A Retargetable Parallel Programming Framework for MPSoC

SEONGNAM KWON, YONGJOO KIM, WOO-CHUL JEUN, SOONHOI HA, AND YUNHEUNG PAEK
Seoul National University

As more processing elements are integrated in a single chip, embedded software design becomes more challenging: It becomes a parallel programming for non-trivial heterogeneous multi-processors with diverse communication architectures and design constraints such as hardware cost, power, and timeliness. In the current practice of parallel programming with MPI or OpenMP, the programmer should manually optimize the parallel code for each target architecture and design constraints. Thus design space exploration of MPSoC (Multi-Processor System-on-a-Chip) costs prohibitively large as software development overhead increases drastically. To solve this problem, we develop a parallel programming framework based on a novel programming model called Common Intermediate Code (CIC). In a CIC, functional parallelism and data parallelism of application tasks are specified independently of the target architecture and the design constraints. Then, the CIC translator translates the CIC into the final parallel code considering the target architecture and the design constraints, to make the CIC retargetable. Experiments with preliminary examples, including H.263 decoder, show that the proposed parallel programming framework increases the design productivity of MPSoC software significantly.

Categories and Subject Descriptors: C.3 [Computer Systems Organization]: Special Purpose and Application-based Systems; D.2.2 [Software]: Software Engineering – Design Tools and Techniques, Computer-aided software engineering (CASE)

General Terms: Design, Experimentation

Additional Key Words and Phrases: embedded software, multi-processor system on a chip, software generation, design space exploration, parallel programming

1. INTRODUCTION

To meet increasingly insatiable demand of system performance, a system with multiple processing elements integrated in a single chip, called MPSoC (Multi-Processor System on a Chip), becomes the norm as semiconductor technology improvement continues. While extensive research has been performed on the design methodology of SoC, most efforts have focused on the design of hardware architecture. But the real bottleneck of MPSoC design will be software design as pre-verified hardware platforms tend to be reused in platform based design. Embedded software design for MPSoC is very challenging since it is a parallel programming for non-trivial heterogeneous multi-

This research was supported by BK21 project, SystemIC 2010 project funded by Korean MOCIE, and Acceleration Research sponsored by KOSEF research program(R17-2007-086-01001-0). This work was also partly sponsored by ETRI SoC Industry Promotion Center, Human Resource Development Project for IT-SoC Architect. The ICT and ISRC at Seoul National University and IDEC provide research facilities for this study. Authors' addresses: S. Kwon, W-C. Jeun, S. Ha (contact author). The Codesign and Parallel processing Laboratory, School of Electrical Engineering and Computer Sciences, Seoul National University, Shinlim-dong, Gwanak-gu, Seoul, 151-744, Korea; Y. Kim, Y. Paek. The Software Optimizations & Restructuring Group, School of Electrical Engineering and Computer Sciences, Seoul National University, Shinlim-dong, Gwanak-gu, Seoul, 151-744, Korea; email: sha@iris.snu.ac.kr
processors with diverse communication architectures and design constraints such as hardware cost, power, and timeliness.

Two major models for parallel programming are the message-passing and the shared address space model. In the message-passing model, each processor has private memory and communicates with other processors via message passing. To obtain high performance, the programmer should optimize data distribution and data movement carefully, which is very difficult task. The Message Passing Interface (MPI)[1994] is a de facto standard interface of this model. In the shared address space model, all processors share a memory and communicate data through the shared memory. The OpenMP[1998] is a de facto standard interface of this model. It is mainly used for a symmetric multi-processor (SMP) machine. Because it is easy to write a parallel program, there are some works such as Sato, et al.[1999], Liu, et al.[2003], Hotta, et al.[2004], and Jeun, et al.[2007] to use the OpenMP as a parallel programming model on other parallel-processing platforms without shared address space such as System-On-Chips and clusters.

In the current practice of parallel programming, the programmer should manually optimize the parallel code considering the specific target architecture and design constraints. If task partition or the communication architecture is changed, significant coding effort is needed to rewrite the optimized code. While an MPI or an OpenMP program is regarded retargetable with respect to the number of processors and processor kinds, we consider that it is NOT retargetable with respect to task partition and architecture change. Another difficulty of programming with MPI and OpenMP is to satisfy the design constraints such as memory requirements and real-time constraints. It is the programmer’s responsibility to confirm satisfaction of the design constraints in the manually designed code. Thus design space exploration of MPSoC (Multi-Processor System-on-a-Chip) costs prohibitively large as software development overhead increases drastically.

