Scenario-aware Data Placement and Memory Area Allocation for Multi-Processor System-on-Chips with Reconfigurable 3D-stacked SRAMs

Meng-Ling Tsai, Yi-Jung Chen, Yi-Ting Chen, and Ru-Hua Chang
Department of Computer Science and Information Engineering
National Chi Nan University
Nantou County, Taiwan
Email: { s100321501, yjchen, s99321030. s99321035}@ncnu.edu.tw

***Abstract*—Integrating Multi-Processor System-on-Chips (MPSoCs) with 3D-stacked reconfigurable SRAM tiles has been proposed for embedded systems with high memory demands. At runtime, the SRAM tiles are configured into several memory areas, which can be reconfigured according to the dynamic behavior of the system. Targeting this architecture, in this paper, we propose a data placement and memory area allocation algorithm. The goal of the proposed algorithm is to optimize the performance of the memory system by minimizing the on-chip memory access latency, the number of off-chip memory accesses, and the number of reconfigurations. Since the behavior of an embedded system can be described by a set of *scenarios*, where each scenario specifies a set of applications that would execute concurrently, the proposed algorithm synthesizes data placements and the memory area allocation for each scenario. Not only the data access patterns within the scenario but also among all scenarios are considered for data placement. We evaluate the proposed algorithm on a set of synthetic and real-world applications. The experimental results show that, compared to the existing data placement method designed for MPSoCs with distributed memory modules, the proposed algorithm achieves up to 11.72% of data access latency reduction.**

I. Introduction

Integrating memory modules and the multi-core processors by the low-latency and high-density Through-Silicon Vias (TSVs) in the third dimension has been considered as a promising way to alleviate the memory bandwidth problem of a multi-core system [11], [14], [18]. Among various 3D-enabled processor-memory integrated architectures, Multi-Processor System-on-Chip (MPSoC) with 3D-stacked reconfigurable SRAMs proposed in [18] provides a special capability of dynamic reconfiguration of the stacked memories. Fig. 1 shows the architecture proposed in [18]. In this architecture, the SRAM layer is stacked on top of the logic layer that is composed of IP cores. The SRAM layer is composed of a set of SRAM tiles interconnected by a 2D-mesh network. At run-time, the SRAM layer is configured to several memory areas, where each of the memory area is composed of a set of contiguous SRAM tiles and is accessed by an individual core in the logic layer [18]. The configuration of memory areas can be dynamically adapted according to the runtime behavior of the system. Since the SRAM access latency is decided by the distance between the requesting IP core and the requested SRAM tile, the stacked memory is a Non-Uniform Memory Access (NUMA) architecture.

978-3-9815370-2-4/DATE14/ © 2014 EDAA

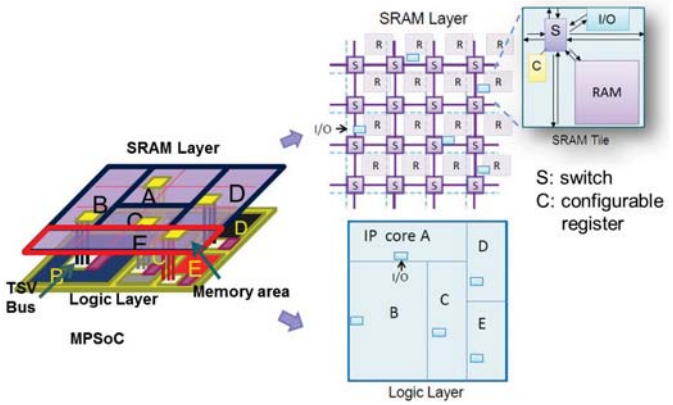

Fig. 1. An MPSoC with 3D-stacked reconfigurable SRAMs.

A straightforward way to utilize the reconfigurable 3D-stacked SRAMs is using the state-of-the-art data placement method designed for NUMA memory architecture, e.g. the method proposed in [3]. Since the behavior of an MPSoC can be described by a set of system *scenarios* [17] [20], where each scenario describes a set of applications that would execute concurrently, we can utilize the methods designed for NUMA to decide the data placement and the configuration of memory areas for each of the scenario. However, these methods only consider the data locality to reduce the average memory access latency in an NUMA architecture. The interference among memory areas is overlooked, and unnecessary reconfigurations may be triggered, which will cause performance degradation. Therefore, to fully exploit the advantage of the reconfigurable 3D-stacked SRAM layer, it is a must to redesign the data placement and memory area allocation methods.

