Lin CS, Lu CH, Lin SW *et al.* VERTAF/Multi-Core: A SysML-based application framework for multi-core embedded software development. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 26(3): 448–462 May 2011. DOI 0.1007/s11390-011-1146-3

VERTAF/Multi-Core: A SysML-Based Application Framework for Multi-Core Embedded Software Development

Chao-Sheng Lin[1] (林朝圣), Chun-Hsien Lu[1] (吕俊贤), Shang-Wei Lin[1] (林尚威), Yean-Ru Chen[2] (陈盈如) and Pao-Ann Hsiung[1] (熊博安), *Senior Member*, *ACM*, *IEEE*

[1]*Department of Computer Science and Information Engineering, "National Chung Cheng University", Chiayi County 62102 Taiwan, China*

[2]*Department of Computer Science and Information Engineering, "National Taiwan University" Taipei 10617, Taiwan, China*

E-mail: {lcs94, lchs91u, linsw}@cs.ccu.edu.tw; d95943037@ntu.edu.tw; pahsiung@cs.ccu.edu.tw

Received March 28, 2010; revised March 19, 2011.

Abstract Multi-core processors are becoming prevalent rapidly in personal computing and embedded systems. Nevertheless, the programming environment for multi-core processor-based systems is still quite immature and lacks efficient tools. In this work, we present a new VERTAF/Multi-Core framework and show how software code can be automatically generated from SysML models of multi-core embedded systems. We illustrate how model-driven design based on SysML can be seamlessly integrated with Intel's threading building blocks (TBB) and the quantum framework (QF) middleware. We use a digital video recording system to illustrate the benefits of the framework. Our experiments show how SysML/QF/TBB help in making multi-core embedded system programming model-driven, easy, and efficient.

Keywords multi-core, model-driven, parallel programming, framework, SysML, design pattern

1 Introduction

The emergence of multi-core architectures[1] is mainly to tackle the issues posed by single-core processors while trying to increase the operating frequency to obtain the computing power, and the issues include memory wall, instruction level parallelism (ILP) wall, and power wall. Multi-core architectures today have been proved to be ideal solutions to increasing the computing power by being adopted in a wide range in the marketing from the server, desktop, and laptop to the embedded systems and digital signal processing design. Though the multi-core has brought us the benefit in increasing the computing power while operating in low frequency and consuming less power, but it also creates challenges for software engineers, especially for those in embedded systems with rigid constraints.

To design an application in a multi-core system, the engineers should first analyze how much computationally intensive work existing in an application. The work is then to be decomposed into small tasks and scheduled to the computing resources in a multi-core system. Here the proper parallel algorithms are required to coordinate the decomposed work according to their behavioral or structural nature in an application, and also, these parallel algorithms need the underlying parallelism control systems as engines to map the decomposed work from logical computations to physical ones. Such engines involve task or thread management and scheduling. To design suitable management policies and scheduling algorithms for parallel algorithms in a multi-core system is tedious to the software engineers while considering the various architecture designs of multi-core systems.

The current state-of-the-art technology in multi-core programming to solve the issues arising from the design of parallel algorithms, and task and thread scheduling, is based on the use of language extensions or libraries, such as *OpenMP*[2], *Cilk*[3] and Intel *Threading Building Blocks* (TBB)[4]. These tools help software engineers to design their applications by applying several parallel algorithms to the decomposed work and managing the tasks and threads for the engineers. Nonetheless, the software engineers still have to get acquaintance with these tools and require parallel computing expertise when parallelizing their applications in multi-core systems.

In order to accelerate the adoption of parallel

Regular Paper
©2011 Springer Science + Business Media, LLC & Science Press, China

programming technologies for the embedded software designers, we extend our existing tool, *Verifiable Embedded Real-Time Application Framework* (VERTAF)[5], for multi-core embedded software design. The proposed *VERTAF/Multi-Core* (VMC) is based on model-driven architecture in software engineering, and takes SysML models as input which contains user-specified model-level explicit parallelism for generating corresponding multi-core embedded software code.

Several issues crop up when developing a model-driven architecture for multi-core embedded software. First of all, how much and what kinds of explicit parallelisms must be specified by a software engineer through system modeling. Second, how we can automatically and correctly realize the user-specified models into multi-core embedded software code. We try to provide partial solutions to the above issues, which are still open to more research work.

The main contributions of this paper are listed below.

• A parallel model for parallel algorithm, namely software pipeline (mixed solution of task parallelism and data parallelism), is proposed by using a *command design pattern*.

• The implementation method may violate the semantics of the adopted system models, e.g., run-to-completion (RTC) semantics in the SysML state machine. We illustrate our experience to the readers in solving the issue at model level by a case study called digital video recording (DVR) system.

• We analyze different thread-level implementations for different purposes, including system behavior control and parallel computing.

• We proposed a code generation flow to show how to implement both the system behavior control and inherent parallelism of an application from the system models by integrating existing libraries, i.e., QF and TBB.

The article is organized as follows. Section 2 describes the languages, libraries and framework, being adopted in VMC, as well as existing related work. Section 3 describes the proposed VMC framework. Section 4 describes the code generation process in VMC. Section 5 uses a digital video recording system example to illustrate how VMC achieves automatic multi-core programming using TBB and QF. Finally, Section 6 gives the conclusions with some future work.

2 Preliminaries and Related Work

VERTAF is a UML-based application framework for embedded real-time software design and verification. The original VERTAF is an integration of software component-based reuse, formal synthesis, and formal verification. It takes three types of extended UML models[6], namely class diagrams with deployments, timed statecharts, and extended sequence diagrams. The sequence diagrams are translated into *Power-Aware Real-Time Petri Nets* and then scheduled for low power design along with satisfaction of memory constraints. The timed statecharts are translated into *Extended Timed Automata* (ETA) and model checked using the SGM (*State Graph Manipulators*) model checker. The class diagram and the statecharts are used for code generation.

VERTAF uses the quantum framework (QF)[7] for software code generation. QF is a framework for rapidly implementing software in an object-oriented fashion. A UML state machine is implemented by a QF active object. Based on the programming principles and APIs provided by QF, VERTAF translates a system modeled by a user with UML state machines into C/C++ embedded software code.

The main purpose of model-driven software development is to alleviate the problem of inherently high complexity in software. In our target embedded systems, multi-core processor architectures not only drastically increase the complexity of embedded software, but also aggravate the whole issue of complexity due to the incomprehensible interactions among multiple threads[8]. Hence, the model-driven development process is even more essential to multi-core embedded software design. However, this requires adaptation of the VERTAF flow to support multi-core embedded software design as illustrated in Fig.1.

In this work, we are extending the code generation of VERTAF using TBB, which as discussed earlier is a C++ library that offers parallelism at higher levels. At the highest level, parallelism exists either in the form of data to operate in parallel, or in the form of tasks to execute concurrently. TBB tasks that take advantage of both data parallelism and task parallelism are most useful for programming multi-core embedded systems. We will use a digital video recording system as an example to illustrate how the data and task parallelisms are integrated into the embedded software code that is generated from SysML models.

