Developing platform specific model for MPSoC architecture from UML-based embedded software models

Sang-Uk Jeon a, Jang-Eui Hong b,*, In-Gwon Song a, Doo-Hwan Bae a

Division of Computer Science, College of Information Science and Technology, KAIST, Republic of Korea
School of Electrical and Computer Engineering, Chungbuk National University, Cheongju, Republic of Korea

A R T I C L E   I N F O

Article history:
Received 15 September 2007
Received in revised form 20 April 2009
Accepted 22 April 2009
Available online 3 May 2009

Keywords:
MPSoC
Embedded software
Platform independent model
Platform specific model
UML
Task allocation

A B S T R A C T

In this paper, we describe a technique to design UML-based software models for MPSoC architecture, which focuses on the development of the platform specific model of embedded software. To develop the platform specific model, we define a process for the design of UML-based software model and suggest an algorithm with precise actions to map the model to MPSoC architecture. In order to support our design process, we implemented our approach in an integrated tool. Using the tool, we applied our design technique to a target system. We believe that our technique provides several benefits such as improving parallelism of tasks and fast-and-valid mapping of software models to hardware architecture.

1. Introduction

In embedded software design, the issues on the performance, parallelism, and module (or practically task) coupling need to be considered in addition to the common issues in general purpose software design (Douglass, 2006). Dealing with such issues introduces not only the consideration of software architecture but also that of hardware platform where software will be deployed. However, the hardware platforms of embedded systems vary in size and complexity. So it is hard for software developers to design software models conforming various hardware platform constraints. In order to deal with the problem, the concept of the Object Management Group (OMG)'s Model-Driven Architecture (MDA) has been introduced in embedded software design (Ha, 2006; Hong and Bae, 2006; Gerard et al., 2005; Lu et al., 2005; Rouxel et al., 2005).

The MDA approach is based on two essential concepts; the Platform Independent Model (PIM) and the Platform Specific Model (PSM) (Kleppe et al., 2003). The PIM captures the essential features of the system. The PSM determines how the PIM executes in the target deployment environment. In the context of embedded software design, the PIM can be defined as the software model. Then the PSM would include the hardware architecture and its configuration, and the result of mapping software tasks to hardware resources. The software model is developed independent of the hardware architecture and its configuration. The mapping of software tasks to hardware resources is a key factor in the development of PSM.

This paper suggests how to develop the PSM, especially how to map software tasks to hardware resources, for the Multi-Processor System-on-Chip (MPSoC) architecture from the PIM which is represented using UML (OMG, 2007). The MPSoC is a system-on-a-chip (SoC) which uses multiple processors. It contains multiple heterogeneous processors, memory blocks, and several I/O resources in a chip. The MPSoC architecture has become a solution for designing embedded systems dedicated to applications that require intensive computations (Atitallah et al., 2007). In the MPSoC architecture, the mapping of software tasks to hardware resources is important since it affects the degree of parallelism among multiple processors and the utilization of hardware resources. In the transaction-oriented general applications, the PSM is a model which contains the information of implementation platforms such as EJB, CORBA, or .NET (Lewis et al., 2006). The PSM is depicted with nodes and edges which mean physically separated computing machines and communicating networks, respectively. However, the PSM of embedded software includes fine-grained hardware resources such as microprocessor, DSP, memory, device controller, and so on (Jerraya and Wolf, 2004). Thus the mapping of PIM for MPSoC architecture should be more detailed than those for distributed and network system.

* Corresponding author. Tel.: +82 43 261 2261; fax: +82 43 273 2265.
E-mail addresses: sujeon@se.kaist.ac.kr (S.-U. Jeon), jehong@chungbuk.ac.kr (J.-E. Hong), igsong@se.kaist.ac.kr (I.-G. Song), bae@se.kaist.ac.kr (D.-H. Bae).
In our approach, we first design the PIM which consists of three UML diagrams – i.e. class, interaction overview and sequence diagrams – and also design a hardware architecture diagram which represents the structure of MPSoC architecture based on the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) (OMG, 2008b). From the PIM, we generate control flow graphs (CFGs). The CFGs are used to represent the behavioral view of a system. Each connected graph in CFGs is identified as a candidate task. Then, by considering parallelism among them, the candidate tasks are further refined and the final tasks are identified. Before allocating those tasks, we calculate the value of correlation factors (CRFs) between each pair of tasks. The CRF value indicates the strength of coupling between two tasks. They are used to decide whether the two tasks should be assigned to the same hardware resource. We define three functions to support the above steps; CFG generation function, task identification function, and CRF calculation function. With the identified tasks and CRF values, the allocation of software tasks to hardware resources is accomplished.

The contribution of this paper lies in the suggestion of methods to identify tasks from UML-based PIM and to evaluate a particular hardware architecture with those tasks through an algorithm that allocates the tasks to hardware resources. Our allocation algorithm is performed using the information of each task regarding its computation type and the costs for communicating with other tasks. In embedded system design, the allocation of tasks to a hardware architecture is usually done by the designer, and its evaluation is performed by tools such as CoFluent Studio (CoFluent Design, 2008). On the other hand, our approach is able to replace the works done by the designer and tools.

