PAPER

A Verification-Aware Design Methodology for Thread Pipelining Parallelization

Guo-An JIAN[†a)], Cheng-An CHIEN[†b)], *Nonmembers*, Peng-Sheng CHEN[†c)], *Member*, *and* Jiun-In GUO[††d)], *Nonmember*

SUMMARY This paper proposes a verification-aware design methodology that provides developers with a systematic and reliable approach to performing thread-pipelining parallelization on sequential programs. In contrast to traditional design flow, a behavior-model program is constructed before parallelizing as a bridge to help developers gradually leverage the technique of thread-pipelining parallelization. The proposed methodology integrates verification mechanisms into the design flow. To demonstrate the practicality of the proposed methodology, we applied it to the parallelization of a 3D depth map generator with thread pipelining. The parallel 3D depth map generator was further integrated into a 3D video playing system for evaluation of the verification overheads of the proposed methodology and the system performance. The results show the parallel system can achieve 33.72 fps in D1 resolution and 12.22 fps in HD720 resolution through a five-stage pipeline. When verifying the parallel program, the proposed verification approach keeps the performance degradation within 23% and 21.1% in D1 and HD720 resolutions, respectively.

key words: *verification, 3D depth map generation, pipeline, parallel computing, behavior model*

1. Introduction

Multicore systems provide powerful computing capabilities that can improve the performance of applications, and such systems have been widely adopted in desktop workstations and consumer electronic devices. However, an application must be parallelized to obtain the performance benefits of multicore systems. Many design patterns for parallel programming have been proposed to exploit the advantages of parallelism [1]. According to the characteristics of the applications, the programmer selects suitable patterns with which to develop and parallelize the applications. Thread pipelining [2], which is one such pattern, processes data through a sequence of stages to extract the potential concurrency (Fig. 1). Assume that D_1–D_7 represents the sequence of data to be processed, and each datum must be processed in the correct order through three stages of the pipeline. Dark gray areas in the figure represent the time taken for waiting, synchronization, and data transfer among the stages. First, stage1 starts to process D_1. After stage1 has finished processing D_1, it sends the result to stage2 and notifies stage2 that it can start processing D_1. Stage1 then processes the next datum D_2, while stage2 processes D_1. Then, stage3 receives and processes D_1, and stage1 and stage2 respectively process D_3 and D_2. For the execution model, the stages in a pipeline can be executed in parallel during most of the processing period. Theoretically, the speed at which an application is run can be increased by close to N times for an N-stage pipeline with sufficient input data. In addition to input data, two factors have a strong effect on the performance of an application constructed by thread pipelining. The first is synchronization among pipeline stages, which ensures the sequence of data is processed in the expected order. The second is the processing time of the pipeline stages, because given that the data are processed based on the order of pipeline stages, each pipeline stage must wait if the previous or next stage has not finished the process. Therefore, the pipeline stage with the longest execution time has a disproportionate influence on the performance of the entire pipeline. To overcome this problem, the pipeline should be appropriately partitioned into several stages with similar execution times, to ensure optimal performance.

The design pattern of thread pipelining implicitly contains task- and data-parallel concepts. Each stage in a pipeline can be individually optimized and parallelized, and seamlessly integrated into the pipeline to enhance the pipeline performance. Many domain-specific applications can be well parallelized using this pattern to realize improvements in speed. Applications related to digital signal processing (DSP) perform a variety of signal processing operations (e.g., filtering and DSP algorithms) on input sensor data. Multimedia-related applications usually apply a sequence of transformations and operations to input streaming data. Each of these applications can be viewed as one or several pipelines, and the transformations and operations that occur during processing can be formulated as pipeline

Manuscript received November 18, 2011.
Manuscript revised June 3, 2012.

[†]The authors are with the Department of Computer Science and Information Engineering, National Chung Cheng University, ChiaYi, Taiwan 621.
[††]The author is with the Department of Electronics Engineering, National Chiao Tung University, HsinChu, Taiwan 300.
a) E-mail: chienka@mail.cs.ccu.edu.tw
b) E-mail: cca95m@cs.ccu.edu.tw
c) E-mail: pschen@cs.ccu.edu.tw
d) E-mail: jiguo@nctu.edu.tw

DOI: 10.1587/transinf.E95.D.2505

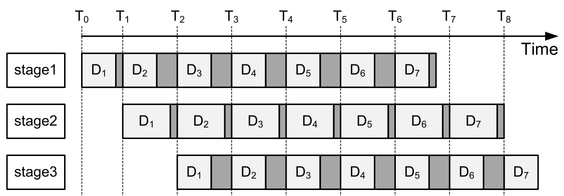


Fig. 1 Concept of thread pipelining.

stages.

