Balancing On-Chip Network Latency in Multi-Application Mapping for Chip-Multiprocessors

Di Zhu, Lizhong Chen, Siyu Yue, Timothy M. Pinkston, Massoud Pedram
Ming Hsieh Department of Electrical Engineering
University of Southern California
Email: {dizhu, lizhongc, siyuyue, tpink, pedram}@usc.edu

***Abstract*—As the number of cores continues to grow in chip multiprocessors (CMPs), application-to-core mapping algorithms that leverage the non-uniform on-chip resource access time have been receiving increasing attention. However, existing mapping methods for reducing overall packet latency cannot meet the requirement of balanced on-chip latency when multiple applications are present. In this paper, we address the looming issue of balancing minimized on-chip packet latency with performance-awareness in the multi-application mapping of CMPs. Specifically, the proposed mapping problem is formulated, its NP-completeness is proven, and an efficient heuristic-based algorithm for solving the problem is presented. Simulation results show that the proposed algorithm is able to reduce the maximum average packet latency by 10.42% and the standard deviation of packet latency by 99.65% among concurrently running applications and, at the same time, incur little degradation in the overall performance.**

***Keywords*—On-chip networks; chip-multiprocessors; application mapping; balanced on-chip latency**

I. Introduction

With tens to possibly hundreds of cores integrated in current and future multiprocessor systems-on-chips (MPSoCs) and chip-multiprocessors (CMPs) [12], networks-on-chips (NoCs) have been proposed as the primary shared media for providing high-performance and scalable communication [6]. Meanwhile, since a single application is unlikely to use up all the computing resources on a many-core chip, multiple applications can usually run concurrently on the system. However, due to the topological layout of the cores (e.g., in tile-based mesh topology), on-chip access latencies to cache and memory controllers are not necessarily the same when initiated from different source locations. It is important to account for this on-chip delay characteristic when mapping applications onto cores to optimize the system performance.

While the issue of application mapping has been receiving increasing attention in many-core chip designs, the problem of mapping multiple applications to CMPs presents several new challenges. First of all, mapping techniques proposed thus far are mainly for MPSoCs [11][13][14][20][21] which, unfortunately, cannot be applied directly to CMPs due to their inherent difference: In MPSoCs, the shared cache/memory is clustered into *some* of the tiles while other tiles contain heterogeneous IP blocks with specific functionalities. In CMPs, the shared cache is distributed to all tiles, each of which contains a homogeneous general-purpose processor core and only some of which have a memory controller. The task of application mapping in a MPSoC consists of assigning caches, IP cores, and other customized blocks to tiles, whereas the task of application mapping in a CMP consisting of assigning the running threads to the fixed and homogeneous physical cores. Consequently, the latency model on which the mapping algorithms are based for MPSoCs no longer holds for CMPs.

Moreover, when mapping multiple applications to CMPs, not only should the overall on-chip latency be reduced as in the single application case, the mapping process should also balance the average packet latency experienced by different applications. That is, each application should expect near equally minimized on-chip network latency as compared with other applications when accessing resources (i.e., shared cache and memory), regardless of where the threads of the application are mapped [16]. Balancing minimized on-chip latency in multi-application mapping is, in fact, much needed in CMPs for three major reasons. First, at the user level, the application-to-tile mapping should be transparent to end users to provide quality-of-service guarantees. This is particularly important when multiple users pay for service in a shared environment, as it is unacceptable if the imbalance of latencies introduced in the mapping process results in service agreement violations for one or more users. Second, at the system level, many techniques in shared cache and memory systems [8][9][22] have been proposed to provide equal or differentiated services among applications, all of which assume uniform on-chip latency. If the on-chip latencies among different applications are not well balanced, the effectiveness of these techniques will be severely affected, if usable at all. Third, at the on-chip network level, several architectural techniques have been proposed to augment the design of router arbitrators and network topologies to provide on-chip latency balance, but at the cost of increased complexity of routers [10] and additional traffic pattern restrictions [16]. If the NoC latency can be balanced early on at the mapping stage, these hardware and software overheads can be greatly mitigated or even entirely avoided.

Although balancing the on-chip latency is a desirable and necessary feature, its realization in multi-application mapping is not straightforward. Balancing packet latencies among applications is often conflict with minimizing overall packet latency of all applications. On the one hand, mapping methods which have minimization of the overall latency as the sole objective are actually counter optimal in terms of latency-balancing, as shown in Section II. On the other hand, a mapping method is not useful if it focuses on balancing latency but leads to greatly increased overall packet latency of on-chip networks. In addition, it is observed that certain core locations have low access latency for one type of traffic (e.g., cache traffic) but have high access latency for another type of traffic (e.g., memory controller traffic), which complicates the design of an effective latency-balancing algorithm. Indeed, the proposed balanced-latency mapping problem is proved to be NP-complete in Section III, indicating that it is quite challenging to find an efficient solution.

In this paper, we address the important issue of achieving balanced on-chip latencies with performance-awareness in the multi-application mapping of CMPs. We use the metric of min-max (i.e., minimizing the maximum of) average packet latency of the applications as the objective function to take into account both balanced and reduced latency aspects, and propose an efficient heuristic-based algorithm called *sort-select-swap* to find the effective mapping solutions. The key idea is to first implement "coarse tuning" in the mapping process by considering the dominant cache traffic, and then conduct "fine tuning" by taking memory traffic into consideration and performing sliding-window based swaps. The main contributions of this paper are the following:

- Identifying the inability of traditional performance-oriented mapping methods to provide balanced latency;
- Formulating the performance-aware latency-balancing mapping problem for CMPs, and proving its NP-completeness;
- Proposing an efficient heuristic-based algorithm with a time complexity of $O(N^3)$ where N is network size.

The rest of this paper is organized as follows. Section II provides more information on the background and the motivation for implementing balanced latencies in NoC-based application mapping. Section III formulates the mapping problem for multiple applications and proves its NP-completeness. Section IV describes the details of our proposed algorithm. Section V presents the simulation results and, finally, Section VI concludes the paper.

II. Background and Motivations

This section first summarizes the related work in application mapping for NoCs, and then it describes the structure of NoC-based CMPs and presents the latency models for shared cache requests and memory controller requests. We highlight that current performance-oriented mapping methods tend to exacerbate the imbalance in the on-chip latencies experienced by different applications. This motivates finding more comprehensive mapping solutions that take the requirements of balanced latency into account as well.

A. *Related Work*

As the number of cores continues to grow with increased non-uniformity within the same chip, the importance of application mapping has been rising rapidly. Earlier work on application mapping mostly targets improving the mapping effectiveness in NoC-based MPSoCs. Murali *et al.* focus on overall latency minimization under minimum routing and traffic splitting for SoCs [20]. Hu *et al.* address energy consumption in the mapping task for tile-based MPSoC architectures [13]. Hansson *et al.* present a combined mapping and routing approach for MPSoCs [11]. Jang *et al.* take into consideration heterogeneous MPSoC architectures and propose efficient mapping solutions for various chip layouts [14]. These techniques assume MPSoC systems which have different characteristics than the targeted CMP systems considered in this work.

