A Pattern-based Refactoring Approach for Multi-core System Design

1,2 Nien-Lin Hsueh, 2 Peng-Hua Chu, 3 Pao-Ann Hsiung, 5 Min-Ju Chuang, 6 William Chu, 7 Chih-Hung Chang, 7 Chorng-Shiuh Koong, 9 Chihhsiong Shih, 3 Chao-Tung Yang

Abstract

Until recently, the most software development tools and techniques were concentrated on views from the sequential model of program execution. Software developers who only familiarize with the sequential model will face unusual challenges of software projects that require multi-core or parallel programming. In this paper, we addresses the challenges that developers face as their projects requiring multi-core or parallel programming by applying design patterns. In general, design patterns are usually regarded as the role of quality-improver for improving software quality, but they could be the role of design-facilitator for facilitating design activities. To address this issue, we propose an approach for specifying the applicable contexts and refactoring steps of patterns systematically, as well as automating the refactoring process for lightening the burden of developers. We also extract a multi-core pattern ParallelWhile which is derived from the template class parallel_while of Intel® TBB library to introduce our approach. Furthermore, we demonstrate our approach by a real-world multi-core embedded system PVE (Parallel Video Encoder), where Command Pipeline pattern is designed for the design enhancement and platform migration. It aids developers identifying operations which could be parallel processing, and revising a sequential design as an Intel® TBB parallel structure. In addition, our approach can promote the extensibility of parallel operations.

Keywords: Multi-Core Embedded System, Design Pattern, Model Transformation, Pattern Application

1. Introduction

Multi-core programming is no more a luxury; it is now a necessity because even embedded processors are becoming multi-core. Various electronic devices with multi-core architecture are popularized in our normal life. However, the state-of-the-art techniques for multi-core embedded software development are not complete due to the tedious work needed in explicitly designing multi-core programs and debugging them [1]. While building a good multi-core embedded software design is difficult, the use of design patterns could be regarded as an useful way to simplify the development work [1, 2].

A design pattern encapsulates successful solutions for resolving recurring design problems, which often results in the product with higher quality [3, 4]. As patterns in Gang-of-Four style [5], design patterns provide various sections for describing the intent, motivation, consequence in nature language and the brief structure with a behavior diagram. They provide “better” solutions to meet non-functional requirements and satisfy original functional requirements [6, 7], as well as configure design elements that compose the solution structures of the patterns. Based on our analysis in previous research [8], a
design pattern could play four kinds of roles in assisting software design: design-facilitator\(^1\), quality-improver, problem-solver or conflict-resolver.

A design-facilitator pattern can facilitate a specific design activity, e.g. decomposition, object allocation, and access control, multi-thread or multi-core design. For example, the Observer design pattern allows us to reduce dependencies between entity objects and boundary objects. A quality-improver pattern handles non-functional requirement and thus improves software quality, such as reusability or extensibility. Taking Abstract Factory pattern as an example, it enhances reusability by providing an interface for creating related product objects without specifying their concrete types. A problem-solver pattern can assist solve a specific design problem, for example, Observer design pattern is used to resolve the inconsistency problem between a group of observer objects which concern with a common subject object. A conflict-resolver pattern can help resolve design conflicts, for instance, Proxy design pattern can be used to resolve the conflict between simplicity and performance.

The design work of multi-core embedded system is usually complicated, and therefore the gap between design and implementation has widened. In this work, we adopt design-facilitator patterns for facilitating multi-core software design and improving the translation from high level design to low level implementation smoothly. The benefit of considering multi-code embedded software design issues, such as parallel processing identification or specific platform implementation, from implementation level to design level is fewer costly and less complicated. In general, developers are usually familiar with a particular design concept, but not how to implement it in a new platform or libraries, e.g. Intel\(^\circ\) Threading Building Block (TBB)\(^2\) or OpenMP\(^3\). Embedding patterns into libraries can help developers comprehend how the library feature works to extract the library’s particular context.