In order to increase the design productivity of embedded software for MPSoC, we develop a parallel programming framework based on a novel programming model, called Common Intermediate Code (CIC). In a CIC, the functional parallelism and data parallelism of application tasks are specified independently of the target architecture and the design constraints. Information on the target architecture and the design constraints is separately described in an xml-style file, called architecture information file. Based on this information, the programmer maps the tasks to the processing components, manually or automatically. Then, the CIC translator automatically translates the task codes in the CIC model into the final parallel code following the partitioning decision. If a new
partitioning decision is made, the programmer needs not modify the task codes but the partitioning information only: Then the CIC translator generates the newly optimized code from the modified architecture information file.

The main contributions of this paper can be summarized as follow: First, we propose a novel parallel programming model that is truly retargetable with respect to architecture change and partitioning decision. Second, the CIC translator alleviates the programmer’s burden to optimize the code for the target architecture. It enables fast design space exploration of MPSoC by reducing the re-programming overhead significantly. Thus we increase the design productivity of parallel embedded software for MPSoC.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 shows the workflow of proposed MPSoC software development methodology. Section 4 explains the proposed programming model, CIC, with its formats and properties. The CIC translator will be explained in Section 5. In Section 6, experiments with preliminary examples, including H.263 decoder, show that the proposed parallel programming model increases the design productivity of MPSoC software significantly. Section 7 concludes this paper.

2. RELATED WORK

Martin [2006] emphasized the importance of parallel programming model for MPSoC to overcome the difficulty of concurrent programming. Conventional MPI or OpenMP programming is not adequate for MPSoC design since the program should be made target specific, message passing or shared address space architecture. To be suitable for design space exploration, a programming model needs to accommodate both styles of architecture. Recently Paulin, et al. [2004] proposed the MultiFlex multi-processor SoC programming environment where two parallel programming models are supported: Distributed System Object Component (DSOC) and Symmetric multi-processing (SMP) models. The DSOC is a message-passing model that supports heterogeneous distributed computing while the SMP supports concurrent threads accessing the shared memory. But it is still the burden of programmer to consider the target architecture when programming the application. Thus it is not fully retargetable. On the other hand, we propose a fully retargetable programming model.

To be retargetable, interface code between tasks should be automatically generated after partitioning decision on the target architecture is made. Since the interfacing between processing units is one of the most important factors that affect the performance of the system, some research has focused on the interfacing between processing units.
Wolf, et al.[2004] defined task transaction level(TTL) interface for integrating HW-SW components. In the logical model for TTL inter-task communication, a task is connected to a channel via a port, and communicates with other tasks through channels by transferring tokens. In this model, tasks call target-independent TTL interface functions on their ports to communicate with other tasks. If the TTL interface functions are defined optimally for each target architecture, the program becomes retargetable. This approach can be integrated in the proposed framework.

For retargetable interface code generation, Jerraya, et al. [2006] proposed a parallel programming model to abstract both HW and SW interfaces. They defined three layers of SW architecture: hardware abstraction layer (HAL), hardware dependent software (HdS), and multi-threaded application. To interface between software and hardware, translation to APIs of different abstraction models should be performed. This work is complementary to our work.

Compared with related work, the proposed approach has the following characteristics that make it more suitable for MPSoC architecture.

1) We especially focus the retargetability of software development framework, and suggest CIC as a parallel programming model. The main idea of CIC is the separation of algorithm specification and its implementation. CIC consists of two sections: task codes and architecture information file. An application programmer writes task codes considering the potential parallelism of the application itself independently of the target architecture. Based on the target architecture, we determine which potential parallelism will be realized in implementation.

2) We use different ways of specifying functional and data parallelism (or loop parallelism). Data parallelism is usually implemented by an array of homogeneous processors or a hardware accelerator, differently from functional parallelism. By considering different implementation practices, we use different specification and optimization methods for functional and data parallelism.

3) Also, we explicitly specify the potential use of hardware accelerator inside a task code using #pragma definition. If use of hardware accelerator is decided after design space exploration, the task code will be modified by a pre-processor. On the other hand, most existent programming models do not consider the use of hardware accelerator so that re-writing the code is necessary if it is decided.

