TECSCE: HW/SW Codesign Framework for Data Parallelism Based on Software Component

Takuya Azumi, Yasaman Syahkal, Yuko Hara-Azumi, Hiroshi Oyama, Rainer Dömer

TECSCE: HW/SW Codesign Framework for Data Parallelism Based on Software Component

Takuya Azumi[1], Yasaman Samei Syahkal[2], Yuko Hara-Azumi[3], Hiroshi Oyama[4], and Rainer Dömer[2]

[1] College of Information Science and Engineering, Ritsumeikan University
takuya@cs.ritsumei.ac.jp
[2] Center for Embedded Computer Systems, University of California, Irvine
{ysameisy,doemer}@uci.edu
[3] Graduate School of Information Science, Nara Institute of Science and Technology
yuko-ha@is.naist.jp
[4] OKUMA Corporation, hi-ooyama@okuma.co.jp

Abstract. This paper presents a hardware/software (HW/SW) codesign framework (TECSCE) which enables software developers to easily design complex embedded systems such as massive data-parallel systems. TECSCE is implemented by integrating TECS and SCE: TECS is a component technology for embedded software, and SCE provides an environment for system-on-a-chip designs. Since TECS is based on standard C language, it allows the developers to start the design process easily and fast. SCE is a rapid design exploration tool capable of efficient MPSoC implementation. TECSCE utilizes all these advantages since it supports transformation from component descriptions and component sources to SpecC specification, and lets the developers decide data partitioning and parallelization at a software component level. Moreover, TECSCE effectively duplicates software components, depending on their degree of data parallelizing, to generate multiple SpecC specification models. An application for creating a panoramic image removing objects, such as people, is illustrated as a case study. The evaluation of the case study demonstrates the effectiveness of the proposed framework.

1 Introduction

Increasing complexities of embedded system and strict schedules in time-to-market are critical issues in the today's system-level design. Currently, various embedded systems incorporate multimedia applications, which are required more and more complex functionalities. Meanwhile, the semiconductor technology progress has placed a great amount of hardware resources on one chip, enabling to implement more functionalities as hardware in order to realize efficient systems. This widens design space to be explored and makes system-level designs further complicated - to improve the design productivity, designing systems at a higher abstraction level is necessary [1].

Hardware/software (HW/SW) codesign of these systems mainly relies on the following challenging issues: (1) data parallelism to improve performance, (2)

support for software developers to implement such complicated systems without knowing system-level languages such as SystemC and SpecC, (3) implementation to directly use existing code without modification, and (4) management of communication between functionalities. To the best of our knowledge, there is no work addressing all of the above issues.

This paper presents a system-level framework (TECSCE) to cope with the preceding issues. This framework aims at enabling even software developers to easily design complicated systems such as multimedia applications which are rich in data parallelism. For this, we integrate a component technology for embedded software, TECS (TOPPERS Embedded Component System [2]), and the system-on-a-chip environment SCE [3], which is based on SpecC language. Since TECS is based on conventional C language, it allows the developers to start the design process easily and fast. SCE is a rapid design exploration tool capable of efficient MPSoC implementation.

The contribution of this work is to present a system-level design method for software developers to deal with massively parallel embedded systems using TECS. In existing HW/SW codesign technologies, a designer needs to manually add or modify HW/SW communication sources (e.g., their size, direction, and allocator) in input behavioral descriptions, which is complex to specify and error-prone. In contrast, in the proposed framework, the developer can design the overall system at a software component level and has no need to specify the HW/SW communication in the input description because TECS defines the interface between components, and the communication sources are automatically generated. Moreover, a new mechanism of duplicating components realizes data partitioning at the software component level for an effective speedup of the applications.

The rest of this paper is organized as follows. Section 2 explains TECS, SCE, and the overview of the proposed framework. Section 3 depicts a case study of adapting the proposed framework. The evaluation of the case study is shown in Section 4. Related work is described in Section 5. Finally, Section 6 concludes this paper.

2 TECSCE

In this section, the overviews of TECS, SCE, and a system-level design framework (TECSCE) integrating TECS and SCE are presented.

2.1 TECS

In embedded software domains, software component technologies have become popular to improve the productivity [2, 4, 5]. It has many advantages such as increasing reusability, reducing time-to-market, reducing software production cost, and hence, improving productivity [6].

