PAPER

Static Mapping with Dynamic Switching of Multiple Data-Parallel Applications on Embedded Many-Core SoCs

Ittetsu TANIGUCHI[†a)], ***Member***, **Junya KAIDA**[†], ***Nonmember***, **Takuji HIEDA**[††], **Yuko HARA-AZUMI**[†††], ***and*** **Hiroyuki TOMIYAMA**[†], ***Members***

SUMMARY This paper studies mapping techniques of multiple applications on embedded many-core SoCs. The mapping techniques proposed in this paper are static which means the mapping is decided at design time. The mapping techniques take into account both inter-application and intra-application parallelism in order to fully utilize the potential parallelism of the many-core architecture. Additionally, the proposed static mapping supports dynamic application switching, which means the applications mapped onto the same cores are switched to each other at runtime. Two approaches are proposed for static mapping: one approach is based on integer linear programming and the other is based on a greedy algorithm. Experimental results show the effectiveness of the proposed techniques.

key words: *many-core SoCs, application mapping, system-level design, embedded systems*

1. Introduction

The embedded System-on-Chip (SoC) architecture has shifted from single-core to multi-core paradigm to realize improvements in power/performance efficiency, and it is now heading towards the many-core era. In order to fully utilize the high parallelism of the many-core architecture, mapping of application software onto cores is one of the important technologies. Especially in embedded SoCs, application mapping needs to take into account not only application-level parallelism (inter-application parallelism) but also data parallelism within applications (intra-application parallelism). One reason is that, unlike scientific applications, the amount of data parallelism inherent in individual embedded applications is limited. Another reason is that many embedded applications are essentially parallel.

This paper proposes two techniques for mapping multiple applications onto homogeneous many-core SoCs for embedded systems. The proposed techniques consider both inter-application and intra-application parallelisms simultaneously, and the mapping is determined at a design time. Additionally, proposed static mapping allows dynamic application switching such that the applications mapped onto the same cores are switched to each other at runtime. When some applications are not executed simultaneously, sharing of cores brings effective CPU utilization among these applications. One of the proposed techniques is an exact solution approach based on Integer Linear Programming (ILP), and the other is based on a greedy algorithm. In order to maximize the benefit of the mapping, two techniques decide the number of cores to be used for each application.

The rest of this paper is structured as follows. Related works are reviewed in Sect. 2. The static mapping problem is described in Sect. 3. Application mapping techniques are proposed in Sect. 4, and Sect. 5 shows experimental results. Finally, Sect. 6 concludes this paper.

2. Related Work

Application mapping for multi/many-core architectures has been an important research topic for many years. Recent studies include [1] which proposes a heuristic algorithm for static application mapping on multi-core embedded systems. The work supports application mapping to hardware accelerators as well as CPU cores, but data parallelism is not considered. In other words, an application is assigned a single core. Techniques presented in [2]–[5] take into account data parallelism within applications (intra-application parallelism) as well as application-level parallelism (inter-application parallelism). Their methods perform scheduling and mapping simultaneously, aiming at minimization of schedule length or pipeline throughput. Our work presented in this paper is similar to their works in a sense that we try to find the optimal number of cores for each application. However, our software model is different from their models – our models target multiple applications running concurrently and repeatedly at different execution rates, while their models take a task graph (i.e., a set of dependent applications) of a single application and try to minimize the execution time of a single activation of the application or to maximize the pipeline throughput. Our software model is widely applicable, and such embedded system is useful. Applications may be independent or dependent.

For such embedded systems, we studied static and exclusive application mapping of multiple data-parallel applications [6]. In our previous research, applications are exclusively mapped onto cores shown in Fig. 1. Exclusive mapping means that no two applications use the same cores, and such mapping takes big advantages in terms of the runtime overhead as demonstrated in [7]. In this paper, we propose two static application mapping, which allows

Manuscript received January 9, 2014.
Manuscript revised June 17, 2014.

[†]The authors are with Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
[††]The author is with Kyushu University, Fukuoka-shi, 819–0395 Japan.
[†††]The author is with Tokyo Institute of Technology, Tokyo, 152–8550 Japan.
a) E-mail: i-tanigu@fc.ritsumei.ac.jp

DOI: 10.1587/transinf.2014EDP7012

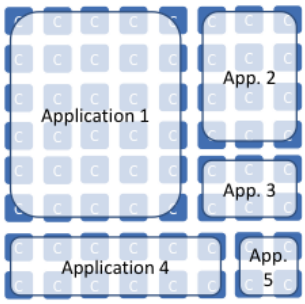

Fig. 1 An example of exclusive application mapping [6].