3. PROPOSED WORKFLOW OF MPSOC SOFTWARE DEVELOPMENT
The proposed workflow of MPSoC software development is depicted in Figure 1. The first step is to specify the application tasks with the proposed parallel programming model, CIC. As shown in Figure 1, there are two ways of generating a CIC program: One is to manually write the CIC program, which is assumed in this paper. The other is to generate the CIC program from an initial model-based specification such as dataflow model or UML. Recently, it becomes more popular to use a model driven architecture (MDA) for systematic design of software (Balasubramanian, et al.[2006]). In an MDA, system behavior is described in a platform independent model (PIM). The PIM is translated to a platform specific model (PSM) from which the target software on each processor is generated. MDA methodology is expected to improve the design productivity of embedded software since it increases the reuse possibility of platform independent software modules: The same PIM can be reused for different target architectures.

![Diagram of software generation from CIC]

Unlike other model driven architectures, the unique feature of the proposed methodology is to allow multiple PIMs in the programming framework. We define an intermediate programming model common to all PIMs including the manual design, so we name it as Common Intermediate Code (CIC). The CIC is independent of the target architecture so that we may explore the design space at the later stage of design. The CIC program consists of two sections, task code section and architecture section, which will be explained in detail in the next section.

The next step is to map the task codes to the processing components, manually or automatically. Optimal mapping problem is beyond the scope of this paper, so we assume that mapping is somehow given in this paper. We are now developing an optimal mapping technique based on a genetic algorithm, considering three kinds of parallelism at the same time: functional parallelism, data (loop) parallelism, and temporal parallelism.
The last step is to translate the CIC program into the target executable C codes based on the mapping and architecture information. In case more than one task codes are mapped to the same processor, the CIC translator should generate the run-time kernel that schedules the mapped tasks, or let the OS schedule the mapped tasks to satisfy the real-time constraints of the tasks. The CIC translator also synthesizes the interface codes between processing components optimally for the given communication architecture.

4. COMMON INTERMEDIATE CODE

The heart of the proposed workflow of MPSoC software is the CIC parallel programming model that separates algorithm specification and architecture information. Figure 2 (a) displays the CIC format that consists of two sections that are explained in this section.

4.1 Task Code

The “Task Code” section contains the definitions of tasks that will be mapped to processing components as a unit. An application is partitioned into tasks that represent the potential temporal and functional parallelism. Data parallelism or loop parallelism is defined inside a task. It is the programmer’s decision how to define the tasks: As the granularity of a task is finer, it will give more chance of optimal exploitation of pipelining and functional parallelism with the cost of programmer’s burden. An intuitive solution is to define a task as reusable for other applications. Such tradeoff should be considered if a CIC is automatically generated from a model based specification.

Figure 2 (c) shows the example of a task code file (.cic file) that defines a task in C. A task should define three functions: `{task name}_init()`, `{task name}_go()`, and `{task name}_wrapup()`. The `{task name}_init()` function is called once when the task is invoked to initialize the task. The `{task name}_go()` function defines the main body of the task and
is executed repeatedly in the main scheduling loop. The \{task_name\}_wrapup() function is called before stopping the task to reclaim the used resources.

The default inter-task communication between tasks, specially for streaming applications, is depicted in Figure 2 (b): A task is connected to channels via port, and communicate with other tasks via send/receive APIs as shown at line 4 of Figure 2 (c). The CIC also supports other communication APIs such as shared memory accesses. The communication channel is properly created by the CIC translator as specified in the architecture information file, which will be explained later.

![Figure 3 Task specification example: H.263 decoder](image)

An example is shown in Figure 3 where an H.263 decoder algorithm is partitioned into six tasks. In this figure, macroblock decoding task contains three functions: Dequantize, Inverse zigzag, and IDCT. These three functions will not be mapped to separate processors if they are not specified as separate tasks in the CIC. Note that data parallelism is specified with OpenMP directives within a task code as shown at line 9 of Figure 2 (c).

For target-independent specification, the CIC uses generic APIs: For instance, two generic APIs are shown in Figure 2 (c) (line 4 and 5). The CIC translator translates the generic API with the appropriate implementations depending on whether an OS is used or not. By doing so, the same task code will be reused despite architecture variation.