Our approach provides several benefits. First, our allocation process is performed at model-level; we can allocate software tasks to hardware resources before writing code. Thus, through the simulation in design phase, we can examine whether the allocation is adequate or not. In addition to that, by providing several functions, our approach makes the allocation process fast, easy, and valid. And the consideration of the types of tasks and hardware resources enables effective resource utilization. Finally, the identification and refinement of concurrent tasks have the effect of maximizing the parallelism of tasks. These advantages come from the consideration of the precise and fine-grained information about the hardware resources of MPSoC architecture. We will explain the source of these advantages, which is our main interest, in the following sections.

The remainder of this paper is organized as follows: Section 2 describes the survey and analysis of existing related efforts. Section 3 includes the transformation process of PIM to PSM and the system model supporting the process. In Section 4, the algorithm for allocating tasks to hardware resources is given, and Section 5 analyzes the result of applying our mapping technique to a target system. Finally the conclusion and further works are described in Section 6.

2. Related works

We have classified the existing research works into two categories. The first category is the research on the transformation technique from PIM to PSM in Model-Driven Architecture (MDA) paradigm. The other is on the task allocation algorithms in multi-processor environment.

Model transformation in MDA paradigm is highly dependent on the designer’s heuristics. Also this transformation is performed at the level of the abstract component such as software packages. A representative research on MDA-based model transformation is studied by IST (2004) project. In Master project, the mapping rules for software tasks using OCL (Object Constraints Language) are suggested. In this work, the components of PIM and PSM are maintained at the repository. Then the components of PIM are transformed to those of PSM using templates or patterns. The result of this transformation is C++ code for the model. However this has the different focus with our research because they consider the distributed application environments as the target platform of PSM while we consider the MPSoC environments. The physical environment of MPSoC is different with that of distributed system in aspects of message routing, communication, and resource failure. Boudiur and Kimour (2006) propose a framework for transforming the UML models such as state transition diagram and object diagram into SystemC (Open SystemC Initiative, 2008). In order to generate code, they use predefined SystemC templates which contain the structural definition of hardware interface. In their framework, UML models are mapped to SystemC templates using XMI syntax, and then the code is generated as the mapping result. However, their suggestion is difficult to apply to MPSoC embedded software because the hardware architecture is not considered.

For the model transformation, several works have suggested model transformation languages (MTLs) such as OMG MOF/QVT (Meta Object Facility/Query View Transformation) (OMG, 2008a), Kafka (Wei et al., 2003), CoReAT (Agrawal, 2003), VIA-TRA2 (OptXware Research & Development LLC, 2008), and ATL (ATLAS Group, 2008). The MTL is used to define how a set of source models is visited to create a set of target models (Bezivin et al., 2005). It defines how the basic operations on models can be performed using a specific set of language constructs such as declarative rules and imperative instruction sequences. The suggested MTLs usually define three languages: one for describing the PIM, another for the PSM, and the other for the rules to transform the first one to the second one. We use the graphical notation, UML, to describe the PIM, which makes its design easy. To handle the UML notations in the implementation, we represent the PIM in XMI format using Eclipse. The PSM is also represented in XMI format. The rules to transform the PIM to PSM can be represented using the existing MTLs. However, in this paper, we choose to represent it in terms of the functions with algorithms instead of using the MTLs.

Most of task allocation algorithms, which belong to the second category, have a strict assumption that multiple processors of the platform architecture are homogeneous type. Therefore the main purpose of these algorithms is to meet the response time and to reduce communication overhead. Plishker et al. (2004) suggest a method of allocating tasks into networked processors of IXP1200 system which is one of embedded systems supporting multi-thread processing. They consider the homogeneous six processors as the target architecture. However, tasks are allocated in the executable code level. Benini et al. (2005) also suggest a task allocation method for MPSoC architecture. They are interested in resource constraints for the allocation. However, it does not consider heterogeneous types of processors in MPSoC architecture. They present a task allocation algorithm for cooperative robots. To allocate tasks to multi-robot systems, they use vacancy chains which represent control chains of empty slots. This algorithm dynamically allocates a task into an empty slot to continuously perform the job given to mobile robot. Thus this algorithm is different from our research in aspects of static and model-based task allocation. Additionally, Hong and Prasanna (2004) focus on the allocation algorithm of the same-sized and independent tasks to distributed computing environments. And Paulin et al. (2004) suggest the parallel programming model which considers the message passing and task scheduling in StepNP multi-processor SoC platform. However these researches are performed in distributed computing environment using network protocol, and the alloca-
tion are performed in the code level. These points are the differences between these researches and ours. Also Fredriksson et al. (2005) present an approach for allocating components to real-time tasks considering several properties such as CPU overhead and memory consumption. Their approach focuses on the identification of tasks from the source model. Our approach further considers the allocation of tasks to hardware resources as well as the identification of tasks.

The commercial tools such as CoFluent Studio (CoFluent Design, 2008) and VisualSim (Mirabilis Design, 2008) support the modeling of functionality and architecture, and the evaluation of mapping the functionality to the architecture. They do not focus on how to map the functional tasks to the architecture, which is the main theme of our research. Instead, they focus on the evaluation of the mapping results. Furthermore, in those tools, the designer is required to model the functional tasks. In this paper, we use the UML model for the design of functionality and suggest how to identify tasks from the UML model.