However, developers are not always familiar with the purpose or usage of each pattern, it may cause design errors or inconsistency by misusing the pattern. Since design errors are not easy to detect and correct in implementation level, we can avoid design errors caused by misusing patterns through model transformation. Therefore, a pattern can be applied correctly even if developers do not understand how to implement it exactly. Moreover, by leveraging on the expertise gained from our previous work on pattern verification [9], a pattern supporting environment is implemented to create design patterns and specify their refactoring processes. Developers can resolve design issues easier by applying design patterns and reusing them for the same issues correctly. To illustrate and evaluate our approach, we present a case study on a real-world multi-core streaming-media system on VMC framework. VERTAF/Multi-Core (VMC) framework [10, 11] takes SysML models to specify model-level explicit parallelism as input and generates corresponding multi-core embedded software code in C++. The generated code can be scheduled and tested for a particular platform such as ARM 11 MPCore and Linux OS. Our work can help generate TBB code by applying multi-core patterns on VMC framework.

Intel TBB is C++ runtime library that abstracts platform details and threading mechanisms to simplify programming for multi-core platforms. Developers can put their effort into designing efficient scalable parallel program without dealing with low level details of threading. The core of the Intel TBB library is the task scheduler designed to hide the complexity of operating systems threads. It manages the internal thread pool and logical tasks that the developer creates, as well as maps them onto physical threads. Although Intel TBB has provides a higher-level and task-based parallelism, a developer must have expert knowledge and comprehensive experience to correctly apply different parallel programming interfaces provided by TBB [11].

In this case study, we intend to replace video encoding operations flexibility and parallel process them efficiently without increasing developers’ efforts. Therefore, the Command Pipeline pattern is proposed to resolve this issue, besides the ParallelWhile pattern for Intel\(^\circ\) TBB platform. The original sequential design is revised as a parallel structure by applying the pattern, and the comparisons between the raw design with enhanced design in performance and power consumption are presented. In summary, there are three features in this work:

---

1. In order to emphasize the ability of this role in this work, the original role activity-facilitator is renamed to design-facilitator.
A pattern-based approach is proposed to facilitate the transformation from raw model to enhanced or specific-platform model, and thus assist multi-core design;

- Two patterns are created for multi-core design to help specific platform implementation, as well as to improve software quality;
- A case study is presented to illustrate and verify our approach.

The reminder of this paper is structured as follows: the following section first details our approach with a multicore pattern example, and then illustrates it with a case study Parallel Video Encoder (PVE) system in Section 3. Section 4 reviews some related works about the applications of design pattern and profile. Finally, we conclude with a summary of the approach and final remarks in Section 5.

2. Our Approach

This section provides the overall architecture of our approach. We introduce a general overview of the steps that comprise our approach. Each of these steps is outlined below. Figure 1 illustrates the concept of our approach. The left part shows the steps to build a new pattern for automatic pattern application; whereas the right part shows that how to transform a raw design to an enhanced design by our approach. The four steps for applying our approach are described as below:

1. Extracting pattern intents. We decouple a pattern’s intent into functional intent \( I_{dp}^- \) and non-functional intent \( I_{dp}^+ \) to describe its functional and non-functional requirements. The aim of this phase is to extract the functional intent from a pattern’s original intent. Table 1 shows \( I_{dp}^- \) and \( I_{dp}^+ \) of Mediator, Abstract Factory and Observer examples.

2. Modeling pattern structures. We also decouple a pattern’s structure into functional structure \( D_{dp}^- \) and non-functional structure \( D_{dp}^+ \) to model the structures for meeting \( I_{dp}^- \) and \( I_{dp}^+ \). The aim of this step is to build the structures in a role-based pattern definition approach. Figure 3 and Figure 4(a) show that the functional and non-functional structures for Parallel/While example.

3. Designing transformation. The aim of this step is to define a series of refactorings on the basis of a pattern’s properties. The transformation describes the refactoring process from \( D_{dp}^- \) to \( D_{dp}^+ \). The end of Section 3.2.2 shows the transformation rules for a multi-core pattern Command Pipeline.

4. Implementing transformation. The aim of this step is to implement the process defined in previous step in specific transformation for execution. Developers can implement the transformation rules in different transformation languages, such as XSLT \(^4\) or ATL \(^5\). Figure 9 shows part of the transformation rules of multi-core pattern Command Pipeline implemented in ATL.