TECS adopts a static model that statically instantiates and connects components. The attributes of the components and interface sources for connecting the components are statically generated by the interface generator. Furthermore,

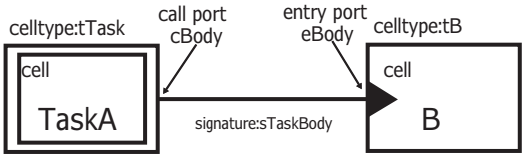

Fig. 1. Component diagram.

TECS optimizes the interface sources. Hence, no instantiation overhead is introduced at runtime, and the runtime overhead of the interface code is minimized [7]. Therefore, these attributes of TECS are suitable for system-level designs.

Furthermore, in system-level designs, parallelism and pipeline processing should be considered. TECS supports parallelism and pipeline processing on a real-time OS for multi-processors in embedded software [8]. The *oneway* calling is provided to support the parallelism. It means that a caller component does not need to wait until a callee component finishes executing. At a software level for multiprocessors environment, the parallelism has been already supported in TECS. Therefore, it is possible to adapt the feature for system-level designs.

Component Model in TECS A *cell* is an instance of component in TECS. *Cell*s are properly connected in order to develop an appropriate application. A *cell* has *entry port* and *call port* interfaces. The *entry port* is an interface to provide services (functions) to other *cell*s. Each service of the *entry port* called the *entry function* is implemented in C language. The *call port* is an interface to use the services of other *cell*s. A *cell* communicates in this environment through these interfaces. To distinguish *call port*s of caller *cell*s, an *entry port array* is used. A subscript is utilized to identify the *entry port array*. A developer decides the size of an *entry port array*. The *entry port* and the *call port* have *signatures* (sets of services). A *signature* is the definition of interfaces in a *cell*. A *celltype* is the definition of a *cell*, as well as the *Class* of an object-oriented language. A *cell* is an entity of a *celltype*.

Figure 1 shows an example of a component diagram. Each rectangle represents a *cell*. The dual rectangle depicts a active *cell* that is the entry point of a program such as a task and an interrupt handler. The left *cell* is a TaskA *cell*, and the right *cell* is a B *cell*. Here, each of tTask and tB represents the *celltype* name. The triangle in the B *cell* depicts an *entry port*. The connection of the *entry port* in the *cell*s describes a *call port*.

Component Description in TECS The description of a component in TECS can be classified into three descriptions: a *signature* description, a *celltype* description, and a *build* description. An example for component descriptions is presented in Section 3 to briefly explain these three descriptions [5] .

2.2 SCE

SCE implements a top-down system design flow based on a specify-explore-refine paradigm with support for heterogeneous target platforms consisting of custom hardware components, embedded software processors, dedicated IP blocks, and

[5] Please refer [2] for the more detailed explanations.

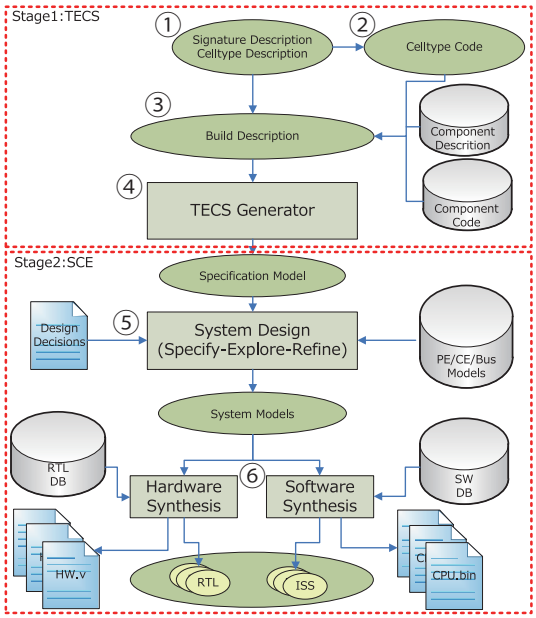

Fig. 2. Design flow using the proposed framework.

complex communication bus architectures. The rest of features and design flow is explained in the next subsection.

2.3 Overview of TECSCE

Figure 2 represents the design flow using the proposed framework. The circled numbers in Figure 2 represent the order of design steps.