If there are HW accelerators in the target-platform, we may want to use them to improve the performance. To open that possibility in a task code, we define a special pragma to identify the code section that can be mapped to the HW accelerator as shown in line 6 of Figure 2 (c). And information on how to interface with the HW accelerator is specified in architecture information file. Then, the code segment wrapped with pragma will be replaced with the appropriate HW interfacing code by the CIC translator.

### 4.2 Architecture Information File

The target architecture and the design constraints are separately specified from the task code in the architecture information section. The architecture section is further divided
into three sections in an xml-style file as shown in Figure 4. The “hardware” section contains the hardware architecture information that is necessary to translate the target-independent task codes to the target-dependent codes. The “constraints” section specifies user-specified constraints such as the real time constraints, resource limitation, and energy constraints. The “structure” section describes the communication and synchronization requirements between tasks.

The hardware section defines the processor id, address range and the size of each memory segment, the use of OS, and task scheduling policy for each processor. For shared memory segments, it indicates which processors share the segment. It also defines information of hardware accelerators, which includes architectural parameters and translation library of HW interfacing code.

The constraints section defines the global constraints such as power consumption and memory requirement as well as per-task constraints such as period, deadline, and priority. And it also includes the execution time of the tasks. Using these information, we will determine the scheduling policies of the target OS or synthesize the run-time system for the processor without OS.

In the structure section, task structure and task dependency are specified. An application task usually consists of multiple tasks that are defined separately in the task code section of the CIC. The task structure of an application task is represented by communication channels between the tasks. Currently two methods of task communication are supported: message queue and shared memory.
For each task, the structure section defines the file name (with “.cic” suffix) of the task code, and its compile options needed for compilation. And each task has the index field of the processor that the task is mapped to. This field is updated after task mapping decision is made: In other words, task mapping can be changed without modifying the task code but by changing the processor mapping id of each task.

5. CIC TRANSLATOR

The CIC translator translates the CIC program into the optimized executable C codes for each processor core. As shown in Figure 5, the CIC translation consists of four main steps: generic API translation, HW interface code generation, OpenMP translation if needed, and task scheduling code generation. From the architecture information file, the CIC translator extracts the necessary information needed for each translation step. Based on the task dependency information that tells how to connect the tasks, the translator determines the number of inter-task communication channels. Based on the period and deadline information of tasks, run-time system is synthesized. With the memory map information of each processor, the translator defines the shared variables in the shared region.

To support a new target architecture in the proposed workflow, we have to add translation rules of generic API to translator, port the subset of MPI libraries used in the OpenMP-translated codes, and generation rule of task scheduling codes tailored for the target OS. Each step of CIC translator will be explained in this section.

5.1 GENERIC API TRANSLATION

Since the CIC task code uses generic APIs for target-independent specification, translation of generic APIs to target-dependent APIs is needed. If the target processor has an OS installed, generic APIs are translated into OS APIs. Otherwise they are translated
into communication APIs that are defined directly accessing the hardware devices. We implement the OS API library and communication API library optimized for each target architecture.

```c
#include <stdio.h>
... file = fopen("input.dat", "r"); ... fread(data, 1, 100, file); ... close(file);
```

(a)

```c
#include <stdlib.h>
... file = OPEN("input.dat", O_RDONLY); ...
READ(file, data, 100); ...
CLOSE(file);
```

(b)

