Performance-Asymmetry-Aware Topology Virtualization for Defect-tolerant NoC-based Many-core Processors

Lei Zhang*, Yue Yu†, Jianbo Dong*, Yinhe Han*, Shangping Ren†, and Xiaowei Li*

*Key Laboratory of Computer System and Architecture
Institute of Computing Technology, Chinese Academy of Sciences
{zlei, dongjianbo, yinhes, lxw}@ict.ac.cn

†Department of Computer Science
Illinois Institute of Technology
{yyu8, ren}@iit.edu

Abstract

Topology virtualization techniques are proposed for NoC-based many-core processors with core-level redundancy to isolate hardware changes caused by on-chip defective cores. Prior work focuses on homogeneous cores with symmetric performance and optimizes on-chip communication only. However, core-to-core performance asymmetry due to manufacturing process variations poses new challenges for constructing virtual topologies. Lower performance cores may scatter over a virtual topology, while operating systems typically allocate tasks to continuous cores. As a result, parallel applications are probably assigned to a region containing many slower cores that become bottlenecks. To tackle the above problem, in this paper we present a novel performance-asymmetry-aware reconfiguration algorithm Bubble-Up *based on a new metric called core fragmentation factor (*CFF*). Bubble-Up can arrange cores with similar performance closer, yet maintaining reasonable hop distances between virtual neighbors, thus accelerating applications with higher degree of parallelism, without changing existing allocation strategies for OS. Experimental results show its effectiveness.*

1 Introduction

Many-core processors integrate tens to hundreds of processing cores on a single chip for massive parallel processing [1]. Network-on-chip (NoC) is considered to be the most promising interconnection solution for many-core processors, for its scalability and high throughput [2].

Fabrication yield is a serious concern for NoC-based many-core chips due to large chip areas. Thus, effective defect tolerance techniques are essential. For example, without considering defect tolerance during the architecture design phase, the yield of Cell processors can be as low as 10% to 20% [3].

For many-core processors, as single core is small and inexpensive when compared to the entire chip, core-level redundancy is considered to be an efficient solution for yield improvements [4, 5], in which an n-core processor is provided with m redundant cores. We can still get n operational cores on-chip in case of up to m core failures.

When faulty cores are replaced by spare ones in NoC-based many-core processors, however, it is possible that the NoC

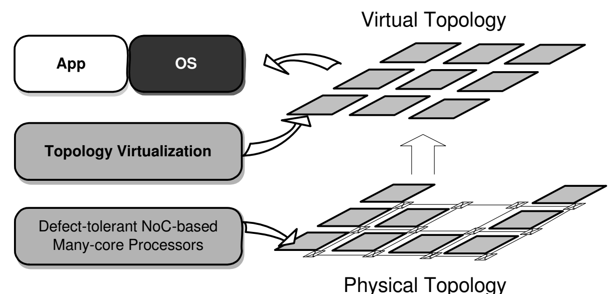


Figure 1. Topology virtualization concept.

topology, i.e., the interconnection relationship between embedded cores, is changed. Thus, different fabricated chips (with possible different core failures) may have different underlying topologies. Operating Systems (OS) and programmers may have to face various different topologies to schedule and optimize parallel applications. This is a big burden and may also cause confusions in marketing.

To address the above problem, we proposed *topology virtulization* techniques to provide OS and application programmers a unified *virtual topology* to isolate underlying hardware complexities [6]. Fig. 1 depicts the concept. As there can be many candidate virtual topologies, under the assumption of homogeneous cores with *symmetric* performances, we proposed two metrics, i.e., distance and congestion factors, to evaluate the performance degradation of on-chip networks. A heuristic called the row rippling column stealing guided simulated annealing (*RRCS-gSA*) algorithm is then devised to find a virtual topology that optimizes the *communication* performance of on-chip networks, i.e., latency, throughput, and traffic distribution.

Manufacturing process variations have become a major concern as they can cause core-to-core (C2C) performance asymmetry in many-core processors. Such kind of C2C performance asymmetry introduces new challenges to the topology virtulization problem. When processing cores are capable of executing the same instruction set at different performance levels, the performance of on-chip communications is no longer the sole factor in determining the overall performance of virtualized many-core systems. For instance, OS and parallel programmers typically allocate tasks to physically or logically adjacent cores [7, 8, 9]. However, without considering C2C performance asymmetry, slower cores may scatter over a virtual topology. It is therefore difficult to find an area of continuous and high performance cores. Thus, parallel tasks may be assigned to a region in a virtual topology that contains many slower cores. In such cases, the advantages of high performance cores diminish and the slower

*The work was supported in part by National Natural Science Foundation of China (NSFC) under grant No. (60906018, 60806014, 60831160526, 60633060, 60921002), in part by National Basic Research Program of China (973) under grant No. (2005CB321604, 2005CB321605), and in part by Hi-Tech Research and Development Program of China (863) under grant No.(2007AA01Z109, 2007AA01Z113, 2009AA01Z126). The work of S. Ren was supported by NSF CAREER Award (CNS0746643).

cores become bottlenecks of parallel applications.