3. Transformation system

3.1. Transformation process

Fig. 1 shows the detailed process of the PSM development starting from PIM. From the PIM, the CFGs are generated. The purpose of this generation is to identify tasks. In the CFGs, each connected graph is identified as a task which is a unit of execution. Details about the generation will be presented in Section 3.2.3. Before allocating software tasks to hardware resources, the values of correlation factors (CRFs) between the tasks are calculated. The CRF value between two tasks indicates how strongly the tasks are coupled. The details about the CRFs also will be explained in Section 3.2.3. With the CRF values, the allocation instance is constructed by allocating the software tasks to hardware resources. The allocation is achieved automatically by our algorithm which will be explained in Section 4. If needed, the developer can manually change the allocation instance. The final PSM is produced after verifying, through the simulation in our approach, the allocation instance.

3.2. Transformation system

The transformation system, as shown in Fig. 1, is defined as follows:

Definition 1 (Transformation system). A transformation system, TS is defined as 

\[ (S, D, F, \alpha, L_m) \]

where

- \( S \): a source model of the transformation system, i.e., a PIM and a hardware architecture diagram.
- \( D \): a target model which is the output of the transformation system, i.e. the allocation instance in Fig. 1.
- \( F \): a set of helper functions used in the process of the transformation.
- \( \alpha \): a set of mapping action \( \alpha \).
- \( L_m \): The allocation function that allocates tasks to hardware resources, which is a combination of mapping actions, \( \alpha \).

3.2.1. Source model, S

The source model of our transformation system includes a PIM, and a hardware architecture diagram which depicts the structure of the MPSoC. The PIM contains three UML 2.0 diagrams; the Class Diagram (CD), the Interaction Overview Diagrams (IODs), and the Sequence Diagrams (SDs). We assume that there exists a use case model that describes the system requirements. The diagrams in the PIM are constructed as the results of the use case realization. In most software development methodologies, the use case realization describes how the use cases are implemented in terms of collaborating objects. The CD describes the static structure of the collaborating objects. The IODs and SDs are used to describe how the objects communicate with each other.

Each class in the CD represents either a software task that performs computations or communicates with I/O devices, or an entity that are used for data storage. In the construction of the CD, the designer is required to specify the stereotype of each class. We use the stereotypes for classes which are used in COMET methodology (Gomaa, 2000). In COMET, there are four stereotypes for the application classes; \(<\text{interface}>\), \(<\text{control}>\), \(<\text{application logic}>\), and \(<\text{entity}>\). Although each stereotype can...
further be specialized, we simply use the four stereotypes for our purpose. In case of the stereotype "<<application logic>>", when used in real-time applications, the classes of the stereotype are usually algorithm classes that encapsulate complex computations (Gomaa, 2000). So we renamed the stereotype as "<<data-intensive processing>>". The four stereotypes are also used to type software tasks.

The IODs and SDs describe the behavior of the software based on the use case scenarios. An IOD represents a use case scenario. In most software development methodologies, an SD is used to describe a use case scenario (Jacobson et al., 1999). However, in our approach, an SD represents a part of a use case scenario. That means, several SDs need to be composed to describe a use case scenario. It is the IOD that shows how those SDs are composed to describe a use case scenario. For example, the main scenario of the 'Determine Distance and Speed' use case of the cruise control system, which is an example in Gomaa (2000), is described using an IOD as shown in Fig. 2. Each InteractionUse node in the IOD refers another IOD or an SD. Fig. 3 shows the detailed behavior of the 'ReadConstantAndCount' node described using an SD. The details about the construction of PIM is described in our previous works (Jeon et al., 2006a,b).

For the description of hardware architecture, we use the MARTE. For the hardware logical model, the profile defines five subpackages: HW_Computing, HW_Storage, HW_Communication, HW_Timing, and HW_Device packages. Each subpackage represents a particular hardware resource's type. As we focus on the software in UML level, we do not consider behavioral properties of the underlying hardware such as pipelining and caching. So we use only a part of those subpackages to abstract the hardware as follows:

- The HW_Computing subpackage defines computing resources such as processor, ASIC (Application Specific Integrated Circuit) and PLD (Programmable Logic Device). Since we target the software on programmable processors, we only use computing resources of HW_Processor type.
- The HW_Storage subpackage includes the HW_Memory resource and the HW_StorageManager resource. A HW_Memory resource could be a processing memory such as cache and RAM, or a storage memory such as ROM and disk drive. We do not consider fine-grained resources such as cache and ROM. So we only use the storage resources of HW_RAM and HW_Drive types. The HW_StorageManager resource, which denotes memory brokers, is not considered.
- The HW_Communication subpackage provides the HW_Media resource that denotes a communication resource able to transfer data with a bandwidth. It could be either a HW_Bus or a HW_Bridge. We use both of them.
- We do not consider the timing resources in the HW_Timing subpackage. Instead, the timing resources will be incorporated into our simulation environment.
- The HW_Device subpackage has two subcategories; the HW_IO and the HW_Support. The former denotes resources that interact with the environment such as sensors and actuators, and the latter denotes support resources like power suppliers. We only use the HW_IO resource.

Fig. 4 shows the metamodel of our hardware architecture based on MARTE. An example MPSoC hardware architecture is shown in Fig. 5.