In this paper, we propose a scenario-aware data placement and memory area allocation algorithm for MPSoCs with reconfigurable 3D-stacked SRAMs to optimize the performance of the memory system. Due to the complexity of modern MPSoCs, the design space of the target synthesis problem is huge. It has been proven that the data allocation problem alone is NP-complete [13]. Therefore, the proposed synthesis algorithm is a heuristic-based method. To optimize the performance of the memory system, our algorithm considers the data access patterns both within a scenario and among all scenarios to perform data placement for each of the scenario. Considering the data access patterns within a scenario helps shorten the

Fig. 2. Formation and reconfiguration of memory areas.

access latency of SRAM tiles, and reduce contentions of an SRAM tile or a memory area so that the number of reconfigurations can be reduced. On the other hand, considering the data access patterns among all scenarios helps data that are accessed by several scenarios not to be falsely replaced during the transition of scenarios. Thus, unnecessary off-chip memory accesses can be avoided. Compared to the existing data placement method for MPSoCs with distributed memory modules, the experimental results show that the algorithm gets up to 5.88% reduction of average data access latency for a scenario on the average. When multiple scenarios are executed in the system, the proposed algorithm achieves up to 11.72% of data access latency reduction when applied to a set of real-world applications.

This paper is organized as follows. The target architecture is introduced in Section II. The related research is reviewed in Section III. The system model and problem formulation are described in Section IV. Section V details the proposed algorithm. The experimental results are discussed in Section VI, and Section VII concludes the paper.

II. Properties of the Target Architecture

The target architecture shown in Fig. 1 is composed of a logic layer and a reconfigurable SRAM layer. The logic layer is composed of IP cores or general purpose processing elements (PEs), and each IP core or PE has one or more I/O port equipped with TSVs to access the SRAM tiles that are stacked on top of the logic layer. The SRAM layer is composed of regular SRAM tiles that are connected by a 2D mesh network, and shares the memory address space with the off-chip memory. As shown in Fig. 1, each SRAM tile has an I/O port, a switch for passing memory requests among tiles and I/O ports, and a configuration register.

At run-time, as shown in Fig. 2, the SRAM layer is configured to several memory areas, where each of them is composed of one or more contiguous SRAM tile. As mentioned in Section I, each memory area is accessed by an individual core in the logic layer, and is operated individually. The configuration of memory areas is indicated by the configuration registers of SRAM tiles. Reconfiguring memory areas is achieved by modifying the configuration registers, which needs 1-cycle delay. The reconfiguration can be triggered by (1) adapting memory areas for the change of system behavior, or (2) a core that retrieves data placed at the SRAM tile of the remote memory area, i.e. the memory area of the other core. For the second case, the reconfiguration is triggered to include the target SRAM tile into the requesting core's local memory area. Note that only data shared among more than one cores may cause this kind of reconfigurations. Although it is convenient to reconfigure the memory area, too many reconfigurations may cause serious performance degradation. When there are data accesses to SRAM tiles in the memory areas to be reconfigured, the reconfiguration process and the new data access requests to the tiles should be halted until the existing data accesses are completed.

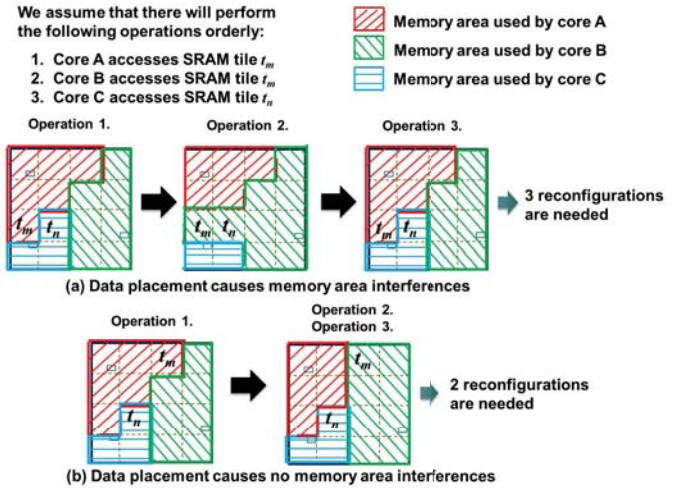

Fig. 3. Example of memory area interferences. Data placement causes (a) memory area interferences, and (b) no memory area interference.

Therefore, reducing the number of unnecessary reconfigurations is critical for improving the performance of the memory system. We observe that, the reconfigurations caused by interferences among memory areas can be avoided by proper data placements. As illustrated in Fig. 3, given the data access sequence, the data placement shown in Fig. 3(a) would cause three reconfigurations since the placement of SRAM tile t_n breaks the contiguity of the memory area used by core B. With the data placement shown in Fig. 3(b), the number of reconfigurations is reduced to two. Therefore, through proper data placements, the interference of memory areas can be reduced and so does the reconfigurations.

