IEICE TRANS. INF. & SYST., VOL.E96–D, NO.10 OCTOBER 2013

LETTER

Static Mapping of Multiple Data-Parallel Applications on Embedded Many-Core SoCs*

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

SUMMARY This paper studies the static mapping of multiple applications on embedded many-core SoCs. The mapping techniques proposed in this paper take into account both inter-application and intra-application parallelism in order to fully utilize the potential parallelism of the many-core architecture. Two approaches are proposed for static mapping: one approach is based on integer linear programming and the other is based on a greedy algorithm. Experiments 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 the single-core to multi-core paradigm because of the need for improved power/performance efficiency, and we are 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 inherently 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 parallelism simultaneously. 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. The 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. Two application mapping techniques considering both inter-application and intra-application parallelism are proposed in Sect. 3, and experiments are shown in Sect. 4. Finally, Sect. 5 concludes this paper.

Manuscript received February 21, 2013.
Manuscript revised June 24, 2013.

[†]The authors are with Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
[††]The author is with Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
[†††]The author is with Kyushu University, Fukuoka-shi, 819–0395 Japan.

*A preliminary version of this work has partially been presented at International SoC Design Conference in November 2012.

a) E-mail: ht@fc.ritsumei.ac.jp

DOI: 10.1587/transinf.E96.D.2268

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 task mapping on multi-core embedded systems. The work supports task mapping to hardware accelerators as well as CPU cores, but data parallelism is not considered. In other words, a task is assigned a single core. The techniques presented in [2]–[5] take into account data parallelism within tasks as well as task parallelism. Their methods take a task graph as input and perform task scheduling and mapping simultaneously, aiming at minimization of schedule length or maximization of 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 task or application. However, our software model is different from theirs in that they take a task graph (i.e., a set of dependent tasks) of a single application as input and try to minimize the execution time of a single activation of the application or to maximize the pipeline throughput, while we target embedded systems where multiple applications run concurrently and repeatedly at different execution rates. The applications may be independent or dependent. To the best of our knowledge, this is the first paper which studies application mapping for such embedded systems.

3. Application Mapping Techniques

This section proposes two techniques for the static mapping of applications onto cores. The techniques determine, for each application, the number of cores onto which the application is mapped, considering both inter-application and intra-application parallelism simultaneously.

3.1 Many-Core Architecture and Application Models

In this paper, we assume homogeneous many-core architectures such as the SMYLEref architecture [6]. If there are N applications in the system, the architecture must contain at

Copyright © 2013 The Institute of Electronics, Information and Communication Engineers

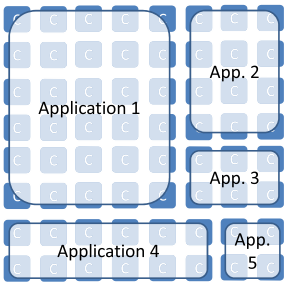

Fig. 1 An example of exclusive application mapping.

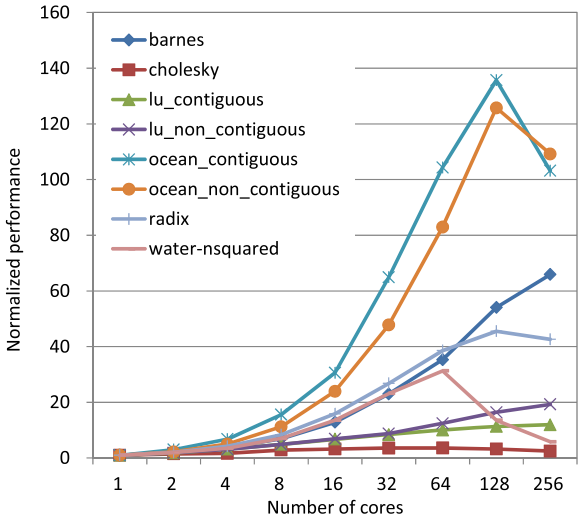

Fig. 2 Normalized performance on different number of cores.

least N cores. The architecture must support data-parallel execution within an application as well as parallel execution of multiple applications. We do not assume any specific interconnection network among cores, but assume 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 implicitly assume that each core has local memory where the application code is stored. In case the application code is stored in shared memory, each core should have cache memory in order to reduce memory access traffic.

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 Problem Description

In this work, applications are mapped onto cores in a *static* and *exclusive* way. Static mapping means that application mapping decision is made at a design time, and the applications never migrate over the cores at run time. This reduces the runtime overhead (in terms of performance and memory requirement) at the cost of lower CPU utilization compared with dynamic mapping. Also, our mapping is exclusive, which means that no two applications use the same cores. An application may use multiple cores for data-parallel execution, but a core is assigned only a single application. Such single-application (single-task) mapping further reduces the runtime overhead as demonstrated in [7].

Figure 1 depicts an example of application mapping, where five applications are mapped onto a 64-core SoC in a static and exclusive manner. As shown in the figure, applications may use the different numbers of cores. For example, Application 1 has a rich amount of data parallelism, and thus the application is assigned a large number of cores. On the other hand, applications with poor data parallelism such as Application 5 are assigned the small number of cores.