Recently, researchers have started to explore the mapping problem in CMPs. Chen *et al.* present a set of comprehensive mechanisms that optimize the mapping of an application onto NoC-based CMPs under the assumption that only one application is running on the chip at a time [3]. However, with increasing number of cores, chip resources can be better utilized if the chip has multiple applications running simultaneously. Murali *et al.* present an efficient method to map multiple use-cases onto MPSoCs rather than CMPs [21]. In contrast, our work aims to address the multi-application mapping problem in CMPs to both optimize overall performance and balance NoC latencies among the applications.

Many techniques have been proposed to provide quality-of-service support for various system components including cache, memory and on-chip networks [7][8][9][10][16][22]. This set of research has very different objectives than balanced NoC latency and, therefore, are orthogonal and complementary to our work. In fact, it is possible to integrate the approach developed in this work with previous mechanisms to further improve the service quality, which can be investigated in the future.

B. *NoC-based CMP Architecture*

Figure 1 shows a typical example of 64-tile multiprocessor with mesh-based NoC structure. Each tile is comprised of a processing core, a private L1 cache, and a slice of shared L2 cache bank. The shared L2 cache is distributed among all the tiles. In most commercial CMPs, when a data block is fetched from memory, the L2 cache bank in which to place the data is determined by hashing on the lower-order bits of the data address [23]. Routers are interconnected to form a mesh network, and tiles are connected to routers via a network interface (NI). Each of the four tiles in the corners (shown as four grey tiles in the figure) includes a memory controller in addition to the regular core/cache structure.

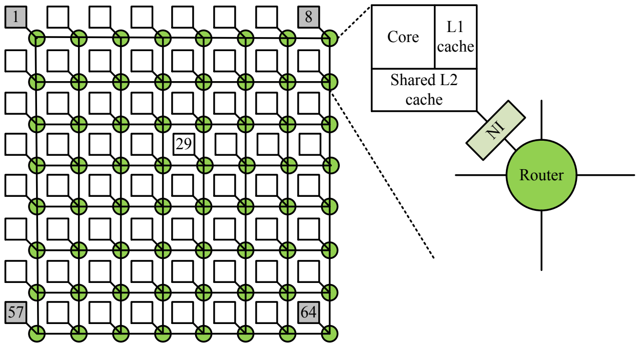


Figure 1. 64-core NoC-based CMP architecture.

When the processing core has a data request, no further packet is needed if the request hits in the private L1 cache. Otherwise, two types of traffic can be generated depending on where the data is located: cache traffic (data is still on-chip) and memory controller traffic (data needs to be fetched from off-chip main memory). Cache traffic includes the requesting packets from the processing core to an L2 cache bank (this packet contains the address of the required data, and is directed towards the tile that has the L2 cache bank to which the address is hashed), the checking/forwarding packets from the L2 cache bank to other private L1 caches, and the reply packets from L2 cache bank to the requesting processing core. In all of these cases, either the source tile or the destination tile is an L2 cache bank.

In the case of memory controller traffic, a requesting packet from the processing core is forwarded to one of the four memory controller tiles through the on-chip network. To increase the efficiency, the forwarding follows the proximity principle, i.e., the packet is sent to the nearest memory controller tile out of the four tiles. Data are then fetched from the main memory and returned to the memory controller after a certain number of cycles.

C. *Packet Latency Models*

To explain the mesh-based NoC latency model, we first introduce the tile numbering rule adopted in this paper. The number $k \in [1, N]$ of a tile is determined by:

$$k = (i_k - 1) \times n + j_k \tag{1}$$

where i_k, j_k are the row number and column number, respectively, and n is the number of tiles in a row. For example, the 29-th tile in Figure 1 (where $n = 8$) is located at the fourth row (from the top), fifth column (from the left).

We derive the latency model based on [5] to calculate the service latency TD_k of a packet generated at the k-th tile and heading for the k'-th tile:

$$TD_k = H_k(k') \times \left(td_r + td_w + td_q\right) + td_s \tag{2}$$

where $H_k(k')$ is the number of hops before a packet reaches its destination tile k'. The terms td_r and td_w are the per-hop latency for the router and wire, respectively, td_q is the average queuing latency per hop. Unlike off-chip networks with pin limitations, on-chip networks typically have very wide link-width (e.g., 128-bit or 256-bit) with multiple virtual channels per link, so the queuing latency td_q is usually very small (0~1 cycles as observed in the simulation). The serialization latency, td_s, is calculated as the ratio of the packet length to the channel bandwidth, which is pre-determined for a given packet format and NoC structure. Note that if the destination tile to which the data address hashes happens to be the same as the source tile, there is no serialization latency since no network communication will be required. To avoid deadlocks, dimension-order routing (i.e., XY routing) is used to minimize design effort and implementation cost [5].

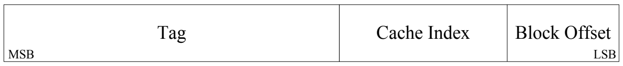


Figure 2. Physical memory address breakdown.

As mentioned, the hashing for the shared L2 cache banks uses lower bits of the data addresses (i.e., cache index) as shown in Figure 2. For example, in a 16 MB L2 cache with a block size of 64 bytes, the block offset is Bit 0 to Bit 5. The next lowest 6 bits, Bit 6 to Bit 11, are the cache index used to hash and decide which tile the cache line is placed. Hence, any consecutive chunk of 64-byte physical address (i.e., one cache line) is uniformly distributed across all the L2 cache banks. This means that, for the cache traffic, the source tile or the destination tile has statistically the same chance to be any tile in the network (including itself). Therefore, for a CMP with $N = n^2$ tiles, the average number of hops $\overline{HC_k}$ of a cache traffic packet generated at the k-th tile before arriving at its destination tile is calculated by:

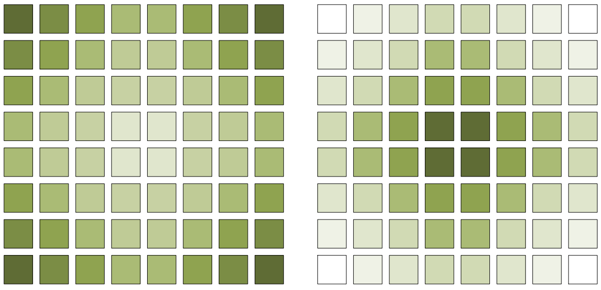
$$\overline{HC_k} = \frac{1}{N}\sum_{i=1}^{N} H_k(i) \tag{3}$$

The value of $\overline{HC_k}$ is smaller for tiles in the center and larger for tiles in the corners. For example, for a corner tile (e.g., tile number 1) on the CMP shown in Figure 3, $\overline{HC_1} = 7$ and for a central tile (e.g., tile number 28) $\overline{HC_{28}} = 4$. Consequently, the threads mapped onto the central tiles have smaller average L2 cache latency than the tiles closer to the perimeter, as shown in Figure 3a), where darker areas indicate tiles with larger packet latencies.