### 3.2.2. Target model, D

The target model, D, is the outcome of our transformation system. As shown in Fig. 1, D is the allocation instance which is generated by allocating software tasks to hardware resources. Fig. 6 shows the metamodel of the allocation instance. The Allocation contains three types of allocations. A ProcessorAllocation represents the allocation of a task whose type is either <<control>> or <<data-intensive processing>> to a resource of HW_Processor type. The MemoryAllocation and I/OAllocation indicate the allocation of an entity task to a HW_RAM type resource and that of an interface task to a HW_I/O type resource, respectively. In the allocation, multiple tasks can be allocated to a hardware resource. However, a task cannot be allocated to multiple hardware resources.

### 3.2.3. Helper functions, F

The transformation from the source model to the target model consists of three helper functions: (1) the function to generate CFGs from UML-based PIM, (2) the function to identify the final tasks using the CFGs, and (3) the function to calculate the correlation factors between the tasks. After the execution of these functions, the allocation function, \( L_m \) will be performed to map the final tasks into hardware resources.

Each helper function requires the CFGs as its input or output. Here, we first define the CFG before explaining the helper functions. A CFG is a directed graph whose vertices and edges represent the model elements of sequence diagram and control flows between them, respectively. The definition of CFG is as follows:

**Definition 2** (Control Flow Graph). A Control Flow Graph, \( G = (V, E, N_v, N_e) \), where

- \( V \): a set of vertices which represent model elements such as events, fork, join, decision, merge in IODs and SDs.
- \( E \subseteq V \times V \): a set of edges.
- \( N_v \): a set of starting vertices of a graph.
- \( N_e \): a set of ending vertices of a graph.
The CFG generation function, \( F_{cfg} \), returns a set of CFGs from the input PIM: \( F_{cfg}(PIM) = \{ CFG_i | i = 1 \ldots n \} \) by traversing IODs and SDs.

The complete algorithms for generating CFG are described in Song (2007). Here, we only present the rough process of generating CFGs through a simple example. Fig. 7a shows a simple IOD each of

1. CFG generation function

Fig. 4. The metamodel for our hardware architecture based on MARTE.

Fig. 5. An example diagram of MPSoC architecture.
whose node represents an SD. First, a CFG is generated from each InteractionUse or Interaction (OMG, 2007) in an IOD, which is represented using an SD. In sd A, there are two objects, :P and :Q. :P sends a synchronous message m1 to :Q. Then, there are two message occurrences (OMG, 2007); one represents the sending of m1 in :P and the other represents the receiving of m1 in :Q corresponding e1 and e2 in Fig. 7a, respectively. The generation of a CFG starts with the creation of vertices representing all occurrences of synchronous and return messages in a SD. Then, those vertices are connected according to the temporal order of their corresponding occurrences in the SD. For the sd A in 7a, vertices e1, e2, e3, and e4 are created. Then, by connecting them according to their temporal order in the SD, a CFG for the sd A is generated as shown in the upper half of Fig. 7b. In the same way, a CFG for the sd B is generated as shown in the lower half of Fig. 7b. The next step is connecting those CFGs according to the temporal order of their corresponding SDs in IOD. In Fig. 7a, the sd A precedes the sd B. So the last vertex of the CFG for sd A, e4, is connected to the first vertex of the CFG for sd B, e5. Through this way, the CFGs for our source model is generated.

There exist several works that analyze the control flow of the scenario-based model (Alur et al., 2003; Garousi et al., 2005). The difference between those works and our generation function is the handling of asynchronous messages that are exchanged between concurrent processes. Existing control flow analysis methods treat the asynchronous messages and the synchronous messages equally because their purpose is to find control dependencies. Then, in the resulting CFGs, the concurrent processes are tied to each other by the control flows, which loses the concurrent processes. On the contrary, our generation function does not tie the concurrent processes by the asynchronous messages. Only the sendings and receivings of the asynchronous messages are specified to the nodes of the CFGs. This results the set of connected graphs whose number is the same with the number of concurrent processes specified in the SDs, which enables us to identify concurrent tasks.
(2) Task identification function

After the generation of all CFGs from the source model, the task identification function, \( F_{\text{task}}(\text{CFG}) = \{ t_i, i = 1 \ldots n \} \), decides final tasks from the CFGs. As we stated in the above, the generated CFGs include a set of connected graphs whose number is the same as the number of concurrent processes specified in the SDs. Basically, the task identification function identifies each connected graph as a task. Then it identifies additional tasks by partitioning the connected graphs with the consideration of parallelism. A ‘fork’ vertex, which corresponds to a par fragment in SD or a fork node in IOD, in the CFG of a task indicates that there exists a parallelism inside the task. If a branch of the ‘fork’ vertex, which corresponds to a thread in the CFG of a task, includes so many message passings or complex computations, it would be better to split the branch and make it another parallel task which can be allocated to a different hardware resource. The task identification function traverses each connected graph until it encounters a fork vertex that has several branches which end at a join vertex. Then it counts the number of vertices in each branch. If the number of vertices in a branch is over a certain threshold, then the task identification function splits the branch from the CFG and regards it as another parallel task.

The task identification function is defined as follows:

Definition 3 (Task Identification Function). Inputs
- \( \mathcal{G} \): The initial set of CFGs

Outputs
- \( \{ \mathcal{G}_i \} \): The set of tasks, which is a set of CFGs.