---

\(^4\) XSL Transformations (XSLT), Version 1.0. World Wide Web Consortium (W3C), Available from: [http://www.w3.org/TR/xslt](http://www.w3.org/TR/xslt)

2.1. Overview

Design patterns encapsulate reusable design knowledge to improve quality attributes or resolve specific design issues [5, 6]. In general, applying patterns still keeps the functional behaviors of original design, but improves one or more non-functional properties. For example, considering the original intent described in the Observer design pattern description [5]:

*Define an one-to-many dependency between objects so that when one object changes state, all its dependants are automatically notified and updated.*

The Observer pattern is designed to address the communication problem between subject objects and their related observer objects. Viewing from the functional aspect, it requires the subject to notify all observers when it changes its state. Viewing from the non-functional aspect, it requires the notification to work automatically *without knowing the types of observers*. Based on the observation, we define a pattern as a tuple \( \langle I, I_F, Q, S_F, S_N, T \rangle \) for evaluating its quality properties in our previous research [9]. In this work, we apply model transformation to address platform implementation instead of pattern evaluation on quality properties, thus the tuple is redefined as \( \langle I_{dp^-}, I_{dp^+}, D_{dp^-}, D_{dp^+}, T \rangle \), where

- \( I_{dp^-} \): the intent for describing what basic requirements does the pattern satisfy.
- \( I_{dp^+} \): the intent for describing how this pattern contribute to improve certain design issues, such as quality improvement or specific platform implementation.
- \( D_{dp^-} \): representing the context of raw design that can realize the \( I_{dp^-} \).
- \( D_{dp^+} \): representing the context of resultant design that can realize the \( I_{dp^+} \).
- \( T \): Transformation, representing the transformation function from \( D_{dp^-} \) to \( D_{dp^+} \).

Figure 2 depicts that the relationships between \( I_{dp^-} \), \( I_{dp^+} \), \( D_{dp^-} \) and \( D_{dp^+} \) for a design pattern \( dp \). The \( D_{dp^+} \) can realize the \( I_{dp^+} \) in the sense that it can satisfy the \( I_{dp^+} \) to a higher degree than compared to its associated \( D_{dp^-} \). The transformation represents the essence of a design pattern, which maps a raw design conforming to \( D_{dp^-} \) into a new design conforming to \( D_{dp^+} \), that is, \( T \). Each refactoring process of a design pattern consists of the addition or removal of model elements, such as classes, operations, attributes or relationships.

2.2. Extracting Pattern Intents

Based on our previous work [9], we were inspired to explore how a design pattern can satisfy the original requirement, and implement specific platform or quality requirements. Therefore, we analyze the pattern’s original intent, that is, \( I_{dp^+} \), to deduce the basic requirement intent \( I_{dp^-} \) as examples as shown in Table 1. In addition to the GoF pattern for quality issue, we extract a multi-core pattern which is derived from the template class \( \text{parallel}\_\text{while} \) of Intel\(^\circledR\) TBB library for platform implementation. According to the Intel\(^\circledR\) TBB tutorial and reference manual, we specify \( I_{\text{parallelWhile}^+} \) as follow: *By using the template class \( \text{parallel}\_\text{while} \) of TBB, the items can be safely processed in parallel.* In terms of functional aspect, the intent of a regular while loop \( (I_{\text{parallelWhile}^-}) \) is described as follow: *Based on a given boolean condition, the loop body is executed repeatedly.*

![Figure 2](image-url)


2.3. Modeling Pattern Structures

In general, the main components of a design pattern are intent and structure, and they should be consistent with each other as a well-designed pattern. For abstraction and reusability, we use UML profile to specify pattern structures. There are many possible starting structures or terminating structures in applying a specific design pattern. Although the structure section of each pattern only describes a sample or possible structure, it identifies the elements that must exist in the terminating structures. Therefore, we adopt France’s approach [12] to specify a pattern’s structure in a formal representation which is a role-based pattern definition approach for specifying a pattern structure from the view of pattern roles instead of specific elements.