Generally, parallel programming is complicated and error-prone [3]. Therefore, despite the numerous benefits that thread pipelining can offer, it remains problematic to reliably verify the correctness of an application parallelized by this approach. Many previous studies have considered verification issues related to parallel programming. Previous verification approaches can be roughly classified into two types: mathematical and formal verifications. We briefly describe these approaches as follows:

(1) **Mathematical approach:** A parallel program can be described and modeled by a sequence of formal statements. According to the theorem proving and induction principles, we can mathematically verify whether a parallel program is correct [4], [5].
(2) **Formal verification:** The target design is modeled through a modeling language such as UML (Unified Modeling Language) [6]–[9]. Then, the designer can verify the target design by using verification tools.

In recent years, behavior-modeling techniques have been widely used in several research areas, including the verification of embedded systems [10], [11], and hardware–software co-design [12]–[15]. According to the concept of behavior-modeling techniques, a behavior model is introduced as a bridge between an original program and the parallel program, for the purpose of verification. It also serves as a firewall that appropriately prevents the bugs caused in the design stage from spreading to the implementation stage during the parallelization. In this paper, we propose a verification-aware methodology to help developers to correctly parallelize applications by using thread pipelining. Replacing traditional verification approaches that directly verify parallel programs, the proposed methodology divides the development of a parallel program into the behavior and parallel phases. In the behavior phase, the original program is appropriately transformed into the corresponding behavior-model program, which sequentially executes and mimics the actions of the final parallelized program. In the parallel phase, the parallel program is intuitively derived from the behavior-model program. The proposed methodology has the advantage of integrating verification mechanisms into the design flow. During verification, the behavior-model program is concurrently executed with the original program or its parallel program to automatically verify the results. To demonstrate the practicality of the proposed methodology, we applied it to the parallelization of a 3D depth map generator with thread pipelining. The parallel 3D depth map generator was further integrated into a 3D video playing system for evaluation of the verification overheads of the proposed methodology and the system performance. The results show the parallel system achieved 33.72 fps in D1 resolution and 12.22 fps in HD720 resolution through a five-stage pipeline. When verifying the parallel program, the proposed verification approach keeps the performance degradation within 21% and 22% in D1 and HD720 resolutions, respectively.

1.1 Contribution

This paper makes the following contributions:
(1) **Verification methodology:** A verification-aware design methodology is presented that helps programmers to identify program bugs and to develop correct parallel programs. Compared to the existing formal verification and mathematical researches, it has the following advantages: (a) Low learning threshold for beginners. The proposed methodology does not rely on any metalanguages (formal specification languages) or software tools for the verification. It has a lower learning threshold, so developers can easily perform the program parallelization and verification. (b) No state-space explosion. For the formal verification and mathematical approach, programs to be verified are modeled by metalanguages or are formally specified by mathematical formulas. Then state machines are used to simulate for the verification. The state-space explosion might occur due to the enormous increment of the number of states in parallel programs. The proposed methodology uses behavior-model programs to replace the detailed state simulation. (c) Low risk of starting again from scratch. The existing verification researches only focus on the verification which is one of the last stages during software development. If developers find bugs which are resulted from design errors, they will be forced to go back to the design stage to re-design the parallel programs. This situation will significantly hurt software development productivity. The proposed methodology integrates the design and verification flows of program parallelization into a standard operation procedure (SOP). By using the proposed methodology, developers are forced to face the verification at the design stage, and the software productivity and reliability of developing parallel programs can be improved according to software engineering literatures.

