observe that the throughputs for the security domains with identical levels in the above runs are equal. It means that the load in one security domain is completely isolated from the changes in other security domains, and the throughput is not altered.

For security evaluation of SHiC-iso and MapPro-iso, as each application maps independent of others, we examine throughput in two cases. First, we evaluate the throughput when one application is mapped alone in the system. Next, we evaluate the throughput when two applications are mapped in the system. As expected, we observe that throughput in the two cases are equal. This shows the effectiveness of isolation process for *Isolated* versions of the two approaches.

To support **non-interference via core mapping**, we evaluate turnaround time of each proposed approach with flushing. Figure 18 shows the turnaround time of running 50 applications for all approaches with and without flushing. We observe that the overhead of context switching is quite small. On average, the overhead for network size of 64 (400) is 1% (5%).

8 CONCLUSION

We introduce Liso (L-shape isolated), a runtime mapping that enables isolation and eliminates interference among applications for NoC resources. The goal of our approaches is to provide more security for applications with common communication resources. The main idea is to have runtime isolated regions across a NoC, for applications with different security domains, and map applications to these regions effectively. Within each security region, applications are mapped efficiently to improve system throughput. Liso yields in performance degradations of 8% and 16%, respectively, relative to MapPro-iso and SHiC-iso (secure baselines). Unlike the previous techniques, Liso and Isolated schemes provide security under efficient and cost-effective minimal routing mechanism, and thus yield in higher performance. We conclude that runtime mapping is a promising software mechanism that transparently provides non-interference communication among different applications in NoC-based many-cores.

REFERENCES

- [1] Adapteva. [n.d.]. Epiphany Technical Reference Documents. Retrieved from <http://www.adapteva.com/all-documents>.
- [2] Kalray. [n.d.]. MPPA Manycore. Retrieved from <http://www.kalrayinc.com/>.
- [3] Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. 2009. Achieving predictable performance through better memory controller placement in many-core CMPs. *ACM SIGARCH Comput. Architect. News* 37, 3 (2009), 451–461.

- [4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhanian. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 29–44.
- [5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 72–81.
- [6] 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 Comput. Architect. News* 39, 2 (2011), 1–7.
- [7] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai et al. 2008. Corey: An operating system for many cores. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, vol. 8, 43–57.
- [8] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich et al. 2010. An analysis of Linux scalability to many cores. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, Vol. 10, 86–93.
- [9] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti. 2015. Noxim: An open, extensible and cycle-accurate network on chip simulator. In *Proceedings of the IEEE 26th International Conference on Application-specific Systems, Architectures Processors*. 162–163.
- [10] Chen-Ling Chou and R. Marculescu. 2010. Run-time task allocation considering user behavior in embedded multi-processor networks-on-chip. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 29, 1 (Jan. 2010), 78–91.
- [11] William James Dally and Brian Patrick Towles. 2004. *Principles and Practices of Interconnection Networks*. Elsevier.
- [12] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. IEEE, 107–118.
- [13] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. 2009. Application-aware prioritization mechanisms for on-chip networks. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 280–291.
- [14] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (1976), 236–243.
- [15] Tanima Dey, Wei Wang, Jack W. Davidson, and Mary Lou Soffa. 2013. ReSense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Trans. Architect. Code Optim.* 10, 4 (2013), 41.
- [16] Jose Duato, Sudhakar Yalamanchili, and Lionel M. Ni. 2003. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann.
- [17] Mohammad Fattah, Masoud Daneshdalan, Pasi Liljeberg, and Juha Plosila. 2013. Smart hill climbing for agile dynamic mapping in many-core systems. In *Proceedings of the 50th ACM Design Automation Conference*. 39:1–39:6.
- [18] Mohammad Fattah, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. 2014. Adjustable contiguity of run-time task allocation in networked many-core systems. In *Proceedings of the 19th IEEE Asia South Pacific Design Automation Conf*. 349–354.
- [19] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2016. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 382–393.
- [20] C. W. Fletcher, Ling Ren, Xiangyao Yu, M. van Dijk, O. Khan, and S. Devadas. 2014. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture*. 213–224.
- [21] J. Flich and J. Duato. 2008. Logic-based distributed routing for NoCs. *IEEE Comput. Architect. Lett.* 7, 1 (Jan. 2008), 13–16.
- [22] Akhila Gundu, Gita Sreekumar, Ali Shafiee, Seth Pugsley, Hardik Jain, Rajeev Balasubramonian, and Mohit Tiwari. 2014. Memory bandwidth reservation in the cloud to avoid information leakage in the memory controller. In *Proceedings of the 3rd Workshop Hardware Architectural Support Security Privacy*. 1–5.
- [23] Mohammad-Hashem Haghighyan, Anil Kanduri, Amir-Mohammad Rahmani, Pasi Liljeberg, Axel Jantsch, and Hannu Tenhunen. 2015. Mappro: Proactive runtime mapping for dynamic workloads by quantifying ripple effect of applications on networks-on-chip. In *Proceedings of the 9th International Symposium on Networks-on-Chip*. ACM, 26.
- [24] Joel Hestness, Boris Grot, and Stephen W. Keckler. 2010. Netrace: Dependency-driven trace-based network-on-chip simulation. In *Proceedings of the 3rd International Workshop on Network on Chip Architectures*. ACM, 31–36.
- [25] John Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, Devon Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom et al. 2010. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference*. 108–109.