For fulfilling $I_{dp^{-}}$ and $I_{dp^{+}}$, each pattern $dp$ is also divided into two kinds of profiles, $D_{dp^{-}}$ and $D_{dp^{+}}$. Figure 3 and Figure 4(a) show that $D_{dp^{+}}$ and $D_{dp^{-}}$ of the ParallelWhile pattern. Each model element in the design is specified as a role tagged with a stereotype in UML profile, whether it is a class, operation, dependency or generalization. As presented in Figure 3, there are nine stereotypes (participants) in $D_{ParallelWhile^{+}}$ for realizing $I_{ParallelWhile^{+}}$ as follows:

- **pwOperation**: An operation with parallel while structure.
- **pwOperator**: A const operator of $pwClass$ for loop body definition.
- **run**: An operation of $parallel\_while$ for execution with $pwClass$ and $itemStream$.
- **parallel_while**: API support, performs parallel iteration over items.
- **pwClass**: Defines the loop body.
- **itemStream**: Defines the stream of items.

<table>
<thead>
<tr>
<th>Design Pattern</th>
<th>$I_{dp^{-}}$</th>
<th>$I_{dp^{+}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mediator</td>
<td>A set of colleague objects communicate with each other directly</td>
<td>A set of colleague objects <em>without knowing about each other</em>, and communicate with each other only indirectly, through the mediator</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>A client object creates/uses a set of related objects <em>(called products)</em></td>
<td>A client object creates or uses these product objects <em>without specifying their concrete types</em></td>
</tr>
<tr>
<td>Observer</td>
<td>A subject object can notify all related objects <em>(called observers)</em> when it changes state</td>
<td><em>Without knowing types of the observers</em>, a subject object can automatically notify all observers</td>
</tr>
</tbody>
</table>

Table 1. The $I_{dp^{-}}$ and $I_{dp^{+}}$ of Mediator, Abstract Factory and Observer.

![Figure 3. Stereotypes extending UML in $D_{ParallelWhile^{+}}$ pattern profile.](image-url)
1. **void** SerialApplyFooToList( Item* root ) {
2.   Item* ptr = root
3.   **while** (ptr != NULL)
4.     Foo( pointer->data );
5.     ptr = ptr->next
6. }

(a) Code of class ItemStream

(b) The situation of $D_{\text{ParallelWhile}^+}$

**Figure 4.** Stereotypes extending UML in $D_{\text{ParallelWhile}^-}$ pattern profile and the application code example.

- **instantiatePWC**: The relationship from $pwOperation$ to $pwClass$ for constructing $pwClass$.
- **instantiateIS**: The relationship from $pwOperation$ to $itemStream$ for constructing $itemStream$.
- **instantiatePW**: The relationship from $pwOperation$ to $parallel\_while$ for constructing $parallel\_while$.

Figure 5 shows that the implementation of $I_{\text{ParallelWhile}^-}$ for fulfilling $I_{\text{ParallelWhile}^+}$. When initiating $pwOperation$ operation, three classes $parallel\_while$, $itemStream$, $pwClass$ are constructed (line 2-4 in Figure 5(c)). The parallel process is executed by the method $\text{run}$ of $parallel\_while$ class with two argument types, $itemStream$ and $pwClass$ (line 5 in Figure 5(c)). Figure 5(b) shows the loop body of $pwClass$ for $D_{\text{ParallelWhile}^+}$. As presented in Figure 5(a), the $itemStream$ starts to get next stream item by $\text{pop\_if\_present}$, and checks the next iteration value. If there is one item and returns true, $parallel\_while$ may concurrently invoke the operator for processing different items. On the contrary, it returns false if there is no more iteration.

According to the description of $I_{\text{ParallelWhile}^+}$, the regular while loop is a control flow statement of an operation. Thus there is only one role in $D_{\text{ParallelWhile}^-}$ for indicating the operation which could be parallel processed as presented in Figure 4(a). Figure 4(b) presents the situation of a $serial\_while$ operation for dealing with infinite loop. From $D_{\text{ParallelWhile}^-}$ to $D_{\text{ParallelWhile}^+}$, the original code (as presented in Figure 4(b)) in principle has been changed as code with underline in Figure 5(a) and Figure 5(b). The remainder code is constant, and whose skeleton could also be created by model transformation.

### 2.4. Designing Transformation

For assisting pattern application, we document the transformation processes in regular rules and describe them in formal transformation language. In this section, we introduce the concept of pattern-based model transformation, and illustrate how to specify a transformation specification for a transformation process.