Algorithm
\[
\begin{align*}
\mathcal{G}_i & = \emptyset \\
\text{for all } & \mathcal{G}_i \text{ s.t. } c \text{ is a connected graph do} \\
\mathcal{G}_i & = \mathcal{G}_i \cup \{ c \} \\
\text{end for} \\
\text{for all } & f \in \mathcal{G}_i \text{ do} \\
\text{for all } & f, j \in c \text{ s.t. } f \text{ is a fork vertex and } j \text{ is the corresponding join vertex of } f \text{ do} \\
\text{for all } & \mathcal{G}_i \text{ do} \\
& \{ v | \mathcal{G}_i \subseteq \{ v | \mathcal{G}_i \}, v \in c.V \}; \\
& \{ (s, t) | (s, t) \in \mathcal{G}_i , c.E \}; \\
& \{ (s, t) | (s, t) \notin \mathcal{G}_i , c.E \} \\
& \{ v | \mathcal{G}_i \subseteq \{ v | \mathcal{G}_i \}, v \in c.V \}; \\
& \{ (s, t) | (s, t) \in \mathcal{G}_i , c.E \}; \\
& \{ (s, t) | (s, t) \notin \mathcal{G}_i , c.E \} \\
& \mathcal{G}_i = \mathcal{G}_i \cup \mathcal{G}_i \\
& \mathcal{G}_i = \mathcal{G}_i \\
& \text{end if} \\
& \text{end for} \\
& \text{end for} \\
& \text{end for} \\
& \text{end for}
\]

In the algorithm of Definition 3, the first for loop identifies each connected tasks and regards it as a task. The second one identifies additional tasks by partitioning the CFG of each task with the consideration of parallelism. In the second for loop, the task identification function first chooses a fork vertex \( f \) and its corresponding join vertex \( j \) from the CFG \( c \) of a task. And it identifies two vertices \( \mathcal{G}_i \) and \( \mathcal{G}_j \) which are the start and the end vertices for a branch of the fork vertex \( f \). Then the CFG \( \mathcal{G}_c \) is a partial order relation in the CFG \( c \). If the number of vertices of the CFG \( \mathcal{G}_c \) is larger than the threshold, the THRESHOLD of the algorithm in Definition 3, the CFG \( \mathcal{G}_c \) is split from the CFG \( c \) and becomes an independent CFG, \( nc \), which represents a new task.

To decide the threshold, we consider the relationship between the execution load of the branch and that of the operation of creating a new task. If the execution load of the branch is smaller than that of the task creating operation, it is not worth creating a new task for the branch. Since the CFGs are generated from the IODs and SDs, most of the vertices in the branch are related to the message passing operation. Thus we compare the execution load of the message passing operation with that of the task creating operation. For the comparison, we use a simulator, EMSIM (Tan et al., 2003), to approximate the instruction counts of a message passing operation and a fork operation in Linux OS on a StrongARM processor. The approximation results show that a message passing operation requires 13 instructions while a fork operation requires 1166 instructions. This indicates that the threshold should be at least 90 times of the number of vertices for a message passing operation, which is 4 in our CFGs. Consequently, we set the threshold to 360. The approximation we performed to decide the threshold is dependent on a specific program language on a specific platform. The threshold needs to be changed according to the target platform.

Fig. 8 shows an example of partitioning tasks. The left part of Fig. 8 represents the initial set of CFGs, \( C_i \), which contains a CFG representing a task. The CFG \( c \) contains a fork vertex \( f \) which has several branches, \( b_1 \) to \( b_n \). The task identification function examines the number of vertices of each branch. Assume that the first branch \( b_1 \), from the vertex \( V \) to \( V \), has more than 360 vertices. Then the task identification function splits \( b_1 \) from \( c \) and makes it an independent CFG, \( nc \), which represents a new task. The resulting set of CFGs, \( C_i \), contains two CFGs, \( c \) and \( nc \), as shown in the right part of Fig. 8.

After identifying all tasks, the function defines the type of each task. The type is derived from the CD in the source model. Basically, each connected graph in the CFG represents an active class in the CD. So the type of each task can be derived straightforwardly. In case of additionally identified tasks by partitioning the CFGs of some tasks, their types are defined by inheriting the types of their original tasks. Beside that, the function derives entity classes from the CD and treats them as tasks of <<entity>> type. Precisely, they are not tasks since they have no thread of control. Our approach treats them as a special kind of task because they are needed in the allocation of tasks to hardware resources. Consequently, the task identification function returns a set of tasks each of whose type is either <<interface>> or <<control>> or <<data-intensive processing>> or <<entity>>.

(3) CRF calculation function

This function calculates CRFs values of all pairs of tasks; i.e., \( \text{CRF}(t_i, t_j) = |\text{CRF}(t_i = 1 \ldots n \times j = 1 \ldots n)| \). The CRF represents the interrelatedness between two tasks. This interrelatedness indicates the strength of coupling between two tasks. So, the higher the CRF value, the stronger the coupling between tasks. Two tasks with high CRF value may be allocated at one hardware resource to reduce the interaction overhead between them. If one of them is a task of <<entity>> type and the other is <<control>> or <<data-intensive processing>> type, the <<entity>> task

\[ C_{\text{crf}} = \{(p_1, p_2) | \{p_1, p_2, \ldots, (p_n, p_n) \} \in C \} \] such that \( C \) is a set of edges of CFG \( c \). For convenience, we denote \( (p_1, p_2) \in C \) as \( p_1 \subseteq C \).
should be allocated at the local memory of the processor to which the other task is allocated.

