2009 International Conference on Computational Science and Engineering

Optimization of Component Connections for an Embedded Component System

Takuya Azumi, Hiroaki Takada
Graduate School of Information Science
Nagoya University
Nagoya Japan
Email:{takuya,hiro}@ertl.jp

Hiroshi Oyama
FA Systems Division
OKUMA Corporation
Niwa-gun, Japan
Email: hi-ooyama@okuma.co.jp

***Abstract*—Software component techniques are widely used to enhance productivity and reduce the cost of software systems development. This paper proposes optimization of component connections for a component system that is suitable for embedded systems. This component system adopts a static model that statically instantiates and connects components. The attributes of the components and the interface code for connecting the components are statically generated by the generator. No instantiation overhead is introduced at runtime, and the runtime overhead of the interface code is minimized. A case study using a serial interface driver is presented to evaluate the execution time overhead, the software code size, and the executable file size. The case study shows the effectiveness of optimization.**

***Keywords*-component-based development; real-time operating system;**

I. Introduction

Recently, three significant problems have appeared in software development for embedded systems. Firstly, the size and complexity of the software for embedded systems is increasing rapidly. Secondly, the required diversity of products and their software is increasing rapidly. Finally, the development time must be significantly decreased.

During the last decade, software component technology for versatile systems has been used to improve productivity, especially in the development of software for desktop applications and distributed information systems. Popular component systems include JavaBeans [1] and ActiveX [2] for desktop applications, and CORBA Component Model (CCM) [3] and COM+ [4] for distributed information systems. The productivity can be increased not only by reducing the coding time, but also by reducing the testing time. However, it is difficult to directly use these component technologies for embedded systems [5] because of the dynamic configuration of components. Recently, researchers have focused on software component technologies for embedded systems [6]. Component technologies for embedded systems, such as PBO [7], Koala component [8], PECT [9], and SaveCCT [10], have been developed. However, such component technologies are not widely used in the domain of embedded systems [11].

TOPPERS [1] [12] Embedded Component System (TECS) [13], [14] has been investigated for several years. It is possible to estimate the memory consumption of an entire application, since TECS uses a static configuration. The static configuration implies that both the configuration of the component behavior and the interconnections between components are static. There are several benefits of using the static configuration. Furthermore, TECS can be used in different embedded systems domains, since a variety of component granularities are supported, and a diversity of components, such as an allocator or an RPC channel, are provided. A small component, such as a component's device driver, is used to distinguish the dependent and the independent hardware parts.

TECS adopts a static model that statically instantiates and connects components. The attributes of the components and interface code for connecting the components is statically generated by the generator. Furthermore, optimization of the interface code is proposed. Hence, no instantiation overhead is introduced at runtime, and the runtime overhead of the interface code is minimized.

The details of TECS are described in Section II. Section III presents a method for optimizing component connections, while Section IV summarizes the results of the current research.

II. TOPPERS Embedded Component System

In this section, the specifications and features of TECS are described in detail.

A. Component Model

A *cell* is an instantiation of the component in TECS. *Cells* 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 *cells*. The service of the *entry port* is called the *entry function*. The *call port* is an interface that uses the services of other cells. In this environment, a *cell* communicates through

[1]TOPPERS (Toyohashi OPen Platform for Embedded Real-time Systems) Project, which is based on the technical development results obtained by applying ITRON, is aimed to develop base software for use in embedded systems.

978-0-7695-3823-5/09 $26.00 © 2009 IEEE
DOI 10.1109/CSE.2009.97

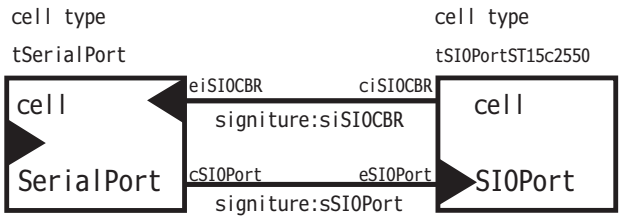

Figure 1. Component Diagram.

these interfaces. The *entry port* and the *call port* have *signatures* (sets of services). A *signature* is the definition of interfaces in a *cell*. Interface abstraction using a *signature* provides control of the dependencies of each cell. The *cell type* is the definition of a *cell*, which is similar to the *Class* of an object-oriented language. A *cell* is an entity of a *cell type*.