The *Unified Modeling Language* (UML)[9] is an industry de-facto standard language used for designing software from various application domains, including embedded systems. UML allows software designers to visualize document models of their software. In order to analyze, design, and verify complex systems, an extension of UML called the OMG *System Modeling Language* (SysML)[10] was recently proposed. SysML reuses several components from UML and extends the system requirements model by supporting more diagrams, such as requirement and parametric diagrams

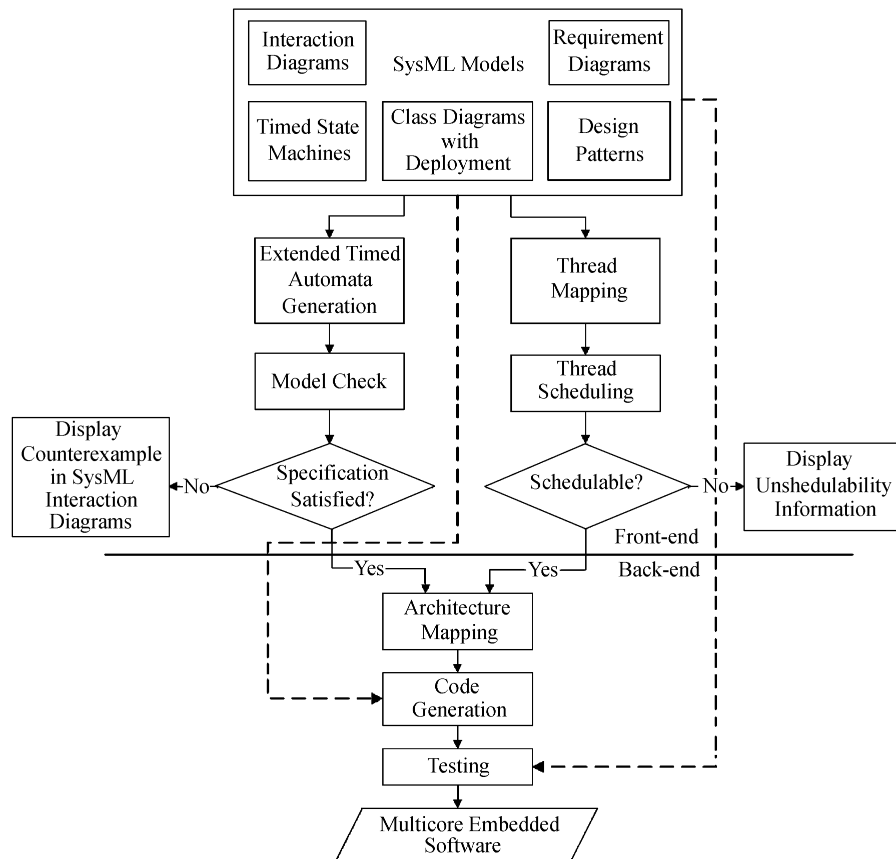

Fig.1. VERTAF/multi-core design flow.

where the former is used for requirements engineering and the latter is used for performance and quantitative analysis. Thus, in this work, instead of using UML as in VERTAF, we started to adopt SysML as our modeling language in VMC.

Commercial tools such as IBM Rational Rose and I-Logix Rhapsody generate code automatically. According to the recent report from IBM software library[11], they deploy the single-core applications to multicore systems using modeling language UML 2.0 and predicting performance gains at system level by applying the Amdahl's law. In contrast to these commercial tools, the code generated by VMC is both scheduled and verified formally. The MoBIES (Model-Based Integration of Embedded Systems) project[12-14] funded by USAs DARPA and the DESS project funded by Europes EUREKA-ITEA are all very large and 5-year longterm research projects. Nevertheless, what VMC has achieved has already surpassed the achievements of both MoBIES and DESS, because MoBIES results were all divided into several small projects in different universities without time integration and DESS mainly proposed theories (guidelines). In contrast, VMC has successfully proposed the theory and implemented it into a useful application framework.

In [15], the authors have provided a design environment for MPSoC based on the tool Gaspard2. The design methodology of Gaspard2 is based on model-driven engineering, and it also adopts *MARTE*[16], which enhances the modeling capability of UML, to model the software and the hardware of applications. The data and system structures are modeled, and through transformation, the computation is implemented in the executable platform based on SystemC at timed programmer view level. Two main differences between VMC and Gaspard2 include the goals of the system modeling and the target executable platform. Our framework focuses on the parallel models for parallel algorithms, as well as, the models of system behaviors. In addition, the code emulator and the testing environment proposed in VMC are the physical hardware platform, not a simulated environment.

Several researches also proposed solutions to the problems which are posed by multi-core systems. SWARM[17] project is also a framework for multi-threaded programming for multi-core systems. The authors proposed a model for the parallel shared memory multi-processor models and algorithm analysis as well

as provided a library for multi-core programming. Different from SWARM, VMC models the multi-core embedded software by SysML and generates code based on the TBB task model and the QF active object model which are higher-level programming paradigms. Though the actual execution of TBB tasks and QF active objects are all performed by the underlying user-level and kernel-level POSIX threads, yet the more abstract task/object level of code generation from system design models avoid low-level bugs to creep into the software and allow easier performance analysis. For example, the *run-to-completion* (RTC) semantics of UML/SysML are all preserved by TBB tasks and QF active objects and thus it is not easy to violate these constraints, while it is very easy to do so when using pure POSIX thread programming. In particular, both TBB and QF libraries have very small memory footprints and are thus very much suitable for embedded systems.

The project Cellss[18] supports simple and flexible programming model for parallel and heterogeneous architectures such as the Cell Broadband Engine (BE) architecture. By maintaining the task-dependency graph of the calls to functions at runtime, Cellss transfers data from and to the Synergistic Processing Elements (SPEs). Similar to OpenMP, Cellss is an alternative programming model and proposes directives for code annotations and functional decomposition. Instead of OpenMP or Cellss, VMC adopts TBB for parallelizing data, task, and data flow, because these real-world concurrency solutions are better implemented by TBB.

Similar to the TBB task stealing, Wagner *et al.*[19] implemented *task processing interface* (TPI) scheduler in the user-space of the operating system and proposed a work stealing algorithm for load balancing in a multi-core environment. VMC currently is based on the original TBB random task stealing; however, it can always be extended to adopt other task stealing algorithms.

3 VERTAF/Multi-Core Framework

VERTAF extended for multi-core programming is now called VERTAF/Multi-Core (VMC). The control and data flows of VMC are represented by solid and dotted arrows, respectively, in Fig.1. From the perspective of system designer, software synthesis is defined as a two-phase process: a software construction phase called front-end and an implementation phase called back-end. This separation helps VMC design to plug-in different target languages, middleware, real-time operating systems, and hardware device configurations.

The front-end phase is divided into three sub-phases, namely SysML modeling phase, real-time embedded software scheduling phase, and formal verification phase. In SysML modeling phase, VMC requires four diagrams as an input of system specification models, namely requirements diagram, block definition diagram, interaction diagram, and state machine. In the scheduling phase, Pthreads are scheduled by Linux OS, and the TBB threads are scheduled by the TBB library along with thread migration among different cores. In the formal verification phase, timed automata models generated from the SysML models are verified using the SGM[20] model checker, along with the VMC built-in models for real-time task scheduling, inter-core task migration, and load balancing policies.

The back-end phase also consists of three phases, namely architecture mapping, code generation, and testing. The architecture mapping phase is the configuration of the hardware system and operating system through the automatic generation of configuration files, make files, header files, and dependency files. We adopt a multi-tier approach for code generation: an operating system layer (Linux), a middleware layer (QF), a multi-core threading library layer (TBB), and an application layer. The generated code is tested for several issues, such as functional validation, non-functional evaluation, and deadlock detection.

4 Multi-Core Code Generation

To alleviate the burden of application designers, VMC supports model-driven development in two ways. First, VMC provides abstract architecture models of multi-core computing along with real-time task scheduling and load balancing mechanisms. Second, VMC supports parallel design patterns such as parallel pipeline to hide latency, parallel loop to reduce latency, and parallel tasks to increase throughput. These design patterns correspond exactly to the three real-world concurrency issues[21].

4.1 Task and Thread Models

A *task* in TBB is a basic unit of computation job, while a thread is a basic unit of computation that can be assigned a task to execute. An application can be represented by a *task graph*. The tasks that are ready are assigned by a task scheduler for execution by threads from a thread pool.

The task and thread models of TBB are quite generic and are suitable for general-purpose computing. However, to satisfy real-time constraints in embedded systems, we need to have threads that are devoted to specific tasks such as input sensing, computation, and actuator outputs. The task/thread model in VMC consists of the following three parts.

• *User-Level Pthreads*. The POSIX threads are *devoted* threads, that is, unlike TBB threads, they are

never reused for executing other tasks. There are two uses of such Pthreads in VMC as follows: (a) to execute the user-specified state machines (represented by QF active objects), and (b) to execute conventional legacy parallel tasks.

• *User-Level TBB Threads*. These are the threads maintained by the TBB scheduler. They can be reused and migrated across different cores. The TBB threads are scheduled by the TBB threading library using a non-preemptive unfair scheduling approach to trades-off between depth first execution and breadth first execution of tasks on the task graph, which are ready for execution.