(a) Code of class ItemStream

(c) The revised $while$
2.4.1. Pattern-based Model Transformation

Based on the general model transformation concept [13], our approach revises a raw design to a resultant design according to the refactoring process of a pattern, thus we call it pattern-based model transformation. As presented in Figure 6, the model transformation MT maps a $D_{dp^-}$ structure to a $D_{dp^+}$ structure. That is, $\forall s \in D_{dp^-}, MT(s) \in D_{dp^+}$, where the notation $\in$ denotes the instantiation relationship between model and metamodel. $\forall s \in D_{dp^-}$ means that the model is an instance model of the metamodel $D_{dp^-}$.

First, we have to design the transformation specification which describes the mapping rules from $D_{dp^-}$ to $D_{dp^+}$ formally. These mapping rules specify which roles are added or removed during the refactoring process, such as add an abstract class, add an operation into a class or remove an association between one and the other class, etc. So far, we have a raw design $s$ which is an instance of $D_{dp^-}$. After describing the transformation specification, we can transform $s$ to a new design $MT(s)$ by executing mapping function $MT()$ based on the transformation specification. Finally, we can get the well-designed model conforming to $D_{dp^+}$ automatically. In this way, the design is no longer revised manually, and developers can reuse the transformation specification and pattern profiles to apply the same design pattern. In addition, developers can track the variation of a design to know what elements are modified.

2.4.2. Rules for Pattern-based Model Transformation

The transformation process from a source model to a target model is composed of some structure modification actions, such as abstraction or association change. Inspiring by Jing et al. [14], they analyze the design patterns documented in [5] to classify the possible pattern evolutions. They identify the primitive-level evolutions which are the addition or removal of modeling elements. The general presentation for adding a model element is $add(ME(PL))$, and the removal of a model element can be described in the same presentation $delete(ME(PL))$, where $ME$ stands for model element and $PL$ represents parameter list. They also propose nine basic model elements (ME) with the parameter list (PL) in their approach.

The model elements in the primitive-level evolutions are the basic components to organize the static structure, that is, class diagram. Therefore, we specify the addition and removal actions as some rules of the pattern transformation specification in terms of a sequence/combination of the primitive-level evolutions in metamodel level. But the primitive pattern evolutions of Jing et al. [14] are specified in model level, we extend the representation with stereotype in metamodel level. Part of pattern evolutions in metamodel are presented in Table 2. For example, we have to add a class stereotyped «AbstractFactory» in the refactoring process from $D_{dp^-}$ to $D_{dp^+}$, which can be specified: $add(Class(AbstractFactory))$. But the other relationship addition is derived from adding
2.5. Implementing Transformation

After specifying the transformation specification, we implement these mapping rules in ATLAS Transformation Language (ATL). ATL is a hybrid of declarative and imperative transformation languages based on OMG OCL (Object Constraint Language)\(^6\). In this work, we use ATL as the language and engine for model transformation on account of the following contentions: a) it is an open-source software that we can popularize our approach and ATL widely; b) the user community is large that we can look for solutions easily; c) there are fixed developers support keeping it stable and growing to maturity; d) there are rich model transformation projects and examples for developers learning; e) it supports various technical spaces maturely such as XML, EMF, and MOF; f) it is implemented as an Eclipse plug-in that can be integrated with modeling environment.

In the next section, we illustrate our approach by specifying Command Pipeline pattern with the transformation specification step by step for assisting multi-core embedded software design. Finally, the transformation specification is implemented in ATL and applied to revise a real case DVR system.

3. Case Study

In this section, we present a real system Digital Video Recording (DVR) [11] as a case study to illustrate our approach. We define Command Pipeline to revise a sequential processing design to a parallel processing design in favor of generating TBB code. Finally, we compare the power consumption and performance between the raw design and resultant design to describe the feasibility of our approach in assisting multi-core embedded software design.

3.1. Parallel Video Encoder: A Case Study

DVR is a real-time multimedia system used to monitor various locations remotely and concurrently. As presented in Figure 7(a), DVR comprises two subsystems: Parallel Video Encoder (PVE) and Video Streaming Server (VSS). PVE is responsible for collecting videos from multiple cameras and encoding them into more compressed data format, such as MPEG. VSS is responsible for servicing remote monitor clients with status information, real-time video streams, on-demand video streams, and storing the encoded video streams in large video databases.