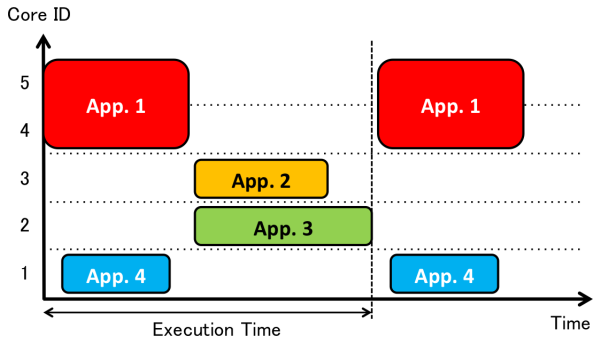

Fig. 2 An execution example of exclusive application mapping (w/o dynamic application switching).

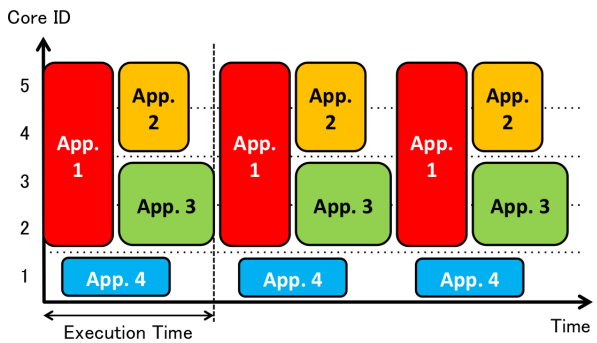

Fig. 3 An execution example of dynamic application switching.

the dynamic application switching, including an ILP-based technique and a greedy algorithm. Contribution of this research is to loose the mapping constraints from previous research [6]. This paper proposes the new mapping techniques to support dynamic application switching such that two applications share the same cores. Proposed mapping techniques introduce the information whether two applications need to be executed in parallel or not, and realize the dynamic application switching. Since proposed mapping handles dynamic application switching at runtime, CPU utilization is improved drastically. Proposed mapping also includes conventional exclusive mapping. To the best of our knowledge, this is the first paper which studies the static application mapping for such embedded systems.

3. Static Application Mapping Problem

3.1 Many-Core Architecture and Application Models

In this paper, we assume homogeneous many-core architectures with shared memory such as the SMYLEref architecture [8]. It is also assumed that the execution time of an application does not depend on the physical position of the application unless the application is assigned the same number of cores.

We assume embedded systems where multiple applications run in parallel. The applications are repeatedly executed at runtime in a cyclic way. Their execution can be periodic, aperiodic or sporadic, and their execution repetition rates may differ between applications. We implicitly assume that the applications are independent of each other. It is still possible to apply this work to dependent applications, but the obtained mapping results may not be optimal depending on how much the applications communicate with each other.

3.2 Dynamic Application Switching

In this work, applications are mapped onto cores in a static way. Static mapping means that application mapping decision is made at design time, and the applications never migrate over the cores at run time. Also, our mapping supports dynamic application switching, which means the applications mapped onto the same cores are switched to each other at runtime. Since some applications may not be executed simultaneously, sharing of cores brings effective CPU utilization among these applications. Therefore, proposed static mapping tries to utilize the cores to share with applications exclusively executed.

Figure 2 and Fig. 3 show execution example of exclusive mapping [6] and mapping supporting dynamic application switching proposed in this paper, respectively. X-axis means the execution time, and Y-axis means the core ID. As shown in Fig. 2, no two applications share the same cores because the applications are exclusively mapped. The exclusive mapping is necessary when applications are guaranteed to be executed in parallel. However, when specific applications are guaranteed to be never executed in parallel, these applications can share the same cores. This is a big opportunity to improve CPU utilization, but previous research [6] does not take into account this opportunity at all. The dynamic application switching proposed in this paper exploits the opportunity to share the same cores. When we preliminarily know the following two applications never execute in parallel, the execution shown in Fig. 3 is available.

- Applications 1 and 2
- Applications 1 and 3

These information becomes the input of the mapping problem. Comparing the two mapping results shown in Fig. 2 and Fig. 3, the dynamic application mapping obviously achieved higher throughput, and CPU utilization is drastically improved. Notice that proposed mapping techniques

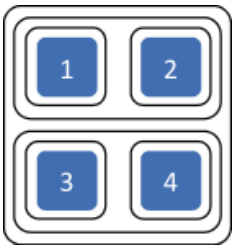

Fig. 4 Tiles.