• *Kernel-Level OS Threads*. Each of the above user-level threads, including POSIX and TBB, is mapped to a kernel thread of the underlying OS such as Linux in VMC. The kernel threads are scheduled by the OS scheduler using a preemptive priority-based scheduling algorithm.

4.2 Model-Level and Code-Level Parallelism

As introduced at the beginning of Section 4, the three real-world concurrency issues[21] include latency hiding, latency reduction, and throughput increasing. The corresponding solutions are parallel pipeline, parallel loop, and parallel tasks, respectively. TBB supports all of these solutions with certain restrictions such as the parallel loop is supported only in four forms, namely `parallel_for`, `parallel_reduce`, `parallel_scan`, and `parallel_while`. Since VMC generates code based on QF and TBB APIs, all of the three corresponding solutions are supported at the code level. However, the main issue to be addressed here is how and what kinds of parallelism to allow application designers to specify at the model level.

The following approach is adopted in VMC. VMC provides a UML profile to support parallel design patterns such that users can apply stereotype tags to SysML models. Currently, VMC users can apply the following sets of stereotype tags:

1) ≪`pipeline`≫ to a transition in the state machine model;

2) ≪`serial`≫ and ≪`parallel`≫ to classes in a block definition diagram describing pipeline parallel model;

3) ≪`parallel_for`≫, ≪`parallel_reduce`≫, ≪`parallel_scan`≫, and ≪`parallel_while`≫ to a method or a part of a method; and

4) ≪`parallel_tasks`≫ to a method, and ≪`task`≫ to a part of a method.

Fig.2 shows the definition of stereotypes for the pipeline parallel model. The definition of stereotype ≪`pipeline`≫ can be applied to a state transition of a state machine. When the VMC code generator detects this stereotype, it immediately checks the value *ParallelModel* defined in stereotype ≪`pipeline`≫ and refers to the dedicated pipeline parallel model. The parallel model is described in a block definition diagram. When code generator detects the stereotype ≪`invoker`≫, it generates the code to initialize the TBB library, instantiate a TBB pipeline class, add all specified filters, and execute the pipeline. Stereotypes to be applied to the filter classes, namely ≪`serial`≫ and ≪`parallel`≫, are for the code generator to generate the serial and parallel filters for the pipeline.

Using this parallel design pattern profile, VMC thus bridges the gap between model-level and code-level parallelisms. Application designers are required to explicitly specify parallelism at the model level because the designers know best what to parallelize and what not to. VMC alleviates the burden of parallel programming through automatic code generation. Designers have to only tag the models with the above stereotypes and VMC takes care of the rest.

4.3 Code Generation

In this subsection, we first introduce the code generating flow, and then describe the issues arising from semantics violation at the model level. Finally, we describe how the parallel code are generated from the parallel models.

Fig.3 shows the flow of the VMC code generator. The code generator consists of five main components

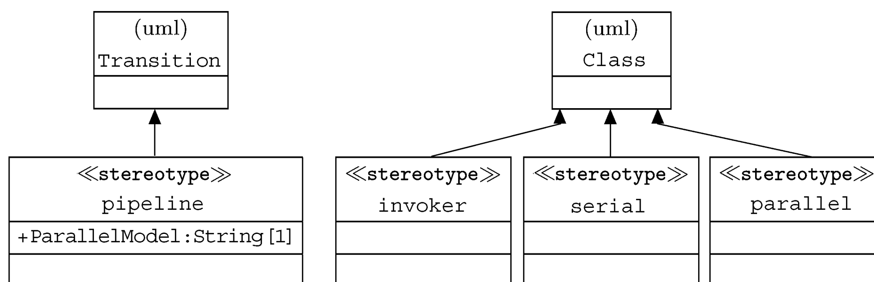

Fig.2. Stereotypes for pipeline design pattern.

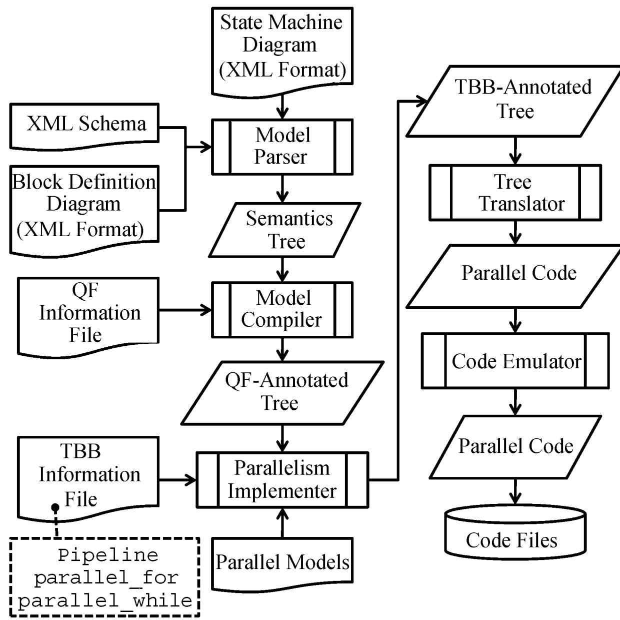

Fig.3. Multi-core embedded software code generating flow.

including model parser, model compiler, parallelism implementer, tree translator, and code emulator.

The model parser is responsible for extracting all related information from user-specified SysML design models, namely block definition diagrams and state machine diagrams which are described by the standard XML schema. The model parser creates a semantics tree structure for the model compiler.

The model compiler is in charge of constructing the backbone of system behavior, that is, the state machine behavior of each system component. The model compiler implements the state machine by the QF library, and the generated snippets of code are inserted into the semantics tree structure, resulting in a new tree structure called QF-annotated Tree.

The parallelism implementer is responsible for generating parallel multi-core code by invoking the TBB library. When a stereotype being associated with a parallel model is processed by the model compiler, the model compiler invokes the parallelism implementer. Subsequently, the parallelism implementer generates the parallel code according to the referred parallel model. The generated code snippets are inserted into the QF-annotated tree, which is then called TBB-annotated tree.

The tree translator traverses the TBB-annotated tree structure which contains both the QF code and the TBB code. It generates code files for the code emulator which in turn executes the compiled code to the target platform to test its functional and non-functional properties. A monitor system is implemented in the code emulator, to collect feedback information from the target platform.

VMC generates multi-core embedded software code automatically from the user-specified SysML state machine models via two open-source, small and lightweight libraries, namely QF and TBB. VMC realizes each SysML state machine as a QF active object by generating code that invokes QF APIs for states, transitions, and communication events. Each active object is executed by a user-level Pthread that maps to a kernel thread in Linux OS. Within an active object, each `do method` that is executed in a state, is encapsulated as a TBB task or a TBB task graph depending on the complexity of the method and its ability to be parallelized. Thus, there are basically two sets of user-level threads, namely Pthreads and TBB threads.

The distinction between these two sets of threads is mainly due to the requirement of UML state machines to satisfy the *run-to-completion* (RTC) semantics. The RTC semantics is required by both the `do methods` in a QF active object and a TBB task. A QF active object cannot be modeled as a TBB task because the active object never terminates execution and thus will violate the RTC semantics if it is a TBB task. Hence, a devoted Pthread is used instead.

Another effect of the RTC semantics is that whenever there is an indefinite polling of some I/O devices such as remote controller, the polling task cannot be a QF method nor a TBB task. VMC addresses this issue by modeling such polling tasks as an independent state machine with a single state, a self-looping transition, and a single triggering event such as data input. Such a specific state machine waits on the single event and thus there is no need to follow the RTC semantics.

Let us use an example to show several snippets of code that are generated by VMC based on machine models specified by designer. As shown in Fig.6, the parallel video encoder (PVE) is a subsystem of digital video recording (DVR) system which will be detailed later in our case study. The PVE subsystem retrieves the raw video frame for the digital camera, and this involves the issue of violation of RTC semantics. We implemented the PVE subsystem by designing two state machines, including video capture and video encoding. Video capture has only one state, namely `Capture_Frame`. The state machine keeps polling the I/O of digital camera devices and stores the raw video frame in a buffer, and then notifies the state machine, video encoding, to process the raw video frame. Thus, the issue of violation of RTC semantics is solved. The following segment of code illustrates the video encoding state machine which comprises two states, namely `RF_Notification` and `ENCODE_OK`.

```
class Video_Encoding:public QActive{public:
```

```
 PVEEncoding();
 ~PVEEncoding();
protected:
 void initial(QEvent const *e);
 QSTATE RF_Notification(QEvent const *e);
 QSTATE ENCODE_OK(QEvent const *e);
private:
 /*member functions and data are declared
 here*/
};
```

The class, `Video_Encoding`, inherits the class `QActive` which is provided by QF and can be used to realize the state machine. The class, `Video_Encoding`, maintains two states, namely `RF_Notification` and `ENCODE_OK`. The two states are realized by the QF class, `QSTATE`.