The main task of PVE is video encoding which is performed by the class VideoEncoder. VideoEncoder provides five operations to finish a video encoding task: Discrete Cosine Transform (DCT()), Quantization (Q()), Huffman Encoding (HE()), Get Raw Frame (GetRF()) and Put Encoded Frame(PutEF()). Figure 7(b) shows the process when VideoEncoder gets a source frame from a

---


---

<table>
<thead>
<tr>
<th>Model Elements</th>
<th>Parameter List</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>classStereotype</td>
<td>Add or remove a class with stereotype ‘classStereotype’ into a pattern</td>
</tr>
<tr>
<td>Interface</td>
<td>interfaceStereotype</td>
<td>Add or remove an interface with stereotype ‘interfaceStereotype’ into a pattern</td>
</tr>
<tr>
<td>Operation</td>
<td>operationStereotype, returnType, accessibility, para1, paraType1, . . .</td>
<td>Add or remove an operation with stereotype ‘operationStereotype’, type of ‘type’, accessibility of ‘accessibility’, and arguments list para1 with type ‘paraType1’ into the class ‘classStereotype’</td>
</tr>
<tr>
<td>Dependency</td>
<td>relationshipStereotype, fromRole, toRole</td>
<td>Add or remove a dependency relationship from class role ‘fromClassStereotype’ to class role ‘toClassStereotype’ into a pattern</td>
</tr>
</tbody>
</table>

«AbstractFactory» class, we have to add a generalization relationship «CFfromAF» between «AbstractFactory» and «ConcreteFactory», that is, add(Generalization(CFfromAF, ConcreteFactory, AbstractFactory)).
camera by \texttt{GetRF()}, and then start video compression. \texttt{DCT()}, \texttt{Q()} and \texttt{HE()} are three kinds of encoding methods which are executed together in series for compressing each raw frame. Finally, the encoded video is exported to VSS through \texttt{PutEF()}.

When the infrastructure of embedded system is multi-core architecture, we want to promote the work efficiency by parallel processing independent operations instead of a series of operations execution sequentially. For example, after \texttt{GetRF()}, VideoEncoder can execute encoding operations \texttt{DCT()}, \texttt{Q()} and \texttt{HE()} simultaneously. If we have various cameras in different sites and they have to be encoded at the same time, it is time-consuming. When such design is deployed on a multi-core architecture, it could not allow the multi-core hardware’s powerful arithmetic capabilities full pay.

3.2. \textit{Command Pipeline Pattern for Multi-Core Architecture}

We specify \textit{Command Pipeline} pattern for simplifying the design of parallelism under multi-core architecture. Developers can consider where the parallelism exists in design phase, and revise a series of independent operations to parallel execution.

3.2.1. \textit{Specification of} $D_{dp^-}$ \textit{and} $D_{dp^+}$

In order to replace operations dynamically and consider parallel processing, we extend the \textit{Command} pattern in the GoF patterns \cite{5} with parallelism identification design issue as \textit{Command Pipeline} pattern. The intent of \textit{Command Pipeline} pattern ($I_{CommandPipeline^+}$) is described as follow:

\begin{itemize}
\item Encapsulate a request as an object for letting invokers parameterize clients with different requests.
\item In addition, it can assist to adjust working loads between cores for avoiding tasks are concentrated on fixed cores.
\end{itemize}

As presented in Figure 8(a), there are eight stereotypes (participants) in $D_{CommandPipeline^+}$ for realizing $I_{CommandPipeline^+}$ as follows:

- **invoker**: The user asks the command to carry out the request through \textit{command}. Each invoker has a property \texttt{pipelineToken} to specify the maximum data units which an operation can deal with.
- **parallelFilter**: An pipeline operation, it can process one or more data units simultaneously, e.g. \texttt{DCT()}, \texttt{Q()} and \texttt{HE()}. It has to implement \textit{command} interface.
- **serialFilter**: A pipeline operation, but it only tackles a data unit at the same time, e.g. \texttt{GetRF()} and \texttt{PutRF()}. It also has to implement \textit{command} interface.
3.2.2. Transformation rules

In this section, we show part of the transformation rules from $D_{CommandPipeline}^-$ to $D_{CommandPipeline}^+$. All the rules are not specific for any transformation language or engine; and they are general descriptions to specify the refactoring steps. In this work, we implement these rules in ATL which are detailed in next section.