The proposed methodology also suffers two overheads. First, the behavior-model program used in the proposed methodology is required to be verified. Then the verified behavior-model program is used to help the verification of the parallel program. Although developers need to do the verification of the behavior-model program, they can thoroughly understand the behavior of the parallel program during the verification. This experience can help developers to efficiently construct the parallel program. Second,the proposed methodology does not aim to prove total correctness of parallel programs. Its goal is to quickly find out most of the bugs while parallelizing sequential programs and to build up workable parallel programs. For critical systems, developers need to perform deductive verification in order to guarantee the total correctness of software components.
(2) **Case study:** The feasibility and practicality of the proposed methodology is demonstrated by applying the approach to developing a parallel 3D depth map generator, which is further integrated into a 3D video playing system for evaluation of the verification overheads of the proposed methodology and the system performance. The experimen-

tal results are presented, showing an improvement in the performance of the 3D video playing system and in the verification overheads of the proposed methodology.

The remainder of this paper is organized as follows. Section 2 presents the proposed verification-aware design methodology, and Sect. 3 describes how to apply the proposed verification mechanism to a parallel 3D depth map generator, and reports the results of simulations and performance measurements. Finally, the conclusions are presented in Sect. 4.

2. Verification-Aware Methodology

Generally, for the parallelization of applications, developers intuitively rewrite the sequential program into the corresponding parallel version using techniques in which they are skilled. However, this approach carries a high risk of errors. In addition, developers may encounter problems during debugging, even though they can quickly parallelize the program using their advanced coding skills. In contrast to the traditional design procedures, we propose a verification-aware methodology to help developers to correctly parallelize applications by using thread pipelining. Figure 2 shows the whole design flow that is used to parallelize an application with the proposed verification-aware methodology. Of note, the verification mechanisms are integrated into the design flow. We leverage a behavior-model program as a bridge for gradually and correctly parallelizing the original program. The proposed methodology divides the development of the parallel program into the behavior and parallel phases. In the behavior phase, the original program is appropriately transformed into the corresponding behavior-model program, which sequentially executes and mimics the actions of the final parallelized program. To prevent the construction of an incorrect behavior model, behavior-level verification is performed to ensure consistency in functionalities between the original program and the behavior-model program. After building the certified behavior-model program, we enter the parallel phase, in which developers start to construct the parallel program based on the certified behavior-model program, and parallel-level verification is performed to fix bugs in the parallel program.

Below, we describe how to model the behaviors of pipelining execution, and we provide detailed descriptions of behavior- and parallel-level verification.

2.1 Behavior Model

The behavior model plays an important role during the process of parallelization because it is a bridge that helps to reduce the coding complexity when directly writing parallel programs from sequential programs. The construction of a behavior-model program involves three steps. The first step is data packetization. For thread pipelining parallelization, data transfer among pipeline stages is usually a major source of synchronization problems. During this step, all the related data required by pipeline stages are appropriately grouped into a packet that is delivered among the pipeline stages to simplify the construction of a pipeline and to avoid redundant interference. Figure 3 shows a pseudo code that demonstrates the technique of data packetization. Assume func1()–funcn() perform the orderly processing of a stream of data. The original program uses shared data to communicate among func1()–funcn(), as shown on the left side of Fig. 3; the program that uses data packets is shown on the right side of the figure. A structure data type, PACKET, can be appropriately defined to form a data packet that contains the necessary data for func1()–funcn(). func1()–funcn() are also suitably modified to operate on the data packets.

The second step in the construction of a behavior-model program is task partitioning, which decides the number of stages in the pipeline. According to the pattern of thread-pipelining, the entire computation must be divided into several tasks so that all the tasks can concurrently operate on different data. These tasks conceptually represent the stages in the pipeline. The workload of each task should be balanced in order to maximize the performance of the whole pipeline. Figure 4 shows a pseudo code that partitions the original works, func1()–funcn(), into m stages: behavior_stage1()–behavior_stagem().

The third step is behavior pipelining, which involves the construction of a behavior-model program to sequentially and faithfully simulate the execution of the pipeline built by the final parallelized program. Here, we focus on