III. Related Works

Several memory resource allocation and data placement methods have been proposed for both traditional 2D and emerging 3D MPSoCs [1], [3], [4], [9], [10], [15]. For traditional 2D MPSoCs, Meyer et al. [15] proposed a synthesis tool to drive simultaneous data mapping, memory allocation, and bus synthesis. Chen et al. [4] proposed a algorithm for synthesizing the allocation of processing elements and memory modules of a resource constrained MPSoC. For 2D Chip-Multiprocessors (CMPs) with NUMA architecture, Chen et al. [3] proposed a data mapping method that minimizes average access latency. Kandemir et al. [10] proposed a dynamic thread and data mapping scheme that utilizes a data mapping method similar to [3] to map data for each phase. For 3D architecture, Hsieh et al. [9] proposed a data mapping method for the 3D-stacked DRAMs to reduce system temperature. Ancajas et al. [1] proposed a dynamic memory re-locator for CMPs with 3D-stacked DRAMs and distributed memory controllers so that the utilization of high speed vertical interconnect can be increased. To the best of our knowledge, none of the existing research works are designed for the architecture discussed in this paper.

IV. System Specification and Problem Formulation

In this section, we present the data structures and models that represent the software behavior and hardware configuration of the target system. Then, we formulate the synthesis problem discussed in this paper.

TABLE I. NOTATIONS USED IN THE SYSTEM MODELS.

Software Model	
SG	$SG = (V_S, E_S)$ indicates scenario graph
V_S	$V_S = \{v_{s_i}, ...\}$ is the set of vertices (scenarios)
E_S	$E_S = \{e_{s_k}, ...\}$ is the set of directed edges
v_{s_i}	$v_{s_i} = \{TG_1, TG_2, ...TG_n\}$ represents the set of applications that are executed currently in scenario v_{s_i}
$p(v_{s_i})$	Execution probability of v_{s_i}
TG_i	$TG_i =< V_{T_i}, E_{T_i} >$ indicates task graph
V_{T_i}	$V_{T_i} = \{v_{T_{i\_j}}, ...\}$ is the set of vertices (tasks)
E_{T_i}	$E_{T_i} = \{e_{T_{i\_j}}, ...\}$ is the set of directed edges
D	$D = \{d_i, ...\}$ is the set of data blocks used in the system
$size(d_i)$	Size (in bits) of d_i
$c(v_{T_i})$	Execution cycles that task v_{T_i} executes on IP core $IP(v_{T_i})$
$d(e_{T_k})$	The ID of the data block transferring by e_{T_k}
Hardware Model	
P	$P = \{p_0, ..., p_l\}$ indicates the set of IP cores
T	$T = \{t_1, ..., t_{m\times n}\}$ indicates the set of SRAM tiles
$io(t_i)$	The ID of IP core connecting to tile t_i
$io(p_i)$	The ID of SRAM tile connecting to IP core p_i
$size(SRAM)$	Capacity of each tile

A. Software Model

The software model utilizes a hierarchical graph to capture the behavior within a scenario and among all scenarios. To represent all the scenarios executed in the system and the relation among the scenarios, we use a directed graph, called *scenario graph* $SG = (V_S, E_S)$, to represent the scenarios. V_S is the set of vertices, and every vertex $v_{s_i} \in V_S$ represents a scenario. A scenario v_{s_i} is also associated with $p(v_{s_i})$ to indicate the execution probability of v_{s_i} [20]. E_S is the set of directed edges. Each edge e_{s_k}, where $e_{s_k} \in E_S$ and $e_{s_k} = \{v_{s_i}, v_{s_j}\}$, represents there is a possibility that scenario v_{s_i} is executed right after v_{s_j}. A scenario v_{s_i} is composed by a set applications, that is, $v_{s_i} = \{TG_1, TG_2, ...TG_n\}$, where TG_i represents the task graph of an application. For each application T_i, we use a task graph $TG_i = (V_{T_i}, E_{T_i})$ to represent its control and data flow. V_{T_i} is the set of vertices that represent the tasks or kernel functions executed in the application, and E_{T_i} is the set of directed edges to indicate the control flow among tasks. Each edge $e_{T_{i\_j}} \in E_{T_i}$ is associated with $d(e_{T_{i\_j}})$ to denote the ID of the data block that is transferred over the edge. Data blocks are collections of scalars or arrays [16]. We assume that D is the set of data blocks used in the system, and each $d_i \in D$ is associated with $size(d_i)$ to indicate the size (in bits) of d_i. Each task graph has the following properties:

- Each $v_{T_{i\_j}} \in V_{T_i}$ is associated with the ID of the IP core that executes $v_{T_{i\_j}}$, and $c(v_{T_{i\_j}})$ to indicate the execution time of $v_{T_{i\_j}}$ in cycles.
- Each $e_{T_{i\_k}} = \{v_{T_{i\_x}}, v_{T_{i\_y}}\}$ indicates $v_{T_{i\_x}}$ stores $d(e_{T_{i\_k}})$ to the memory first, and $v_{T_{i\_y}}$ retrieves $d(e_{T_{i\_k}})$ from the memory subsequently. If $v_{T_{i\_x}} \in \emptyset$ or $v_{T_{i\_y}} \in \emptyset$, it indicates storing $d(e_{T_{i\_k}})$ to the memory or retrieving $d(e_{T_{i\_k}})$ from the memory only.

B. Hardware Model

The hardware model captures the configurations of the logic layer and the 3D-stacked SRAM layer as shown in Fig. 1. We utilize $T = \{t_1, ..., t_{m\times n}\}$ to represent the set of $m \times n$ SRAM tiles at the SRAM layer. Each $t_i \in T$ is associated with $io(t_i)$. If $io(t_i) \in \emptyset$, it indicates tile t_i does not connect with logic layer. Else, $io(t_i)$ is set to the ID of the IP core that t_i is connected to. The size of an SRAM tile is denoted by $size(SRAM)$. For the logic layer, we use $P = \{p_0, ..., p_l\}$ to indicate the set of IP cores, and each p_i is associated with $io(p_i)$ to identify the tile position of its I/O port. All the notations used in our models are listed in Table I.

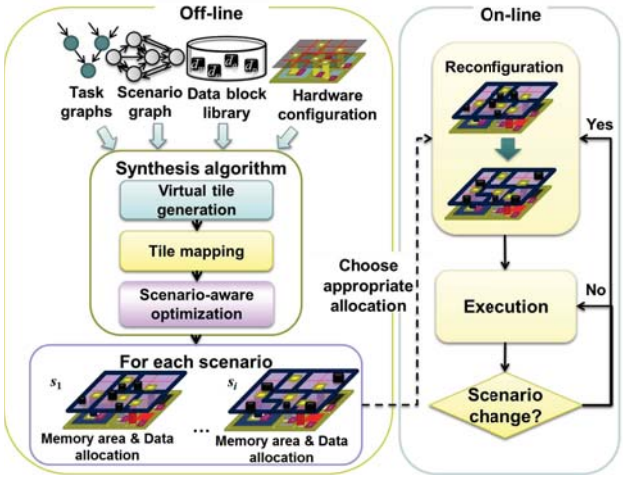

Fig. 4. Flow of the scenario-aware data placement and memory area allocation algorithm.

C. Problem Formulation

Given Scenario graph SG, task graphs TG_i, data block library D, IP cores P, the set of SRAM tiles T and capacity of each tile $size(SRAM)$.

Synthesis target Data placement $\theta : D \rightarrow T$ and SRAM tile allocation $\omega : T \rightarrow P$ for each scenario $v_{s_i} \in V_S$.

- **Data placement** Map each $d_i \in D$ to one of the SRAM tile $t_i \in T$ or the off-chip memory. We use the function $\theta : D \rightarrow T$ to represent the operation.
- **Memory area allocation** Allocate each SRAM tile t_i to one of the IP core's local memory area. We use the function $\omega : T \rightarrow P$ to represent this step.

Goal Minimizing the average data access latency.

V. SCENARIO-AWARE DATA PLACEMENT AND MEMORY AREA ALLOCATION ALGORITHM

The execution flow of the proposed synthesis framework is shown in Fig. 4. At the off-line phase, the proposed algorithm synthesizes the data placement and memory area allocation for each of the scenarios. At the on-line phase, the synthesized configurations are adopted based on the scenario in execution.

We adopt a heuristic-based method. The goal of the proposed method is improving memory system performance by reducing the number of the reconfigurations, average on-chip data access latency, and the number of off-chip accesses. As shown in Fig. 4, our algorithm is composed of three major steps; *virtual tile generation*, *tile mapping*, and *scenario-aware optimization*. In the *virtual tile generation* step, the algorithm first finds the set of data blocks that should be placed in the same on-chip SRAM tile. The goal of this step is to reduce the number of reconfigurations by minimizing the contention for an SRAM tile, and to maximize the number of on-chip memory accesses by increasing the locality of an SRAM tile. With the set of *virtual tiles* (VTs) formed in the first step, the *tile mapping* step allocates each VT to a physical SRAM tile position to minimize the average on-chip data access latency and the interference among memory areas. Finally, we perform *scenario-aware optimization* to fine-tune the data allocation so that the data that are utilized in several scenarios are kept in on-chip SRAMs. The initial memory area allocation for each IP core in a scenario is then decided once the data placement is done. Details of each synthesis step are presented in the following sections.