Figure 1 shows an example of a serial port. Each rectangle represents a *cell*. The left *cell* is a SerialPort *cell* which is a target-independent part of the serial port, and the right *cell* is an SlOPort *cell* which is a target-dependent part of the serial port. Here, tSerialPort and tSIOPortST15c2550 represent the *cell type* name. Each triangle in the SerialPort or SIOPort *cell* depicts an *entry port*. The connection of the *entry port* in the *cell*s describes a *call port*.

A *call port* can only connect to an *entry port*. Therefore, in order to connect several *entry port*s, a *call port array* is used. An *entry port* can connect to other *call port*s. However, in this case, it is impossible to identify which *call port*s are connected. *Attribute* and *variable* keywords are used to increase the number of different types of *cell*s. For example, a serial communication *cell* has an *attribute* to define the send or receive buffer sizes. Although each *attribute*, such as the buffer size, can not be changed during execution time, each *variable*, such as the file name of a file cell, can be changed. Thus, it is assumed that the *attributes* are automatically mapped to ROM, and that the *variables* are automatically mapped to RAM. A *singleton cell* is a particular type of *cell*, of which there is only one in the system. The *singleton cell* is used to reduce the overhead because the *cell* can be optimized.

B. Development Flow

Figure 2 shows the development flow in TECS. In Section II-C, the *signature*, *cell type*, and *build* description are explained in detail. The *signature* description is used to define a set of function heads of a *cell type*. The *cell type* description is used to define the *entry port*s, *call ports*, *attributes*, and *variables* of a *cell type*. The *build* description is used to declare the *cell*s and create the connections between *cell*s in order to assemble an application. An *interface generator* generates several interface *C*- (.h or .c) and template codes for the comment source from the *signature*, *cell*

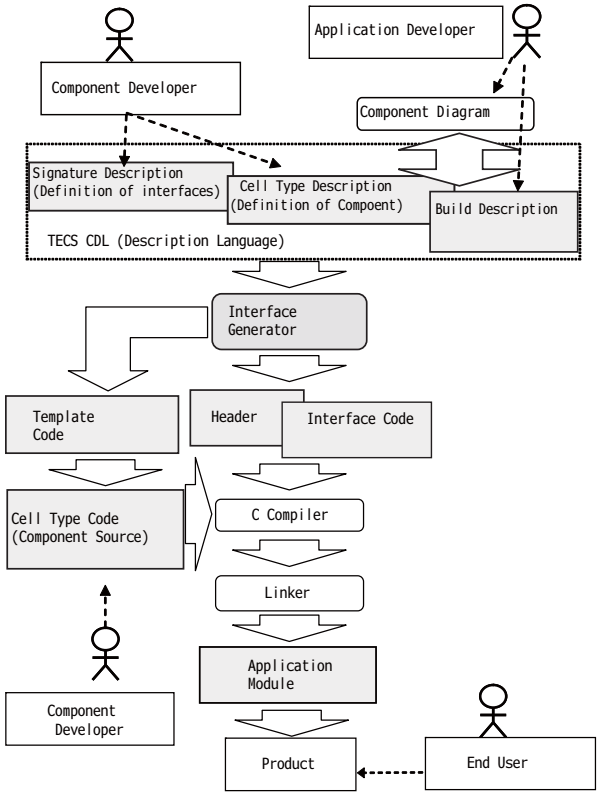

Figure 2. Development Flow.

type, and *build* descriptions. Developers in this framework are divided into two parts: a component developer and an application developer. The role of the component developer is to define the *signature*s and *cell type*s, as well as write the implementation code (Component Source) of the *cell*s. Generally, a component is provided by the source code. On the other hand, the role of the application developer is to develop an appropriate application by connecting the *cell*s.

C. Component Description

The description of a component in TECS can be divided into three parts: a *signature* description, a *cell type* description, and a *build* description. The *signature* and the *cell type* descriptions are described by component developers. The *build* description is written by the application developers.

1) Signature Description: The *signature* description is used to define a set of function heads. Figure 3 represents the *signature* descriptions to define the serial port interface. A *signature* name, such as sSerialPort, follows a *signature* keyword to define the *signature*. The initial part of the *signature* name ("s") represents the *signature*. A set of function heads is enumerated in the body of this keyword.