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
... file = open("input.dat", O_RDONLY);
... read(file, data, 100);
... close(file);
```

(c)

Figure 6 Generic API translation: (a) An example of generic API translation, (b) the translation rule of READ API for POSIX, and (c) the translation rule of READ API for Linux system call.

The inputs to the translator are a CIC code, pattern information and parameters for each generic API, and the file that describes the translation rule. The pattern of an API depicts the typical usage of the API in the code. Figure 6 (a) shows an example of using generic APIs in a task code for file access: OPEN, READ, and CLOSE. The figure illustrates two possible translations depending on the target platform. Figure 6 (b) and (c) show the translation rules of READ API for POSIX and Linux system call respectively. The translation rule consists of three parts: initialize part(line 1-4), transform part(line 5-8), and close part(not shown in Figure 6 (b) and (c)). In the “initialize” part, include files, variable declaration, and variable initialization for this API can be specified. In the “transform” part, direct translation rule of each API is specified. Finally, in the “close” part, closing instructions for API can be specified. Each translation rule is stored in a separate rule file for each API. These translation rules are listed in the pattern list file for each target. For example, if there are two translation target: POSIX and Linux system call, there are two pattern list files. In each pattern list file, translation rules and rule files for generic APIs are listed. Maeng, et al.[2006] explain detailed information about generic API translator.

5.2 HW INTERFACE CODE GENERATION
If there is a code segment wrapped with a HW pragma, and its translation rule exists in an architecture information file, the CIC translator replaces the code segment with the HW interfacing code considering the parameters of HW accelerator and buffer variables that are defined in the architecture section of the CIC. The translation rule of HW interfacing code for a specific HW is separately specified as a HW interface library code.

Note that some HW accelerators work together with other HW IPs. For example, a HW accelerator may notify the processor of its completion through interrupt. Then an interrupt controller is needed. The CIC translator generates a combination of the HW accelerator and the interrupt controller as will be shown in the next section.

5.3 OPENMP TRANSLATION

If an OpenMP compiler is available for the target, task codes with OpenMP directives can be used easily. Otherwise we need to translate the task code with OpenMP directives to the parallel code somehow. In the current implementation, we translate it to the MPI codes using a minimal subset of MPI library due to the following reasons: (1) MPI is a standard to be easily ported to various software platforms. (2) Porting the MPI library is much easier than modifying the OpenMP translator itself for the new target architecture. Figure 7 shows the structure of the translated MPI program.

As shown in the figure, the translated code has the master-worker structure: the master processor executes the entire core while worker processors execute the parallel region only. When the master processor meets the parallel region, it broadcasts the shared data to worker processors. Then all processors execute the parallel region concurrently. The master processor synchronizes all the processors at the end of the parallel loop and collects the results from the worker processors. For performance optimization, we have to
minimize the amount of inter-processor communication between processors. We have implemented some optimization techniques that are omitted due to space limitation.

5.4 SCHEDULING CODE GENERATION

The last step of the proposed CIC translator is to generate the task scheduling code for each processor core. There will be many tasks mapped to each processor with different real-time constraints and dependency information. And remind that a task code is defined by three functions: `{task name}_init()`, `{task name}_go()`, and `{task name}_wrapup()`. The generated scheduling code initializes the mapped tasks by calling `{task name}_init()` and wraps up the tasks after scheduling loop finishes its execution by calling `{task name}_wrapup()`.

```
typedef struct {
  void (*init)();
  int (*go)();
  void (*wrapup)();
  int period, priority, …;
} task;

task taskInfo[] = {{task1_init, task1_go, task1_wrapup, 100, 0}, {task2_init, task2_go, task2_wrapup, 200, 0}};

void scheduler() {
  while(all_task_done()==FALSE) {
    int taskId = get_next_task();
    taskInfo[taskId]->go();
  }
}