Virtual tile generation
Input: number of SRAM tiles $m \times n$, and $D = \{d_1, ..., d_n\}$
Output: a set of virtual tiles $vt[]$
1 for each d_k used in the scenario
2 classify_DBs_by_user();
3 for each category
4 while(there exists a DB not assigned to a VTC)
5 VTC_forming_by_knapsack();
6 sorting_VTV_by_access_frequency($VTCs$);
7 choose the $VTCs$ within the top $m \times n$ grades as VTs;

Fig. 5. Pseudo code of virtual tile generation.

A. Virtual Tiles Generation

As mentioned earlier, the goal of this step is to select the data blocks that are placed in the same SRAM tile so the contention for the SRAM tile is minimized and the number of on-chip memory accesses is maximized. To reduce the contention for an SRAM tile, the number of cores that access data blocks placed in it should be minimized. Therefore, we first classify the data blocks accessed in a scenario according to the cores that would access them. To avoid unnecessary contention for an SRAM tile, only the data blocks of the same category can be placed in the same SRAM tile.

To maximize the number of on-chip data accesses, the access frequency of each on-chip SRAM tile should be maximized. So, for each category of data blocks, we select the set of data blocks that have total size no more than $size(SRAM)$ and maximum data access frequency to form a virtual tile candidate (VTC). Taking the size of each data block as its weight, the access frequency as its value and the SRAM tile size as the limit, we can reduce the problem of forming VTC to the *Knapsack Problem*. The VTC forming process repeats until all the data blocks in a category are selected to a VTC.

The access frequency, or the value, of each data block d in scenario s, denoted by $U(s, d)$, can be defined as

$$U(s, d) = \sum_{p_j \in P} F_s(p_j, d), \tag{1}$$

where $F_s(p_j, d)$ is the total number of bits that IP core p_j accesses data block d in scenario s. $F_s(p_j, d)$ is calculate by

$$F_s(p_j, d) = size(d) \times access_s(p_j, d), \tag{2}$$

where $access_s(p_j, d)$ is the number of times that data block d is accessed by p_j in scenario s. We use the dynamic programming approach to solve the Knapsack problem.

After the forming of VTCs, the $m \times n$ VTCs with the highest access frequencies are selected as the virtual tiles (VTs) for the $TileMapping$ step. The access frequency of each VTC in scenario s, denoted by $T(s, VTC)$, is defined as

$$T(s, VTC) = \sum_{\forall d_j \text{ accessed in } s} U(s, d_j). \tag{3}$$

The VTs accessed by only one core are called as *private virtual tiles* (PVTs), and the VTs accessed by more than one core are called as *shared virtual tiles* (SVTs). The pseudo code of this step is shown in Fig. 5.

B. Tile Mapping

This step maps the VTs selected from the previous step to physical SRAM tile positions. The goal is minimizing the average on-chip data access latency is minimized, and reducing the interference among memory areas as mentioned in Section II. The first priority of the Tile Mapping step is minimizing the average on-chip data access latency. So, for each scenario, starting from the VT with the highest access frequency, we map each VT to the free physical tile with the least access cost. The access cost of mapping a VT vt to a physical tile $\varphi(vt)$ is quantified by the function $cost(\varphi(vt))$, which is defined as

$$cost(\varphi(vt)) = \sum_{d \in vt} \sum_{p \in P_{user}} \mid io(p) - \varphi(vt) \mid F(p, d). \tag{4}$$

In the function, P_{user} denotes the set of IP cores that would access data stored in vt, and $\mid io(p) - \varphi(vt) \mid$ denotes the Manhattan distance between the I/O port of p and the physical SRAM tile $\varphi(vt)$ [3].

Since there may be more than one physical SRAM tile with the minimum access cost, we select the tile position for PVTs and SVTs separately. For PVTs, to prevent interference among memory areas, the tile with the I/O port of the PVT's only user has the highest priority. The tiles with I/O ports of other IPs have the least priority. For SVTs, the tiles with I/O ports should have the least priority. For the candidate tiles without I/O ports, we set the tile that is close to the I/O port of the PVT's user and far from the I/O ports of the other IPs has the highest priority. This design is for keeping the contiguity of the memory area and minimizing the interference of memory areas. We define $d(\varphi(vt))$ to calculate the difference between the distance of vt to its user IP, and the distance of vt to its non-user IPs when allocation $\varphi(vt)$ is utilized. The candidate physical tile with the largest $d(\varphi(vt))$ value is selected. $d(\varphi(vt))$ is defined as

$$d(\varphi(vt)) = \left(\sum_{p_j \in P_{non\_user}} \mid io(p_j) - \varphi(vt) \mid \right) - \left(\sum_{p_i \in P_{user}} \mid io(p_i) - \varphi(vt) \mid \right). \tag{5}$$