To investigate the above problem, in this paper we first derive a new metric, called the core fragmentation factor (*CFF*) to evaluate the virtual topologies considering both the performance asymmetries and hop distances among cores. A novel heuristic performance-asymmetry-aware topology reconfiguration algorithm, called the *Bubble-Up* is then presented based on *CFF*. Such kind of "core defragmentation" technique provides OS and programmers a large area of continuous and high performance cores yet maintaining reasonable hop distances between virtual neighbors. This facilitates acceleration of applications with higher degree of parallelism, without changing existing allocation strategies for OS. Experimental results show its effectiveness.

The rest of the paper is organized as follows: Section 2 motivates this work. Section 3 investigates the impact of C2C performance asymmetry and hop distances on the execution time of parallel applications. The core fragmentation factor (*CFF*) is then derived to measure both the core performance differences and hop distances in a virtual topology. Section 4 describes in detail the Bubble-Up algorithm. Section 5 presents and discusses experimental results. We conclude the paper in Section 6.

2 Motivations

C2C performance asymmetry in many-core processors arises due to within-die (WID) variations, which consist of random and systematic components. Systematic variations due to lithographic aberrations exhibit spatial correlation, which depends only on the distance between two transistors [10]. Therefore, nearby transistors share similar parameter values. In contrast, random variations induced by dopant fluctuations affect transistors randomly and independently, and thus have no spatial correlation.

According to empirical data from [11], the effects of random and systematic components of WID variation are approximately equal in 130-nm technology, and random variations dominate with scaling. That means we will discern quite small spatial correlation features as it is covered up by random effects. As a result, C2C performance asymmetry will become even worse, and manifest in two aspects: 1) individual core's frequency deviates more from each other, and 2) the distribution of cores' frequencies becomes more random.

For these performance-asymmetric many-core processors, previous work on topology virtulization for NoC-based many-core processors with core-level redundancy in [6] no longer applies as it assumes homogeneous on-chip cores with symmetric performance, and optimizes the performance of on-chip communications only. Consider the following example:

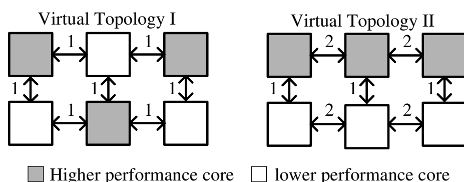


Figure 2. Topology virtualization considering C2C performance asymmetry.

As shown in Fig. 2, different cores have different performances in Virtual Topology I. The numbers on the arrows indicate the hop counts. On the other hand, if we logically invert the two cores in the middle as in Virtual Topology II, it is clear that hop counts between cores increase due to the inversion and thus communication efficiency degrades. However, Virtual Topology II changes the underlying random variations into *virtual* spatial correlated variations and cores with similar performances are brought together without increasing too much the hop distances between cores.

Most existing processor allocation techniques are *continuous*, meaning that the processors assigned to an application are physically or logically adjacent [7, 8, 9]. From the view point of OS, Virtual Topology II provides a larger area of continuous and high performance cores and hence makes it possible to accelerate applications with higher degree of parallelism. For example, 3-thread parallel applications will benefit from running on the upper 1×3 submesh in Virtual Topology II. In Topology I, however, every submesh of size larger than 1×1 will inevitably contain at least one slower core. We will show in the following section that the execution time for the parallel programs are constrained by the slowest core allocated to the programs. As a result, for any parallel applications, if the degree of parallelism is at least 2, slower cores will become the bottleneck in Topology I even if fast cores are used in parallel.

The above observations are quite similar to page defragmentation methods of memory, where fragments are rearranged to provide continuous storage spaces for applications. In topology virtualization, "*core defragmentation*", i.e., rearranging cores with similar performance closer without totally messing up communications, facilitates operating systems to accelerate applications with higher degree of parallelism without changing existing allocation strategies.

The question now is how to quantitatively evaluated the virtualized many-core systems, considering both the impact of C2C performance asymmetry and on-chip communications. In the following, we investigate the impact of the above two factors on the performance of parallel programs, and derive an evaluation metric for parallel programming models.