- Step1: A framework user (hereafter, a developer) defines *signature*s (interface definitions) and *celltype* (component definitions).
- Step2: The developer implements *celltype* source (component source code) in C language. They can use the template code based on *signature*s and *celltype* descriptions.
- Step3: The developer describes an application structure including definitions of *cell*s (instances of component) and the connection between *cell*s. In this step, the developer decides the degree of data partitioning. If it is possible to use existing source code (i.e., legacy code), the developer can start from Step3.
- Step4: The SpecC specification model based on the component description, including definitions of *behavior*s and *channel*s, is generated by a TECS generator. The specification model is a functional and abstract model that is free of any implementation details.
- Step5: The designer can automatically generate system models (Transaction-level models) based on design decisions (i.e. mapping the behaviors of the specification model onto the allocated PEs).
- Step6: The hardware and software parts in the system model are implemented by hardware and software synthesis phases, respectively.

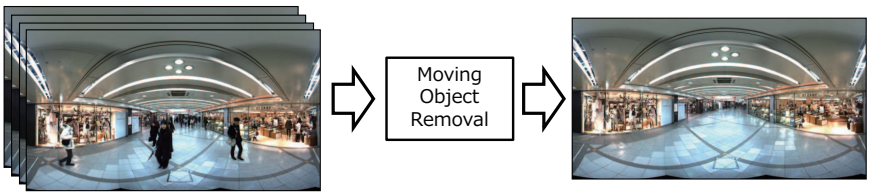

Fig. 3. Target application. Left images are input images. Right image is a result image.

SCE supports generating a new model by integrating the design decisions into the previous model.

3 Case study for proposed framework

In this section, the proposed framework is explained through a case study. First, a target application is described. Then, two kinds of mechanism to generate specification models (Step4 in Figure 2) are depicted.

3.1 Target Application

The target application named MovingObjectRemoral for a case study of the framework is an application for generating a panoramic image removing objects, such as people. In the panoramic image view system, such as Google Street View, a user can see images from the street using omnidirectional images. Figure 3 illustrates the target application. The application creates the image without people as shown in the right image of Figure 3 based on the algorithm [9] by using a set of panoramic images which are taken at the same position.

Since creating an image by removing obstacles needs a number of original images, each of which has too many pixels, the original program is designed only for off-line use. Because the output image depends on the place and environment, we do not know how many source images are needed to create the output image. Therefore, currently, we need enormously long time to take images at each place. Our final goal is to create the output image *in real-time* by using our framework.

3.2 TECS components for the target application

Figure 4 shows a TECS component diagram for the target application. Each rectangle represents a *cell* which is a component in TECS. The left, middle, and right *cell*s are a Reader *cell*, an MOR (MovingObjectRemoral) *cell*, and a Writer *cell*, respectively. The Reader *cell* reads image files, slices the image, and sends the sliced image data to the MOR *cell*s. The MOR *cell* collects background colors (RGB) of each pixel based on the input images. The Writer *cell* creates the final image based on the data collected by the MOR *cell*. Here, tReader, tMOR, and tWriter represent the *celltype* name.

Figure 5 shows a *signature* description between tReader and tMOR, and between tMOR and tWriter. The *signature* description is used to define a set of function heads. A *signature* name, such as sSliceImage, follows a *signature*

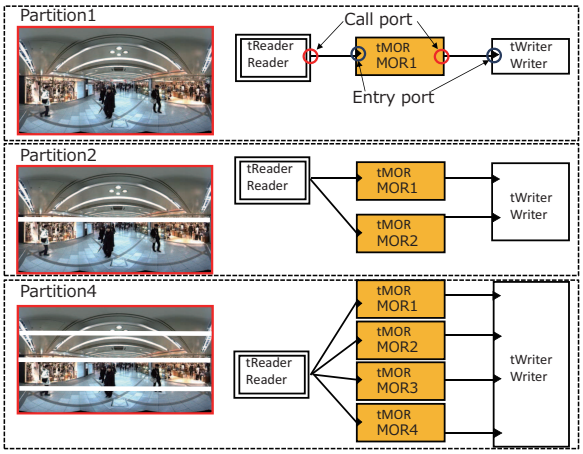

Fig. 4. Component diagram for target application.

```
1  signature sSliceImage {
2    [ oneway ] void  sendBlock([ in ]const slice *slice_image);
3 };
```