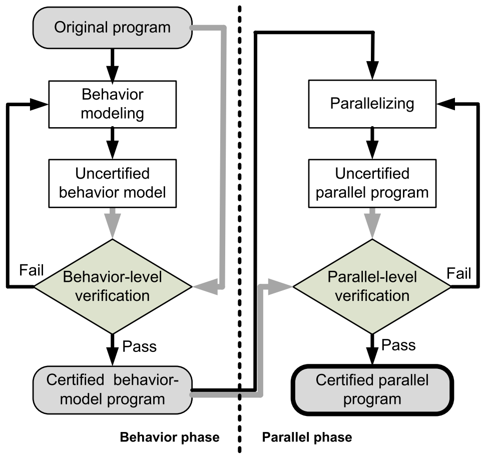
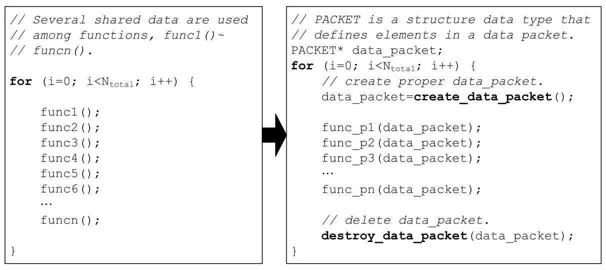

Original program
Behavior modeling
Uncertified behavior model
Behavior-level verification
Fail
Pass
Certified behavior-model program
Parallelizing
Uncertified parallel program
Parallel-level verification
Fail
Pass
Certified parallel program
Behavior phase
Parallel phase

Fig. 2 Design flow of the proposed verification-aware methodology.



```
// Several shared data are used
// among functions, func1()~
// funcn().

for (i=0; i<N_total; i++) {

    func1();
    func2();
    func3();
    func4();
    func5();
    func6();
    ...
    funcn();

}
```

```
// PACKET is a structure data type that
// defines elements in a data packet.
PACKET* data_packet;
for (i=0; i<N_total; i++) {
    // create proper data_packet.
    data_packet=create_data_packet();

    func_p1(data_packet);
    func_p2(data_packet);
    func_p3(data_packet);
    ...
    func_pn(data_packet);

    // delete data_packet.
    destroy_data_packet(data_packet);
}
```

Fig. 3 Pseudo code at the step of data packetization.

```
// Functions func1()~funcn() are properly
// divided into behavior_stage1() ~
// behavior_stagem().

PACKET* data_packet;
for (i=0; i<Ntotal; i++) {
    // create proper data_packet.
    data_packet=create_data_packet();

    behavior_stage1(data_packet);
    behavior_stage2(data_packet);
    ...
    behavior_stagem(data_packet);

    // delete data_packet.
    destroy_data_packet(data_packet);
}
```

Fig. 4 Pseudo code at the step of task partitioning.

```
// initialize data_packet1~data_packetm to NULL.
PACKET* data_packet1=NULL, ···, data_packetm=NULL;

for (i=0; i<(m+(Ntotal-1)); i++) {
    // create proper data_packet.
    if (i<Ntotal)
        data_packet1=create_data_packet();
    else
        data_packet1=NULL;

    if (data_packet1 != NULL)
        behavior_stage1(data_packet1);
    if (data_packet2 != NULL)
        behavior_stage2(data_packet2);
    if (data_packet3 != NULL)
        behavior_stage3(data_packet3);
    ...
    if (data_packetm != NULL)
        behavior_stagem(data_packetm);

    // delete data_packet.
    if (i>=m) destroy_data_packet(data_packetm);

    // simulate delivery of data_packet.
    data_packetm = data_packetm-1;
    ...
    data_packet3 = data_packet2;
    data_packet2 = data_packet1;
}
```

Fig. 5 Pseudo code at the step of behavior pipelining.

the behavior of the pipeline. To mimic the pipeline execution, we add several statements to control the execution of the stages. Each function no longer performs its computation over the same data packet, and the loop iteration has to be changed to $N_{stage} + (N_{total} - 1)$, where N_{stage} is the number of stages in the pipeline and N_{total} is the number of input data. Figure 5 shows the pseudo code at the step of behavior pipelining for the case shown in Fig. 4.

2.2 Verification

Here, we describe the verification mechanisms that are integrated into the proposed verification-aware methodology. As shown in the rhombuses in Fig. 2, the verification process is required to perform during the behavior and parallel phases. The behavior-level verification assesses the consistency of functionalities between the behavior-model and the original programs, and the parallel-level verification assesses the consistency of functionalities between the behavior-model and parallel programs. To improve the practicability, the behavior- and parallel-level verifications are run synchronously, which means that the behavior-model program is concurrently executed with the original program or parallel program to automatically verify the results. This strategy results in a much higher likelihood of identifying bugs in the application.