3 Problem Formulation

In order to see how C2C performance asymmetry affects the execution time of parallel applications, we run SPLASH-2 [**?**] benchmark programs on a dual-core Xeon machine, which allows software to set the frequency of each core to 350, 700, 1050, 1400, 1750, 2100, 2450, and 2800 MHz, respectively. The execution time (T) of 2-thread *fft* in different configurations against the reciprocals of the working frequencies of the two cores are plotted as shown in Fig. 3. Note that, results for other SPLASH-2 benchmarks are similar, thus are omitted due to page limits. We use two first-order polynomials (i.e., planes) to fit (in a least-square sense) these points with $\frac{1}{f_1} \geq \frac{1}{f_2}$ and $\frac{1}{f_1} \leq \frac{1}{f_2}$ respectively, as follows:

$$T = \begin{cases} \frac{1.978\times10^7}{f_1} + \frac{3.743\times10^8}{f_2} + 1.289 \times 10^4 & \text{when } \frac{1}{f_2} \geq \frac{1}{f_1} \\ \frac{3.772\times10^8}{f_1} + \frac{1.858\times10^7}{f_2} + 1.119 \times 10^4 & \text{when } \frac{1}{f_2} \leq \frac{1}{f_1} \end{cases}$$

Note that the coefficient of the reciprocal of the slower core is always at least one order of magnitude larger, and thus the effect of the faster core on T vanishes. Also, the coefficients for the slower cores in the two polynomials are roughly the same.

These indicate that the performance of applications running on a pair of cores is constrained by the slower one. Therefore, pairing two cores with significant performance differences diminishes the advantages of faster cores.

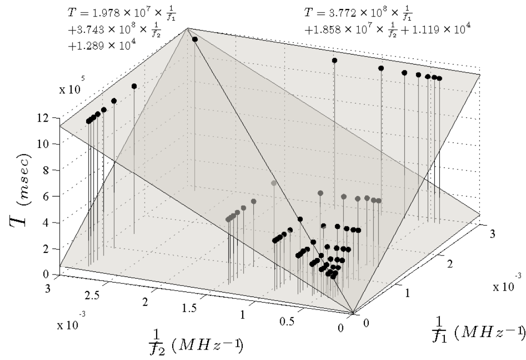
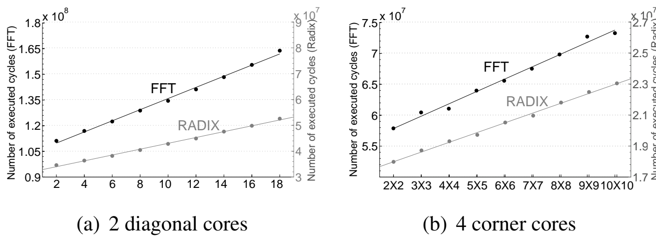


Figure 3. The impact of C2C performance asymmetry on 2-thread *fft*.

On the other hand, in order to see how hop-dependent communication latencies of participating cores affect the execution time of applications, we run SPLASH-2 programs on Simics/GEMS full-system simulator [12]. Simics is used to boot unmodified Solaris 9. 2D mesh NoCs with various scales are modeled using a detailed interconnection network simulator GARNET [13], which is incorporated inside GEMS. The number of executed cycles for *fft* and *radix*, when they are binded to two or four cores with different hop distances, are illustrated in Fig. 4(a) and 4(b). Note that, the 2-thread programs *fft-2* and *radix-2* are running on two diagonal cores on an $n \times n$ mesh, while the 4-thread *fft-4* and *radix-4* are running on the four corner cores, respectively.



(a) 2 diagonal cores (b) 4 corner cores

Figure 4. Effects of hop counts on execution times on an $n \times n$ mesh.

It is clear that the total executed cycles grow linearly with increasing hop counts. Moreover, hop counts affect applications differently, i.e., different slopes as in Fig. 4, because applications can be computation- or communication-intensive.

Furthermore, the above experiments also indicate that the performance of parallel applications not only depends on individual speeds of participating cores, but also the hop distances between them.

Consider a computation with two processes (a master process and a slave process) that is carried out on two neighboring cores. The communication schemes can be buffered or blocking send/receive [14], as illustrated in Fig. 5. The total execution time for the computation can be estimated as:

- buffered send: $\max(C_1, T_1 + C_2 + T_2)$
- blocking send: $T_1 + \max(C_1, C_2 + T_2)$

where C_1 and C_2 are the computation times on $Core$ 1 and $Core$ 2, respectively; and T_1 and T_2 are the communication latenci

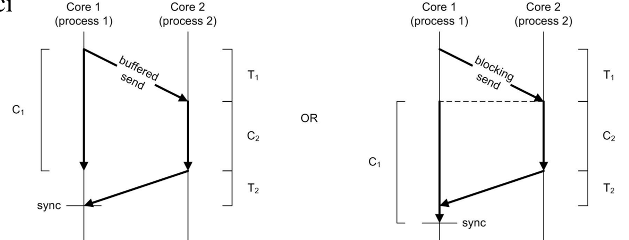


Figure 5. Execution model for buffered and blocking send/receive.