A detailed explanation of the interface description is given below.

```
 1 //to use dependent part
 2 signature sSIOPort {
 3   void   open(void);
 4   void   close(void);
 5   bool_t putChar([in] char_t c);
 6   int_t  getChar(void);
 7   void   enableCBR([in] uint_t cbrtn);
 8   void   disableCBR([in] uint_t cbrtn);
 9 };
10 //call back
11 signature siSIOCBR {
12   void   readySend(void);
13   void   readyReceive(void);
14 };
15 //to use serial port for application
16 signature sSerialPort{
17   ER      open(void);
18   ER      close(void);
19   ER_UINT read([out,size_is(len)]
            char_t *buf, [in] uint_t len);
20   ER_UINT write([in,size_is(len)]
            char_t *buf, [in] uint_t len);
21 }
```

Figure 3. Signature Description of Serial Port.

- Input or Output: The *in*, *out*, and *inout* keywords are used to distinguish whether a parameter is an input or/and an output. These keywords are understandable when a parameter is a pointer. It is important to use these keywords with respect to memory allocation in a distributive framework.
- Pointer: A pointer indicates an array or a value in TECS. In this case, the *buf* parameter represents an array (Lines 19-20 in Figure 3).
- Array Size: It is necessary to describe the size of an array by using the *size_is* keyword in TECS (Lines 19-20 in Figure 3).

2) Cell Type Description: The *cell type* description is used to define the *entry port*s, *call port*s, *attribute*s, and *variable*s of a *cell type*. Figure 4 and 5 describe the *cell type* description used to define components for the serial port. A *cell type* can have *entry port*s, *call port*s, *attribute*s, and *variable*s. A *cell type* name, such as tSerialPort, follows a *celltype* keyword to define *cell type*. The initial part of a *cell type* name ("t") represents the *cell type*. To declare an *entry port*, an *entry* keyword is used. Two words follow the *entry* keyword: a *signature* name, such as sSIOPort, and an *entry port* name, such as eSIOPort. The initial part of the *entry port* name ("e") represents an *entry port*. Likewise, to declare a *call port*, a *call* keyword is used. The initial part of the *call port* name ("c") represents a *call port*.

The *attr* and *var* keywords that are used to increase the number of different *cell*s are attached to the *cell type* and are initialized when each *cell* is created. For example, a serial port *cell* has an attribute to define the buffer size. The set of *attribute*s or *variable*s is enumerated in the body of these

```
 1 //cell type definition of
     independent part
 2 celltype tSerialPort{
 3   //entry port for application
 4   entry sSerialPort eSerialPort;
 5   //entry port for dependent part
 6   entry siSIOCBR    eiSIOCBR;
 7   //call port to use dependent part
 8   call sSIOPort     cSIOPort;
 9
10   attr{
11    //receive buffer size
12    uint_t    rcv_bufsz = 128;
13    //send buffer size
14    uint_t    snd_bufsz = 128;
15   };
16   var{
17    //receive buffer
18    [size_is(rcv_bufsz)]
              char_t *rcv_buffer;
19    //send buffer
20    [size_is(snd_bufsz)]
              char_t *snd_buffer;
21   };
22 };
```

Figure 4. Cell Type Description of Independent Part.

```
 1 //cell type definition of dependent part
 2 celltype tSIOPortST16c{
 3   //entry port to provide
 4   entry sSIOPort eSIOPort;
 5   //call port for call back
 6   call siSIOCBR       ciSIOCBR;
 7   attr{
 8       /* definition of attributes */
 9   };
10   var{
11       /* definition of variables */
12   };
13 };
```

Figure 5. Cell Type Description of Dependent Part.

keywords. These keywords can be omitted when a *cell type* does not have an *attribute* and/or a *variable*.

3) Build Description: The build description is used to declare *cell*s and to connect between *cell*s for creating an application. Figure 6 shows a build description to instantiate and connect the *cell*s. To declare a *cell*, the *cell* keyword is used. Two words follow the *cell* keyword: a *cell type* name, such as tSerialPort, and an *cell* name, such as SerialPort (Lines 2-7 in Figure 6). In this case, eSIOPort (*entry port* name) of SIOPort (*cell* name) connected to cSIOlPort (*call port* name) of SerialPort (*cell* name). The *signatures* of the *call port* and the *entry port* must be the same in order to connect the *cell*s.