In general, the execution time and energy consumption of an application depend on the number of cores which the application uses. Figure 2 shows the normalized performance of eight application programs from the SPLASH-2 benchmark suite executed on the Graphite cycle-accurate multi-core simulator [8]. For each program, we changed the number of cores from 1 to 256, and measured the number of execution cycles. The cores are connected in a 2D-mesh structure, and each core is based on Intel x64 ISA (1GHz, single issue) with private L1 I-cache (32KB, 4 ways), private L1 D-cache (32KB, 4 ways), and L2 coherent cache (unified, 512KB, 8 ways). In addition, the architecture has main memory shared by all cores. Figure 2 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. 2, 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 radix 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.

3.3 An ILP-Based Technique

We formulate the application mapping problem stated in Sect. 3.2 as an Integer Linear Programming (ILP) problem. By solving the ILP problem using a commercial or free ILP solver, an optimal application mapping solution is obtained.

Let $map_{i,j}$ be a 0/1 decision variable whose value is 1 if application i is assigned j cores. Otherwise, $map_{i,j}$ is 0. Also, let $Ncore$ indicate the number of cores available on the target SoC. Then, the ILP formulation of the static application mapping problem is formulated as follows:

$$\text{Maximize :} \quad \sum_i \sum_j map_{i,j} \times gain_{i,j} \tag{1}$$

$$\text{Subject to :} \quad \forall i, \sum_j map_{i,j} = 1 \tag{2}$$

$$\sum_i \sum_j j \times map_{i,j} \leq Ncore \tag{3}$$

Objective function (1) to be maximized expresses the total gain of the applications. Formulas (2) and (3) are the constraints about mapping. Formula (2) says that each application should be assigned at least one core. Formula (3) shows that the total number of assigned cores should not exceed the total number of cores available on the SoC.

3.4 A Greedy Algorithm

Since the ILP problem defined in Sect. 3.3 is NP-hard, exact solutions may not be obtained in a practical time for a large set of applications and a large number of cores. Therefore, we have developed a fast greedy algorithm for the mapping problem.

Let map_i denote the number of cores assigned to application i. Then, our greedy algorithm works as follows.

1. $map_i = 1$ for each application.
2. For each application, calculate the gain increase, i.e., $gain_{i,map_i+1} - gain_{i,map_i}$, in case we allocate an additional core to the application.
3. Select the application with the largest gain increase, and set $map_i = map_i + 1$.
4. Repeat steps 2 to 4 until no core is available.

The computational complexity of the greedy algorithm is $O(Napp \cdot Ncore)$, where $Napp$ and $Ncore$ are the number of applications and that of cores, respectively.

4. Experiments

We have evaluated the two application mapping techniques presented in Sect. 3, i.e., the ILP-based approach and the greedy algorithm. We used IBM ILOG CPLEX 12.5 as an ILP solver. We used three sets of application programs based on the SPLASH-2 benchmark suite as shown in Fig. 2. Set 1 includes water_nsquared, ocean_non_contiguous, ocean_contiguous and lu_non_contiguous, while Set 2 includes radix, lu_contiguous, cholesky and barnes. Set 3 includes all of the eight programs. As gain values, we used the normalized performance whose baseline is the execution on a single core as shown in Fig. 2.

The experimental results for Sets 1, 2 and 3 are shown in Figs. 3, 4 and 5, respectively, where the Y-axis presents the normalized gain. Because, to our knowledge, no existing techniques are directly applicable to our application mapping problem, we compared our techniques with a simpler mapping technique with which cores are evenly allocated to

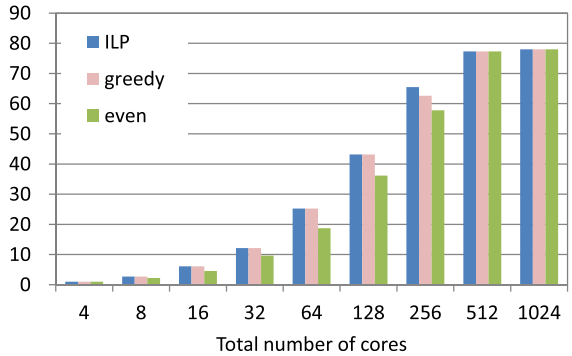

Fig. 3 Results of three mapping techniques for Set 1.

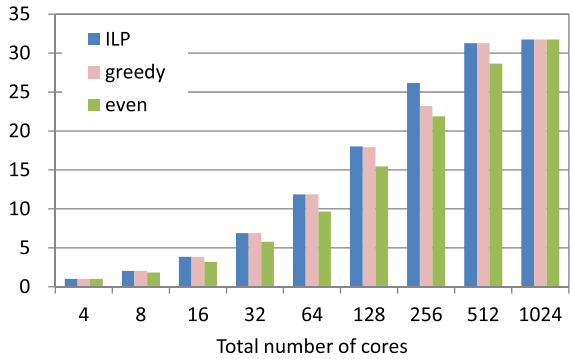

Fig. 4 Results of three mapping techniques for Set 2.