int main() {
  init();            /* {task_name}_init() functions are called */
  scheduler(); /* scheduler code */
  wrapup();     /* {task_name}_wrapup() functions are called */
  return 0;
}
```

(a) void * thread_task_0_func(void *argv) {
  …
  task_0_go();
  get_time(&time);
  sleep(task_0->next_period – time);  // sleep for remained time
  …
  }

int main() {
  …
  pthread_t thread_task_0;
  sched_param thread_task_0_param;
  …
  thread_task_0_param.sched_priority = 0;
  pthread_setschedparam(…, &thread_task_0_param);
  …
  task_init();  /* {task_name}_init() functions are called */
  pthread_create(&thread_task_0, 
  &thread_task_0_attr, thread_task_0_func, NULL);
  …
  task_wrapup();  /* {task_name}_wrapup() functions are called */
  }

(b) Figure 8 Pseudo code of generated scheduling code: (a) if OS is available and (b) if OS is not available

The main body of the scheduling code differs depending on whether there is an OS available for the target processor. If there is an OS that is POSIX-compliant, we generate a thread-based scheduling code as shown in Figure 8 (a). A POSIX-thread is created for each task (line 17-18) with an assigned priority level if available. The thread, as shown in lines 3-5, executes the main body of the task, `{task name}_go()`, and schedules the thread itself based on its timing constraints by calling `sleep()` method. If the OS is not POSIX-compliant, the CIC translator should be extended to generate the OS-specific scheduling code.

If there is no available OS for the target processor, the translator should synthesize the run-time scheduler that schedules the mapped tasks. The CIC translator generates a data structure of each task containing three main functions of tasks(`init()`, `go()`, and `wrapup()`).

With this data structure, a real time scheduler is synthesized by the CIC translator. Figure
8 (b) shows the pseudo code of a generated scheduling code. Generated scheduling code may be changed by replacing the function “void scheduler()” or “int get_next_task()” to support another scheduling algorithm.

6. PRELIMINARY EXPERIMENTS

To verify the viability of the proposed programming, we built a virtual prototyping system that consists of multiple sub-systems of arm926ej-s connected to each other through a shared bus as shown in Figure 9. H.263 Decoder as depicted in Figure 3 is used for preliminary experiments.

Figure 9 The target architecture for preliminary experiments

6.1 DESIGN SPACE EXPLORATION

We specified the functional parallelism of the H.263 decoder with six tasks as illustrated in Figure 3 where each task is assigned an index. For data-parallelism, the data-parallel region of motion compensation task is specified with an OpenMP directive. In this experiment, we explored the design space of parallelizing the algorithm considering both functional and data parallelism simultaneously. As evident in Figure 3, tasks 1 to 3 can be executed in parallel. So, they are mapped to multiple-processors with three configurations as shown in Table 1 (a). For example, task 1 is mapped to processor 1, and the other tasks are mapped to processor 0 for the second configuration.

Table 1 (a) task mapping to processors and (b) execution cycles for nine configurations

<table>
<thead>
<tr>
<th>Processor Id</th>
<th>The configuration of task mapping</th>
<th>The number of processors for data-parallelism</th>
<th>The configuration of task mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Task 0, Task 1, Task 2, Task 3, Task 4, Task 5</td>
<td>155,415,942, 154,159,995, 168,640,527</td>
<td>N/A, Task 1</td>
</tr>
<tr>
<td>1</td>
<td>Task 1</td>
<td>155,415,942, 154,159,995, 168,640,527</td>
<td>N/A, Task 2</td>
</tr>
<tr>
<td>2</td>
<td>N/A</td>
<td>155,415,942, 154,159,995, 168,640,527</td>
<td>N/A, Task 2</td>
</tr>
</tbody>
</table>

(a) (b)
For each configuration of task mapping, we parallelized task 4 using one, two and four processors. As a result, we have prepared nine configurations in total as illustrated in Table 1 (b). In the proposed framework, each configuration is simply specified by changing the task mapping information in the architecture information file. And the CIC translator generates the executable C codes automatically.

Table 1 (b) shows the performance result for these nine configurations. For functional parallelism, the best performance can be obtained by using two processors. H.263 decoder algorithm uses 4:1:1 format frame, so computation of Y macro block decoding is about four times larger than that of U and V macro blocks. Therefore macro block decoding of U and V can be merged in one processor during macro block decoding of Y in another processor. For data parallelism, there is no performance gain obtained by exploiting data parallelism. It is because the computation workload of motion compensation is not large enough to outweigh the communication overhead that is incurred by parallel execution.

6.2 HW INTERFACING CODE GENERATION

Next, we accelerated the code segment of IDCT in the macroblock decoding tasks(task 1 to task 3) with a HW accelerator as shown in Figure 10 (a). We use the RealView SoC Designer to model the entire system including the HW accelerator. Two kinds of IDCT accelerator are used. One uses an interrupt signal for completion notification, and other uses polling to detect the completion. The latter is specified in the architecture section as illustrated in Figure 10 (b), where the library name of HW interfacing code is set to IDCT_slave and its base address to 0x2F00000.

Figure 10 (a) Code segment wrapped with a HW pragma and architecture section information of IDCT (a) when interrupt is not used, and (c) when interrupt is used.

Figure 11 (a) shows the assigned address map of IDCT accelerator and Figure 11 (b) shows the generated HW interfacing code. This code is substituted for the code segment