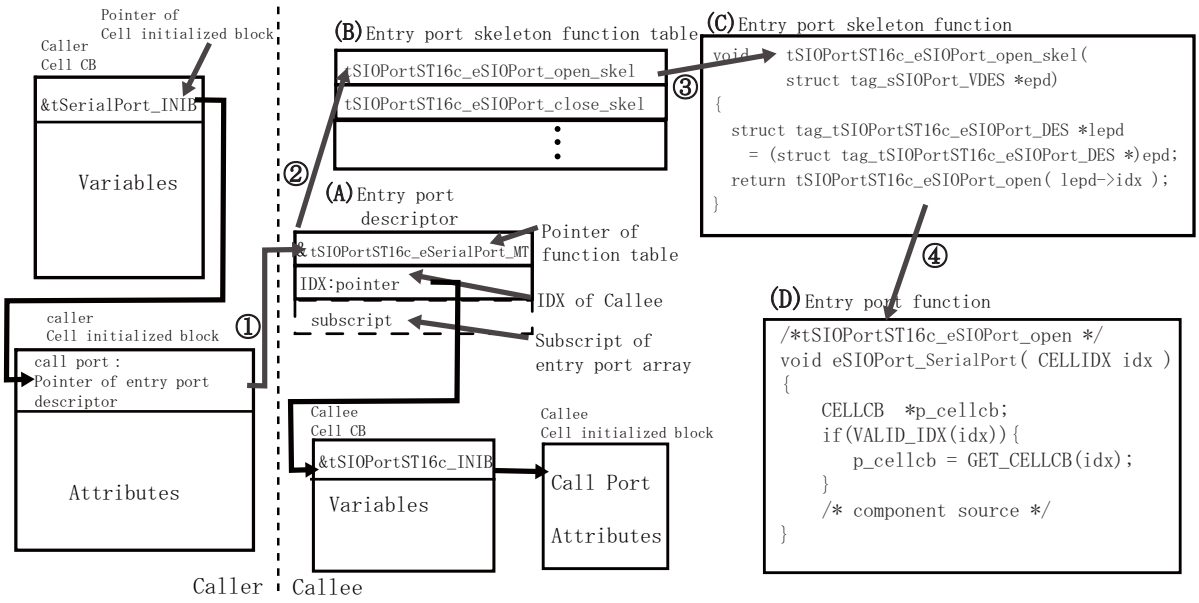

Figure 7. Component Connection Structure.

```
 1 //instantiation of independent part
 2 cell tSerialPort SerialPort {
 3   //connect the entry port of SIOPort
 4   cSIOPort = SIOPort.eSIOPort;
 5   rcv_bufsz = 256;
 6   snd_bufsz = 256;
 7 };
 8 //instantiation of dependent part
 9 cell tSIOPortST16c SIOPort {
10  //connect the entry port of SerialPort
11   ciSIOCBR = SerialPort.eiSIOCBR;
12 };
```

Figure 6. Build Description.

III. Component Connection Optimization

A. Component Connection Structure

Figure 7 shows the component connection structure. In this case, the structure represents the SerialPort *cell* (left-hand side) that calls the open function of the SIOPort *cell* (right-hand side). The numbers in Figure 7 describe the order of the calling procedure.

The developers only implement the *entry port function*. The other parts, *entry port descriptor*, *entry port skeleton* function table, and *entry port skeleton*, are generated by the interface generator.

The words in Figrue 7 are explained below.

- *Cell* initialized block
  A *cell* initialized block (*cell* INIB) is a data structure that has the *call port*s and the *attribute*s of each *cell*. The *call port* stores the pointer of *entry port descriptor*s. The data of the *Cell* initialize block are mapped to ROM.
  Figure 8 represents the structure of a *cell* INIB for tSerialPort. The structure is defined based on the *cell type* description for each *cell type*. The structure has *attribute*s (Lines 3-4). In case of a *call port*, the structure stores the pointer that connects with the pointer of the *entry port descriptor*s (Line 5).
- *Cell* control block
  A *cell* Control Block (*cell* CB) is a data structure that stores the pointer of the *cell* INIB and the *variable*s of each *cell*. The data of the *cell* CB are mapped to RAM. The arrays shown in Figure10 are allocated by the interface generator. Figure 11 represents the initialization of *cell* CB for SerialPort. The structure

```
1 typedef const struct tag_tSerialPort_INIB{
2     /* attributes */
3     uint_t          rcv_bufsz;
4     uint_t          snd_bufsz;
5     /* call port */
6 }tSerialPort_INIB;
```