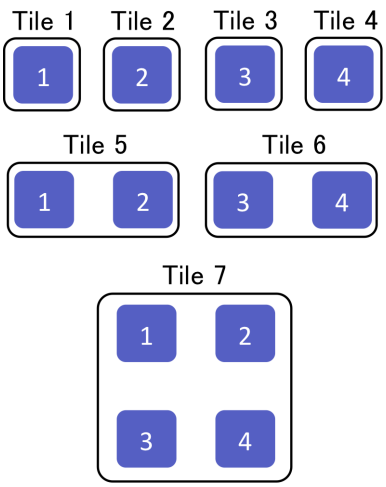

Fig. 5 Tiling example.

are applicable not only periodic execution, but also aperiodic or sporadic execution.

3.3 Problem Description

In order to describe a mapping problem, we introduce a concept of tile. A tile is a set of cores on which a single application can be mapped. Figure 4 shows an SoC with four cores. For simplicity without loss of generality, we assume that each application may use one, two or four cores. In this case, the 4-core SoC has the following seven ways of tiling.

- Tile 1: Core 1
- Tile 2: Core 2
- Tile 3: Core 3
- Tile 4: Core 4
- Tile 5: Cores 1 and 2
- Tile 6: Cores 3 and 4
- Tile 7: Cores 1, 2, 3 and 4

For easy understandings, Fig. 5 also shows the above mentioned tiling examples. We say that two tiles are overlapped if those tiles have at least one identical core. In case of Fig. 5, Tiles 1 and 5 are overlapped, but Tiles 3 and 5 are not overlapped. Apparently, if two applications need to run in parallel, the tiles of the two applications must not be overlapped.

In general, the execution time and energy consumption of an application depend on the number of cores which

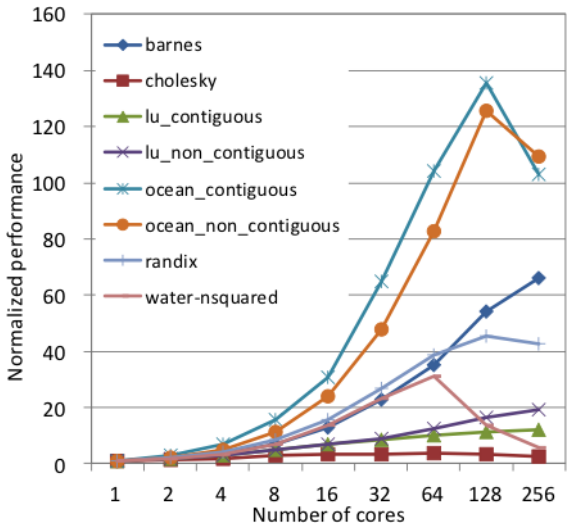

Fig. 6 Normalized performance on different number of cores [6].

the application uses. Figure 6 shows the normalized performance of eight application programs from the SPLASH-2 benchmark suite executed on the Graphite cycle-accurate multi-core simulator [9]. For each program, we changed the number of cores from 1 to 256, and measured the number of execution cycles. The graph shows that eight programs feature different performance scalability curves. For example, the performance of *ocean_contiguous* scales up nicely until 128 cores, but it drops at the point of 256 cores. *Barnes* continuously scales up to 256 cores, but the performance improvement is relatively lower than *ocean_contiguous*. *Cholesky* does not scale up at all.

As we see in Fig. 6, different applications present different performance scalability curves, meaning that the optimal number of cores to be assigned depends on the application. In addition, we have to remind that the total number of cores is limited. For example, let us consider a scenario where we need to map *barnes* and *randix* onto a 64-core SoC. Of course, we cannot allocate 64 cores to both of the two applications because we have only 64 cores in total. In this case, assigning 32 cores to each application is a natural solution.

In the example above, we used the normalized performance as a metric for application mapping, but in practice we need to consider other factors such as energy consumption. Hereafter, for generality, let gain be a metric which indicates not only performance, but also energy consumption and other important factors.

Let $gain_{i,j}$ indicate the gain of i-th application when the application is assigned j cores. We assume that $gain_{i,j}$ for each application is given prior to application mapping. Then, the static application mapping problem is defined as follows: Given $gain_{i,j}$ for each application and the total number of cores available, determine the number of cores for each application so that the total gain is maximized.

4. Static Application Mapping Techniques

4.1 An ILP-Based Technique

In order to obtain optimal results of static application mapping with dynamic switching, this section explains an ILP-based technique.