```c
#pragma hardware IDCT(output.data, input.data) {
    /* code segments for IDCT */
}
```

Figure 10 (a) Code segment wrapped with a HW pragma and architecture section information of IDCT (a) when interrupt is not used, and (c) when interrupt is used.
wrapped HW pragmas. In Figure 11 (b), bold letters are changeable according to the parameters specified in a task code and the architecture information file.

<table>
<thead>
<tr>
<th>Address (Offset)</th>
<th>I/O</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read</td>
<td>Semaphore</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Write</td>
<td>IDCT start</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Read</td>
<td>Complete flag</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Write</td>
<td>IDCT clear</td>
<td></td>
</tr>
<tr>
<td>64-191</td>
<td>Write</td>
<td>Input data</td>
<td></td>
</tr>
<tr>
<td>192-319</td>
<td>Read</td>
<td>Output data</td>
<td></td>
</tr>
</tbody>
</table>

(a)            (b)

Figure 11 (a) The address map of IDCT and (b) its generated interfacing code

Note that interfacing code uses polling at line 6 of Figure 11 (b). If we use the accelerator with interrupt, and an interrupt controller is additionally attached to the target platform as shown in Figure 10 (c) with information on the code library name, *irq_controller*, and its base address, 0x4A801000. New IDCT accelerator has the same address map of previous one except the complete flag. The address of complete flag(address 8 of Figure 11 (a)) is assigned to “interrupt clear”.

Figure 12 (a) shows the generated interfacing code for the IDCT with interrupt. Note that the interfacing code does not access the HW to check the completion of IDCT, but checks the variable “complete.” In the generated code of interrupt handler, this variable is set to “1” (Figure 12 (b)). Initialize code for interrupt controller (“initDevices()”) is also generated and called in `{task_name}_init()` function.

6.3 SCHEDULING CODE GENERATION

We generated the task scheduling code of H.263 decoder changing the working condition, OS support and scheduling policy. At first, we used the eCos real-time OS for arm926ej-s in RealView SoC Designer, and generated the scheduling code, the pseudo code of which is shown in Figure 13. In function “cyg_user_start()” of eCos, each task is created as a thread. CIC translator generates the parameters needed for thread creation such as stack...
variable information and stack size (fifth and sixth parameter of cyg_thread_create()).
And, \{task\}_go is placed in a while loop inside the created thread (lines 10-14 of
Figure 13). Functions \{task\}_init() is called in “init_task()”.

Note that “TE_main()” is also created as a thread. “TE_main()” checks whether
execution of all tasks is finished, and calls \{task\}_wrapup() in “wrapup_task()”
before finishing the entire program.

```
1. void cyg_user_start(void) {
2.   cyg_thread_create(taskInfo[0]->priority, TE_task_0,
3.       (cyg_addrword_t)0, "TE_task_0", (void*)&TaskStk[0],
4.       TASK_STK_SIZE-1, &handler[0], &thread[0]);
5.   …
6.   init_task();
7.   cyg_thread_resume(handle[0]);
8.   …
9. }
```

```
10. void TE_task_0(cyg_addrword_t data) {
11.   while(!finished)
12.     if (this task is executable) taskInfo[0]->go();
13.     else cyg_thread_yield();
14. }
```

```
15. void TE_main(cyg_addrword_t data) {
16.   while(1)
17.     if (all_task_is_done()) {
18.       wrapup_task();
19.       exit(1);
20.     }
21. }
```

Figure 13 Pseudo code of an automatically generated scheduler for eCos

For a processor without OS support, the current CIC translator supports two kinds of
scheduling code: default and rate-monotonic scheduling (RMS). Default scheduler just
keeps the execution frequency of tasks considering the period ratio of tasks. Figure 14 (a)
and (b) show the pseudo code of function “get_next_task()”, which is called in the
function “scheduler()” of Figure 8 (b), for the default and the RMS respectively.

```
1. int get_next_task() {
2.   a. find executable tasks
3.   b. find the tasks that has the smallest value of time count
4.   c. select the task that is not executed for the longest time
5.   d. add period to the time count of selected task
6.   e. return selected task id
7. }
```