Assume equal partition of a task between the two cores for load balancing, the execution time with *Core 1* as the master and *Core 2* as the slave for buffered send/receive is thus

$$T_{<1,2>} = \max(\frac{1}{f_1}, \frac{1}{f_2} + 2c \cdot hop_{1,2}) \quad (1)$$

where c is an application-dependent constant, and f_1 and f_2 are the corresponding frequencies of the two cores, respectively. It is clear that large performance gap or long hop distances in between a core pair will diminish the advantage of a high performance core.

A new metric called core fragmentation factor (CFF) to evaluate the quality of a virtual topology considering both C2C performance asymmetry and on-chip communication is defined as the largest $T_{<i,j>}$ for all virtually neighboring cores. More precisely, for a given $m \times n$ virtual topology V, $CFF(V) = \max T_{<i,j>}$, where $1 \leq i \leq m$, $1 \leq j \leq n$, and $i,\ j$ are virtual neighbors in V.

4 Performance-Asymmetry-Aware Topology Reconfiguration Algorithm

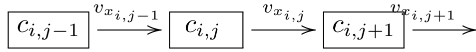
In this section, we present a novel asymmetry-aware reconfiguration algorithm for 2D mesh, called *Bubble-Up* (BU). The purpose of this algorithm is to generate virtual topologies that contain more larger submeshes of high-quality in terms of low *CFF*s. Bubble-Up is based on the following simple analogy: from the discussions in Section 3, we prefer to pair cores with similar performances yet a moderate hop distance as virtual neighbors. This is quite similar to a general physical law in nature, i.e., Newton's law of universal gravitation $F = G\frac{m_1 m_2}{r^2}$ where the affinity between a pair of objects increases as the two masses m_1 and m_2 increase and their distance r decreases. Thus if we consider an $m \times n$ mesh as a cup of liquid with diverse densities, the k faulty cores can be modeled as bubbles that will eventually be pushed up to the surface. Due to the performance differences, as the faulty cores bubble up, non-faulty cores will be drawn together and fill the hollow spaces.

From the above discussions, at each reconfiguration step, the movement of each non-faulty core is driven by three types of forces: *attractions*, *pressures*, and *gravities*. Attractions are used to draw ideal virtual neighbors, i.e., cores with a smaller CFF value, closer; and the other two are introduced to guarantee the stability and the convergence of the reconfiguration algorithm, respectively:

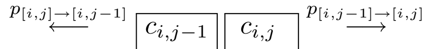
1. *Attraction*: each non-faulty core $c_{i,j}$ is attracted by the 8

cores around it [1]. The strength of the attraction is determined by the reciprocal of (1), i.e. $T^{-1}_{<x,y>}$, and the direction is determined by the relative positions of virtual neighbors. Intuitively, similar to the universal gravities, these attractions draw cores that have small CFF values closer. Since in (1), the hop count is the actual number of hops between cores in the physical topology, when two cores of similar performances are dragged closer virtually, the attraction between them become smaller due to their large actual hop distance. This avoids the case where two remote cores with similar performances become virtual neighbors. We can therefore expect that the entire system reaches an equilibrium where all attractions balance, thus reaching a configuration that minimizes the core defragmentation factor of the reconfigured topology. To facilitate computations of the velocities of a core in x^+- and y^+-directions, we decouple the summation of 8 attractions into x^+- and y^+-components and denote them as $f_{x_{i,j}}$ and $f_{y_{i,j}}$, respectively.

2. *Pressure*: Introducing attractions solely will bring the following anomaly:



In the above figure, suppose that due to attractions, $c_{i,j-1}$, $c_{i,j}$, and $c_{i,j+1}$ have x^+-direction velocities $v_{x_{i,j-1}}$, $v_{x_{i,j}}$, and $v_{x_{i,j+1}}$, respectively, and $v_{x_{i,j-1}} > v_{x_{i,j}} > v_{x_{i,j+1}}$. We do not want $c_{i,j-1}$ to pass all the way through $c_{i,j}$ and $c_{i,j+1}$ (and presumably more) in one iteration, and attracted backward in the next iteration due to the change of attractions. Therefore, in order to guarantee the continuity of the movements of consecutive non-faulty cores and thus the stability of the reconfiguration algorithm, we introduce pressures between non-faulty cores that are virtual neighbors. These pressures follow Newton's Third Law of Motion and are treated as *variables* as shown in the following discussions.