Let $gain_{i,j}$ be the gain of application i in case it is mapped to tile j. Then we introduce a decision variable $map_{i,j}$: $map_{i,j}$ takes 1 if application i is mapped to tile j, otherwise 0. Then, the object of this problem is to maximize following function:

maximize:

$$\sum_i \sum_j map_{i,j} \times gain_{i,j} \tag{1}$$

In order to describe feasibility of the mapping result, we introduce two symbols: $parallel_{i1,i2}$ and $overlap_{j1,j2}$. $parallel_{i1,i2}$ indicates whether two applications need to be executed in parallel or not. $parallel_{i1,i2}$ takes 1 if applications $i1$ and $i2$ need to be executed in parallel, otherwise 0. $overlap_{j1,j2}$ indicates whether two tiles are overlapped or not. $overlap_{j1,j2}$ takes 1 if tiles $j1$ and $j2$ are overlapped, otherwise 0. Then we call the mapping is feasible (not overlapped) when following formula becomes true for all combinations of applications and tiles.

$$\begin{aligned} \forall(i1, i2, j1, j2), (parallel_{i1,i2} = 1) \rightarrow \\ \Big(\big(map_{i1,j1} \cdot overlap_{j1,j2} = 0\big) \\ \vee \big(map_{i2,j2} \cdot overlap_{j1,j2} = 0\big)\Big) \end{aligned} \tag{2}$$

Now we assume that applications $i1$ and $i2$ are never executed in parallel ($parallel_{i1,i2} = 0$). Then the applications $i1$ and $i2$ can be freely mapped to any tiles. On the other hand, we assume that applications $i1$ and $i2$ need to be executed in parallel ($parallel_{i1,i2} = 1$). Then the applications $i1$ and $i2$ are respectively mapped to the tiles $j1$ and $j2$ only if the tiles $j1$ and $j2$ are not overlapped ($overlap_{j1,j2} = 0$). If the tiles $j1$ and $j2$ are overlapped ($overlap_{j1,j2} = 1$), $map_{i1,j1}$ or $map_{i2,j2}$ takes 0. Therefore, the feasible mapping is obtained.

Additionally, following formula becomes true.

$$\forall i, \sum_j map_{i,j} = 1 \tag{3}$$

Solving the ILP formulation by ILP solver, we obtain the optimal mapping result. However, it takes long time to solve the large-scale problem, and effective mapping algorithm is necessary.

4.2 A Greedy Algorithm

This section describes proposed mapping algorithm to solve the same problem formulated by ILP in our previous section. The proposed mapping algorithm is based on simple greedy, and selects the better tile for each application to increase gain as much as possible. The initial solution is a solution such that each application is randomly assigned to each unique core, and any overlap is not allowed to keep feasibility of the initial solution. Thus the tiles such that each core is only assigned must be prepared in the set of tiles. Then proposed algorithm is described as follows:

1. The initial solution is generated.
2. Following procedures are iterated until all applications are updated only once.
   a. Find a combination of the application and the tile to earn the largest gain increment. Then the updated mapping must keep its feasibility.
   b. Finish finding new mapping, and decides the selected tile as the mapping results for the application.
3. Output the mapping result.

Since the algorithm iterates the update for each application, the complexity of the algorithm becomes $O(N_{tile} \cdot N_{appl}^2)$, where N_{tile} and N_{appl} means the number of tiles and applications, respectively. Notice that the proposed algorithm keeps feasibility of the mapping results. Therefore the number of cores must be more than the number of applications in order to obtain the feasible initial solution.

For easy understanding, we demonstrate the proposed algorithm. Now we assume a 4-core SoC with the tiling shown in Fig. 5. Then $overlap_{j1,j2}$ is extracted as shown in Table 1. Since Tile 1 and Tile 5 are overlapped, $overlap_{1,5}$ and $overlap_{5,1}$ take 1. On the other hand, $overlap_{1,6}$ and $overlap_{6,1}$ take 0 because Tile 1 and Tile 6 do not share any cores. We also assume four applications, and $gain_{i,j}$ and $parallel_{i1,i2}$ are defined in Table 2 and Table 3.

Figure 7 shows mapping procedures by the proposed algorithm. Step 1 in Fig. 7 shows an initial solution, and each application is assigned to each unique core. In order to

Table 1 Example of $overlap_{j1,j2}$.