Figure 8. Structure of *cell* INIB for tSerialPort.

```
1 typedef struct tag_tSerialPort_CB{
2     /* variables */
3     char_t*         rcv_buffer;
4     char_t*         snd_buffer;
5     /* pointer to cell INIB */
6     tSerialPort_INIB  *_inib;
7 }tSerialPort_CB;
```

Figure 9. Structure of *cell* CB for tSerialPort.

```
1 /* statically allocated buffers  */
2 char_t SerialPort_CB_rcv_buffer_INIT[256];
3 char_t SerialPort_CB_snd_buffer_INIT[256];
```

Figure 10. Static Allocation of Variable.

```
 1 struct tag_tSerialPort_CB
               tSerialPort_CB_tab[]={
 2   {
 3    /* variable */
 4    /* receive buffer：rcv_buffer */
 5    SerialPort_CB_rcv_buffer_INIT,
 6    /* send buffer：snd_buffer */
 7    SerialPort_CB_snd_buffer_INIT,
 8    /* pointer to cell INIB */
 9    &tSerialPort_INIB_tab[0]
10   }
11 };
```

Figure 11. Initialization of *cell* CB for SerialPort.

stores the pointer of the buffers allocated in Figure 11 (Lines 5-7) and the pointer of the *cell* INIB (Line 9).

- IDX
  IDX is an identifier to obtain the *cell* CB. There are two types of IDX: a subscript to the array-managed *cell* CB, and the pointer to the *cell* CB. Figure 7 shows an example where IDX is a pointer.
- *Entry port descriptor*
  An *entry port descriptor* (Figure 7(A)) is a structure that has *entry port*s and the *cell* information, such as a pointer to the *entry port skeleton* function table, IDX, and a subscript to the *entry array*.
- *Entry port skeleton* function table
  An *entry port skeleton* function table (Figure 7(B)) is a function table that has a set of function pointers to the *entry port skeleton*.
- *Entry port skeleton*
  An *entry port skeleton* (Figure 7(C)) is a function for calling an *entry port function* that has the caller *cell* information, such as IDX and the subscript to the *entry array*.
- *Entry port function*
  An *entry port function* (Figure 7(D)) is a function for providing the functionality of the *cell*s. An *entry port function* is based on template code generated by the interface generator.

B. Connection Optimization

The *call port* and *entry port* of the *cell*s can be optimized when certain conditions are met.

1) Optimization of a call port: There are four types of optimization for *call port*s that reduce the execution time.

1. Omission of the *entry port skeleton* function
   The *entry port* function is directly called without passing through the *entry port skeleton* function if and only if the number of parameters of the *entry port* functions connected to a *call port* is constant. The number of parameters depends on whether a *cell type* is singleton or not, and whether an *entry port* is an *entry port* array or not.
2. Omission of function table
   The *entry port* function is directly called without referring to the *entry port skeleton* function table if and only if each *call port* of all the *cell*s is connected to the *cell*s that are instantiated by a *cell type*.
3. Omission of the *entry port descriptor*
   The *cell* CB of a calling *cell* is mapped to the *cell* INIB of the caller's *cell* if and only if the *call port* optimization 2 and the *entry port* connected to a *call port* are not the *entry port* array.
4. Single *cell*
   The *call port* of a caller *cell* directly includes the *cell* CB and *cell* INIB of the caller cell if and only if the *call port* optimization 3 and all the *call ports* of all the *cells* that are instantiated by a *cell type* are connected to a *cell*.

2) Optimizations of entry port: There three types of optimization for *entry port*s that reduce memory consumption.

1. Deletion of the *entry port skeleton* function
   An *entry port skeleton* function can be deleted if and only if there are no references to an *entry port skeleton* function when the *call port* optimization 1 is applied to the *call port*.
2. Deletion function table
   An *entry port skeleton* function table can be deleted if and only if there are no references to a function table when the *call port* optimization 2 is applied to the *call port*.
3. Deletion of the *entry port descriptor*
   An *entry port descriptor* can be deleted if and only if there are no references to an *entry port descriptor* when the *call port* optimization 3 is applied to the *call port*.