- [26] Jingcao Hu, Umit Y. Ogras, and Radu Marculescu. 2006. System-level buffer allocation for application-specific networks-on-chip router design. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 25, 12 (2006), 2919–2933.
- [27] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the USENIX Annual Technical Conference*. 323–336.
- [28] Fangfei Liu and Ruby B. Lee. 2014. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 203–215.
- [29] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. 2016. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro* 36, 5 (2016), 8–16.
- [30] Andrea Lodi, Silvano Martello, and Daniele Vigo. 1999. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS J. Comput.* 11, 4 (1999), 345–357.
- [31] Jim Ng, Xiaohang Wang, Amit Kumar Singh, and Terrence Mak. 2016. Defragmentation for efficient runtime resource management in NoC-based many-core systems. *IEEE Trans. Very Large Scale Integr. Syst.* 24, 11 (2016), 3359–3372.
- [32] Anuj Pathania, Vanchinathan Venkataramani, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. 2017. Defragmentation of tasks in many-core architecture. *ACM Trans. Architect. Code Optim.* 14, 1 (2017), 2.
- [33] Anastasios Psarras, Junghee Lee, Ioannis Seitanidis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. 2016. PhaseNoC: Versatile network traffic isolation through TDM-scheduled virtual channels. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 35, 5 (2016), 844–857.
- [34] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference Computer Communications Security*. 199–212.
- [35] Richard Schooler. 2010. Tile processors: Many-core for embedded and cloud computing. In *Proceedings of the 14th IEEE Workshop High Performance Embedded Computing*.
- [36] Michael D. Schroeder and Jerome H. Saltzer. 1972. A hardware architecture for implementing protection rings. *Commun. ACM* 15, 3 (1972), 157–170.
- [37] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. 2015. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th ACM International Symposium on Microarchitecture*. 89–101.
- [38] Jakub M. Szefer. 2013. *Architectures for Secure Cloud Computing Servers*. Ph.D. Dissertation. Princeton University.
- [39] Francisco Triviño, José L. Sánchez, Francisco J. Alfaro, and José Flich. 2012. Network-on-chip virtualization in chip-multiprocessor systems. *J. Syst. Archit.* 58, 3 (Mar. 2012), 126–139.
- [40] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. 2014. Timing channel protection for a shared memory controller. In *Proceedings of the 20th IEEE Int. Symp. High Performance Computer Architecture*. 225–236.
- [41] Yao Wang and G. Edward Suh. 2012. Efficient timing channel protection for on-chip networks. In *Proceedings of the 6th IEEE/ACM Int. Symp. Networks on Chip*. 142–151.
- [42] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Int. Symp. Computer Architecture*. 494–505.
- [43] Zhenghong Wang and Ruby B. Lee. 2008. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41th IEEE/ACM Int. Symp. Microarchitecture*. 83–93.
- [44] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2013. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proceedings of the 40th International Symposium on Computer Architecture*. 583–594.
- [45] David Wentzlaff and Anant Agarwal. 2009. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operat. Syst. Rev.* 43, 2 (2009), 76–85.
- [46] 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.

Received October 2017; revised December 2018; accepted May 2019