	Tile $j1$						
Tile $j2$	1	2	3	4	5	6	7
1	1	0	0	0	1	0	1
2	0	1	0	0	1	0	1
3	0	0	1	0	0	1	1
4	0	0	0	1	0	1	1
5	1	1	0	0	1	0	1
6	0	0	1	1	0	1	1
7	1	1	1	1	1	1	1

Table 2 Example of $gain_{i,j}$.

	Application i			
Tile j	1	2	3	4
1	10	20	5	10
2	10	20	5	10
3	10	20	5	10
4	10	20	5	10
5	20	40	10	15
6	20	40	10	15
7	20	80	15	30

increase gain as much as possible, the algorithm checks the largest gain increment for each application. For Application 1, changing the mapping from Tile 1 to Tile 5 brings the largest gain increment ($gain_{1,5} - gain_{1,1} = 10$) keeping the feasibility of solution because two applications executed in parallel are not mapped on overlapped tiles. For example, Application 1 mapped on Tile 6 and Application 3 mapped on Tile 3 are not executed in parallel. For Application 2, mapping to Tile 7 brings the largest gain increment ($gain_{2,7} - gain_{2,2} = 60$) keeping the feasibility because Application 2 is not executed with any other applications in parallel as shown in Table 3. However, Application 3 and 4 are not changed the mapping to Tile 5, 6 or 7 because Application 1, 3 and 4 are executed in parallel, and given tiles do not allow to increase the gain under the constraint. Then the mapping of Application 2 is changed from Tile 2 to Tile 7 like Step 2 in Fig. 7.

In the same way, the mapping of Application 1 is changed from Tile 1 to Tile 5 like Step 3 in Fig. 7. Notice that Application 3 and 4 are not updated the mapping as mentioned before, and the mapping result in Step 3 in Fig. 7 becomes the output.

Table 3 Example of $parallel_{i1,i2}$.

	Application $i1$			
Application $i2$	1	2	3	4
1	0	0	1	1
2	0	0	0	0
3	1	0	0	1
4	1	0	1	0

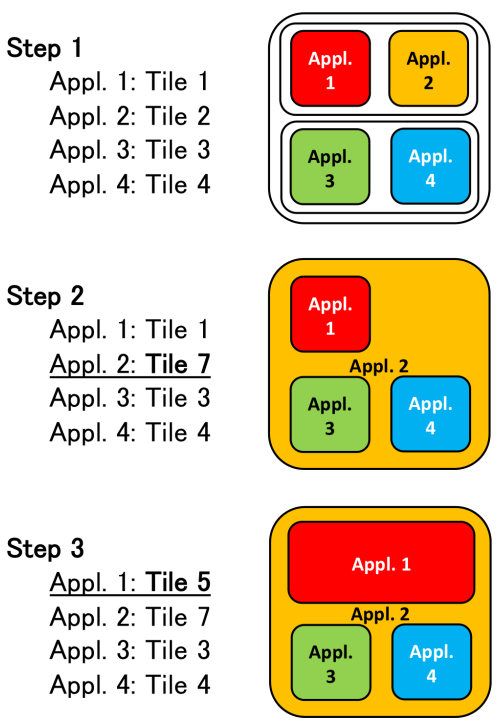

Fig. 7 Example of mapping algorithm.

5. Experiments

In order to demonstrate the efficiency of proposed mapping techniques, we compared the results obtained by the ILP-based technique and the greedy algorithm. Notice that the ILP-based technique obtains optimal results. The ILP problem was solved with IBM CPLEX12.5, and all experiments were performed on Intel Xeon (2.0GHz, 16cores/32thread) and 128GB memory machine.

We prepared three sets of application programs based on the SPLASH-2 benchmark suite. Benchmark set 1, 2, and 3 include 4, 8, and 16 application programs, respectively. Benchmark set 1 includes *lu_non_contiguous*, *ocean_contiguous*, *ocean_non_contiguous*, and *water – nsquared* from the SPLASH-2 benchmark suite. Benchmark set 2 includes eight application programs shown in Fig. 6. Benchmark set 3 includes the same two programs for each application program.

The values of the two-dimensional matrix $parallel_{i1i2}$ are randomly decided based on density d, which indicates the percentage of value 1. $d = 100\%$ indicates $parallel_{i1i2} = 1$ for any two applications, meaning that all applications should run in parallel. On the other hand, $d = 0\%$ indicates that all cores are available to every application. Notice that $d = 100\%$ also means any two applications cannot run in parallel, and this corresponds to the exclusive mapping [6].