There is no pressure between a pair of cores that has at least one faulty core and thus non-faulty cores can "fill" the hollow spaces left by faulty ones. We denote the pressure from core c_x to core c_y as $p_{x\to y}$, where c_x is one of the four virtual neighbors of c_y.

3. *Gravity:* To guarantee that the faulty cores are eventually "pushed up", we give each non-faulty core a constant gravity, denoted as g, in the y^- direction. Intuitively, again as in most physical processes in nature, non-faulty cores will eventually fill the hollow spaces of faulty cores in order to minimize the potential of these gravities imposed on them. This guarantees the convergence of the algorithm, i.e., we will eventually get an $m' \times n$ virtual mesh without faulty cores. The constant g determines the speed of convergence of the reconfiguration algorithm: for large g that dominates core-pair attractions, non-faulty cores fill the hollow space fast; while for small g, the algorithm converges slowly, but the effects of core-pair attractions will draw cores that reduce CFF closer.

Having introduced the "forces", we are ready to establish the momentum equations that guide the movements of cores. Let $\vec{v}$ be the velocity, and $\vec{f}$ be the summation of forces define above, according to Newton's Second Law of Motion, we have

$$\frac{\partial \vec{v}}{\partial t} = \vec{f} \tag{2}$$

From (2), we have

$$\left(\frac{\partial v_x}{\partial t}\right)_{i,j} = f_{x\,i,j}^{(t)} + p^{(t)}_{[i,j-1]\to[i,j]} - p^{(t)}_{[i,j+1]\to[i,j]} \tag{3}$$

and

$$\left(\frac{\partial v_y}{\partial t}\right)_{i,j} = f_{y\,i,j}^{(t)} - g - p^{(t)}_{[i-1,j]\to[i,j]} + p^{(t)}_{[i+1,j]\to[i,j]} \tag{4}$$

Taking the first order Taylor expansion of the velocity at each core in x- and y-directions at time $t+\Delta t$, we have

$$\begin{cases} v_{x\,i,j}^{(t+\Delta t)} &= v_{x\,i,j}^{(t)} + \left(\frac{\partial v_x}{\partial t}\right)_{i,j}\Delta t \\ v_{y\,i,j}^{(t+\Delta t)} &= v_{y\,i,j}^{(t)} + \left(\frac{\partial v_y}{\partial t}\right)_{i,j}\Delta t \end{cases} \tag{5}$$

Inserting (3) and (4) into (5), and setting $\Delta t = 1$, we have

$$\begin{cases} v_{x\,i,j}^{(t+1)} - p^{(t)}_{[i,j-1]\to[i,j]} + p^{(t)}_{[i,j+1]\to[i,j]} &= v_{x\,i,j}^{(t)} + f_{x\,i,j}^{(t)} \\ v_{y\,i,j}^{(t+1)} + p^{(t)}_{[i-1,j]\to[i,j]} - p^{(t)}_{[i+1,j]\to[i,j]} &= v_{y\,i,j}^{(t)} + f_{y\,i,j}^{(t)} - g \end{cases} \tag{6}$$

where terms on the left are variables and terms on the right are known at time t [2].

After solving the system of linear equations, we obtain the values of $v_{x\,i,j}^{(t+1)}$ and $v_{y\,i,j}^{(t+1)}$, and are able to determine the movement of each non-faulty cores. More specifically, for each faulty core (conceptually a hollow space), we look at the velocities of its four virtual neighbors and swap it with the non-faulty core with the largest velocity towards it. After each reconfiguration step, we obtain a new physical-virtual topology mapping which is used to determine $f_{x_{i,j}}$ and $f_{y_{i,j}}$ for the next iteration step.

We summarize the Bubble-Up algorithm in Algorithm 1.