The CRF value is calculated by summing up two values: the Message-Related-Factor (MRF) value and the Fork-Related-Factor (FRF) value. The MRF value represents the message relatedness between two tasks. For instance, assume that a task $t_i$ sends a message $m$ to a task $t_j$ and the message $m$ has two parameters, one of which is the type of integer and the other is the type of double. Then the message size is calculated by summing up following values:

- The size of message identifier, which we assigned the value of 4.
- The size of an integer-type parameter, which we assigned the value of 4.
- The size of a double-type parameter, which we assigned the value of 8.

Thus the size of message $m$ is 16. Usually, in 32-bit computer architecture, a message identifier consumes 1 word, 4 bytes. So we assign the value of 4 to the size of it. In those languages, the sizes of integer and double variables are 4 and 8 bytes long, respectively. We first define the size of a message.

**Definition 4 (Message Size).** The size of a message $m$ is defined as follows:

$$MSize(m) = S_{mid} + \sum_{p: Par(m)} Size(p).$$

where
- $S_{mid}$: the size of message identifier, mid.
- $Par(m)$: the set of parameters in $m$.
- $Size(p)$: the size of the data type, $p$.

The MRF value between two tasks is calculated by the summation of the sizes of all messages that are exchanged between them. Here, we additionally consider control structures such as loop and decision which enclose the messages. If a message is included in a loop $l$, the size is multiplied by the number of iterations of the loop, $l(l)$. Here, the designer is required to specify $l(l)$. In case of branches in a decision vertex, it is multiplied by the probability $P(m)$ that the branch is selected. For a decision vertex, the specification of $P(m)$ for each branch is also needed. If the designer does not specify the probabilities for the branches, they are all assumed be 1 divided by the number of branches in the decision vertex. If the designer does not specify control structures in the PIM, both the values $l(l)$ and $P(m)$ are simply assumed to be 1.

**Definition 5 (MRF).** The calculation function of Message-Related-Factor, MRF is defined as follows:

$$MRF(t_i, t_j) = \sum_{m \in M(G_G)} \left( \prod_{l : l(l)} P(m) MSize(m) \right),$$

where
- $M(G_G)$: a set of synchronous messages from a vertex $G_G$ in the CFG of the task $t_i$ to a vertex $G_G$ in the CFG of the task $t_j$.
- $l(l)$: the predicted number of iterations in the loop $l$.
- $L(m)$: a set of loops enclosing the message $m$.
- $P(m)$: the probability that the branch, that contains the vertex which sends the message $m$, is selected in decision.
- $S_{mid}$: the size of message identifier, mid.

The FRF means the relatedness between two tasks by fork/join operation. If one task forks another task, then the value of FRF between the two tasks is 1, otherwise 0.

**Definition 6 (FRF).** The calculation function of Fork-Related-Factor, FRF is defined as follows:

$$FRF(t_i, t_j) = \begin{cases} 1, & \text{if } t_i \text{ forks } t_j \text{ or vice versa.} \\ 0, & \text{otherwise.} \end{cases}$$

With MRF and FRF, we define the CRF.

**Definition 7 (CRF).** The calculation function of task correlation factor, CRF is defined as follows:

$$CRF(t_i, t_j) = MRF(t_i, t_j) + W_f \times FRF(t_i, t_j).$$

Here, the weight value $W_f$ of FRF represents the message size when the fork or join operation occurs. It can be assigned with any value, but should be higher than the average value of MRF. This is to distinguish concurrent tasks from other tasks.
4. Allocation algorithm

4.1. Algorithm overview

In this section, we define the allocation function $L_m$. The input to this function are a set of tasks and their types, a set of target hardware resources and their types, and the CRF values. Fig. 9 shows the overview of our task allocation process. The allocation function $L_m$ fetches a candidate task to be allocated and a hardware resource to allocate. When fetching it, through the corresponding type set as shown in Fig. 9, it is guaranteed that the type of the hardware resource corresponds to that of the task. For example, the corresponding type of the hardware resource for ‘Control’ typed task is ‘µP’ (microprocessor). Then, the function allocates the task to the hardware resource considering the CRF values.

4.2. Variable for hardware architecture description

MPSoC architecture may contain a number of hardware resources in homogeneous type – for example, 4, 8, 16, or 32 ARM926EJ-S processors. Also some of the processors may not have local memory. Since MPSoC architecture is described with such multi-dimensional information as the component type, the number of the components, and their property. We define a structure to represent the MPSoC architectural information as shown in Fig. 10. Fig. 10 shows the representation of an example MPSoC architecture.

MPSoC architecture has two DSPs: the first one has its local memory, and the second one does not.

4.3. Decision table for allocation

We classify the basic configuration for general MPSoC architecture into the following eight categories:

1. In case that the number of µP is just 1.
2. In case that the number of µP is over than 1.
3. In case that the number of DSP is just 1.
4. In case that the number of DSP is over than 1.
5. In case that there exists local memory for µP.
6. In case that there exists local memory for DSP.
7. In case that there exists shared memory
8. In case that there exist some kinds of I/O device.