As also mentioned, memory controller packet forwarding is based on the proximity principle. The whole chip is divided into four quadrants relative to its center. All packets generated by tiles in one quadrant are sent to the memory controller in that quadrant. Therefore, the average number of hops $\overline{HM_k}$ for a memory controller request packet generated at the k-th tile is calculated by:

$$\overline{HM_k} = \min\{i_k - 1, n - i_k\} + \min\{j_k - 1, n - j_k\} \tag{4}$$

The average on-chip latency of memory controller access is smaller for tiles closer to the corners, as shown in Figure 3b).



(a) L2 cache delay per packet (b) memory controller delay per packet

Figure 3. Packet latencies on an 8-by-8 mesh network.

With the above expression of $\overline{HC_k}$ and $\overline{HM_k}$ in combination with equation (2) for the latency model, two arrays $\{TC(k)\}$ and $\{TM(k)\}$ can be generated to estimate the on-chip network latency of cache traffic and memory traffic originating from the k-th tile, respectively.

D. *Imbalance in Traditional Algorithms*

While on-chip latency balancing is an important design aspect in CMPs, traditional application mapping algorithms that target minimizing the overall packet latency of all the applications are actually counter optimal in terms of latency-balancing. This is because they map the applications with higher data access rates to the tiles with smaller average on-chip latencies, which is more beneficial in minimizing overall packet latency than mapping applications with less traffic to those tiles. This increases the latencies of those low traffic-load applications and results in significantly imbalanced per-application average packet latencies (or APLs for short).

To illustrate this effect, we conduct evaluations on four different configurations (i.e., sets) of input applications, namely C1, C2, C3, and C4. Each configuration contains four applications, each consisting of 16 threads. The application traces are gathered from running the PARSEC 2.0 benchmarks [18] (detailed evaluation methodology can be found in Section V.A). The mapping task is to map these 4x16 threads onto an 8-by-8 mesh network.

A large number (> 10^4) of random mappings are generated first, and for each mapping we derive the maximum APL among the applications (i.e., each application has an APL, and the largest one among all applications is the max-APL) as well as the standard deviation of the applications' APLs (referred to as dev-APL). Larger max-APLs and dev-APLs indicate worse imbalance among the four applications. We then calculate the average of the max-APL and dev-APL for all the random mappings. These two values are adopted to denote the average level of latency balance between applications' APLs for a randomly generated mapping.

Global stands for a mapping method that optimizes the overall network latency of all applications, i.e., reduces the global APL (referred to as g-APL). The g-APL is the average latency of all packets generated, calculated by the sum of all the packet latencies divided by the total communication volume. Table 1 shows the g-APL, max-APL, and dev-APL values for both the aforesaid random average result and *Global*. It can be seen that although the g-APL is reduced by 4.78% through the *Global* method compared to the random average, the max-APL is increased by 9.85% and the dev-APL is three to four times that of the random result. This means that the *Global* algorithm makes the APL of one or more of the applications dramatically larger in order to reduce the global APL. Consequently, the *Global* mapping reduces overall packet latency but at the cost of greatly exacerbating the imbalance among applications.

As an illustrating example, Figure 4 shows the *Global* mapping results of the four applications in the C1 configuration, each application containing 16 threads. The four applications are assigned with ID 1 to 4 in ascending order of total communication rates (i.e., Application 1 has the lightest traffic). Note that the memory controller traffic takes only a small fraction of overall requests for each application, thereby making the cache communication latency the dominant factor of the application APLs. As we can see from the mapping results, the threads of Application 1 are assigned tiles with the worst average L2-cache access time, resulting in an APL of 25.15 cycles, which is 17.80% more than the overall average APL of 21.35 cycles. This clearly demonstrates that a mapping algorithm that aims to reduce overall packet latency is, in fact, counter-optimal in terms of latency balancing among the multiple simultaneously running applications.

TABLE 1. IMBALANCE EXACERBATION BY GLOBAL OPTIMIZATION.

	g-APL (cycles)		max-APL (cycles)		dev-APL	
	Random	***Global***	***Random***	***Global***	***Random***	***Global***
C1	22.63	21.35	22.76	25.15	0.534	2.09
C2	22.58	21.63	22.71	24.63	0.547	1.63
C3	22.70	21.55	22.77	25.15	0.535	1.88
C4	22.54	21.60	22.66	24.93	0.542	1.77
Avg	22.61	21.53	22.73	24.97	0.54	1.84

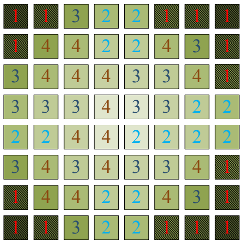


Figure 4. Global mapping results of C1.

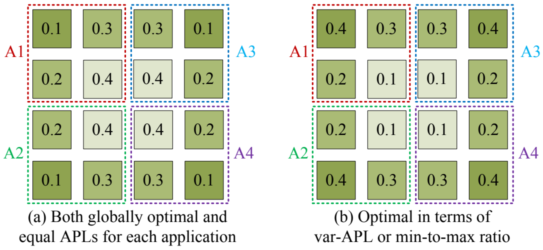
III. ON-CHIP LATENCY BALANCING MAPPING

This section first analyzes various potential metrics for latency balancing and illustrates that the max-APL should be selected as the criterion in the proposed problem formulation. We then state the formulation of the balanced latency mapping problem and prove its NP-completeness.

A. Metrics of Latency Balancing

As mentioned before, latency balancing in multi-application mapping strives to minimize the variation in average packet latency (APL) across applications while keeping APL low, regardless of where the threads of the applications are mapped. Nevertheless, the overall performance, i.e., the global APL, should also be taken into consideration, otherwise the mapping method will introduce large total latency.

Prior to the max-APL, we have investigated two other popular metrics: the standard deviation and the ratio of minimum to maximum of the APLs [25]. Unfortunately, they both suffer from one weakness if used as the objective function: optimizations based on these two objectives cannot ensure overall network performance and may result in such a solution that makes the APLs of each of the applications equally large. The example below illustrates this.



(a) Both globally optimal and equal APLs for each application

(b) Optimal in terms of var-APL or min-to-max ratio

Figure 5. Comparison of two optimal mapping methods.