Fig. 5. Signature description for the target application.

```
 1 [ singleton,  active]
 2  celltype tReader {
 3    call sSliceImage cSliceImage[];
 4 };
 5  celltype tWriter {
 6    entry sSliceImage eSliceImage[];
 7 };
 8  celltype tMOR{
 9    entry sSliceImage eSliceImage;
10    call sSliceImage cSliceImage;
11    attr{
12      float32_t rate = 0.75;
13    };
14    var{
15      int32_t count = 0;
16      slice out_slice_image;
17      slice slice_images[MAX_COUNT];
18    };
19 };
```

Fig. 6. Celltype description for the target application.

keyword to define the *signature*. The initial character ("s") of the *signature name* sSliceImage represents the *signature*. A set of function heads is enumerated in the body of this keyword. TECS provides the *in*, *out*, and *inout* keywords to distinguish whether a parameter is an input and/or an output. The *in* keyword is used to transfer data from a caller *cell* to a callee *cell*. The *oneway* keyword means that a caller *cell* does not need to wait for finishing a callee *cell*. Namely, the *oneway* keyword is useful when a caller *cell* and a callee *cell* are executed in parallel.

Figure 6 describes a *celltype* description. The *celltype description* is used to define the *entry port*s, *call port*s, *attribute*s, and *variable*s of each *celltype*. The *singleton* keyword (Line 1 in Figure 6) represents that a singleton *celltype* is a

```
 1 const int32_t SliceCount = 2;
 2 [ generate(RepeatJoinPlugin," count=SliceCount")]
 3  cell tReader Reader {
 4    cSliceImage[0] = MOR_000.eSliceImage;
 5 };
 6 [ generate(RepeatCellPlugin," count=SliceCount")]
 7  cell tMOR MOR_000 {
 8    cSliceImage = Writer.eSliceImage[0];
 9 };
10  cell tWriter Writer{
11 };
```

Fig. 7. Build description for the target application.

particular *cell*, only one of which exists in a system to reduce the overhead. The *active* keyword (Line 1 in Figure 6) represents the entry point of a program such as a task and an interrupt handler. A *celltype* name, such as tReader, follows a *celltype* keyword to define *celltype*. The initial character ("t") of the *celltype* name tReader represents the *celltype*. To declare an *entry port*, an *entry* keyword is used (Line 6 and 9 in Figure 6). Two words follow the *entry* keyword: a *signature* name, such as sSliceImage, and an *entry port* name, such as eSliceImage. The initial character ("e") of the *entry port* name eSliceImage represents an *entry port*. Likewise, to declare a *call port*, a *call* keyword is used (Line 3 and 10 in Figure 6). The initial character ("c") of the *call port* name cSliceImage represents a *call port*.

The *attr* and *var* keywords that are used to increase the number of different *cell*s are attached to the *celltype* and are initialized when each *cell* is created. The set of attributes or variables is enumerated in the body of these keywords. These keywords can be omitted when a *celltype* does not have an attribute and/or a variable.

Figure 7 shows a *build* description. The *build* description is used to declare *cell*s and to connect between *cell*s for constructing an application. To declare a *cell*, the *cell* keyword is used. Two words follow the *cell* keyword: a *celltype* name, such as tReader, and a *cell* name, such as Reader (Lines 3-5, Lines 7-9, and Lines 10-11 in Figure 7). In this case, eSliceImage (*entry port* name) of MOR_000 (*cell* name) is connected to cSliceImage (*call port* name) of Reader (*cell* name). The *signature*s of the *call port* and the *entry port* must be the same in order to connect the *cell*s.

3.3 *cellPlugin*

At the component level (Step 3 in Figure 2), the proposed framework realizes data partitioning. A new plugin named *cellPlugin* is proposed to duplicate *cell*s for data partitioning and connect the *cell*s. There are two types of *cellPlugin*: RepeatCellPlugin and RepeatJoinPlugin.