5 Experimental Results

In this section, we show the effectiveness of the proposed Bubble-Up algorithm by comparing with prior RRCS-gSA algorithm. RRCS-gSA tries to maximize on-chip communication by maintaining the physical regularity of a virtual topology

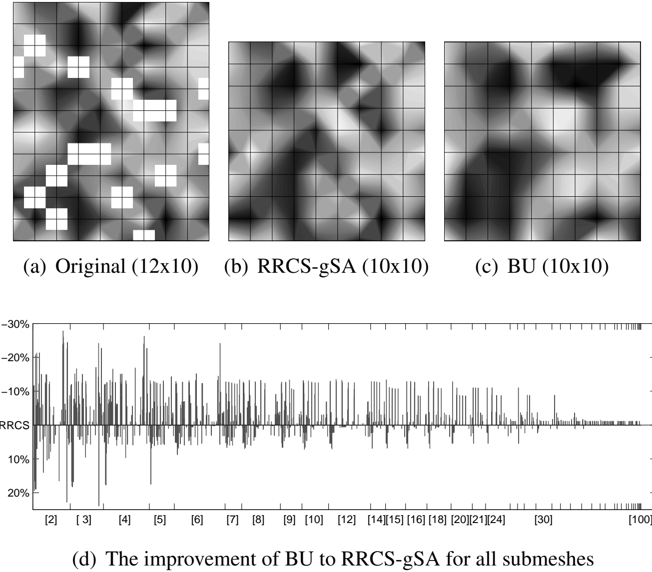
[1] For cores on the boundaries and the corners of the mesh, the numbers are different. Faulty cores are treated equally by using their frequencies in calculating the attractions. Also, one can include more layers of surrounding cores to mitigate locality.

[2] To prescribe boundary conditions, we extend the $m \times n$ mesh to $(m+2) \times (n+2)$ and set the velocities of outermost imaginary cores to 0. Therefore, at each time step, we have $6(m+2)(n+2)$ variables: $2(m+2)(n+2)$ variables for v_x and v_y at time $t+1$, and $4(m+2)(n+2)$ variables for the pressures at each core in 4 directions at time t. In (6), we have $2(m+2)(n+2)$ equations, one for each velocity; also, recall that when we introduce *pressures*, we claim that they follow Newton's Third Law of Motion which gives us another $2(m+2)(n+2)$ equations (with proper boundary conditions on the outermost imaginary cores). For the rest of the $2(m+2)(n+2)$ equations, we resort to the incompressibility constraint commonly used in the area of computational fluid dynamics, i.e., $\nabla \cdot \vec{v} = 0$. The last $2(m+2)(n+2)$ equations only guarantee the solution of the system and are out of the scope of this paper.

Algorithm 1: *Bubble-Up Algorithm*

Input: $m \times n$ mesh with $k < (m - m')n$ faulty cores
Output: $m' \times n$ reconfigured mesh

1. **while** *the lower* $m' \times n$ *mesh contains faulty cores* **do**
2. **solve** the system of linear equations (6);
3. **foreach** *faulty core at position* (i, j) **do**
4. **find** $\max\left\{v_{x_{i,j-1}}^{(t+1)}, -v_{x_{i,j+1}}^{(t+1)}, -v_{y_{i-1,j}}^{(t+1)}, v_{y_{i+1,j}}^{(t+1)}\right\}$, i.e., velocities of 4 virtual neighbors of (i, j) (omit faulty ones) towards it;
5. **swap** faulty core (i, j) and the non-faulty virtual neighbor with the largest velocity;
6. **end**
7. **update** the physical-virtual mapping;
8. **caculate** all $f_{x_{i,j}}^{(t+1)}$ and $f_{y_{i,j}}^{(t+1)}$ for the next iteration by the reciprocal of (1) based on the mapping;
9. **end**



(a) Original (12x10) (b) RRCS-gSA (10x10) (c) BU (10x10)

(d) The improvement of BU to RRCS-gSA for all submeshes

Figure 6. Comparison of RRCS-gSA and Bubble-Up algorithms.

in row and in column unit. Please refer to [6] for details. In our experiments, for core fragmentation factor, the application-dependent constant c is tuned to give computation and communication the same weights, since many-core processors are for general-purpose computing.

5.1 Experiment I

In this experiment, we generate a 12×10 2D mesh with core performances varying from 1.8Ghz to 2.2Ghz as shown in Fig. 6(a). In the figure, regions of cores with higher performance are darker, and the white squares denote the 16 randomly generated faulty cores. Fig. 6(b) shows the virtual mesh generated by RRCS-gSA. Since RRCS-gSA does not consider core performance asymmetries and only optimizes on-chip communications, low-performance cores (lighter regions) scatter all over the resultant 10×10 virtual mesh. According to the discussions in Section 3, this increases the probability of submeshes containing low performance cores that become bottlenecks.

In contrast, in Fig. 6(c), as non-faulty cores fall down to fill the spaces of faulty ones, attractions (1) among them draw cores that reduce CFF closer, forming large clusters of cores with similar performances without severely changing the communication topology.