In state machine `Video_Encoding`, the transition between states, `RF_Notification` and `ENCODE_OK`, are applied three stereotypes on it, namely ≪`seral_filter`≫, ≪`parallel_filter`≫, and ≪`pipeline`≫. When the code generator reaches these stereotypes, the code generator refers to the class diagram which is a command design pattern and provided by users with respect to the stereotype. For example, when reaching the stereotype, ≪`pipeline`≫, the code generator refers to the class diagram as shown in Fig.4. In this example only two filters, `GetRF` and `DCT`, are shown to illustrate the generated code.

The snippet of code for serial filter, `GetRF`, is shown below.

```
class _GetRF:public tbb::filter{
 /*member data are declared here*/
public:
 _GetRF(usertype* ptrBuffer):
 filter(serial){
   /*initialize member data here*/
 }
 void* GetRF(void* item){
   //user manual code
 };
 void* operator(void* item){
 Token1* tokenOut =
 <static_case>(Token1*)GetRF(null);
 return tokenOut;
 }
};
```

The class, `_GetRF`, inherits the filter class provided by TBB, and its parent constructor is initialized by keyword `serial` defined by TBB. Users have to write the code in the member function `GetRF` to process data according to their objectives for the filter. For the parallel filter, the parent constructor is initialized by the keyword `parallel`.

As for the stereotype ≪`pipeline`≫, the code generator generates the following code to initialize TBB pipeline and adds filters to it.

```
tbb::task_scheduler_init init;
tbb::pipeline ppline;
_GetRF __GetRF(&container);
/*Declare and initialize
other filters here*/
ppline.add_filter(__GetRF);
/*add other filters here*/
ppline.run(numToken);
ppline.clear();
```

Firstly, the code generator initializes the TBB environment for the pipeline and then instantiates the pipeline. Subsequently, the code generator instantiates and initializes the filters that are specified in the state machine, such as the GetRF and DCT filters, and then adds these filters to the pipeline.

4.4 Validation

VMC validates models and code using different approaches. SysML models are flattened into timed

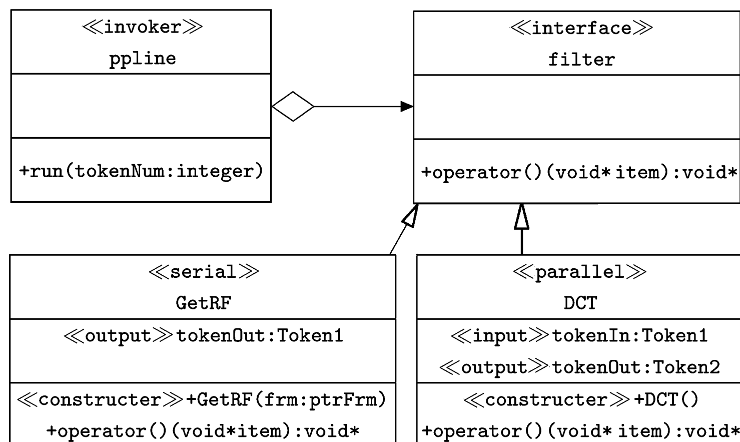

Fig.4. Parallel model for TBB pipeline.

automata models that are then integrated with VMC provided architecture and computing models for the multi-core processor environment with task migration and scheduling. The SGM model checker[20] is then used to verify the timed automata models against user-specified properties. This part is out-of-scope here and the details can be found in [22].

As far as the code validation is concerned, a testing environment is used for validating if the multi-threaded code execution satisfies user-given constraints that are specified in SysML requirements diagram. User-given constraints can be classified into two categories: *application-independent* such as CPU utilization and power consumption, and *application-dependent* such as encoding rate and streaming rate in classical multimedia applications.

For application-independent constraints, we synthesize an additional active object, called `Monitor`, to collect and record the CPU core utilizations by invoking system calls provided by the underlying operating system. The period of this invocation can be specified by the user depending on the desired resolution of data collection. The CPU core utilization statistics are written into a log file for further analysis. The estimation of power consumption by an application is evaluated in `Monitor` according to the power model proposed by Lien *et al.*[23], which is based on the CPU core utilizations of the application.

For application-dependent constraints, the information-gathering task is left to the corresponding active object to keep the relation loosely-coupled between `Monitor` and other active objects. The more loosely-coupled the relation is, the easier the `Monitor` code can be synthesized in a general way. For example, an encoder active object should calculate its encoding rate by itself because only it knows how many raw frames are encoded in a certain time period. All that the `Monitor` has to do is to ask the encoder active object about its encoding rate and write it into a log file. Thus, every active object of an application inherits a virtual function `showInfo()` from the abstract class `Information` predefined by VMC code generator. In the inherited virtual function `showInfo()`, each active object formulates whatever information it wants to pass to `Monitor` by encoding it into a character string. The `Monitor` will then invoke this function to get the information string and write it to a log file once in each specified time period. In addition, the `Monitor` maintains a list that records all the active objects of which it should call the corresponding `showInfo()` functions. Note that `Monitor` does not understand the content of the information strings obtained from other active objects; it just gets the strings and writes them to a log file. In such a way, the `Monitor` active object can be generally synthesized in VMC without manual handling.

With `Monitor` executing concurrently other active objects of an application, all the dynamic information about the application can be obtained in a log file. System designers can analyze the log file to validate the application. If the constraints are not met by the generated multi-core embedded software, he/she can refine the system model or constraints and then validate the generated multi-core embedded software again until all the specification constraints are met. For example, if the power consumption is too high, this can be observed from the log file and then the designer can try to reduce the number of cores and re-run the code generation process such that the new version of the code can be executed along with `Monitor` to check if the power consumption is reduced or not.

5 Digital Video Recording: A Case Study

We use a real-world example called *digital video recording* (DVR) system to illustrate how VMC works and the benefits of applying VMC to multi-core embedded software development. DVR is a real-time multimedia system for online and on-demand video streaming. The overall architecture of DVR is illustrated in Fig.5, which shows that DVR has two subsystems, namely

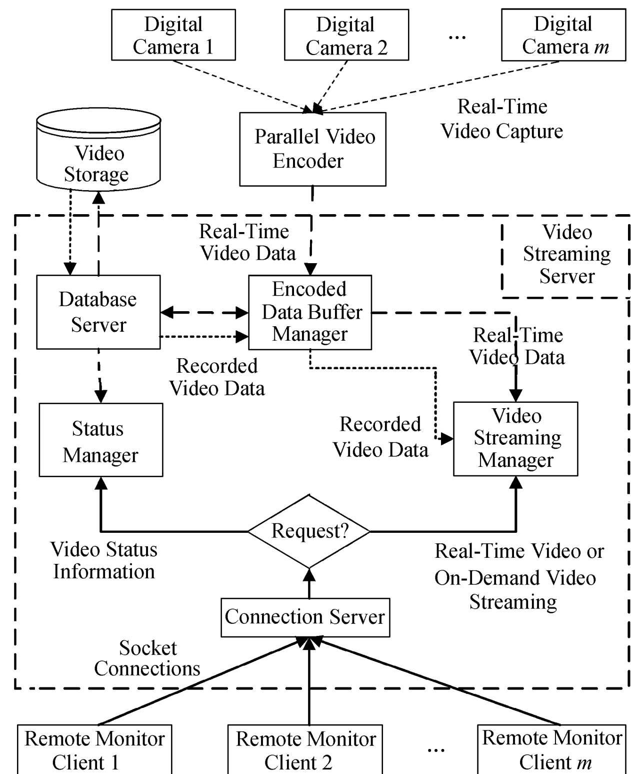

Fig.5. Architecture of digital video recording system.