In this experiment, we suppose the application is mapped to neighboring 2^n cores. Figure 8 shows tiling example for 16 cores. In this experiment, for 16 cores, we prepared tiles with 1 core, 2 cores, 4 cores, 8 cores, and 16 cores. The number of tiles with 1 core, 2 cores, 4 cores, 8 cores, and 16 cores are 16, 8, 4, 2, and 1, respectively. Therefore the number of tile for 2^n cores is $2^{n+1} - 1$. We performed the experiment on 2^n cores ranging from the number of applications to 1024 cores.

Table 4, 5, and 6 show comparisons of gain for each benchmark set. We varied the density d from 0% to 100%, and the number of cores. "Greedy" and "ILP" mean the results obtained by proposed greedy algorithm and ILP-based technique. Notice that "—" mark indicates that the mapping result cannot be obtained by out of memory. Table 7

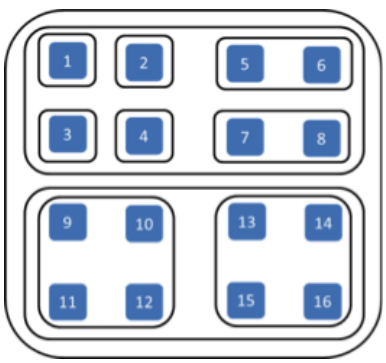

Fig. 8 Example of tiling for 16 cores.

Table 4 Comparison of gain (benchmark set 1 (#application=4)).

	d = 0%		d = 25%		d = 50%		d = 75%		d = 100%	
#Cores	Greedy	ILP	Greedy	ILP	Greedy	ILP	Greedy	ILP	Greedy	ILP
4	18.4	18.4	9.0	9.0	9.0	9.0	6.1	6.1	4.0	4.0
8	38.7	38.7	18.4	18.4	14.3	18.4	12.7	13.4	10.6	11.0
16	75.0	75.0	38.7	38.7	30.8	38.7	27.3	28.2	24.4	24.4
32	144.7	144.7	75.0	75.0	60.3	75.0	53.7	58.0	48.5	48.5
64	231.0	231.0	144.7	144.7	119.0	144.7	109.3	109.3	100.8	100.8
128	309.1	309.1	231.0	231.0	192.2	231.0	184.1	187.9	172.6	172.6
256	312.0	312.0	309.1	309.1	266.3	309.1	262.4	273.8	250.5	262.0
512	312.0	312.0	312.0	312.0	309.1	312.0	309.1	312.0	309.1	309.1
1024	312.0	312.0	312.0	312.0	312.0	312.0	312.0	312.0	312.0	312.0

Table 5 Comparison of gain (benchmark set 2 (#application=8)).

	d = 0%		d = 25%		d = 50%		d = 75%		d = 100%	
#Cores	Greedy	ILP	Greedy	ILP	Greedy	ILP	Greedy	ILP	Greedy	ILP
8	61.8	61.8	29.3	29.7	16.3	21.8	12.8	18.6	8.0	8.0
16	113.7	113.7	57.4	58.8	39.6	43.5	37.9	40.1	23.8	23.8
32	206.4	206.4	107.4	111.5	74.9	83.3	78.3	79.1	49.8	49.8
64	318.5	318.5	195.6	204.5	145.9	157.7	154.2	157.2	104.8	105.0
128	423.6	423.6	316.9	316.9	241.3	261.8	257.0	264.0	182.3	182.6
256	439.0	439.0	422.4	422.4	343.1	381.7	368.7	381.7	272.4	283.9
512	439.0	439.0	438.4	438.4	432.1	435.5	421.6	422.4	391.9	391.9
1024	439.0	439.0	439.0	439.0	439.0	439.0	439.0	439.0	435.4	435.5

Table 6 Comparison of gain (benchmark set 3 (#application=16)).

	d = 0%		d = 25%		d = 50%		d = 75%		d = 100%	
#Cores	Greedy	ILP	Greedy	ILP	Greedy	ILP	Greedy	ILP	Greedy	ILP
16	227.4	227.4	65.3	92.2	39.8	67.5	22.0	46.8	16.0	16.0
32	412.9	412.9	168.3	175.8	112.2	131.5	83.4	97.1	47.6	47.6
64	637.1	637.1	314.0	332.8	239.7	250.8	188.2	195.9	111.9	111.9
128	847.3	847.3	514.9	542.3	424.1	436.8	346.9	351.4	210.2	217.8
256	877.9	877.9	723.4	753.5	644.3	654.7	534.0	563.0	328.0	371.8
512	877.9	877.9	855.1	877.9	806.0	833.8	760.9	766.0	538.3	568.9
1024	877.9	—	877.9	—	874.5	—	860.0	—	736.4	—