4.4. Algorithm of mapping actions

After selecting a row in the decision table, the allocation function $L_m$ processes the mapping actions, $x_4$, $x_5$, and $x_8$ in Action field of Table 1. The behaviors of these actions, $x_4$, $x_5$, and $x_8$, are defined in the next section.
of the algorithm of mapping actions in Table 1. We also define three subfunctions – <<all1>>, <<cr>>, and <<rr>>. The functionalities of them are explained in the upper part of the algorithm in Fig. 11. Basically, our algorithm allocates tasks of control type (CL) to μPs and those of data-intensive processing (DP) type to DSPs. If there is only one μP in hardware architecture, all CL-typed tasks are allocated to it by performing action \( z_0 \). If there are several μPs in the architecture, all CL-typed tasks are allocated into those μPs by round-robin approach within action \( z_1 \). In the allocation of DP-typed tasks to DSPs, the same approach is applied by performing \( z_2 \) and \( z_3 \). If the hardware architecture has only one μP and no DSP, all CL-typed and DP-typed tasks are allocated to it by performing action \( z_4 \). For the tasks of entity (EN) type, they are allocated to the local memories of μPs or DSPs. The way of allocation is the same as that of CL-typed and DP-typed tasks to μPs or DSPs, which is performed by \( z_5 \) and \( z_6 \). If there is no local memory, all EN-typed tasks are allocated to the shared memory by \( z_7 \). The action \( z_8 \) allocates tasks of interface type to the corresponding I/O devices. The round-robin allocation process, in \( z_1 \), \( z_2 \), and \( z_3 \), uses both of the configuration information of hardware architecture and the CRF values among tasks.

In the implementation of our algorithm, a task which is selected from the set of tasks is allocated to any hardware resource and then eliminated from the set. Therefore no task is allocated twice to the resource. Also, the actions which deal with the same type of tasks – for example, action \( z_3 \) and \( z_4 \), or action \( z_2 \) and \( z_3 \) – are not performed together in the allocation function \( L_{\text{as}} \), as shown in Table 1. They are mutually exclusive.

## 5. Experiment and analysis

### 5.1. Experimental setting

We implemented the helper functions defined in Section 3.2.3 in an integrated tool, ESMUL (ESMUL, 2007). The tool supports the drawings of the source model in our approach, three UML diagrams and the hardware architecture diagram. It also supports the simulation of the software model, UML diagrams, based on the hardware architecture. Using the tool, we model a target system to apply our mapping process and techniques. The functionality of the target system is to read and encrypt source data, then send the encrypted data through network. For the encryption, we adopt the FIPS-197 AES (Advanced Encryption Standard) algorithm which was published by NIST (2001). We modeled the static structure of the system as shown in Fig. 12. The behavior of the system is also modeled. Fig. 13a shows the top-level behavior of the system, which is depicted with an IOD. The detailed behavior of the ‘Encrypting’ node in Fig. 13a is modeled using another IOD, as

![Fig. 12. The static structure of the target system.](image-url)
shown in Fig. 13b. Again, the detailed behavior of each node in Fig. 13b is modeled using an SD.

5. Results

5.2. Task identification

Using the tool, we generated the CFGs for the model and identified final tasks as shown in Fig. 14. The IDs of tasks whose types are «control», «data-intensive processing» and «interface» are notated with the prefix of ‘AT’, and the prefix of ‘ET’ for the case of the «entity» tasks. We use the prefix ‘ET’ to distinguish the task type to other three task types. In Fig. 14, the task AT0: fileController.0, AT1: fileController.1, and AT2: fileController.2 are parallel tasks which are split from the fileController task.

5.2.2. Calculation of CRF values

Fig. 15 shows the CRF values of all pairs of the final tasks that we obtained through the CRF calculation function. In the calculation, the weight value \( W_f \) of Definition 7 is assumed to 100. In our target system, the tasks whose types are either control or data-intensive processing do not send messages to each other. Instead, they communicate to each other through the entity tasks. Thus the MRF value between them is 0. This means that we can assign any positive value to the weight value \( W_f \). It could be 1 or 2 or other positive values. In this experiment, we simply assign 100 to \( W_f \).

As shown in Fig. 15, the CRF value between the entity ET5 and the task AT6 is 214,720. And the entity ET5 has zero CRF value with other tasks except AT6, which means no interactions. Thus it will be allocated to the local memory of the processor to where the task AT6 was allocated. In case of the entity ET2, its CRF value with the task AT6 and AT7 is 8,158,040 and 3,430,000, respectively. Since the entity ET2 has various associations with several tasks, it should be allocated at shared memory to be accessible from those tasks.