- **Create new model elements in the target model:**
  - `add(Class(invoker))`, create a Class instance stereotyped `invoker`.
  - `add(Interface(command))`, create an Interface instance stereotyped `command`.
  - `add(Dependency(invokeCommand, invoker, command))`, add a dependency relationship stereotyped `invokeCommand` from `invoker` to `command`.

- **For each operation in the source model stereotyped `serial`:**
  - `add(Class(serialFilter))`, create a Class instance stereotyped `serialFilter`.
  - `remove(Operation(serial))`, remove the operation stereotyped `serial` from source model.
  - `add(Realization(commandRealization, serialFilter, command))`, add a realization relationship stereotyped `commandRealization` from `serialFilter` to `command`.

- **For each operation in the source model stereotyped `parallel`:**
  - `add(Class(parallelFilter))`, create a Class instance stereotyped `parallelFilter`.

The $I_{CommandPipeline}^-$ is described as: *To perform a series of operations to finish a task*. Therefore, each operation is a kind of «serial» method, and operations which could be parallel processed without impacting with the others is a kind of «parallel» method in $D_{CommandPipeline}^-$ as shown in Figure 8(b).
3.2.3. Implementation in ATL

We describe below a subset of the transformation rules by using the ATL. As presented in Figure 9, the main refactoring steps change an independent «parallel» operation of a class to a class tagged with «parallelFilter» in transformation rule Op2ClassP. Initially, an ATL helper is defined to extract the stereotype of each model element. It extracts a model element which is an operation with the stereotype «parallel» first (line 2), and creates a class with the class name as the original operation name (line 3-5).

Further, it adds a common operation execute in the new class (line 7-9), and a realization named ParallelImplement which is linked from the new class to the interface PipelineOperation (line 11-14), assuming that the interface PipelineOperation is created and stereotyped «command» by executing previous rules. Finally, the new class is stereotyped as «parallelFilter» (line 16), the realization between it and the interface PipelineOperation is tagged with «commandRealization» (line 18), and the operation ‘execute’ is stereotyped «commandExecute» (line 17).

Obviously, we have implemented the rules add(Class(parallelFilter)), add(Operation(commandExecute, parallelFilter)), and add(Realization(commandRealization, parallelFilter, command)) in target model, and the model element removal is implicit, e.g. the remove(Operation(parallel)). If any model elements are not described in ATL through the transformation, they do not appear in the target model.

3.2.4. The resultant design

In order to transform the raw design to an enhanced design, we tag encoding operations with «parallel» stereotype to identify what methods could be parallel processed, and remains are tagged with «serial» as presented in Figure 10(a). The resultant design of VideoEncoder is generated based on the pattern transformation specification, and the target model is shown as Figure 10(b). The resultant design can be used to generate Intel® TBB code seamlessly and extended parallel operations easily. If developers intend to increase other video encoding operations for extension, they add a new «parallelFilter» class to implement PipelineOperation interface instead of modifying VideoEncoder class directly. That is, the resultant design conforms to OCP (Open Closed Principle) for software design: “software entities should be open for extension, but closed for modification” [15].
Although the main benefit of our proposed patterns is to improve the extensibility of the resultant design, it also cannot impair the basic multi-core properties: performance and power consumption. Therefore, we generate the TBB code based on the resultant design which is compared with the raw design in performance and power consumption. We evaluate two implementations in the same environment with a quad processor and two cameras whose capture rates are set for 16 to 20 fps (frames per second). The evaluation shows that the resultant design can improve multi-core design quality without lowering performance or increasing power consumption.

As shown in Figure 11(a), the raw design of PVE is a sequential processing structure, therefore there is only one core works and fully loading, but the others are idling. The encoding rates for two cameras are 2.89 and 3.01 fps respectively. The raw design cannot bring the multi-core context into full pay, and it wastes the great capacity for multi-tasking of this device. Comparing with Figure 11(a), the workload of resultant design in Figure 11(b) is distributed into four cores averagely. The encoding rates for two cameras are upped to 15 and 16 fps. Obviously, the resultant design can balance processor utilization effectively, and then promote the encoding rates to increase throughput.