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 allowing connections from multiple *remote monitor clients* (RMC), for servicing the clients with status information, real-time video streams, and on-demand video streams, and for storing the encoded video streams in large video databases.

Encoded data buffer manager (EDBM) which is a subsystem of VSS buffers the compressed data from PVE. Each buffer corresponds to each digital camera appearing in the DVR system. These buffers are accessed by database server and video streaming manager (VSM). Database server is responsible for storing the compressed data into video storage, so that remote monitor client (RMC) can request the on-demand video for playback, and status manager lists all accessible videos for the users. VSM accesses the compressed data form EDBM and streaming the buffered data to RMC when there is real-time streaming request from RMC.

In the rest of this section, we will describe how task parallelism, data parallelism, and data flow parallelism, i.e., parallel pipeline, are automatically realized in the embedded software code generated from user-specified models of the PVE. We will also describe how conventional thread parallelism is integrated into the embedded software code generated from user-specified models of the VSS.

5.1 Parallel Video Encoder

The PVE subsystem is responsible for the capturing of raw video data from all digital cameras, the encoding of the raw video from each camera into more compressed data format for efficient network transmission and for smaller storage space requirement, and the transmission of the encoded video data to the buffer manager in the VSS subsystem. In this subsection, we show how PVE is a very good illustration example for the three real-world concurrency issues[21].

5.1.1 Task Parallelism

Capturing and processing video from each camera is an independent task. However, due to the requirement of RTC semantics in UML and QF, we need to segregate the capture and the processing of the video into two different state machines as illustrated in Fig.6. The video capture state machine is devoted to capturing video from a camera, while the video encoding state machine performs the real-time encoding of video. Thus, for a set of n cameras, there are $2n$ QF active objects that are executed by $2n$ Pthreads.

5.1.2 Data Parallelism

Since video data is composed of a large number of frames and the encoding process is iteratively applied to a data block of 8×8 pixels in a frame, there is a high degree of data parallelism in video encoding. Further, since the color model of the video in DVR is RGB, with 8-bits per pixel color, the encoding process can be parallelized into a multiple of 3, that is, one set of threads for each of the three colors. For example, a frame size of 640×480 pixels consists of $80 \times 60 \times 3 = 14400$ data blocks. The degree of data parallelism can be ranged from 3 to 14 400 blocks. Allowing high parallelization might consume too much system resources and cause more timing overhead than the time saved through parallelization. Thus, a tradeoff between parallelism and resource usage is required to achieve high system efficiency. In the method for encoding, the stereotypes ≪`parallel_for`≫, ≪`parallel_while`≫, ≪`parallel_task`≫, ≪`task`≫ can all be used for parallelizing the encoding method.

5.1.3 Data Flow Parallelism

Besides the task parallelism for multiple camera video inputs and the data parallelism for multiple data blocks within each frame, we can also apply data flow parallelism to PVE because the video encoding process applied to each data block is itself a sequence of functions. For most multimedia standards such as MPEG, the sequence of functions consists of *discrete cosine transform* (DCT), *quantization* (Q), and *Huffman encoding* (HE). This sequence of functions can be parallelized as a pipeline to hide latency such that more than one data block is processed at any time instant.

In Fig.6, note how data flow parallelism is specified through the three stereotypes: ≪`pipeline`≫, ≪`serial_filter`≫, and ≪`parallel_filter`≫.

Tagged values such as `num_tokens` and `num_buffers` are specified, respectively, in the ≪`parallel_filter`≫ and ≪`pipeline`≫ stereotypes to represent the number of tokens (the TBB terminology for degree of parallelism in a parallel filter) and the maximum number of buffers (the TBB terminology for the maximum degree of parallelism in a system). The encoding pipeline in PVE has two serial filters, namely `GetRF` that gets and decomposes a raw frame for parallel processing by the parallel filters and `PutEF` that collects all encoded data blocks and composes an encoded frame for transmission to the video buffer. The three parallel filters in PVE pipeline are responsible for computing in parallel the functions: DCT, quantization, and Huffman encoding.

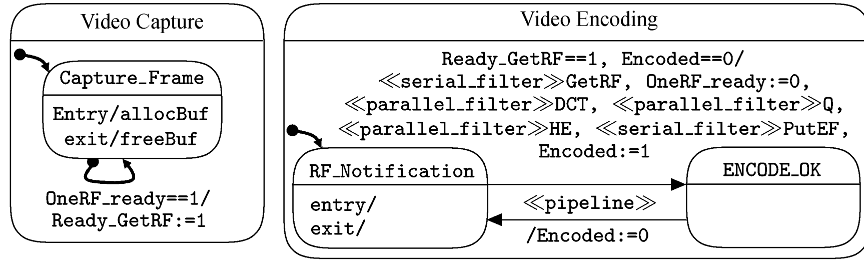

Fig.6. State machines of the parallel video encoder.

5.2 Video Streaming Server

We use the *video streaming server* (VSS) subsystem to illustrate how legacy multi-threaded software can be integrated into VMC. As a result of the integration, the threads in legacy multi-threaded software, the POSIX threads for executing QF active objects, and the TBB threads work together seamlessly. The main functions of VSS include 1) accepting multiple connections from remote clients, 2) streaming multiple real-time videos and/or on-demand videos to the remote clients, 3) providing requested server status information to the remote clients, and 4) recording the encoded videos into storage devices. The architecture of the VSS subsystem is shown in Fig.7 and the functionalities of each component in VSS are described as follows.

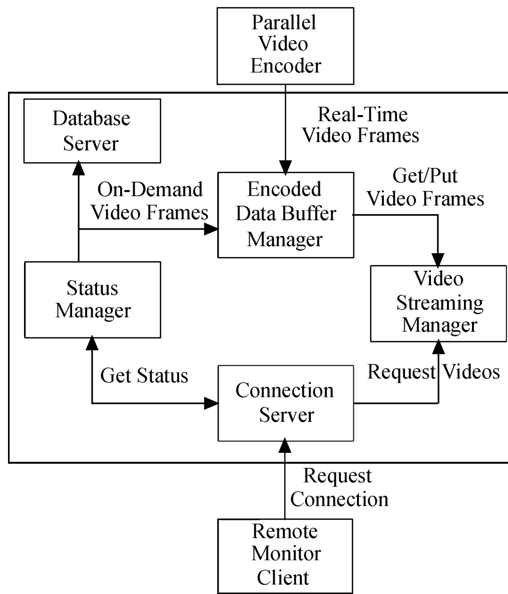

Fig.7. Architecture of video streaming server.

5.2.1 *Legacy Threads*

Legacy threads are simply multiple threads that exist in legacy software. This is illustrated in *connection server* (CS) and *video streaming server* (VSM) as follows.

The connection server is responsible for handling connections and invoking services corresponding to multiple client requests. Traditionally, this has almost always been implemented as an iterative or concurrent TCP server using either the `select` or the `fork` mechanism. The state machine for the connection server is shown in Fig.8. In the `DISPATCH` state, the server simply forks a new *legacy* thread for servicing a new request from a client. It is simply unreasonable to forsake well-established proven concurrent artifacts such as a concurrent TCP server. This example shows that the VMC framework does not *force* one to model everything for TBB or QF. The reason for not applying the TBB principle here is that the parallelism is explicitly designed into the system and it is required for providing real-time services to the clients.

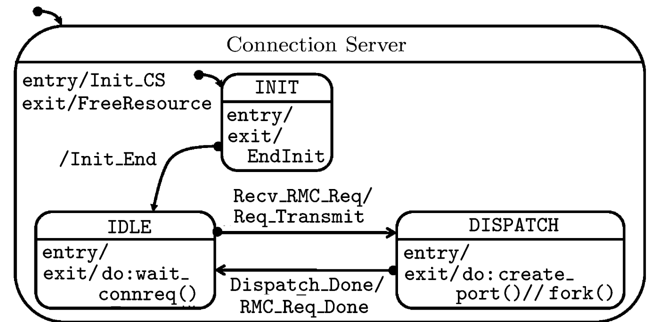

Fig.8. State machine model of the connection server.