After the above mapping, we can obtain the memory area allocation for each IP core. However, we find that, some SVTs are more frequently accessed by an IP core compared to other IP users, and mapping these SVTs based on its total data access frequency may not achieve a good result. To avoid

Tile mapping
Input: $T = \{t_1, ..., t_{m \times n}\}$, $D = \{d_1, ..., d_n\}$, and virtual tiles $vt[]$
Output: Tile mapping function φ
1 sorting($vt[]$); // according to their $T(s, VTC)$
2 for i = 1 to $m \times n$
3 for each physical tile $pt[j] \in T$
4 $vt[i].cost[j] = cost(vt[i], pt[j])$;
5 set_candidates();
set physical tiles with least costs to vt[i]'s candidates
6 cal_distance();
//calculate the $d(\varphi(vt))$ for each vt[i]
7 sorting($vt[i].pt\_candidate[]$); //sorting $vt[i].pt\_candidate[]$ from maximum to minimum according to their distance
8 if ($vt[i]$ is PVT)
9 move candidates with I/Os to first position of $vt[i].pt\_candidate[]$;
10 if ($vt[i]$ is SVT)
11 move candidates with I/Os to last position of $vt[i].pt\_candidate[]$;
12 for r = 1 to num_pt_candidate_of_$vt[i]$
13 if($vt[i].pt\_candidate[r]$ is free)
14 Map $vt[i]$ to $vt[i].pt\_candidate[r]$;
15 else //$vt[i].pt\_candidate[r]$ has been allocated to $vt[u]$
16 if ($vt[u].pt\_candidate[]$ has next candidate)
17 Re-map $vt[u]$ to next PT candidate;
18 Map $vt[i]$ to $vt[i].pt\_candidate[r]$;
19 else if ($r! = vt[i].pt\_candidate[].size()$ $r++$;

Fig. 6. Pseudo code of tile mapping.

this, we fine tune the mapping by swapping two adjacent tiles. For each physical tile, we evaluate if swapping the tile and one of its adjacent tile would result in better results than the original mapping. The swapping is quantified by the function $totalcost(\varphi)$, which is defined as

$$totalcost(\varphi) = cost(\varphi(vt_1)) + cost(\varphi(vt_2)). \quad (6)$$

If $totalcost$ of swapping is smaller than the original mapping, we perform the swapping. The process repeats until there is no better mapping. Note that the swapping are not performed on physical tiles with I/O ports to avoid causing extra reconfiguration overhead as mentioned previously. The pseudo code of the *Tile Mapping* process is shown in Fig. 6.

C. Scenario-aware Optimization

Without considering the data access behavior among scenarios, the data accessed by several scenarios may be falsely moved to the off-chip memory when there is a change of scenario. So, off-chip memory accesses would be needed for the future use of the data. To prevent this, we keep the data that are utilized in several scenarios in on-chip SRAM tiles even if the new scenario does not access the data. For each data block d, we calculate its expected value of being accessed (denoted by $G(d)$) in the system by

$$G(d) = \sum_{s_i \in S} p(v_{s_i}) \times U(s_i, d). \quad (7)$$

Data blocks with high $G(d)$ values are more likely to be stored in on-chip SRAMs across all scenarios. In this paper, the data blocks with the $G(d)$ values at top 20% of all the data blocks are selected to be stored in on-chip memory permanently. Once a data block d_p is selected to be stored in the on-chip SRAM permanently, for each scenario s_i, if d_p is not accessed by s_i, we choose data block d that has comparable data block size and the least $U(s_i, d)$, and is in the same category of d_p to be replaced by d_p.

VI. Experimental Results

A. Experimental Setup

To evaluate the proposed scenario-aware data placement and memory area allocation algorithm, we construct a simulation platform that based on HORNET [8], which is a configurable and cycle accurate network-on-chip simulator. We simulate the performance of the target architecture when the synthesis results of our algorithm are utilized at run-time. The proposed algorithm is evaluated with a set of synthetic workload and a set of real-world workload, respectively. The synthetic applications are generated by Task Graphs For Free (TGFF) [5], and the forming of system scenarios and the execution probability of each scenario are randomly generated. The real-world applications are selected from E3S [6], which is a set of task graphs of EEMBC [7] workloads. The system scenarios and their execution probabilities are obtained from [19]. The properties of the task sets and scenarios are listed in Table II and Table III, respectively. In our experiments, we assume the logic layer is composed of a dual core Cortex A9 [2], a video/audio processor, and an image processor [12]. The SRAM layer is composed of 4×4 regular tiles, where each SRAM tile is of 64KB capacity. We assume the on-chip network utilize the wormhole routing [3]. The detailed system configuration is listed in Table IV.