RepeatCellPlugin supports duplication of *cell*s depending on the *slice count* i.e. the number of data partitions. and connection between the *call port* of the duplicated *cell*s and the *entry port*s of the connected *cell* in the original *build*

description (Line 6 in Figure 7). RepeatJoinPlugin provides connection between the *call port* of the duplicated *cell*s generated by RepeatCellPlugin (Line 2 in Figure 7). Note that it is easy to duplicate MOR *cell*s for realizing data partitioning and parallelization as shown in Figure 4.

3.4 *cd2specc*

In this subsection, policies of transformation from a component description to a specification model in SpecC language are described. A basic policy of transformation is that a *cell* and an argument of function of *signature* correspond to a *behavior* and a *channel* in SpecC language, respectively. The tReader, tMOR, and tWriter *celltype*s correspond to tReader, tMOR, tWriter behaviors generated by *cd2specc*, respectively. The following pseudo code describes the examples of generated SpecC code.

Pseudo Code 1 tMOR behavior

```
 1 behavior tMOR(channel definitions){
 2  void eSliceImage_sendBlock
                        (slice_image){
 3    for i to HIGHT / SliceCount do
 4     for j to WIDTH do
 5      store pixel color
 6      sort
 7     end for
 8    end for
 9    send new image to Writer
10  }
11  void main(){
12    while true do
13      receive slice_image data
14      call eSliceImage_sendBlock
15    end while
16  }
17 }
```

Pseudo Code 1 shows a tMOR behavior. If a behavior has an *entry function*, the behavior receives parameters to call the *entry function*. In this case, tMOR behavior receives sliced images by using channels to call *entry function* (eSliceImage_sendBlock) in Pseudo Code 1. Although there are several ways to realize tMOR, here we show in the pseudo code an algorithm to do so easily. This is often used for sorting algorithm based on brightness of each pixel to find the background color for each pixel. In this case, the brighter color depending on the rate value (Line 12 in Figure 6) is selected.

A SpecC program starts with execution of the main function of the root behavior which is named Main as shown in Pseudo Code 2. The roles of the main behavior are instantiation of behaviors, initialization of channels, connection of channels between behaviors, and management of execution of the other behaviors.

All behavioral synthesis tools typically do not support all possible C language constructs, such as recursion, dynamic memory allocation. Thus, TECS component source obeying these restrictions can be synthesized. Since recursion and dynamic memory allocation are not usually used for embedded software, these restrictions are not critical.

Figure 8 shows a specification model of SpecC language when *slice count* is two. The model consists of four behaviors and four communication channels.

Pseudo Code 2 Main behavior

```
1  behavior Main(channel definitions){
2    // declaration of channels
3    // declaration of behaviors
4    tReader Reader(...);
5    tMOR    MOR_000(...);
6    tMOR    MOR_001(...);
7    //...
8    Writer  Writer(...);
```

```
9   int main(){
10    par{
11        Reader.main();
12        MOR_000.main();
13        MOR_001.main();
14        //...
15        Writer.main();
16    }
17  }
18 }
```

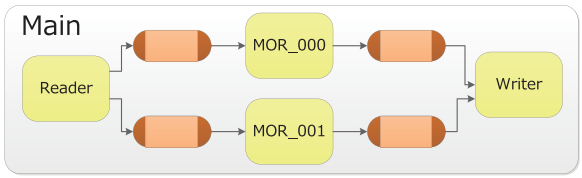

Fig. 8. Specification model of SpecC language when *slice count* is two.

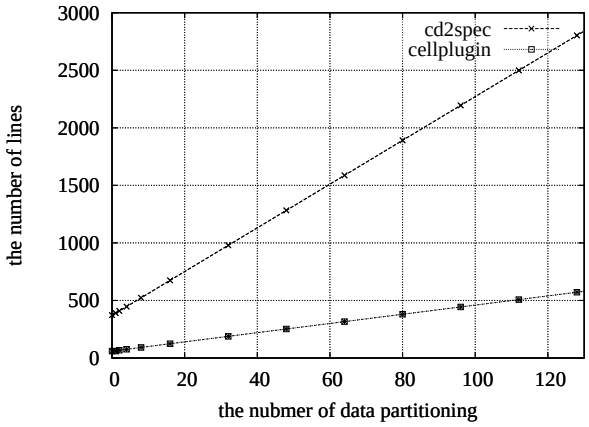

Fig. 9. Size of generated code in SpecC language.