Assume that there are four applications, each with four threads, totaling 16 threads to be mapped onto the 16 tiles of a 4-by-4 mesh. Suppose the four threads of Application 1 have an L2 cache access rate of 0.1, 0.2, 0.3, and 0.4, respectively. The same are the thread access rates of Applications 2 to 4. For simplicity, suppose all the applications require zero memory accesses. Assume td_r=3, td_w=1, and $td_s = 1$. An optimal solution can be easily obtained as shown in Figure 5(a), which achieves the globally minimal APL as well as equal APLs (10.3375 cycles) among Applications 1 to 4. However, if we choose the standard deviation of the four APL values or the min-to-max ratio as the objective, we find that Figure 5(b) is also among the optimal mappings with regard to the corresponding standard deviation and min-to-max objectives: the standard deviation is zero and the min-to-max ratio has a maximum value of 1. As can be seen, although all the applications have the same APL, they experience equally bad latencies with an APL of 11.5375 cycles, meaning that standard deviation and min-to-max cannot minimize overall packet latency while achieving balanced APL.

To avoid the drawbacks of the above two metrics as the objective function, we adopt the max-APL, which uses the maximum of all applications as the metric. By minimizing the max-APL, both the global performance and the balance among individual applications are taken into consideration, as it prevents any of these applications from having a significantly large latency. Note that although standard deviation may not be suitable as objective function in designing an algorithm, it can still be used as an evaluation metric for mapping solutions to reflect the degree of balance.

B. Problem Statement

Suppose a given NoC-based CMP has N tiles and a set of applications $\{a_i\}$ where $i \in \{1,2,\dots,A\}$ with a total number of threads N. Each thread is mapped to one tile on the chip[1]. Assuming that the threads and tiles are indexed from 1 to N, the problem of application mapping is to find a permutation $\pi(j) = k$ where $j, k \in \{1,2,\dots,N\}$ denoting that the j-th thread is mapped to the k-th tile.

There are two parameters regarding each thread of an application: shared cache request rate when the requested data is on-chip and memory controller request rate when the requested data is off-chip. The request rate is defined by the number of request packets per unit time. Let c_j denote the cache request rate of the j-th thread, and m_j denote the memory controller request rate of the j-th thread. For simplicity, we define that i-th application a_i consists of the threads indexed from $N_{i-1}+1$ to N_i ($N_0 = 0$ and $N_A = N$). For each tile of the NoC, we consider $TC(k)$ and $TM(k)$ as explained in Section II.C. Then, with a given mapping solution $\pi(j)$, the APL of application a_i is calculated by

$$d_i = \frac{\sum_{j=N_{i-1}+1}^{N_i}\left(c_j TC\big(\pi(j)\big) + m_j TM\big(\pi(j)\big)\right)}{\sum_{j=N_{i-1}+1}^{N_i}\left(c_j + m_j\right)} \qquad (5)$$

where $c_j TC\big(\pi(j)\big)$ is the total packet latency caused by cache accesses when thread j is mapped to $\pi(j)$ and, similarly, $m_j TM\big(\pi(j)\big)$ is the total packet latency due to memory controller accesses. Therefore, the problem of latency balanced mapping is to minimize the maximum of each application's APL:

$$d_{\max} = \max\{d_i\} \qquad (6)$$

Formally, we formulate the *On-chip latency Balanced Mapping (OBM)* problem as follows:

Given:

1) The number of tiles (or threads) N;
2) Application set $\{a_i\}$, where $i = 1,2,\dots,A$, the i-th application a_i consists of the threads indexed from $N_{i-1}+1$ to N_i, where $N_0 = 0$ and $N_A = N$;
3) The L2 cache communication array $C = \{c_j\}$, where c_j indicates the request rate of L2 cache of the j-th thread;
4) The memory controller communication array $M = \{m_j\}$, where m_j indicates the memory controller request rate of the j-th thread; and
5) Arrays $\{TC(k)\}$ and $\{TM(k)\}$, denoting the APLs from the k-th tile to the distributed L2 cache and to the memory controller, respectively;

Find: Thread-to-tile mapping $\pi(j) = k$, where $j, k \in \{1,2,\dots,N\}$ and

Minimize: the max-APL

$$d_{\max} = \max_{i=1,2,\dots,A}\{d_i\} \qquad (7)$$

where d_i is the APL of the i-th application defined in (5). Given router parameters in (2), $TC(k)$ and $TM(k)$ are dependent on the position of the k-th tile only.

C. NP-Completeness of OBM

Theorem: the formulated *OBM* problem is NP-complete (NPC).

Proof: In order to prove the NP-completeness of *OBM*, we first transform it into a decision problem:

Decision version of OBM (*DOBM*): With the conditions given in the previous Section III.B, does there exist a mapping solution that makes the APL of each application no larger than a given value γ?

In order to prove the NP-completeness of *DOBM*, we need to prove the following two statements:

1) $DOBM \in NP$;
2) For a known NPC problem L, we have $L \le_P DOBM$.

We first prove that within polynomial time of calculation we can verify whether a given mapping solution satisfies the *DOBM* problem. It takes $O(N)$ calculations to get the APLs for all the applications, and there are A applications requiring $O(A)$ times of comparisons with the threshold value γ. The total number of calculations is therefore $O(A+N) = O(N)$, thus proving $DOBM \in NP$.

We next prove that a known NPC problem L is polynomial-ly reducible to *DOBM*. We adopt a well-known NPC

[1] If the number of threads N' is less than N, we add $(N - N')$ pseudo threads with zero memory controller traffic to those applications and solve the same problem. A more generalization would be for multiple threads to map to one tile. This is not considered in this paper.

problem, the set-partition problem [4] as L in our proof. It is stated as follows: Given a set of numbers $S = \{s_k\}, k \in \{1,2,\ldots,N\}$, does there exist two sets A_1 and A_2 with equal size, satisfying $A_1 \cup A_2 = \{1,2,\ldots,N\}$ such that $\sum_{k\in A_1} s_k = \sum_{k\in A_2} s_k$?

Assume we have a subroutine Y that solves *DOBM*, i.e., Y returns whether there exists such a mapping that the APLs of all applications are no larger than γ. In order to solve the above problem L, we set up a *DOBM* problem of the following form: Build an N-tile chip such that the set of the APLs of each tile's L2 cache accesses is equal to S, i.e., $TC(k)$ is equal to s_k in the set-partition problem. Assume there are a total of two applications with equal size, a_1 and a_2, making $N_1 = N_2 = N/2$. Then, $\forall i, j, k$, assume $TM(k) = 0, c_{ij} = 1$. In this simpler version of the *DOBM* problem, the APLs of a_1 and a_2 become

$$d_1 = \frac{\sum_{j=1}^{N/2} s_{\pi(j)}}{N/2}, d_2 = \frac{\sum_{j=N/2+1}^{N} s_{\pi(j)}}{N/2} \tag{8}$$