5.2.3. Evaluation of task allocation

After calculating the CRF values, we mapped the tasks to hardware resources. We consider the following three cases:

- **Case 1** The MPSoC architecture is configured with 4 μPs, 2 DSPs, and 2 shared memories, and the result of task mapping is shown in Table 2 which is obtained from the application of our mapping algorithm.
- **Case 2** The hardware configuration is the same as the Case 1. However, in this case, the type matching between tasks and hardware resources is not considered.
- **Case 3** The hardware consists of just 1 μP and 1 DSP.
We estimated the elapsed time to perform simulations of the three cases. For simplicity, we tried our simulation with reduction of the number of iterations to encrypt the source data; the condition clause cipheround <10 in Fig. 13b was modified with cipheround <3. We also construct a graph for each case representing the status of resource occupation of each hardware resource in simulation. Initially, we estimated only the total elapsed time. However, due to the broad time range, the corresponding graphs look similar to each other, which makes us hard to analyze the experiments. So, we additionally estimated the elapsed time to a certain break point; until the target system encrypts source data, and writes the encrypted data into buffer for 1/5 portion of whole source data. The simulation results are shown in Table 3.

Fig. 15 shows the graphs for the right-side column of Table 3. Fig. 16a shows the graph of Case 1, which represents high parallelism of tasks and data processing and fast response time. The graph of Case 2, Fig. 16b, also shows high parallel processing of tasks. The elapsed time of Case 2 is 1.7 times of that of Case 1 until the predefined break point. This is because the tasks of data-intensive processing type are allocated to $ \mu $P, thus it requires more time to input/output processing of data. Fig. 16c shows the case that all tasks are allocated to 1 $ \mu $P and 1 DSP. In this case, the utilization of hardware resources is approximately 100%, but the elapsed time until the break point is 4.6 times of that of Case 1. As shown in Table 3, the total elapsed time of Case 3 is about 2 times of that of Case 1. This means that the behavior of the target system is highly dependent on the number of DSPs because the system contains huge data processing logic.

5.3. Discussion

From our experimental observation, we confirm that the task allocation with considering the task types gives high parallelism and fast response time of task and data processing. Also the allocation technique provides high resource utilization for MPSoC architecture. In existing works on task allocation, the designer is required to model the software tasks including the communication costs between them. Our approach does not require the communication costs. Instead, we calculate them as the form of CRF values based on the behavior model. This helps the designer reduce the burdens of specifying the communication costs between tasks.

Although the experiments above validates our approach, we still have several limitations. The first one is the insufficient con-
sideration of hardware resources. Our mapping algorithm mainly focuses on the types of software tasks and hardware resources. More information on hardware resources, such as the communication costs between processors and the bandwidth of a bus, is not considered. Our future works include the consideration of such information in our mapping algorithm. Secondly, since our mapping algorithm is a heuristic one, it does not produce the optimal allocation of tasks. The allocation in our algorithm can be used as a guide to the designer. For the optimization, the manual adjustment by the designer will be needed.

Although our research focuses on the allocation of software tasks considering parallelism between them, the schedulability of them is also an important issue to consider. One way to analyze the schedulability of software tasks in the model-level is to annotate the information regarding it in the model and apply schedulability analysis techniques. Instead of annotating such information in the source model, we consider the scheduling of software tasks in our simulation environment. In the simulation, each virtual processor schedules the allocated tasks based on its scheduling policy. The designer can choose the scheduling policy of each processor.

6. Conclusion

This paper describes an approach to develop PSM of MPSoC-based embedded software from UML 2.0-based PIM. The PSM includes the hardware architecture and its configuration, and an allocation instance which represents the allocations of tasks to hardware resources. In the PSM, we focus on the allocation instance. In order to develop the allocation instance, we define the transformation system and its micro-level process as shown in Fig. 1, and define and implement three helper functions to support the process: (1) CFG generation function to identify tasks from PIM, (2) CRF calculation to determine the interrelationship between tasks, and (3) algorithm to allocate the tasks to hardware resources.

Compared with existing works, the first characteristic of our work is the consideration of the precise features for tasks and hardware resources within the mapping process. We considered this due to the fact that our target environment is an MPSoC architecture rather than a distributed computing environment or network. The second is that our work is model-based. The input of our transformation system is not code or executables but UML model. This gives the benefits of the effective design of MPSoC-based embedded software in early development phases. The feedback by design-mistake in code-level allocation may lead to significant reworks and cost-overrun. The last is the consideration of data parallelism as well as task parallelism. If a task is divided into several subtasks by inherent parallelism, independent data processing will be possible for each subtask. The example of this parallelism is similar to ‘par’ pragma in OpenMP program (Mohr et al., 2001).

Our proposed techniques are greatly required to develop MPSoC-based embedded applications in various areas such as advanced missile control system, mobile communication and multimedia devices, satellite system, avionics system, and so on. Applying our suggestions to those application areas will enable the fast and reliable design of embedded software, and reduction of the redesign of hardware prototype board.

Fig. 16. Resource occupation graph for each case in Table 3.
There are some works to mature our approach. In the allocation process, we only consider the type of tasks and hardware resources. Considering detailed information of hardware resources, such as bus bandwidth, will be helpful to elaborate our transformation system. As we stated in Section 3.2.3, the approximation we performed to decide the threshold in the task identification function is dependent on a specific program language on a specific platform. We will improve the approximation method with more generality. Also we plan to elaborate our allocation technique considering other non-functional requirements such as memory usage and energy consumption. Finally, we will conduct additional experiments with more complex system. The experiments will be helpful to elaborate and confirm the scalability of our approach.

Acknowledgements

This research was supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Institute of Information Technology Advancement) (IITA-2009-C1090-0902-0032) and was partially supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract.

References