C. Improving Call Port Optimization

The optimization of the *call port* can be improved when the composition of the *cell*s are known using Figure 12 as an example. In this example, the optimization of the *call port* of *cell*s (A1 or A2) instantiated by the same *cell type* tA is described. The numbers in Figure 12 represent the *call port* optimization number from Section III-B1.

In case (A), it is not possible to adapt to the omission of the function table optimization because the connected *cell types* of cell A1 and A2 are different (tC or tD). Furthermore, the number of parameters in the entry functions is different because tB is not a *singleton cell type* and tC is a *singleton cell type*. Therefore, this case cannot be optimized.

In case (B), it is also not possible to adapt to the omission of the function table optimization. It is possible to adapt the omission of the *entry port skeleton* function optimization

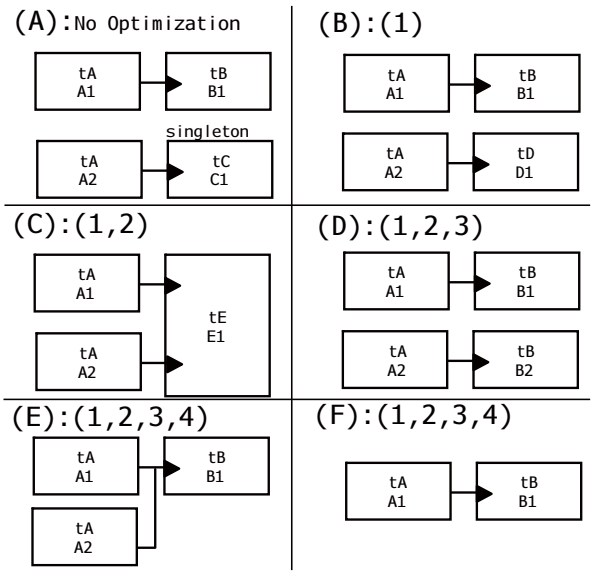

Figure 12. Examples of *Call Port* Optimization

because the *cell type* connected to the A1 or A2 *cell* is not a *singleton* nor an *entry array*. In cases (C-F), it is possible to adapt to the omission of the *entry port skeleton* function operation.

In case (C), it is possible to adapt to the omission of the *entry port skeleton* function optimization because the *cell type* connected to A1 or A2 *cell* is the same *cell type* (tE). It is not possible to adapt to the omission of the *entry port descriptor* optimization, because the *entry port*s connected to A1 and A2 *cell* are *entry port array*s.

In case (D), it is possible to adapt to the omission of the *entry port skeleton* function optimization because the *cell type* connected to A1 or A2 *cell* is the same *cell type* (tB). As well, it is possible to adapt to the omission of the *entry port descriptor* optimization, because the *entry port*s connected to A1 and A2 *cell*s are not an *entry port array*. It is not possible to adapt the single *cell* optimization because the *cell*s connected to A1 and A2 *cell*s are different (B1 or B2).

In case (E), it is possible to adapt to all the optimizations because the *cell* connected to A1 or A2 *cell* is the same *cell* (B1).

In case (F), it is also possible to adapt to all the optimizations because the *cell* installed by the tA *cell type* is only A1.

As mentioned in Section III-B2, optimization for the *entry port*s is adapted if and only if there are no references to the information when the *call port* optimization implements the *call port*.

Table I
EXPERIMENTAL PLATFORMS.

Processor	SH3(SH7727)
Board	MS7727CP02
Clock	96MHz
Compiler	gcc(3.3)
Compiler Option	-O2

Table II
OVERHEAD OF FUNCTION CALLS.

The number of character	1character	5characters
Non component	29.15μs	63.67μs
Component without optimizations	31.27μs	69.16μs
Component with optimizations	29.16μs	64.18μs

D. Experimental Results

This subsection presents a set of experimental results obtained by comparing a non-component program with a component program in order to demonstrate the function call overhead and the effectiveness of optimization. Table I depicts the experimental platforms.

Three different measurements of the function calls are performed: non-component program, component program without optimization, and component program with optimization. The serial driver of TOPPERS/ASP which is a real-time operating system is used as a non-component program. In the non-component case, the glue code between the dependent and non dependent parts is hand-coded.

All the optimizations described in Section III-B1 are adapted to the *call port*. The measurement range of execution is from the point where an application calls the send function of the serial driver to the point where the procedure returns to the caller application.