We then call the subroutine Y to find if there exists a mapping $j \to \pi(j)$ so that each application's APL is no larger than γ, where

$$\gamma = \frac{1}{N}\sum_{k=1}^{N} TC(k) \tag{9}$$

Note that γ is constant for a given chip layout. If Y holds, i.e., $d_1 \le \gamma$ and $d_2 \le \gamma$, we have $\frac{1}{2}Nd_1 \le \frac{1}{2}\sum_{k=1}^{N} TC(k)$ and $\frac{1}{2}Nd_2 \le \frac{1}{2}\sum_{k=1}^{N} TC(k)$. As $\frac{1}{2}Nd_1 + \frac{1}{2}Nd_2 = \sum_{k=1}^{N} TC(k)$, we conclude that $\frac{1}{2}Nd_1 = \frac{1}{2}Nd_2 = \frac{1}{2}\sum_{k=1}^{N} TC(k)$. Therefore, if Y holds, it means that

$$\exists \pi(j)\ s.t. \sum_{j=1}^{N/2} s_{\pi(j)} = \sum_{j=N/2+1}^{N} s_{\pi(j)} = \gamma \times \frac{N}{2} = \frac{1}{2}\sum_{k=1}^{N} s_k \tag{10}$$

L holds if and only if Y holds. The solutions to the two subsets for L are:

$$A_1 = \{\pi(j) | j = 1, \ldots, N/2\}; A_2 = \{\pi(j) | j = N/2+1, \ldots, N\} \tag{11}$$

Subroutine Y is called only once, proving $L \le_P DOBM$. Therefore the NP-completeness of *DOBM* is proved, and equivalently the *OBM* problem is NPC. ■

IV. Proposed Algorithm

The NP-completeness of the *OBM* problem motivates us to explore an efficient heuristic algorithm. Some of the previous research efforts on NoC mapping problems have tried general neighborhood search algorithms such as simulated annealing or genetic search [14][17], but these two algorithms are too time-consuming to reach a satisfying solution.

Based on the characterizations of NoC-based CMPs, we present an efficient heuristic algorithm called *sort-select-swap* to solve the *OBM* problem. The key idea comes from the fact that the majority of the on-chip network communication is shared cache traffic, i.e., the L2 cache request rate c_j is several times larger than the memory controller request rate m_j. Therefore we first implement *coarse tuning* by assigning the tiles to applications so that each application gets almost the same number of tiles with larger cache latencies and the same number of tiles with smaller cache latencies. Then *fine tuning* is done by taking into account the on-chip traffic to memory controllers and performing sliding-window-based swaps.

During the process, the following optimization task is common and necessary in both coarse and fine tuning to reduce imbalance: After selecting the tiles for each application, we need to find a mapping method which properly assigns its threads to these selected tiles, so that the APL of this application is minimized. This is consistent with the primary objective of minimizing the max-APL because, with a given application-to-tile assignment, the max-APL can be reduced by minimizing each application's APL.

A. Single Application Mapping Optimization

Before elaborating the solution to the *OBM* problem, we first formulate and solve the single-application mapping (*SAM*) problem. The *SAM* problem is described as follows: Given N_a tiles and an application a with N_a threads, find the thread-to-tile mapping that minimizes APL of application a.

Given:
1) The number of tiles (or threads) N_a;
2) The L2 cache request rate array $C = \{c_j\}$ and the memory controller request rate array $M = \{m_j\}$ as in Section III.B; and
3) Arrays $\{TC(k)\}$ and $\{TM(k)\}$, indicating the APLs from the k-th tile to the distributed L2 cache and to the memory controller, respectively.

Find: Mapping $\pi_a(j) = k$, where $j, k \in \{1,2,\ldots,N_a\}$, and

Minimize: The APL of application a:

$$d_a = \frac{\sum_{j=1}^{N_a}\left(c_j TC(k) + m_j TM(k)\right)}{\sum_{j=1}^{N_a}(c_j + m_j)} \tag{12}$$

We show that the above *SAM* problem is an easy problem and propose a polynomial-time solution to it.

Due to the evenly distributed nature of the L2 cache in CMPs and the proximity principle of memory controller communication as discussed before, the APL of one thread assigned to a certain tile depends only on the communication rates c_j and m_j of thread j, and the $TC(k)$ and $TM(k)$ of tile k. It is independent of which tiles other threads are assigned to. In other words, once an assignment of a thread is finished, mapping results of other threads will not affect the APL of the current thread as long as the threads are mapped to different tiles. This mapping problem is, thus, an instance of the combinational assignment problem and can be solved using the Hungarian method [15] with a complexity of $O\left({N_a}^3\right)$.

We use the following Hungarian-based method to solve the *SAM* problem.

Algorithm 1: Hungarian-based *SAM* solution

Inputs: The number of threads or tiles N_a, tile latency arrays TC and TM.

```
Step 1: Generate the $N_a \times N_a$ cost matrix $\{cost_{jk}\}$, where
$cost_{jk}$ indicates the APL of $j$-th thread placed on the $k$-th
tile (L2 cache delay plus the on-chip memory request delay):
        $cost_{jk} = c_j \times TC(k) + m_j \times TM(k)$        (13)
Step 2: Call Hungarian algorithm with input matrix $\{cost_{jk}\}$
to generate a permutation $\pi(j)$ of $\{1,2,\dots,N_a\}$ that minimizes
the total cost. The output of Hungarian is the minimum
cost $(d_a)_{\min}$ as well as the permutation $\pi_a(j)$.
Return: The minimized APL $(d_a)_{\min}$ and mapping $\pi_a(j)$.
```

The first and second step have $O({N_a}^2)$, $O({N_a}^3)$ complexity, respectively. The overall complexity of the *SAM* solution is therefore $O({N_a}^3)$.

B. *Sort-Select-Swap Algorithm*

Provided with the above *SAM* solution, we develop the complete *sort-select-swap* algorithm for the *OBM* problem applied to multiple application mapping as follows.

We first *sort* all the tiles according to their L2 cache APLs (i.e., $TC(k)$). The second step is to *select* appropriate tiles to assign to each application in such a way that the large-cache-latency tiles and the small-cache-latency tiles are equally distributed among different applications. For example, we have an application a with 16 threads. We divide the sorted tile list into 16 sections with equal length and select the tile in the middle from each section for this application, as shown in Figure 6. The selection is then followed by calling the Hungarian-based *SAM* method to minimize the APL within this application. Similarly, every application is assigned to tiles in this manner. After this selection step, every thread in each application is assigned to a tile.

The third step is to perform fine tuning by *swapping* certain thread-to-tile assignments across applications. We use a sliding-window-based swap and choose the best result greedily to implement fine tuning. Tiles are still organized in the list view conceptually, as shown in Figure 7. Each window contains four tiles. All 24 possible permutations (the factorial of 4) of the mappings for these four tiles are explored, and we choose the permutation that leads to the minimum max-APL. The window starts with a step size of 1, i.e., four consecutive tiles are picked, and slides from the beginning to the end of the tile list. Then we increase the step size of the window and perform window-sliding again, as depicted in Figure 7. Finally, after all the window-sliding is done, the algorithm calls the *SAM* method once more for each application to reduce its APL, thereby possibly further reducing the overall max-APL of the generated mapping. The pseudo code is shown as follows.