```
1. int get_next_task() {
2.   a. find executable tasks
3.   b. select the task that has the smallest period
4.   c. update task information
5.   d. return selected task id
6. }
```

Figure 14 Pseudo code of “get_next_task()” without OS support: (a) default and (b) RMS scheduler

6.4 PRODUCTIVITY ANALYSIS

For the productivity analysis, we recorded the elapsed time to manually modify the
software (including the debugging time) when we change the target architecture and task
mapping. Such manual modification is performed by an expert programmer who is a
Ph.D. student.

For fair comparison of automatic code generation and manual coding overhead, we
made the following assumptions. First, the application task codes are prepared and
functionally verified. We chose an H.263 decoder as the application code that consists of
six tasks as illustrated in Figure 3. Second, simulation environment is completely
prepared for the initial configuration as depicted in Figure 15 (a). We chose RealView
SoC Designer as the target simulator. And, we prepared two different kinds of HW IPs for IDCT function block. Third, software environment for the target system is prepared, which includes run-time scheduler and target-dependent API library.

![Diagram](attachment:image.png)

Figure 15 Four target configurations for productivity analysis: (a) initial architecture, (b) HW IDCT is attached, (c) HW IDCT and interrupt controller are attached, and (d) additional processor and local memory are attached.

At first, we needed to port the application code to the simulation environment of Figure 15 (a). The application code consists of about 2400 lines of C codes, in which 167 lines are target-dependent. The target-dependent codes should be re-written using target-dependent APIs defined for the target simulator. It took about 5 hours to execute the application on the simulator of our initial configuration (Figure 15 (a)). The simulation porting overhead is directly proportional to the amount of target-dependent codes. In addition, the overhead increases as the total code size increases since we need to identify the target-dependent codes throughout the entire application code.

Next, we changed the target architecture to Figure 15 (b) and (c) by using two kinds of IDCT HW IP. The interface code between the processor and IDCT HW should be inserted. It took about 2 and 3 hours to write and debug the interfacing code with IDCT HW IP without and with interrupt controller respectively. The sizes of the interface without and with interrupt controller are 14 and 48 lines of code respectively. Note that the overhead will increase if the HW IP has more complex interfacing protocol.
Last, we modified the task mapping by adding one more processor as shown in Figure 15 (d). For this analysis, we needed to make an additional data structure of software tasks to link with the run-time scheduler on each processor. It took about 2 hours to make the data structure of all tasks and attach to the default scheduler. Then, it took about 0.5 hour to modify the data structure according to the task mapping decision. Note that to change the task mapping configuration, algorithm part of software code need not be modified.

We summarize the overhead of manual software modification in Table 2.

On the other hand, in the proposed framework, design space exploration is simply performed by modifying architecture information file only, not task code. Modifying the architecture information file is much easier than modifying the task code directly, and it needs only a few minutes. Then CIC translator generates the target code automatically in a minute. Surely, it requires huge time to establish the translation environment for a new target. But once the environment is setup for each candidate processing element, we believe that the proposed framework improves design productivity dramatically for design space exploration of various architecture and task mapping candidates.

<table>
<thead>
<tr>
<th>Description</th>
<th>Code line</th>
<th>Time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial porting overhead to the target simulator</td>
<td>167 of 2400</td>
<td>5</td>
</tr>
<tr>
<td>Making HW interface code of IDCT (Figure 15 (a) ➔ Figure 15 (b))</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>Modifying HW interface code to use interrupt controller (Figure 15 (a) ➔ Figure 15 (c))</td>
<td>48</td>
<td>3</td>
</tr>
<tr>
<td>Making initial data structure for scheduler</td>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>Modification of data structure according to the task mapping decision</td>
<td>12</td>
<td>0.5</td>
</tr>
</tbody>
</table>

7. CONCLUSION

In this paper, we presented a retargetable parallel programming framework for MPSoC, based on a new parallel programming model called common intermediate code. The CIC specifies the design constraints and the task codes separately. Furthermore,
functional parallelism and data parallelism of application tasks are specified independently of the target architecture and the design constraints. Then, the CIC translator translates the CIC into the final parallel code considering the target architecture and the design constraints, to make the CIC retargetable.

Preliminary experiments with a H.263 decoder example prove the viability of the proposed parallel programming framework: It increases the design productivity of MPSoC software significantly. There are many issues to be researched further in the future, which includes optimal mapping of CIC tasks to a given target architecture, exploration of optimal target architecture, optimizing the CIC translator for specific target architectures.

ACKNOWLEDGMENTS
We deeply appreciate the constructive and insightful comments of the anonymous reviewers to improve the quality of the article.

REFERENCES

PEACE official homepage, http://peace.snu.ac.kr