Table 7 Comparison of runtime [sec] (benchmark set 3 (#application=16)).

	d = 0%		d = 25%		d = 50%		d = 75%		d = 100%	
#Cores	Greedy	ILP	Greedy	ILP	Greedy	ILP	Greedy	ILP	Greedy	ILP
16	< 0.1	0.5	< 0.1	21.0	< 0.1	11.6	< 0.1	20.6	< 0.1	1.2
32	< 0.1	1.8	< 0.1	15.1	< 0.1	21.2	< 0.1	20.0	< 0.1	5.0
64	< 0.1	7.2	< 0.1	10.9	< 0.1	54.7	< 0.1	38.4	< 0.1	14.7
128	< 0.1	30.0	< 0.1	41.7	< 0.1	90.8	< 0.1	121.4	< 0.1	25.4
256	< 0.1	117.3	< 0.1	162.4	< 0.1	253.2	< 0.1	404.1	< 0.1	285.3
512	< 0.1	461.6	< 0.1	640.8	< 0.1	826.9	< 0.1	1173.7	< 0.1	1109.7
1024	< 0.1	—	< 0.1	—	< 0.1	—	< 0.1	—	< 0.1	—

also shows comparison of CPU time for benchmark set 3 under the same condition of Table 6.

First of all, previous ILP-based technique did not obtain the results in all cases because of out of memory. Especially, for benchmark set 3, ILP-based technique only obtained the results less than one third of all combinations of density and cores. However, the proposed greedy algorithm obtained in all cases. As shown in Table 7, runtime for all cases were less than 0.1 sec. Since the complexity of proposed greedy-based algorithm is polynomial order, the proposed algorithm is practical in terms of runtime.

In terms of the quality of the solutions, the greedy algorithm obtained almost optimal results in most cases. However, some results have measurable difference between optimal results, and relative error was about 15% to 25%. Especially, in case d = 75% on benchmark set 3, relative error was more than 50%, and further improvement is necessary. Notice that the quality of the solution is largely depends on the initial solution, the set of tiles, etc. Preparing various tiles brings further opportunity to find a better solution in greedy-based iteration, and it is expected that the results be improved more and more. Therefore, the proposed greedy algorithm is quite practical in terms both of runtime and quality.

Figure 9 is plotted results of ILP shown in Table 5. As shown in Fig. 9, smaller density d, which means looser

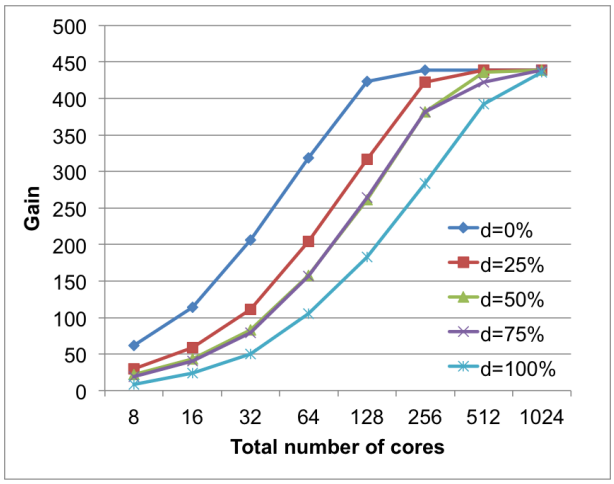

Fig. 9 Results of ILP-based mapping for benchmark set 2 (#application=8).

mapping constraints, brings larger gain for all cases. For example, in case total number of cores was 64, the case $d = 0\%$ earns more than three times larger gain than the case $d = 100\%$, the result obtained by the exclusive mapping [6]. This means the gain largely increases by allowing parallel execution and dynamic switching.

6. Conclusions

In this paper, we have proposed static mapping with dynamic switching of multiple applications on embedded many-core SoCs. The mapping techniques proposed in this paper take into account both inter-application and intra-application parallelisms in order to fully utilize the potential parallelism of the many-core architecture. Additionally, proposed mapping supports dynamic application switching, which means the applications mapped onto the same cores are switched to each other at runtime. Two approaches are proposed for static mapping: one approach is based on integer linear programming and the other is based on a greedy algorithm. Experimental results show that the proposed mapping obtained the results less than 0.1 second for each case, and the effective and practical mapping is available.

At present, this work does not assume that applications have deadline constraints. In the future, we will take into account deadline constraints of individual applications.

Acknowledgments

The authors would like to thank Professor Koji Inoue and Professor Hiroshi Sasaki for their support to conduct the experiments. This work was in part supported by NEDO.