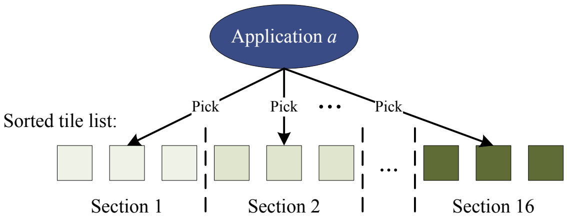


Figure 6. Select from the sorted tile list.

Algorithm 2: *sort-select-swap*

```
Inputs: Application set $\{a_i\}$, the number of threads or tiles
$N$, Cache access latency array $\{TC(k)\}$ and memory access
latency array $\{TM(k)\}$.
Step 1: Sort all the tiles based on their L2 cache APLs. Denote
the sorted tile list as $\{l_k\}$.
Step 2: Select tiles and assign them to applications
for application $a_i$ from $a_1$ to $a_A$, do
    Divide the tile list $\{l_k\}$ into $\Delta N_i = N_i - N_{i-1}$ sections
    with equal length;
    Pick the tiles in the middle of each section from $\{l_k\}$;
    Call the Hungarian-based SAM method to assign these
    $\Delta N_i$ tiles to the threads of $a_i$ so that the APL for $a_i$ is
    minimized;
    Remove the assigned tiles from the list $\{l_k\}$.
Step 3: Greedy sliding-window-based swap
for step size $s$ from 1 to $N/4$, do
    for $i$ from 1 to $N - 3s$, do
        //The four tiles in the current window $\{i, i+s, i+2s, i+3s\}$
        Generate all 24 permutations of the current window;
        Find the permutation that minimizes the max-APLs;
        Change the four tiles' mapping to this permutation.
for application $a_i$ from $a_1$ to $a_A$, do
    Call the Hungarian-based SAM method to adjust the tile-
    to-thread assignment of the $\Delta N_i$ tiles within application
    $a_i$ so that the APL for $a_i$ is minimized.
Return: Mapping $\pi(j) = k$
```

The proposed *sort-select-swap* algorithm has polynomial time complexity:

Step 1: Sorting takes $O(N \log N)$ times of calculation.

Step 2: There are A applications each requires an assignment. In each assignment, picking ΔN_i tiles has $O(\Delta N_i)$ time complexity, the Hungarian algorithm has $O({\Delta N_i}^3)$ time complexity and deleting the assigned members from the current list $\{l_k\}$ takes $O(N)$ time. Altogether the assignment step has a time complexity of $O(N) + \sum O({\Delta N_i}^3) = O(N^3)$ since $\sum \Delta N_i = N$.

Step 3: The number of windows generated is $O(N^2)$. During the processing of each window, there are 24 permutations, each requiring one APL calculation. This calculation is finished within $O(N)$ time according to APL definition in (6). Similar to step 2, the final Hungarian has $O(N^3)$ time complexity. Overall swapping has a time complexity of $O(N^2) \cdot O(N) + O(N^3) = O(N^3)$.

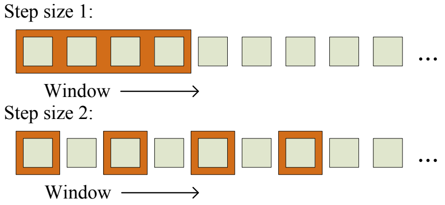


Figure 7. Sliding-window-based swap.

Therefore, the overall time complexity of the proposed *sort-select-swap* algorithm is $O(N^3)$. With this low computation complexity, the proposed mapping algorithm can be used when applications are dynamically added or removed in the CMPs. This is achievable because the runtime of the proposed mapping algorithm is short and application change happens at a much coarser time-granularity. To realize this, we simply collect the $\{c_j\}$ and $\{m_j\}$ statistics at runtime during a certain interval after some applications are added/removed, and then solve the OBM problem to determine the new mapping solution which will be used until the next application change.

V. SIMULATION

A. Simulation Setup

We evaluate the effectiveness of different mapping algorithms by utilizing traces gathered from running multi-threaded PASREC 2.0 benchmarks [2] on full-system simulation using Simics [18]. The GEMS [19] and Garnet [1] simulators are integrated into Simics for detailed timing of the memory system and on-chip network, respectively. We use the latest NoC power model DSENT [24] for power estimation, based on 45nm technology and 1 V power supply. We assume a canonical 3-stage credit-based wormhole router with look-ahead routing optimization. With 128-bit link width, short 16-bit packets are single-flit while long packets carrying 64-byte data plus a head flit have 5 flits. Table 2 lists the key parameters of our simulation.

Since different PARSEC applications have various intensities of network load (i.e., the sum of shared cache requests and memory controller requests), we construct eight different configurations (i.e., C1 to C8) with varying loads in the evaluation. Each configuration has four 16-thread applications. Table 3 provides the average values and standard deviations of the cache and memory traffic of the four applications for each of the eight configurations C1 to C8. Since the cache communication rate is, on average, 6.78 times that of the memory controller traffic rate, we classify based on the characteristics of cache traffic. The eight configurations include those with relatively higher average rates (e.g., C4) or lower rates (e.g., C2), and higher standard deviations (e.g., C8) or lower standard deviations (e.g., C7).

TABLE 2. KEY PARAMETERS FOR SIMULATION.

Network topology	8x8 mesh
Router	3-stage, 2GHz
Input buffer	5-flit depth
Link bandwidth	128 bits/cycle
Cores	Sun UltraSPARC III+, 2 GHz
Private I/D L1$	32KB, 2-way, LRU, 1-cycle latency
Shared L2 per bank	256KB, 16-way, LRU, 6-cycle latency
Cache block size	64 Bytes
Virtual channel	3 VCs per protocol class
Coherence protocol	MOESI
Memory controllers	4, located one at each corner
Memory latency	128 cycles

TABLE 3. AVERAGE VALUES AND STANDARD DEVIATIONS OF COMMUNICATION RATES OF EIGHT CONFIGURATIONS.

Configuration	**Cache**		**Memory**	
	Average	***Std-dev***	***Average***	***Std-dev***
C1	7.008	88.3	0.899	9.84
C2	1.8855	17.52	0.381	2.21
C3	10.881	112.34	1.51	18.42
C4	11.063	107.27	1.548	17.56
C5	9.04	129.27	1.371	19.91
C6	9.222	125.81	1.409	19.21
C7	1.992	14.69	0.399	2.01
C8	8.881	131.87	1.334	20.45

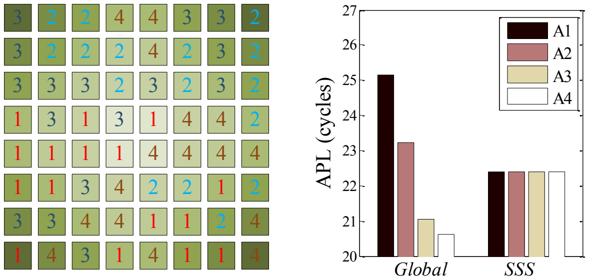
We compare the following four algorithms:

1) Global optimization (*Global*): as discussed in Section II.D, it minimizes overall latency of all the threads;
2) Monte Carlo method (*MC*) for the *OBM* problem: from a large number (e.g. 10^4) of random mappings, it selects the one with the minimum max-APL;
3) Simulated annealing-based algorithm for the *OBM* problem (*SA*): we define a random "move" in *SA* as swapping the mapping of two randomly chosen threads; and
4) The proposed *sort-select-swap* algorithm for the *OBM* problem (*SSS*).