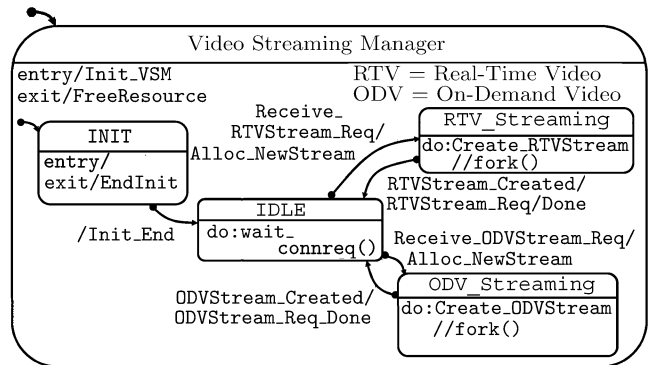

Fig.9. State machine model of the video streaming manager.

The video streaming manager is also a typical concurrent manager that creates new streams at runtime to serve client requests. In DVR, because a minimum QoS of 15 frames per second (fps) is required for video streaming, VSM manages a pool of legacy threads. The state machine of VSM is illustrated in Fig.9, where a new thread is used for servicing each new request, either for a real-time video streaming or an on-demand video streaming.

5.2.2 *TBB Tasks/Threads and QF Threads*

The VMC framework uses TBB tasks mainly for two reasons as follows: (a) a job is parallelizable, but there is no real-time constraints, or (b) a job is parallelizable, but the underlying hardware device is not. The first case is illustrated by the *status manager* (SM) and the second case by the *database server* (DS) and the *encoded data buffer manager* (EDBM). Note that the EDBM also utilizes multiple QF threads for executing concurrent states.

The status manager retrieves the list of recorded video files and the list of on-line digital video cameras from the database server and passes the information to remote clients. Since the status requests do not have real-time constraints, there is no need for devoted threads, instead, VMC realizes these jobs as TBB tasks to be executed by the TBB scheduler using a set of TBB threads. The stereotype ≪`parallel_task`≫ is used to specify request servicing as a set of parallel TBB tasks.

The database server provides recorded video files to VSM and allows storing of real-time video data from EDBM. Multiple client requests and multiple camera video inputs require the database server to be a concurrent one. However, since DVR considers a single hard-disk for database storage, allowing multiple *devoted* threads for each read or write request is unnecessary because ultimately all the requests must be serialized by the OS disk scheduler. Instead, VMC maps such read and write jobs as TBB tasks. Parallelism is still needed so that the disk accesses can be made efficient through the OS disk scheduler.

The encoded data buffer manager (EDBM) is responsible for buffering the video streams including both the real-time ones from PVE and the stored ones from the database server. In the case of real-time videos, EDBM buffers the video data, sends them to the database for storage and future retrieval, and also sends them to the remote clients through VSM. Since the EDBM buffers are physically located in the main memory which usually has a single access port, all memory accesses are, in fact, serialized at the lowest level. Thus, similar to the database server, multiple devoted threads are also unnecessary and VMC realizes these memory accesses as TBB tasks with TBB synchronization mechanisms. However, unlike the single QF thread for the database server, EDBM has a concurrent state, as shown in Fig.10, and thus two QF threads are required: one for sending the real-time videos to the database and the other for sending buffered videos to VSM.

5.3 Remote Monitor Client

The remote monitor client (RMC) allows users to interact with the DVR server through a graphical user interface in the following ways: 1) acquiring the status information of the DRV server, 2) real-time video streaming, 3) on-demand video streaming, and 4) debugging and testing. The RMC can also be used to

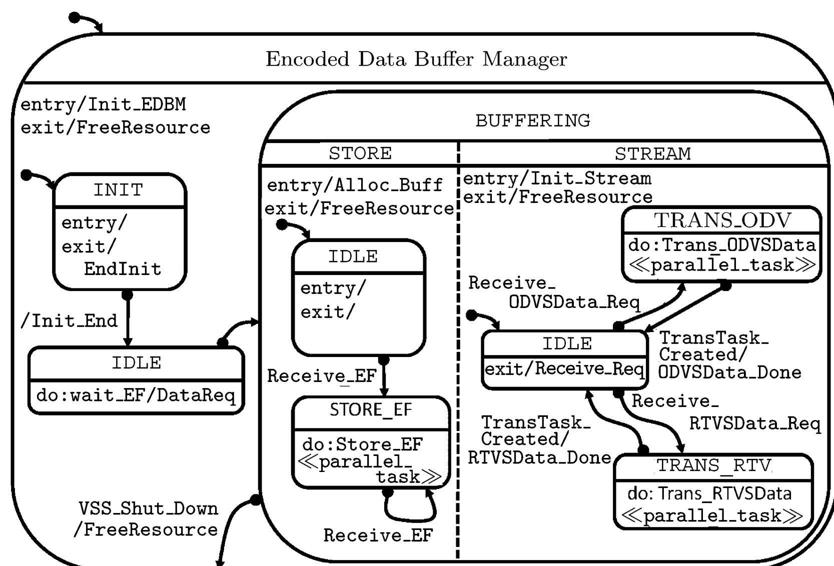

Fig.10. State machine model of the encoded data buffer manager.

gather the server performance statistics for improving the video streaming QoS guarantees.

5.4 Experimental Results

We now measure the DVR system by several metrics, including performance, power consumption, and load balancing, on two different multi-core platforms. The first platform is Intel Core2 Quad CPU Q6600 with clock frequency 2.4 GHz, 8 MB cache and 4 GB RAM. The second platform is Intel Xeon Processor E5520 with clock frequency 2.26 GHz, 8 MB cache, and 8 GB RAM. Note that each core has two hardware threads. The first platform has one processor with four cores in it, and the second one has two processors where each processor consists of four cores.

We implemented three versions of the DVR system. The first version is pure QF version, that is, this version does not parallelize the PVE encoding flow, called the QF version. For each macroblock of a raw frame, it is encoded sequentially through DCT and Quantization, and finally all macroblocks are compressed by Huffman encoding. The second version is implemented with QF and TBB pipeline being applied to the encoding flow, called the TBB version. The implementation of the third version employs two tools, including QF and OpenMP, called the OpenMP version. We use OpenMP to optimize the encoding flow in two different ways, including the coarse-grained, namely the entire encoding flow, and the fine-grained, e.g., DCT operation which consumes most of the computation time of the encoding flow.

In the first experiment, we observed the load balancing of each version. We compared the QF version and TBB versions, in which the system configuration was of four cores, one real-time streaming, two cameras, and the capture rate ranging from 16 to 20 frames per second (fps). The time unit of a system tick is 1/8000 seconds in all the experiments in this paper. The target platform is Intel Core2 Quad CPU Q6600. As shown in Fig.11, the TBB version achieves better load balancing among cores due to the random task stealing between the TBB threads.

In the second experiment, we measured the performance of DVR in terms of the frame encoding rate in frames per second (fps). The configuration of this experiment is the same as in the first experiment. As illustrated in Fig.12, the average encoding rate of QF version is 12 fps, while the average encoding rate in the TBB version is about 16 fps. The superior performance in the TBB version is due to the TBB pipeline code generated by VMC into the TBB version of DVR after the designer associated a pipeline stereotype to the PVE state machine. In the TBB pipeline, concurrent tasks are used to process (encode) more than one block of a raw frame at the same time. Thus, TBB pipeline reduces the time for encoding each raw image frame.

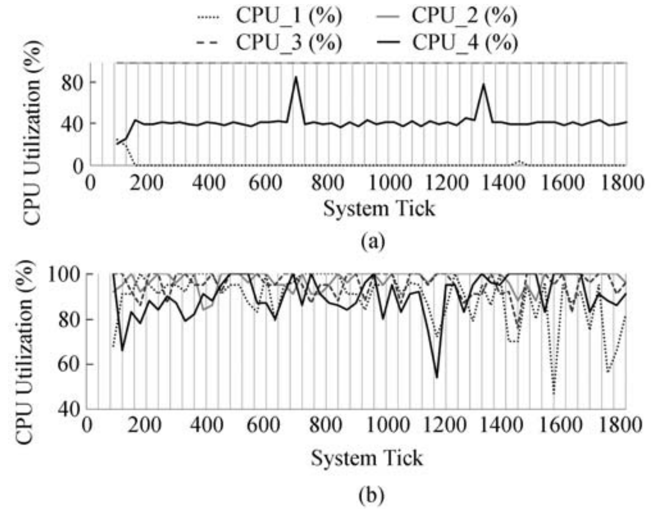

Fig.11. Comparison of load balance in Intel Core2 Quad CPU. (a) QF version. (b) TBB version.