TABLE II. Properties of task sets.

	Task set	Memory footprint
Synthetic workload	t_1	532KB
	t_2	288KB
	t_3	440KB
	t_4	472KB
	t_5	392KB
Real-world workload	telecom	2KB
	MPEG2	95KB
	consumer	1099KB
	office automation	97KB

TABLE III. Properties of scenarios.

	Scenario	Task sets	$p(v_{s_i})$
Synthetic workload	s_1	t_1, t_2, t_4, t_5	0.1
	s_2	t_1	0.5
	s_3	t_2, t_3, t_4	0.3
	s_4	t_1, t_3, t_4	0.1
Real-world workload	Phone call	telecom	0.4
	In room	MPEG2	0.1
	In office	officeautomation, consumer	0.2
	On train	telecom, consumer	0.3

B. Analysis of the Experimental Results

In this set of experiments, we compare the proposed synthesis algorithm to the data placement method proposed in [3], which is designed for traditional 2D CMPs with distributed on-chip memories. We first discuss the performance improvement achieved within a synthetic scenario. Fig. 7 shows the average on-chip data access latency of each data flit achieved by the proposed method and the one proposed in [3]. We can see that, the proposed algorithm achieves up to 14.94% of data access latency reduction. The performance improvement mainly comes from the reduction of contentions for the same SRAM and memory area interference. Moreover, since our algorithm would allocate tiles that would be accessed by several IPs (tiles that would cause many reconfigurations) at the boundary of the memory area to avoid interference, tiles that are more likely to be accessed by a specific IP core can be placed closer to the I/O port of the IP core and thus reduce the average on-chip memory access latency. Fig. 7 shows the percentage of reconfiguration reduction achieved by our method when compared to [3]. Our method successfully reduces the number of reconfigurations and achieves up to 43.33% of reconfiguration reduction compared to the method proposed in [3].

From the above results, we observe that, our algorithm achieves obvious performance gain due to reducing the contention of SRAM tiles, especially for workloads with shared data. To understand how the amount of shared data affect the performance results achieves by the proposed algorithm, we create synthetic workloads that have the same control and data flow graph, and memory footprint, but various amount of shared data. Fig. 8 shows the average data access latency per flit and the percentage of reconfiguration reduction for this set of experiments. The results show that, when the amount of shared data increase from 37% to 48%, our algorithm does show a great performance improvement. However, with 61% shared data workloads, since the high percentage of shared data

TABLE IV. Parameter of the system configuration.

Parameter	Value
Total die area	$8mm^2$
SRAM layer dimension	4×4
Cores	Dual core Cortex A9, Audio/video processor, image processor
flit size	32 bits
TSV width	256 bits
SRAM size	64KB / tile
On-chip memory access latency	1 cycle
Off-chip memory access latency	100 cycles

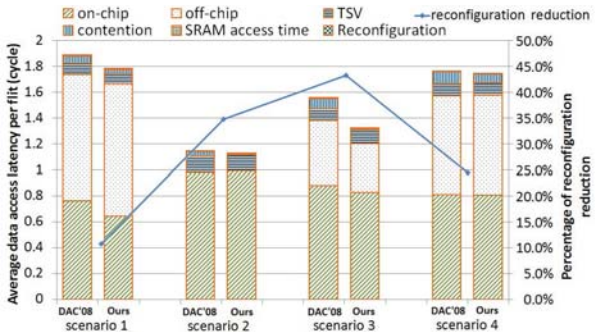

Fig. 7. Average data access latency per flit for various scenarios.

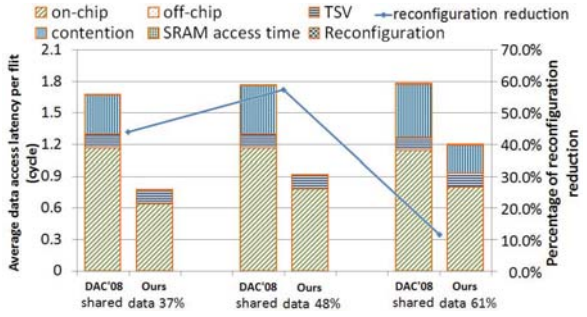

Fig. 8. Average data access latency per flit of task graphs with different percentage of shared data.

makes the contention for an SRAM tile unavoidable, for our algorithm, the percentage of reconfiguration reduction drops significantly and the amount of data access latency reduction diminishes. However, even in this case, our algorithm still achieves up to 33% reduction in average data access latency.