Fig. 6(d) shows the CFF reductions of BU over RRCS-gSA. The x-axis indicates the size of all kinds of submeshes, ranging from 2 (i.e., 1×2 or 2×1) to 100 (i.e., 10×10) in the generated virtual topologies. It is very clear that BU is superior as more submeshes get improved (upturned lines). We can also observe that when the scale increases, especially submeshes larger than 30 as in the figure, much more submeshes are better in BU than in RRCS-gSA. Therefore, highly parallelized applications can expect higher performance when running on the virtual topologies generated by BU.

We further choose the top 500 best submeshes [3], i.e., submeshes with the lowest core fragmentation factor. The metric interval from the best to the worst is divided into three classes, i.e., Class 1, 2, and 3. Fig. 7(a), 7(b), and 7(c) show the distributions of submeshes in each class, respectively. In the figures, for the best Class 1, the maximum submesh size (i.e., 9) is smaller than other two classes. This is reasonable since lager scale submeshes probably contain more slower cores. BU is superior to RRCS-gSA: the biggest submesh size in Class 1 is 9 for BU while 5 for RRCS-gSA. Similar results can be observer from Fig. 7(b) and 7(c). In addition, for the best Class 1, there are much more submeshes in BU than in RRCS-gSA; although for Class 2 and 3, RRCS-gSA becomes better, BU still dominates. We can conclude that virtual topologies generated by BU have more and larger high-quality submeshes than RRCS-gSA.

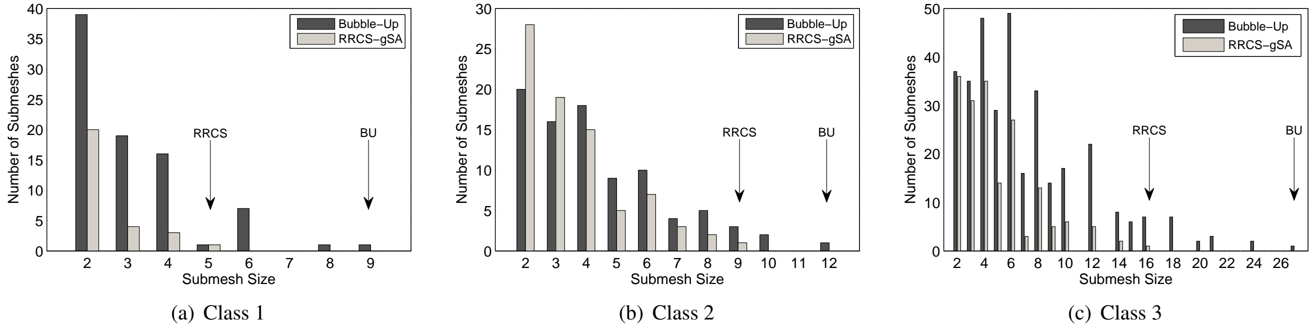
5.2 Experiment II

To show the impact of virtual topologies on parallel applications, in this experiment, we run SPLASH-2 benchmarks on top of the Simics/GEMS 16-core platform. The original core failure map and virtual topologies generated by BU and RRCS-gSA are shown in Fig. 8(a). Note that RRCS-gSA generates a virtual topology with smaller neighboring hop counts, while BU puts cores that reduce core fragmentation factors together.

Eight 2-thread applications are allocated on eight 2×1 submeshes. For example, *fft-2* are running on the upper-left two cores as shown in the middle of Fig. 8(a). Four 4-thread applications are running on the four quadrants, while a 16-thread *radix* covers the entire virtual topologies.

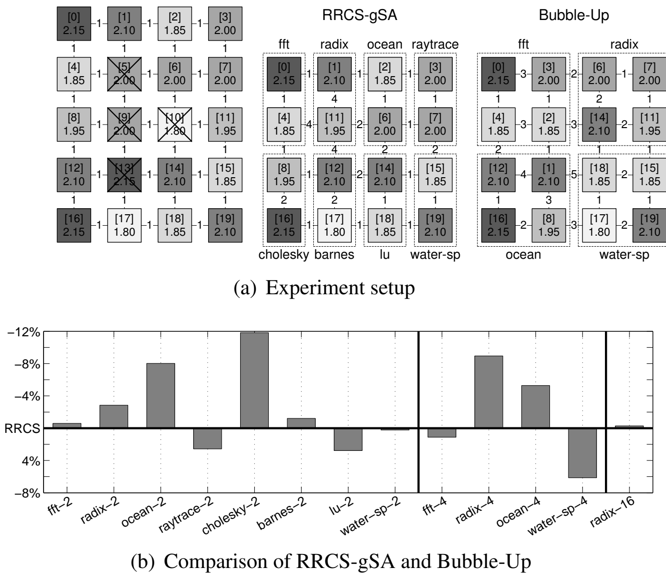
From Fig. 8(b), we can see that three 2×1 submeshes, running *cholesky-2*, *ocean-2*, and *barnes-2*, respectively, achieve much higher performance, while the virtual topology generated by RRCS-gSA contains two 2×1 submeshes with only slightly (less than 3%) application performance benefit. These can be easily verified from the configurations of virtual topologies in Fig. 8. For scale 2×2, BU contains 2 better submeshes, one of which achieves 9% performance improvement, while RRCS-gSA contains only 1 better. For *radix-16*, all available cores are used. The performance will be mainly affected by on-chip communication. As BU maintains reasonable hop distances, the performance are roughly equivalent to that of RRCS-gSA.