We measured the power consumption of the video streaming server without using an additional

![Diagram](image1)

**Figure 10.** (a) The VideoEncoder class of PVE with stereotypes of $D_{Command Pipeline}$; (b) The new design conforming to $D_{Command Pipeline}$ after transformation.

![Graph](image2)

**Figure 11.** The improvement of PVE’s performance after applying patterns.
hardware meter. Lien et al. [16] propose a power model for measuring the power consumption of a streaming-media server as, $P = D + (M - D)U^{0.5}$, where $P$, $D$, $M$ and $U$ represent for power consumption, base power, full-load power and CPU utilization, respectively. By the power model, we evaluate the power consumption of two designs as follows:

$$P_{\text{raw design}} = 69 + (142 - 69)(0.3)^{0.5} = 108.98 \text{ (W)}$$

$$P_{\text{resultant design}} = 69 + (142 - 69)(0.9)^{0.5} = 138.25 \text{ (W)}$$

The power consumption of resultant design is higher than raw design about 30W expectably since the quad processor is fully utilized, but the encoding rate is increased five times. From what has been discussed above, the Command Pipeline pattern can assist developers to identify the parallel processes operations to utilize multi-core resources efficiently. The key concern that developers have to consider is the identification of the parallel tasks.

4. Related Work

Taibi and Ngo [17] propose BPSL to specify the structural as well as behavioral aspects of design patterns. This is achieved by combining two subsets of logic, one from First Order Logic and the other from Temporal Logic of Actions. However, the specification language of formal method-based approach is not a widespread language that the approach is not easy to be integrated with general software design environments. Thus, some works specify design patterns in UML-based approaches. France et al. propose the RBML notation to specify the solution domain of design patterns [12], and Kim et al. provide tool support for generating an UML model from an RBML specification [18]. They use a role-based representation to formally specify pattern solution in the meta-model level. The resulting specifications can be used to develop tool support for automatic evaluation of pattern applicability. Mak et al. [19] also use UML profile with role-based representation to provide a more complete and precise UML-based modeling of pattern leitmotifs. In their work, a set of stereotypes is defined to encapsulate the high level relationships among roles, and these relationships convey the true abstract nature of role associations that reveals designers’ intuition to the invariants of pattern leitmotifs. Therefore, the abstract property of design pattern can be retained by avoiding premature commitments.

Soundarajan et al. [20] focus on the testing and maintenance phases of the software development life-cycle instead of design phase for pattern application. In order to prevent misunderstanding of different team members for a pattern, they formalize the pattern format as “pattern contracts” which include a set of role-contract for a design pattern. However, formalizing a design pattern may reduce its flexibility, they provide “pattern subcontract” to specialize pattern application. Developers can apply a design pattern by describing pattern subcontract which maps each class and its methods as the role and role methods based on system design. Dong et al. [14] propose a service-oriented architecture for design pattern evolution and analysis based on two-level transformations to show the possible evolutions of each design pattern explicitly. They identify nine modeling elements that can be added or deleted as the primitive-level evolutions. On the other hand, the possible evolutions of each design pattern are categorized as the pattern-level evolutions. The pattern-level evolutions are composed of the basic transformation steps of primitive-level evolution. The pattern-level evolutions also can be classified as five categories according to different design evolutions. The evolutions of each design pattern are described as XSLT transformation rules to check the consistency of the evolution result.

5. Conclusion

In this study we have shown how it is possible to improve multi-core design by pattern-based model transformation. In addition, we propose Command Pipeline and ParallelWhile patterns to facilitate the transition from single core to multi-core design for performance improvement and load balance. It allows us to pay less power consumption, but get better performance remarkably.

We also apply Command Pipeline pattern to a real case for transforming a sequential processing design into a parallel processing one. To raise the level of abstraction in multi-core embedded system design from implementation phase to design phase, developers could devote their attentions to important contents without considering low-level details. In the future, we plan to discover, analyze
and specify more design patterns, as well as specify their transformation rules to advance a raw design up to an enhanced design automatically.

Acknowledgements
This research was sponsored by Department of Industrial Technology, Ministry of Economic Affairs in Taiwan ROC under the Grant NSC99-2220-E-035-001.

References