To evaluate the performance gain obtained from considering the behavior both within a scenario and among all scenarios, we performance experiments on a sequence of thirty scenarios, where the proposed algorithms with and without the scenario-aware optimization are applied. The experimental results show that, with scenario-aware optimization, the performance average data access latency is further reduced by 9.72% when compared to the one without scenario-aware optimization.

For the real-world applications, our algorithm achieves up to 15.40% of average data access latency reduction when compared to the method proposed in [3] for single scenario execution. When executing a sequence of thirty scenarios, the average data access latency reduction achieves up to 11.72% when the proposed algorithm is applied.

The implementation overheads of the proposed synthesis algorithm is the memory space for storing the placements of DBs of each type of the scenario. For each DB, we use one word to specify its starting address. In our experiments, we need no more than 2KB memory space for storing the data placement results.

VII. Conclusion

In this paper, we propose a scenario-aware data placement and memory allocation method for MPSoCs with reconfigurable 3D-stacked SRAMs. The proposed algorithm takes the data access patterns both within a scenario and among all scenarios into consideration. The heuristic-based method synthesizes data placements so that the number of reconfigurations, the average on-chip data access latency, and the number of off-chip memory accesses can be reduced. The experimental results show that, when performing on a set of real-world applications, the proposed algorithm achieves up to 11.72% reduction in average data access latency when compared to the method proposed for 2D CMPs with NUMA memory architecture.

Acknowledgment

This work is supported in part by research grants from NSC 102-2221-E-260-030-, and 101-2221-E-260-037-. We thank Professor Takeshi Tokuyama from Graduate School of Information Sciences, Tohoku University, for his valuable comments and suggestions.

References

[1] D. M. Ancajas et al. Dmr3d: dynamic memory relocation in 3d multicore systems. In *Proc. DAC '13*, pages 291–294, 2010.

[2] ARM. Processors. http://www.arm.com/zh/products/processors/cortex-a/cortex-a9.php.

[3] G. Chen et al. Application mapping for chip multiprocessors. In *Proc. DAC '08*, pages 620–625, 2008.

[4] Y.-J. Chen et al. Pm-cosyn: Pe and memory co-synthesis for mpsocs. In *Proc. DATE '10*, page 157, 2013.

[5] R. P. Dick et al. Tgff: Task graphs for free. In *Proc. international workshop on CODES '98*, pages 97–101, 1998.

[6] E3S. Embedded system synthesis benchmarks suite. http://ziyang.eecs.umich.edu/ dickrp/e3s/.

[7] EEMBC. Embedded microprocessor benchmark consortium. http://www.eembc.org/home.php.

[8] HORNET. Hornet-1.0 online available at. http://csg.csail.mit.edu/hornet/.

[9] A.-C. Hsieh et al. Thermal-aware memory mapping in 3d designs. *ACM TECS*, 13(1), 2013.

[10] M. Kandemir et al. Dynamic thread and data mapping for noc based cmps. In *Proc. DAC '09*, pages 852–857, 2009.

[11] T. Kgil et al. Picoserver: Using 3d stacking technology to enable a compact energy efficient chip heterogeneous multiprocessors. In *Proc. ASPLOS '06*, 2006.

[12] Y. Kitasho et al. Development of low power and high performance application processor (t6g) for multimedia mobile applications. In *Proc. ASPDAC '11*, 2011.

[13] Y. K. Kwok et al. Benchmarking and comparison of the task graph scheduling algorithms. *JPDC*, 59(3):381–422, 1999.

[14] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proc. ISCA '04*, page 453464, 2008.

[15] B. H. Meyer et al. Simultaneous synthesis of buses, data mapping and memory allocation for mpsoc. In *Proc. CODES+ISSS '07*, 2007.

[16] S. Pasricha et al. Cosmeca: Application specific co-synthesis of memory and communication. architectures for mpsoc. In *Proc. DATE '06*, pages 700–705, 2006.

[17] J. M. Paul et al. Benchmark-based design strategies for single chip heterogeneous multiprocessors. In *Proc. CODES+ISSS '04*, 2004.

[18] H. Saito et al. A chip-stacked memory for on-chip sram-rich socs and processors. *IEEE JSSC*, 45(1):15–22, 2010.

[19] L. Schor et al. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proc. CASES '12*, 2012.

[20] A. Schranzhofer et al. Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms. *IEEE TII*, 6(4):692–707, 2010.