Table II shows the calling overhead results. Function call overhead is evaluated in two cases where the number of characters is one or five. The numbers in Table II are the average execution times for 10,000 iterations. Component programs with optimization were approximately 2 μs faster for the one-character case and 5 μs faster for the five-character case compared with the component program without optimization. Furthermore, the overhead of a component program with optimization is less than 1 μs compared to the non-component program. These results imply that the execution time overhead for the component program with optimizations is relatively small, which indicates that the appropriate interface code has been generated by the interface generator.

Next, the line numbers of the source code written by a developer are compared. The values in Table III represent the independent part, the dependent part, the glue code (glue), the component description (CD), and the total number of lines of code. Comments and empty lines are not measured.

Table III
COMPARISON OF CODE SIZE.

	independent	dependent	glue	CD	total
Component	456	185	0	125	766
Non component	549	278	119	0	946

Table IV
EXECUTABLE FILE SIZE.

Section	text	data	bss	rodata	total
Non component	3,632B	0B	568B	100B	4,300B
Component without optimization	3,632B	100B	512B	190B	4,436B
Component with optimization	3,120B	100B	512B	88B	3,820B

Table III shows that the total code size of the component program is smaller than that of the non-component program. This is because the source code lines generated by the interface generator are more than those of the component description. Furthermore, in the TECS case, developers only modify a few lines of the *build* description to change other serial ports. On the other hand, in the non-component case, developers have to implement glue codes for each serial controller. Therefore, by using TECS, it is easy to change system configurations.

Finally, Table IV shows the size of each section. The text section includes the program operation, the data section includes the *variable*s of the *cell*, the bss section contains the send or receive buffer, and the rodata section contains the *attribute*s of the *cell*. It is possible to reduce RAM consumption because the *attribute*s of each *cell* are mapped to ROM (rodata). Additionally, the text and rodata sections are reduced because of the deletion of the *entry port* descriptor and the *entry port* skeleton function if the proposed optimization technique is used. Furthermore, the glue code is simplified in components that are optimized. Therefore, the total file size of the component program with optimization is smaller than the non-component program.

IV. CONCLUSIONS

The present paper described TOPPERS Embedded Component System (TECS). It is possible to minimize the execution time overhead and the memory consumption of an entire application because TECS adopts a static configuration. A static configuration means that both the configuration of the component behavior and the interconnections between components are static. Additionally, an optimization of the interface code was proposed. No instantiation overhead was introduced at runtime, and runtime overhead of the interface code was minimized. A serial interface driver was used as a case study to evaluate the execution time overhead, the software code size, and executable file size. The case study demonstrated the effectiveness of the proposed optimization.

REFERENCES

[1] Sun Microsystems, Inc., "JavaBeans," http://java.sun.com/products/javabeans/.

[2] Microsoft Corporation, "Activex," http://www.microsoft.com/activex/.

[3] OMG, "CORBA Component Model 4.0," http://www.omg.org/technology/documents/formal/components.htm.

[4] Microsoft Corporation, "Microsoft Component Object Model," http://www.microsoft.com/com/.

[5] S. Lin, J. Wu, and Z. Hu, "A contract-based component model for embedded systems," in *Proc. Fourth International Conference on Quality Software, 2004.*, Sep. 2004, pp. 232–239.

[6] I.-L. Yen, J. Goluguri, F. Bastani, and L. Khan, "A component-based approach for embedded software development," in *Proc. Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing,*, May 2002, pp. 402–410.

[7] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Design of dynamically reconfigurable real-time software usingport-based objects," *Software Engineering*, vol. 23, no. 12, pp. 759–776, Dec. 1997.

[8] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *IEEE Computer*, vol. 33, no. 3, pp. 78–85, Mar. 2000.

[9] K. C. Wallnau, "Volume iii: A component technology for predictable assembly from certifiable components." in *Technical report, Software Engineering Institute, Carnegie Mellon University*, 2003.

[10] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli, "The SAVE approach to component-based development of vehicular systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 655–667, May 2007.

[11] I. Crnkovic, "Component-based software engineering for embedded systems," in *Proc. 27th International Conference on Software Engineering*, May 2005, pp. 712–713.

[12] TOPPERS Project, http://www.toppers.jp/en/index.html.

[13] T. Azumi, M. Yamamoto, Y. Kominami, N. Takagi, H. Oyama, and H. Takada, "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, pp. 46–50.

[14] T. Azumi, H. Oyama, and H. Takada, "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, pp. 204–209.