B. Simulation Results

1) Mapping Results of Different Algorithms

In Section II.D, we showed the mapping result of *Global* for configuration C1, in which the *Global* increases the imbalance among applications. Figure 8 (a) presents the mapping results of the proposed *SSS* for the same configuration. The applications are sorted in ascending order of average communication rate. As can be seen, Application 1, which has the lightest on-chip traffic, is no longer placed in the four corners of the chip in the *SSS* mapping method, whereas the *Global* mapping result shown in Figure 4 assigns the corner cores to Application 1. Figure 8 (b) compares the APLs of the four applications in C1, which shows a significant improvement in the balance of on-chip packet latency in *SSS*. For instance, compared with *Global*, the APL of Application 1 in *SSS* reduces from 25.15 to 22.40 cycles, resulting in a 10.89% decrease. The APLs of the four applications in *SSS* are nearly the same. In the following subsections, more detailed data on max-APLs, standard deviations, and network performance and power are discussed.



3	2	2	4	4	3	3	2
3	2	2	2	4	2	3	2
3	3	3	2	3	2	3	2
1	3	1	3	1	4	4	2
1	1	1	1	4	4	4	4
1	1	3	4	2	2	1	2
3	3	4	4	1	1	2	4
1	4	3	1	4	1	1	4

(a) Mapping result of *SSS*

(b) APLs for the four applications

Figure 8. Mapping result and APL comparison of C1.

2) Max-APL Comparison

Figure 9 compares the max-APL results of the four mapping methods. As shown in the figure, the proposed *SSS* is

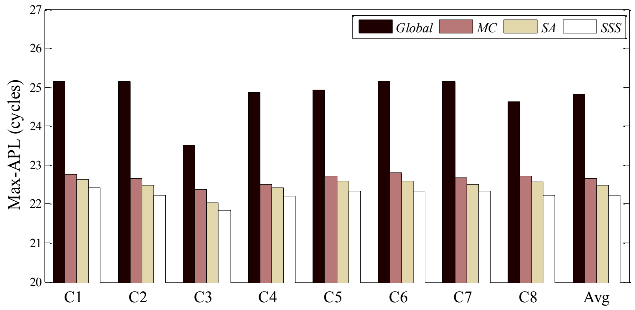


Figure 9. Max-APLs for different configurations.

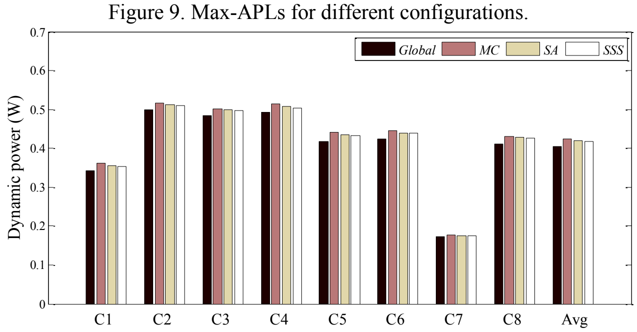


Figure 11. Dynamic power for different configurations.

able to reduce the max-APL by an average of 10.42% compared to *Global*. This result illustrates that *SSS* can provide good latency balancing for different combinations of applications. The high max-APL of *Global* indicates that it optimizes mapping at the cost of greatly exacerbated imbalance.

MC and *SA* can also achieve mapping results with max-APL improvement to *Global* by 8.74% and 9.44%, respectively. However, note that the *MC* and *SA* are given higher runtime than *SSS* to generate these results. The Monte Carlo method and the simulated annealing are search-based algorithms. They both have to perform a sufficient amount of random moves in order to obtain a satisfying result. The high complexity of search-based algorithms makes them nearly impossible to be adopted in dynamic scenarios.

3) *Standard Deviation*

As mentioned before, although standard deviation of APLs (dev-APL) may not be suitable for the objective function used in a particular mapping algorithm, it is still a direct and well-acknowledged indicator for measuring the variance among multiple values. Table 4 lists the dev-APL of the four mapping methods for the eight different configurations. It can be seen that *Global* has the largest dev-APL among the four mapping algorithms. Both *MC* and *SA* have moderate reduction in dev-APL. In comparison, the proposed *SSS* can reduce dev-APL significantly by 99.65%, 95.45%, and 83.15% compared to *Global*, *MC* and *SA*, respectively, demonstrating its superior advantage in balancing latencies among multiple applications.

TABLE 4. DEV-APL FOR DIFFERENT CONFIGURATIONS.

	C1	C2	C3	C4	C5	C6	C7	C8
Global	2.094	1.630	1.877	1.774	2.140	2.030	1.262	2.160
MC	0.087	0.162	0.042	0.037	0.036	0.114	0.298	0.123
SA	0.060	0.020	0.091	0.114	0.060	0.241	0.110	0.022
SSS	0.006	0.005	0.007	0.010	0.005	0.002	0.002	0.014

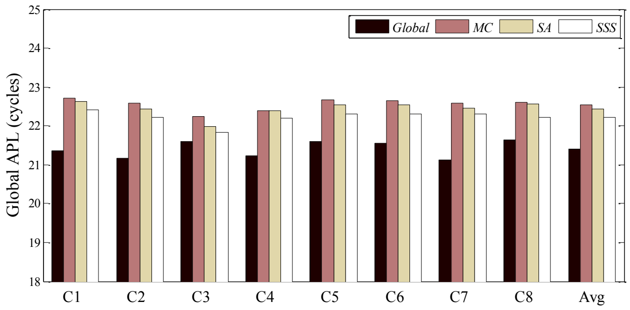


Figure 10. g-APLs for different configurations.

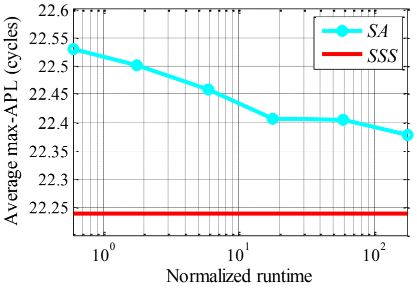


Figure 12. Simulated annealing results as a function of runtime.