References

[1] Y. Ando, S. Shibata, S. Honda, H. Tomiyama, and H. Takada, "Fast design space exploration for mixed hardware-software embedded systems," SoC Design Conference (ISOCC), 2011 International, pp.92–95, 2011.

[2] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A framework for exploiting task and data parallelism on distributed memory multicomputers," IEEE Trans. Parallel Distrib. Syst., vol.8, no.11, pp.1098–1116, Nov. 1997.

[3] H. Yang and S. Ha, "Ilp based data parallel multi-task mapping/scheduling technique for mpsoc," SoC Design Conference, 2008. ISOCC '08. International, pp.I–134–I–137, 2008.

[4] H. Yang and S. Ha, "Pipelined data parallel task mapping/scheduling technique for mpsoc," Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09, pp.69–74, 2009.

[5] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz, "An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications," IEEE Trans. Parallel Distrib. Syst., vol.20, no.8, pp.1158–1172, 2009.

[6] J. Kaida, Y. Hara-Azumi, T. Hieda, I. Taniguchi, H. Tomiyama, and K. Inoue, "Static mapping of multiple data-parallel applications on embedded many-core socs," IEICE Trans. Inf. & Syst., vol.E96-D, no.10, pp.2268–2271, Oct. 2013.

[7] H. Xiao, T. Isshiki, A.U. Khan, D. Li, H. Kunieda, Y. Nakase, and S. Kimura, "A low-cost and energy-efficient multiprocessor system-on-chip for uwb mac layer," IEICE Trans. Inf. & Syst., vol.E95-D, no.8, pp.2027–2038, Aug. 2012.

[8] M. Kondo, S. Nguyen, T. Hirao, T. Soga, H. Sasaki, and K. Inoue, "Smyleref: A reference architecture for manycore-processor socs," Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific, pp.561–564, 2013.

[9] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on, pp.1–12, 2010.

Ittetsu Taniguchi received B.E., M.E., and Ph.D. degrees from Osaka University in 2004, 2006, and 2009, respectively. He is currently a lecturer at Ritsumeikan University, Japan. From 2007 to 2008, he was a Ph.D. researcher at IMEC, Belgium. His research interests include system level design methodology, and combinatorial optimization problems. He is a member of IEEE, ACM, IEICE, and IPSJ.

Junya Kaida received B.E. and M.E. from Ritsumeikan University in 2012 and 2014, respectively. His research interests include many-core architecture and application mapping techniques.

Takuji Hieda received Bachelor of Engineering and Master and Doctor of Information Science and Technology degrees from Osaka University in 2005, 2007, and 2011, respectively. From 2011 through 2013, he was a research fellow in the Research Organization of Science and Engineering, Ritsumeikan University. He has been a Research Fellow in the Faculty of Information Science and Electrical Engineering, Kyushu University. His research interests include compiler optimization and design space exploration for embedded processors. He is a member of IEEE, IEICE, and IPSJ.

Yuko Hara-Azumi received her Ph.D. degree in computer science from Nagoya University in 2010. From 2010 to 2012, she was a JSPS postdoctoral research fellow at Ritsumeikan University. In 2012, she joined the Graduate School of Information Science, Nara Institute of Science and Technology, as an assistant professor. Since 2014, she has been with the Graduate School of Science and Engineering, Tokyo Institute of Technology, where she is currently an associate professor. Her research interests include system-level design automation for embedded/dependable systems. She currently serves as organizing and program committees of several premier conferences including ICCAD, ASP-DAC, RTCSA, and so on. She is a member of IEEE, IEICE and IPSJ.

Hiroyuki Tomiyama received his Ph.D. degree in computer science from Kyushu University in 1999. From 1999 to 2001, he was a visiting postdoctoral researcher with the Center of Embedded Computer Systems, University of California, Irvine. From 2001 to 2003, he was a researcher at the Institute of Systems & Information Technologies/KYUSHU. In 2003, he joined the Graduate School of Information Science, Nagoya University, as an assistant professor, and became an associate professor in 2004. In 2010, he joined the College of Science and Engineering, Ritsumeikan University as a full professor. His research interests include design automation, architectures and compilers for embedded systems and systems-on-chip. He currently serves as editor-in-chief for IPSJ Transactions on SLDM. He has also served on the organizing and program committees of several premier conferences including ICCAD, DAC, DATE, ASP-DAC, CODES+ISSS, and so on. He is a member of ACM, IEEE, IPSJ and IEICE.