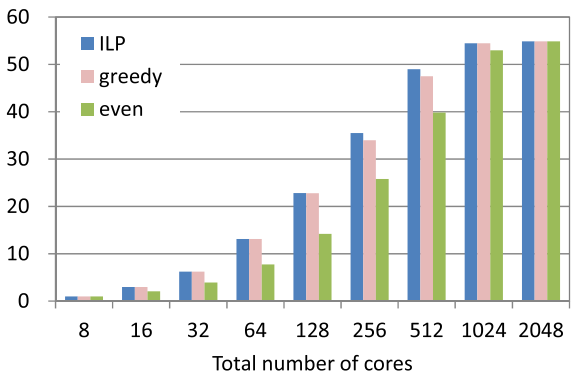

Fig. 5 Results of three mapping techniques for Set 3.

Table 1 Computation time [milliseconds].

	Set 1		Set 2		Set 3	
# cores	ILP	greedy	ILP	greedy	ILP	greedy
4	280	0.00	252	0.00	n.a.	n.a.
8	512	0.81	390	0.75	260	0.00
16	504	1.60	434	1.55	562	1.12
32	634	2.01	562	1.88	586	1.72
64	530	2.40	566	1.80	564	2.39
128	840	2.77	666	2.35	582	3.74
256	608	2.81	558	2.62	580	3.20
512	620	3.09	594	3.04	580	4.55
1024	246	3.01	232	3.15	510	5.89
2048	n.a.	n.a.	n.a.	n.a.	198	6.17

all applications. For example, if we have 32 cores for four applications, each application is assigned eight cores. This mapping technique is labeled "even" in the figures.

In any cases, our ILP technique is the best, but we observe that our greedy algorithm also yields the optimal or near-optimal mapping solutions in many cases. The "even" technique, on the other hand, could hardly find good mapping solutions because it does not take into account the characteristics of the applications. This is observed clearly for Set 3 where the "even" technique is less efficient than for Sets 1 and 2. As seen in Fig. 2, ocean_contiguous and ocean_non_contiguous scale up nicely along with the number of cores, while cholesky does not. Set 3 contains these applications with such different characteristics, and therefore, the "even" technique is not good for Set 3.

In general, the results in Figs. 3, 4 and 5 present similar characteristics. As an example, let us take a closer look at the results for Set 3 in Fig. 5. Since Set 3 contains eight applications, when the total number of cores is eight, each application is assigned a single core by any of the three mapping techniques, and this is the only feasible solution (thus, the best). In the rightmost case, 2048 cores are more than sufficient for the eight applications, because each application is assumed to be assigned at most 256 cores as shown in Fig. 2. Therefore, any of the three mapping techniques easily found the optimal mapping solution, i.e., 256 cores for barnes, lu_contiguous and lu_non_contiguous, 128 cores for radix, ocean_contiguous and ocean_non_contiguous, and 64 cores for cholesky and water_nsquared. When the number of cores is between 16 and 1024, there exist a number of feasible mapping solutions to be explored, and that is why the results of the three techniques differ.

The runtimes of the ILP solver and the greedy algorithm on Intel Core i5 (1.7GHz, 2 cores / 4 threads) with 4GB memory are presented in Table 1. The runtime of the ILP solver was within 1 second in our experiments†, and the greedy algorithm was much faster.

†In our experiments, the ILP solver was fast enough because we restricted that each application is assigned 2^n cores where n is a positive integer. This limitation comes from our simulation setups, not from our mapping techniques. If we assign cores in a finer granularity, the runtime will be much longer.

5. Conclusions

In this paper, we have proposed static application mapping techniques for embedded many-core SoCs. The proposed techniques take into account both inter-application and intra-application parallelism in order to efficiently utilize the potential parallelism of the many-core architecture. Experiments show the effectiveness of the mapping techniques.

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.

Acknowledgements

The authors would like to thank Professor Hiroshi Sasaki for his support in conducting 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," International SoC Design Conference (ISOCC), 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," International SoC Design Conference (ISOCC), 2008.

[4] H. Yang and S. Ha, "Pipelined data parallel task mapping/scheduling technique for MPSoC," Design Automation and Test in Europe (DATE), 2009.

[5] N. Vydyanathan, S. Krishnamoorthy, G.M. Sabin, U.V. Catalyurek, T. Kurc, P. Sadayappan, and J.H. 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, Aug. 2009.

[6] M. Kondo, S.T. Nguyen, T. Hirao, T. Soga, H. Sasaki, and K. Inoue, "SMYLEref: A reference architecture for manycore-processor SoCs," Asia and South Pacific Design Automation Conference (ASP-DAC), 2013.

[7] H. Xiao, T. Isshiki, A. Ullah 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] J.E. 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.