4) *Impact on Global APL*

In order to achieve balanced on-chip packet latencies, it is possible for the application mapping to increase the overall packet latency. The proposed *SSS* is a performance-aware latency balancing mapping algorithm as it uses the max-APL as the criterion by which it may be able to achieve better latency balancing with much less performance loss compared to other criteria such as standard deviations. Figure 10 plots the normalized global APLs (g-APLs) of four mapping methods. As expected, *Global* has the minimal g-APL since minimizing g-APL is its sole objective. The performance loss percentages of the other three algorithms are all within 6% because they all have minimization of max-APL as their optimization objective. Among the three algorithms, *SSS* only slightly increases g-APL compared to *Global*, by less than 3.82% and is better than *SA* (4.82% loss) and *MC* (5.35% loss). This means that the benefits of balanced latency in the proposed *SSS* are not obtained at large penalty in overall packet latency.

5) *Algorithm Runtime Comparison*

Search-based algorithms such as simulated annealing are a tradeoff between runtime and performance. Figure 12 presents the max-APL results of *SA* when it is allowed to run for different CPU time. The result is normalized to the runtime of *SSS* and plotted in logarithmic scales. Due to the randomness of the simulated annealing method, we show the average of the max-APLs of the eight configurations. The max-APL derived in *SA* decreases as the allowed algorithm runtime increases, but it has a diminishing gain. As shown in Figure 12, *SSS* outperforms *SA* even when *SA*'s runtime is 100X larger than that of *SSS*. In addition, while the max-APL difference between *SA* and *SSS* is not very large, we have seen from Table 4 that the dev-APL between the two methods has an average of around 6X difference when *SA* is allowed to have similar runtime as *SSS*.

6) *Power Consumption*

In addition to the network performance, we also evaluate the NoC power consumption of different mapping algorithms. While the static power is approximately the same for different schemes, the dynamic NoC power depends on the total number of packets injected to the network per unit time and the average number of hops per packet. Figure 11 presents the dynamic power comparison. As can be observed, the proposed *SSS* algorithm has almost negligible power overhead with less than 2.7% compared to *Global*, and is slightly better than both *MC* and *SA*, illustrating that *SSS* does not penalize overall NoC power.

In summary, *Global* achieves the minimal global average packet latency. However, the *Global* solutions experience the worst imbalance among the four methods under comparison. *SA* can obtain near-optimal solutions at the cost of very large runtime. Similarly, as another randomized algorithm, *MC* can also find the optimal solution theoretically provided with sufficiently long runtime, but its performance in practice is much worse than *SA*. In comparison, our proposed *SSS* algorithm is able to achieve near-optimal solutions in terms of latency balancing in a very short runtime and, at the same time, incur little overall packet latency and power consumption overhead.

VI. Acknowledgement

This research is supported, in part, by the National Science Foundation (NSF) grant CCF-1321131 and the Software and Hardware Foundations program of the NSF.

VII. Conclusion

This paper addresses the important issue of balancing on-chip network latency in multi-application mapping for chip multiprocessors. We formulate the problem of on-chip latency balanced mapping (*OBM*) for multiple concurrently running applications. After proving its NP-completeness, we propose an efficient heuristic-based algorithm that leverages the characteristics of shared cache and memory controller traffic. Simulation results show that the proposed algorithm can achieve an average reduction of 99.65% in standard deviation and 10.42% in maximum average packet latency, with less than 3.84% overhead in overall packet latency and 2.7% more power consumption. This demonstrates the viability of exploiting thread-to-tile mapping to balance the on-chip latencies among different applications while incurring little overall network performance and power degradation.

References

[1] N. Agarwal, T. Krishna, L. S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," In Proceedings of the Performance Analysis of Systems and Software, 2009.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," In Proceedings of the international conference on Parallel Architectures and Compilation Techniques, 2008.

[3] G. Chen, F. Li, S. W. Son, and M. Kandemir, "Application mapping for chip multiprocessors," In Design Automation Conference, 2008.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms," MIT press, 2001.

[5] W. J. Dally and B. Towles, "Principles and practices of interconnection networks," Morgan Kaufmann, 2003.

[6] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," In Design Automation Conference, 2001.

[7] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-aware prioritization mechanisms for on-chip networks," In IEEE/ACM International Symposium on Microarchitecture, 2009.

[8] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," ACM Sigplan Notices, 2010.

[9] R. Gabor, S. Weiss, and A. Mendelson, "Fairness and throughput in switch on event multithreading," In IEEE/ACM International Symposium on Microarchitecture, 2006.

[10] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, "Kilo-NOC: a heterogeneous network-on-chip architecture for scalability and service guarantees," In ACM SIGARCH Computer Architecture News, 2011.

[11] A. Hansson, K. Goossens, and A. Rădulescu, "A unified approach to constrained mapping and routing on network-on-chip architectures," In Proceedings of the IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, 2005.

[12] J. Howard, et al., "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," In Proceedings of the Solid-State Circuits Conference Digest of Technical Papers, 2010.

[13] J. Hu, and R. Marculescu, "Energy-aware mapping for tile-based NoC architectures under performance constraints," In Proceedings of the Asia and South Pacific Design Automation Conference, 2003.

[14] W. Jang, and D. Z. Pan, "A3MAP: Architecture-aware analytic mapping for networks-on-chip," ACM Transactions on Design Automation of Electronic Systems, 2012.

[15] H. W. Kuhn, "The Hungarian method for the assignment problem," Naval Research Logistics, 2005.

[16] M. M., Lee, J. Kim, D. Abts, M. Marty, and J. W. Lee, "Probabilistic distance-based arbitration: Providing equality of service for many-core CMPs," In IEEE/ACM International Symposium on Microarchitecture, 2010.

[17] Z. Lu, L. Xia, and A. Jantsch, "Cluster-based simulated annealing for mapping cores onto 2D mesh networks on chip," In Design and Diagnostics of Electronic Circuits and Systems, 2008.

[18] P. S. Magnusson et al., "Simics: A full system simulation platform," In IEEE Computer, 2002.

[19] M. M. Martin, et al. "Multifacet's general execution-driven multiprocessor simulator toolset," ACM SIGARCH Computer Architecture News, 2005.

[20] S. Murali, and G. De Micheli, "Bandwidth-constrained mapping of cores onto NoC architectures," In Proceedings of the conference on Design, automation and test in Europe, 2004.

[21] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli, "A methodology for mapping multiple use-cases onto networks on chips," In Proceedings of the conference on Design, automation and test in Europe, 2006.

[22] M. K. Qureshi, and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," In IEEE/ACM International Symposium on Microarchitecture, 2006.

[23] L. Spracklen, and S. G. Abraham, "Chip multithreading: opportunities and challenges," In International Symposium on High-Performance Computer Architecture, 2005.

[24] C. Sun et al., "DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," In International Symposium on Networks-on-Chip, 2012.

[25] H. Vandierendonck, and A. Seznec, "Fairness metrics for multi-threaded processors," Computer Architecture Letters, 2011.