[3] The total number of possible submeshes of size larger than 1×1 in a 10×10 mesh is $\binom{10+1}{2} \times \binom{10+1}{2} - 100 = 2925$.



(a) Class 1 (b) Class 2 (c) Class 3

Figure 7. Comparison of submesh distributions generated by RRCS-gSA and Bubble-Up.



(a) Experiment setup

(b) Comparison of RRCS-gSA and Bubble-Up

Figure 8. Execution time comparison of RRCS-gSA and Bubble-Up.

6 Conclusion

Topology virtualization techniques are proposed for NoC-based many-core processors with core-level redundancy, to isolate hardware changes caused by on-chip defective cores. Prior work focuses on homogeneous cores with symmetric performance and optimizes on-chip communications only. However, core-to-core performance asymmetry due to manufacturing process variations poses new challenges for the construction of virtual topologies. OS and programmers typically allocate tasks to continuous cores. Without considering C2C performance asymmetry, slower cores would scatter all over a virtual topology. Thus, applications with higher degree of parallelism are probably assigned to a region containing many slower cores that become bottlenecks.

In this paper, we introduce a new metric called the *core fragmentation factor* to measure the overall performance of virtualized performance-asymmetric many-core processors. Based on this metric, we propose a novel asymmetry-aware *Bubble-Up* reconfiguration algorithm. Experimental results show that such algorithm can generate efficient virtual topologies that contain more and larger high-quality submeshes, thus accelerate applications with higher degree of parallelism, without changing existing allocation strategies for OS.

References

[1] S. Borkar, “Thousand core chips: a technology perspective,” in *Proc. of the 44th Design Automation Conference*, 2007, pp. 746–749.

[2] W. J. Dally and B. Towles, “Route packets, not wires: on-chip inteconnection networks,” in *Proc. of the 38th Design Automation Conference*, 2001, pp. 684–689.

[3] E. Sperling, “Turn down the heat please,” 2006. [Online]. Available: http://www.edn.com/article/CA6350202.html

[4] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger, “Exploiting microarchitectural redundancy for defect tolerance,” in *Proc. of the 21st International Conference on Computer Design*, 2003.

[5] L. Zhang, Y. Han, Q. Xu, and X. Li, “Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology,” in *Proc. of the conference on Design, automation and test in Europe*, 2008, pp. 891–896.

[6] L. Zhang, Y. Han, Q. Xu, X. Li, and H. Li, “On topology reconfiguration for defect-tolerant noc-based homogeneous manycore systems,” *IEEE Transactions on Very Large Scale Integration Systems*, vol. 17, 2009.

[7] K. Li and K. H. Cheng, “A two dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system,” in *Proc. of the ACM annual conference on Cooperation*, 1990, pp. 22–27.

[8] P. J. Chuang and N. F. Tzeng, “Allocating precise submeshes in mesh connected systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 2, pp. 211–217, 1994.

[9] B. S. Yoo and C. R. Das, “A fast and efficient processor allocation scheme for mesh-connected multicomputers,” *IEEE Trans. Comput.*, vol. 51, no. 1, pp. 46–60, 2002.

[10] S. R. S. et al, “Varius: A model of process variation and resulting timing errors for microarchitects,” *IEEE Transactions on Semiconductor Manufacturing*, vol. 21, 2008.

[11] T. Karnik, “Probabilistic and variation-tolerant design: Key to continued moore’ s law scaling,” in *Invited talk in ACM/IEEE Intl TAU Workshop on Timing Issues*, 2004.

[12] [Online]. Available: http://www.cs.wisc.edu/gems/

[13] L.-S. Peh, N. Agarwal, N. Jha, and T. Krishna, “Garnet: A detailed on-chip network model inside a full-system simulator,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 33–42.

[14] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, 2003.