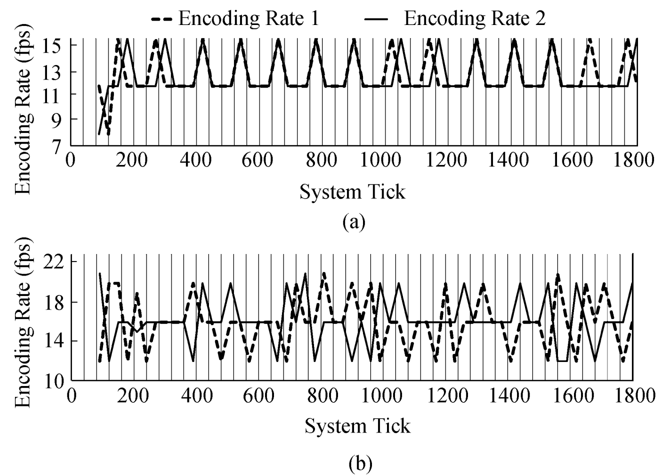

Fig.12. Performance of encoding rate in Intel Core2 Quad CPU. (a) QF version. (b) TBB version.

We then experimented with different comparisons of the second experiment such as reducing the number of cores from four to two and three. We realized this by limiting all the QF and TBB threads to using only two or three cores through the Linux system call, that is `sched_setaffinity`. With three cores, the average encoding rate of the QF version was still 12 fps, but that of the TBB version dropped to 6.13 fps. With two cores, the encoding rate of the QF version dropped to 9.78 fps, and that of the TBB version was 6.58 fps. We can observe that the less the number of cores used, the worse the performance provided by TBB. Because TBB introduces the overhead of splitting and merging

parallel tasks. The phenomenon is more obvious when the number of cores is reduced. As for the streaming rate in all experiments aforementioned, the streaming rate is consistent with the encoding rate, because the streaming tasks are I/O bound.

The third experiment shows the power consumption of the QF and TBB versions. We evaluated the power consumption according to the core utilization. We adopted the power model which is illustrated by Lien *et al.*[23], the following equation:

$$P = D + (M - D) \cdot \alpha U^{\beta} \quad (1)$$

where P represents the average power consumption in Watts, D represents base power in Watts when core is idle, M represents the the full-load power consumption in Watts, U is for the core utilization, α and β are platform-specific parameters set to be 1 and 0.5, respectively. We also set D to 69 Watts and M to 105. The configuration of this experiment is the same as the first one with four cores, one real-time streaming, two cameras, and capture rate of 16 to 20 fps.

As shown in Fig.13, the TBB version consumes totally 24 094 Watts during a period of 1800 system ticks (0.22 seconds), while the QF version consumes only 20 140 Watts. The TBB version consumes 3954 Watts more than the QF version. TBB keeps all CPU cores busy while performing the load-balancing, thus it also consumes more energy than the QF version. Nevertheless, the performance is also enhanced as shown in the first experiment.

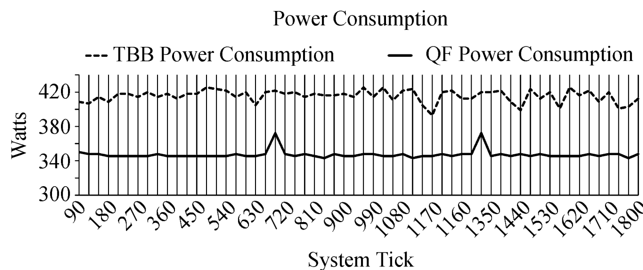

Fig.13. Comparison of power consumption in Intel Core2 Quad CPU.

We now compare another different implementation of the DVR system, namely the OpenMP version, to the TBB version. The following experiments were performed on Intel Xeon Processor E5520.

The first experiment shows several different optimizations for the DVR system by OpenMP. We choose the one with the best optimization to compare with the TBB version. In the beginning, we focus on the optimization of the DCT function for each macroblock (an eight by eight integer matrix). DCT processes each macroblock by firstly transforming each column and then transforming each row, and each transformation consists of 3-level of nested loops. We applied the OpenMP pragma `parallel for` to the column or row transformations separately or simultaneously, with assigning different number of threads. The snippet of code is as follows, where the number of threads is set to 4.

```
//Column transform
#pragma omp parallel \
for private (i,j,k) num_threads(4)
for(j=0;j<8;j++)
 for (i=0; i<8; i++){
  temp[i][j]=0;
  for (k=0; k<8; k++)
   temp[i][j]+=column[i][k]*pInput[k][j];
}
```

Another optimization is using task parallelism where each task consists of entire encoding flow, including DCT, quantization, and Huffman encoding, by applying the OpenMP directive *#pragma omp sections*.

The system configuration includes two digital cameras, one real-time streaming, and employing all cores. We found that the 4-thread configuration has better optimization than the others for the row and column transformations in terms of the encoding rate in fps.

Fig.14 shows that the optimization of row transformation is better than that of column transformation, because the optimization of raw transformation can

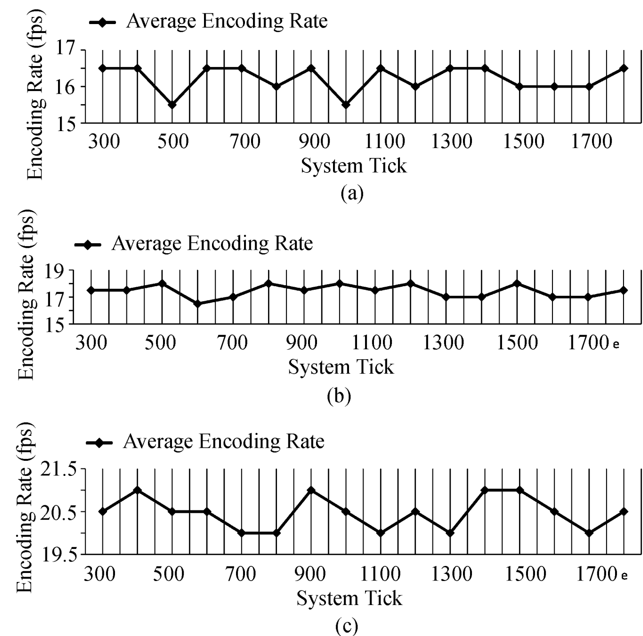

Fig.14. DCT and encoding flow optimization with OpenMP. (a) OpenMP version (column transformation with four threads). (b) OpenMP version (row transformation with four threads). (c) OpenMP version (#pragma sections for encoding flow).

reach an average encoding rate around 18 fps, while the column transformation optimization results in only 16 fps. This is due to the cache locality, that is, the raw transformation incurs fewer cache misses than the column transformation. When implementing optimizations for both row and column transformations, the average encoding rate is still limited to 16 fps or less due to the fact that the bottleneck of the DCT function is in column transformation.

The bottom figure in Fig.14 shows the performance of parallelizing entire encoding flow at the task level. The average encoding rate is about 21 fps which is better than that of optimized row transformation. The reason is that the small task parallelism in row transformation incurs higher scheduling overhead than the task parallelism on the entire encoding flow level, which task is large enough to benefit from the overhead.

Now let us consider the load balancing and performance of the TBB version shown in Fig.15. We can observe that the load balance is better than those implemented by OpenMP, which only eight cores reach 100% CPU utilization in the optimized row and column transformation in DCT, and four cores reach 100% CPU utilization in the optimized entire encoding flow. The average encoding rate of the TBB version is about 30 fps which is better than any other experiments in this paper. However, we must notice that the higher the core utilization is, the more the power is consumed.