The numbers of channels and MOR instances are depended on the number of *slice count*.

4 Evaluation

For the experimental evaluation of the TECSCE design flow, we used the application described in Section 3 to show effectiveness of *cellPlugin* and *cd2specc* for improving design productivity.

First, we measured the number of lines of each component description generated by *cellPlugin* and each SpecC code generated by *cd2specc*. The values in Figure 9 represent the total number of lines of generated code. When the number of data partitioning is zero, the value shows the lines of common code, e.g., def-

Table 1. Results of Execution Time (ms) (*slice count* is Eight)

Algorithm	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7	CPU8
Bubble	16465.0	16343.0	16215.2	16162.5	16276.9	16325.0	16372.7	16396.3
Insert	2261.3	2360.5	2423.9	2425.4	2423.2	2384.4	2345.2	2317.8
Average	942.1	973.4	997.9	997.7	997.6	997.8	997.8	997.5
Bucket	944.9	973.2	987.7	980.8	998.8	999.3	999.4	999.2

initions of channel types, template code of behaviors, and implementation code based on *entry function*s. As can be seen from Figure 9, the lines of the code proportionally grow to *slice count*. In TECSCE, the developers only change the parameter for *slice count* in order to manage the data partitioning. The results indicate that the communication code between behaviors have a significant impact on productivity. Therefore, it can be concluded that *cellPlugin* and *cd2specc* are useful, particularly for large *slice count*.

Next, we evaluated four algorithms to realize the MOR: *Bubble*, *Insert*, *Average*, and *Bucket*. *Bubble* is a basic algorithm for MOR based on a bubble sort to decide the background color. *Insert* is based on an insertion sort. *Average* is assumed that the background color is the average color value. *Bucket* is based on a bucket sort.

Each MOR behavior was mapped onto different cores based on ARM7TDMI (100MHz). The execution time of processing 50 images with 128x128 pixels on every core is measured when *slice count* was eight. An ISS (Instruction Set Simulator) supported by SCE was used to measure the cycle counts for estimation of the execution time. Table 1 shows the results of execution time for each core when *slice count* is eight. These results indicate that the generated SpecC descriptions are accurately simulatable.

All of the series of images are not necessary to collect the background color for the target application because the series of images are almost the same. Therefore, if a few input images can be obtained per second, it is enough to generate the output image. In our experiments, two images per second were enough to generate an output one. It is possible to use this application in real-time when each input image with 256x512 is used on this configuration (eight cores, ARM 100MHz, and *Bucket* algorithm). If the developers want to deal with bigger images in real-time, there are several options: to use higher clock frequency, to increase the number of data partitioning, to use hardware IPs, and so forth.

5 Related Work

HW/SW codesign frameworks have been studied for more than a decade.

Daedalus [10] framework supports a codesign for multimedia systems. It starts from a sequential program in C, and converts the sequential program into a parallel KPN (Kahn Process Network) specification through a KPNgen tool.

SystemBuilder [11] is a codsign tool which automatically synthesizes target implementation of a system from a functional description. It starts with system specification in C language, in which a designer manually specifies the system

functionalities as a set of concurrent processes communicating with each other through channels.

SystemCoDesigner [12] supports a fast design space exploration and rapid prototyping of behavioral SystemC models by using an actor-oriented approach.

The system-on-chip environment (SCE) [3] is based on the influential SpecC language and methodology. SCE implements a top-down system design flow based on a specify-explore-refine paradigm with support for heterogeneous target platforms consisting of custom hardware components, embedded software processors, dedicated IP blocks, and complex communication bus architectures.

System-level designs based UML [13, 14] are proposed to improve the productivity. One [13] is for exploring partial and dynamic reconfiguration of modern FPGAs. The other [14] is for closing the gap between UML-based modeling and SystemC-based simulation.

To the best of our knowledge, there is no work addressing all of the issues mentioned in Section 1. TECSCE solve all of the issues since *cd2specc*, which a part of TECSCE, makes the overall system at a software component level in order to hide the many implementation details such as communication between functionalities. The framework users do not need to specify the HW/SW communication in the input description because the communication sources are automatically generated from component descriptions TECS specifically defines the interface between components. Therefore, TECSCE realizes that existing code can be used without modification and without knowing system-level languages such as SystemC and SpecC. Moreover, *cellPlugin*, which is a part of TECSCE, supports that duplication of components realizes data partitioning at a component level for an effective speedup of the applications.