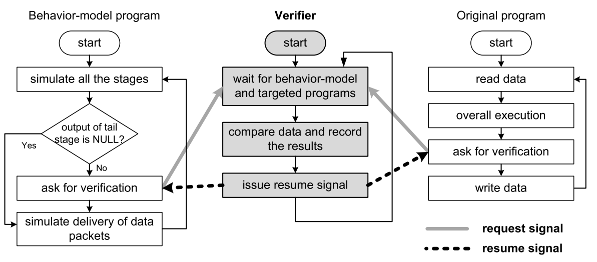


Fig. 6 Working flow of behavior-level verification.

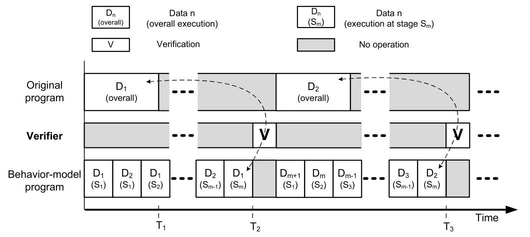


Fig. 7 Timing diagram of behavior-level verification.

2.2.1 Behavior-Level Verification

Figure 6 shows the workflow of behavior-level verification. The behavior-model program is set as the target to be verified, and the original program is set as a comparison that provides reliable data for verification. During the behavior-level verification, the original program does not process the next data until it has finished processing the current data. Therefore, verification is not started until the first processed data have passed completely through the pipeline. A verifier routinely waits for request signals from the behavior-model and original programs, compares data, records the results of comparisons and status information, and issues resume signals back to the behavior-model and original programs. Figure 7 shows a timing diagram in which it is assumed that the original program, the behavior-model program, and the verifier are executed by individual threads. An overall execution on data n represents the execution of all pipeline-related computations regarding these data. For the case shown on the right-hand side of Fig. 3, the pipeline-related computations are func1()–funcn(). At timestamp T_1, the original program finishes the overall execution on the data D_1; meanwhile, the data D_1 are processed at stage2 for the behavior-model program. The verifier must wait until the behavior-model program completes the process on D_1. At timestamp T_2, the behavior-model program completes the process on D_1 and then the verifier is signaled to start verifying the current results. After finishing the verification, the original and behavior-model programs resume processing the next data.

2.2.2 Parallel-Level Verification

For simplicity, we assume that the parallel program contains a pipeline with several pipeline workers. Each pipeline worker is dedicated to one pipeline stage, and each pipeline stage can be only assigned to one pipeline worker. Because the mapping between physical processor cores and pipeline workers is flexible, this assumption does not result in any loss of generality for the approach. Figure 8 shows the workflow of parallel-level verification. The parallel program is set as the target to be verified, and the certified behavior-model program is set as a comparison that provides reliable data for verification. The verifier and behavior-model program are concurrently executed with all pipeline workers. When a pipeline worker finishes the work on one data packet, it pauses the execution and issues a request signal to the verifier. Similarly, the behavior-model program simulates the behavior of pipeline execution. Given that the data packets from all pipeline stages must be verified, the verifier may spend a lot of time checking the data. If the verifier focuses on the parts where data are written-out, the efficiency of the checking data is significantly improved. Figure 9 shows a timing diagram for parallel-level verification. At timestamp T_1, the pipeline work1 finishes work on D_1, and as does the behavior-model program. The verifier starts the verification and then issues the resume signal. Next, due to parallel execution, pipeline worker1 and pipeline worker2 finish work on D_2 and D_1 before the behavior-model program finishes the same work. Therefore, the verifier is required to wait and cannot start the verification until timestamp T_2.

2.3 Software Design Change

Here we describe the issues relating to software design change. When the design of data packets or pipeline stages is changed, developers should go through the proposed design flow again in order to ensure the correctness of functionality of parallel programs. For these cases, developers should re-write the behavior-model program and then perform behavior-level verification to get a certified behavior-model program. Then developers have to re-construct the parallel program based on the revised behavior model. Finally parallel-level verification should be performed to check consistency of functionality between the behavior model and parallel programs. Although developers should go through the proposed design flow when the design of data packets or pipeline stages is changed, it does not mean everything has to be re-done. The efforts about thread management and synchronization can be completely reused. In addition, the development experience can be passed down to reduce software development time and related cost.