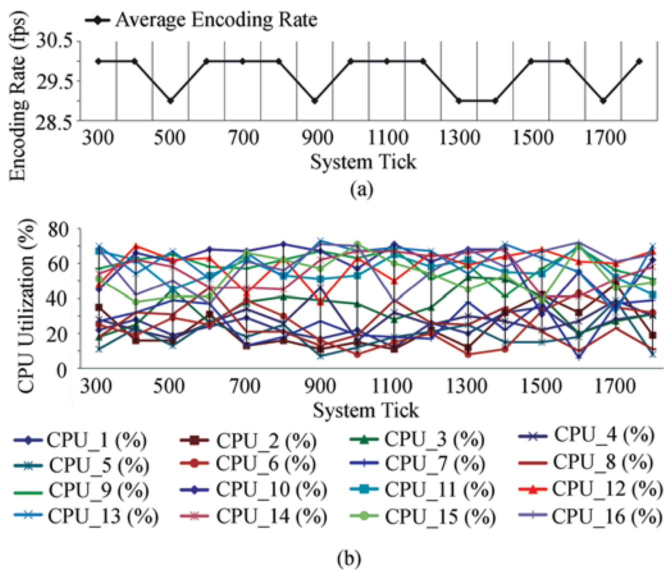

Fig.15. TBB pipeline implementation in Intel Xeon Processor E5520. (a) TBB version (Intel Xeon Processor E5520). (b) TBB version (Intel Xeon Processor E5520).

6 Conclusions

In this paper, we proposed a novel framework VERTAF/Multi-Core (VMC) for the system designer to easily design and test their application on multi-core systems. We illustrate how we address the problems (RTC semantics violation) arising from the gap between model and implementation. A parallel model, namely *command design pattern*, is proposed for the designer to be able to reuse the model and exploit the computing power of multi-core. Additionally, we also illustrated how we designed the code generating flow to automatically generate parallel code from models. In future work, we will focus on the feedback and modification of design flow, which are important issues in the software engineering process.

References

[1] Akhter S. Multi-Core Programming: Increasing Performance Through Software Multi-Threading. Intel Press, 2006.

[2] OpenMP. http://www.openmp.org/, 2008.

[3] Intel Inc. http://software.intel.com/en-us/articles/intel-cilk-plus/, 2010.

[4] Reinders J. Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly Media, Inc., 2007.

[5] Hsiung P A, Lin S W, Tseng C H, Lee T Y, Fu J M, See W B. VERTAF: An application framework for the design and verification of embedded real-time software. *IEEE Transactions on Software Engineering*, Oct. 2004, 30(10): 656-674.

[6] Rumbaugh J, Booch G, Jacobson I. The UML Reference Guide. Addison Wesley Longman, 1999.

[7] Samek M. Practical StateCharts in C/C++. CMP Books, 2002.

[8] Lee E A. The problem with threads. *IEEE Computer*, May 2006, 39(5): 33-42.

[9] UML. http://www.omg.org/gettingstarted/what_is_uml.htm, 2010.

[10] SysML. http://www.omgsysml.org/, 2010.

[11] Model driven development–simplifying multicore systems deployment. Technical Report, IBM Corporation Software Group, October 2009.

[12] de Niz D, Rajkumar R. Time Weaver: A software-through-models framework for embedded real-time systems. In *Proc. LCTES 2003*, San Diego, USA, Jun. 11-13, 2003, pp.133-143.

[13] Kodase S, Wang S, Shin K G. Transforming structural model to runtime model of embedded real-time systems. In *Proc. the Design Automation and Test in Europe Conference*, Munich, Germany, Mar. 3-7, 2003, pp.170-175.

[14] Wang S, Kodase S, Shin K G. Automating embedded software construction and analysis with design models. In *Proc. the International Conference of Euro-uRapid*, Frankfurt, Germany, Dec. 2-3, 2002, pp.A/5.1-A/5.6.

[15] Piel E, Ben Atitallah R, Marquet P, Meftali S, Niar S, Etien A, Dekeyser J L, Boulet P. Gaspard2: From MARTE to systemc simulation. In *Workshop on Modeling and Analyzis of Real-Time and Embedded Systems with the MARTE UML Profile* (*DATE 2008*), March 2008.

[16] Rioux L, Saunier T, Gerard S, Radermacher A, de Simone R, Gautier T, Sorel Y, Forget J, Dekeyser J L, Cuccuru A, Dumoulin C, Andre C. MARTE: A new profile RFP for the modeling and analysis of real-time embedded systems. In *Workshop UML for SoC Design* (*DAC 2005*), June 2005.

[17] Bader D, Kanade V, Madduri K. SWARM: A parallel programming framework for multi-core processors. In *Proc. IPDPS 2007*, Long Beach, USA, Mar. 26-30, 2007, pp.1-8.

[18] Perez J, Bellens P, Badia R, Labarta J. Cellss: Making it easier to program the cell broadband engine processor. *IBM Journal of Research and Development*, 2007, 51(5): 593-604.

[19] Wagner J, Jahanpanah A, Traff J. User-land work stealing schedulers: Towards a standard. In *Proc. CISIS 2008*, Mar. 4-7, 2008, pp.811-816.

[20] Wang F, Hsiung P A. Efficient and user-friendly verification. *IEEE Transactions on Computers*, January 2002, 51(1): 61-83.

[21] Cantrill B, Bonwick J. Real-world concurrency. *ACM Queue*, September 2008, 6(5): 16-25.

[22] Tsao C C. An efficient collaborative verification methodology for multiprocessor SoC with run-time task migration [Master's Thesis]. "National Chung Cheng University", July 2008.

[23] Lien C H, Bai Y W, Lin M B. Estimation by software for the power consumption of streaming-media servers. *IEEE Transactions on Instrumentation and Measurement*, October 2007, 56(5): 1859-1870.

Chao-Sheng Lin received the B.S. degree in architecture and urban design from Chinese Culture University, Taipei, China, in 1998, and the M.S. degree in the Department of Computer Science and Information Engineering from "National Chung Cheng University", Chiayi, Taiwan, China, in 2007. He is now working toward the Ph.D. degree in the Department of Computer Science and Information Engineering at "National Chung Cheng University". He has two-year working experience in software engineering and had been the vice senior software engineer in Synchronous Communication Corp. His research interests include formal verification, reconfigurable systems, and multi-core software engineering.

Chun-Hsien Lu received the B.S. degree in computer science and information engineering from the "National Chung Cheng University", in 2006. Currently, he is a Ph.D. candidate in computer science and information engineering of "National Chung Cheng University", China. His main research interests include reconfigurable computing, network on chip, multi-core embedded system.

Shang-Wei Lin received his B.S. degree in management information system in 2003 and his Ph.D. degree in computer science and information engineering in 2010 both from "National Chung Cheng University". Currently, he is a post-doctoral researcher at School of Computing, National University of Singapore. His research interests include formal verification, formal synthesis, scheduling, and object-oriented software synthesis.

Yean-Ru Chen received the B.S. degree in computer science and information engineering from the "National Chiao Tung University", Hsinchu, Taiwan, China in 2002. From 2002 to 2003, she was employed as an engineer in SoC Technology Center, Industrial Technology Research Institute, Hsinchu, Taiwan, China. She received the M.S. degree in computer science and information engineering from the "National Chung Cheng University", in 2006. She is currently a Ph.D. candidate in Graduate Institute of Electronics Engineering of "National Taiwan University", Taipei, China. Her current research interests include model checking, safety-critical systems, security-critical systems, electronic system level (ESL) design and multi-core embedded software.

Pao-Ann Hsiung received his B.S. degree in mathematics and his Ph.D. degree in electrical engineering from the "National Taiwan University", in 1991 and 1996, respectively. Since 2007, he has been a full professor in the Department of Computer Science and Information Engineering, "National Chung Cheng University". He has published more than 200 papers in international journals and conferences. He was a recipient of the 2010 Excellent Research Award and the 2004 Young Scholar Research Award, "National Chung Cheng University", and the 2001 ACM Taipei Chapter Kuo-Ting Li Young Researcher award. He is a senior member of the IEEE and the ACM, and a life member of the IICM. He is on the editorial board of several international journals and on the program committee of more than 80 international conferences. His main research interests include reconfigurable computing and system design, multi-core programming, cognitive radio architecture, embedded design and verification, embedded software synthesis and verification, real-time system design and verification, hardware-software codesign and coverification, and component-based object-oriented application frameworks.