6 Conclusions

This paper proposed a new codesign framework integrating TECS and SCE, which enables software developers to deal with massive parallel computing for multimedia embedded systems. The advantage of our framework is that developers can directly exploit software components for system-level design without modifying input C sources (component sources). Moreover, since TECS supports data partitioning and SCE supports MPSoCs as target architectures, our framework can deal with more complex applications (such as MOR) and can help parallelize them for efficient implementation. The evaluation demonstrated the effectiveness of the proposed framework including *cellPlugin* and *cd2specc* and the capability of operating the MOR application in real-time. Furthermore, almost all multimedia applications can be adapted to the same model of our framework. *cellPlugin* and *cd2specc* are open-source software, and will be avaibale to download from the website at [15].

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant Number 40582036. We would like to thank Maiya Hori, Ismail Arai, and Nobuhiko Nishio for providing the MOR application.

References

1. Sangiovanni-Vincentelli, A.: Quo vadis, SLD? Reasoning about the Trends and Challenges of System Level Design. IEEE **95**(3) (2007) 467–506
2. Azumi, T., Yamamoto, M., Kominami, Y., Takagi, N., Oyama, H., Takada, H.: A new specification of software components for embedded systems. In: Proc. 10th IEEE International Symposium on Object/component/service-Oriented Real-Time Distributed Computing. (May 2007) 46–50
3. Dömer, R., Gerstlauer, A., Peng, J., Shin, D., Cai, L., Yu, H., Abdi, S., Gajski, D.D.: System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design. EURASIP Journal on Embedded Systems **2008** (Jan. 2008) 1–13
4. AUTOSAR: AUTOSAR Specification. http://www.autosar.org/
5. Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., Tivoli, M.: The SAVE approach to component-based development of vehicular systems. Journal of Systems and Software **80**(5) (May 2007) 655–667
6. Lau, K.K., Wang, Z.: Software component models. IEEE Transactions on Software Engineering **33**(10) (Oct. 2007) 709–724
7. Azumi, T., Oyama, H., Takada, H.: Optimization of component connections for an embedded component system. In: Proc. IEEE/IFIP 7th International Conference on Embedded and Uniquitous Computing. (Aug. 2009) 182–188
8. Azumi, T., Oyama, H., Takada, H.: Memory allocator for efficient task communications by using RPC channels in an embedded component system. In: Proc. the 12th IASTED International Conference on Software Engineering and Applications. (Nov. 2008) 204–209
9. Hori, M., Takahashi, H., Kanbara, M., Yokoya, N.: Removal of moving objects and inconsistencies in color tone for an omnidirectional image database. In: Proc. of ACCV2010 Workshop on Application of Computer Vision for Mixed and Augmented Reality. (Nov. 2011) 62–71
10. Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A.D., Polstra, S., Bose, R., Zissulescu, C., Deprettere, E.F.: Daedalus: Toward composable multimedia mp-soc design. In: Proc. International 45th Design Automation Conference. (Jul. 2008) 574–579
11. Honda, S., Tomiyama, H., Takada, H.: RTOS and codesign toolkit for multiprocessor systems-on-chip. In: Proc.In 12th Asia and South Pacific Design Automation Conference. (Jan. 2007) 336–341
12. Keinert, J., Streübrbar, M., Schlichter, T., Falk, J., Gladigau, J., Haubelt, C., Teich, J., Meredith, M.: Systemcodesigner an automatic esl synthesis approach by design space exploration and behavioral synthesis for streaming applications. ACM Trans. Des. Autom. Electron. Syst. **14**(1) (Jan. 2009) 1:1–1:23
13. Vidal, J., de Lamotte, F., Gogniat, G., Diguet, J.P., Soulard, P.: UML design for dynamically reconfigurable multiprocessor embedded systems. In: Proceedings of the Conference on Design, Automation and Test in Europe. (Mar. 2010) 1195–1200
14. Mischkalla, F., He, D., Mueller, W.: Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems. In: Proceedings of the Conference on Design, Automation and Test in Europe. (Mar. 2010) 1201–1206
15. TECS: `http://www.toppers.jp/tecs`.