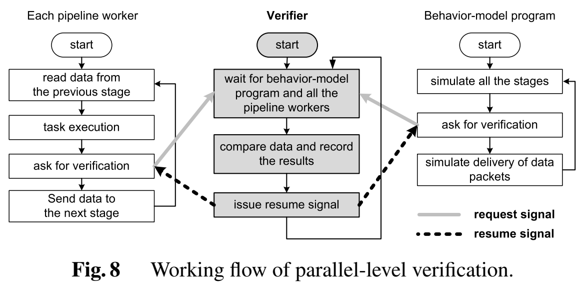

Each pipeline worker
start
read data from the previous stage
task execution
ask for verification
Send data to the next stage
Verifier
start
wait for behavior-model program and all the pipeline workers
compare data and record the results
issue resume signal
Behavior-model program
start
simulate all the stages
ask for verification
simulate delivery of data packets
request signal
resume signal

Fig. 8 Working flow of parallel-level verification.

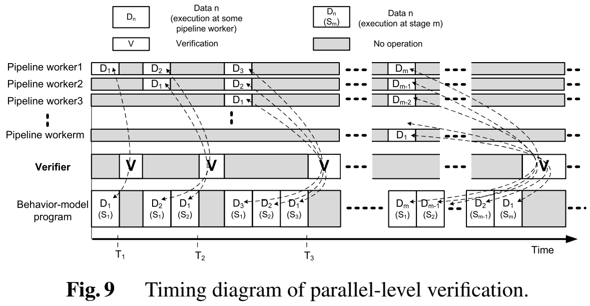


Fig. 9 Timing diagram of parallel-level verification.

3. Case Study

In this section, we present a case study in which the proposed verification-aware methodology is used to help to efficiently and correctly parallelize a 3D depth map generator. The parallel 3D depth map generator is then integrated into a 3D video playing system to demonstrate its capability.

3.1 3D Depth Map Generator

Figure 10 shows the depth map generation algorithm [16], which generates depth maps from single 2D images. The algorithm can operate on three types of input images: images of general scenes with a vanishing point (VP), images of scenery containing sky and mountains, and close-up images. First, the algorithm classifies input images into three categories. Images containing few vanishing lines are identified as close-up images; those containing a sufficient number of vanishing lines are classified as general images; and those containing a sufficient number of vanishing lines, as well as sky and mountains, are classified as scenery images. The following techniques are used to generate accurate depth maps: edge detection using Sobel filtering, line detection using a 5 × 5 Hough transform, vanishing region detection, segmentation, depth map merging, depth map post-processing by simplified joint bilateral filtering (SJBF), and block-based contrast filtering for identifying foreground objects in close-up images. These processing steps are also optimized to reduce the computational complexity while re-

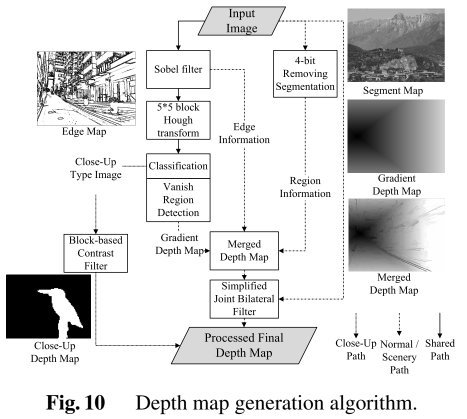


Fig. 10 Depth map generation algorithm.

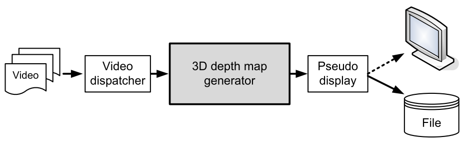


Fig. 11 Structure of the 3D video playing system.

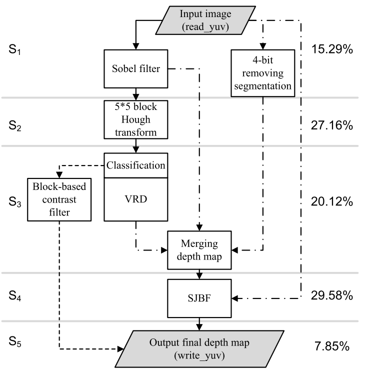


Fig. 12 Task partition for the 3D depth map generator.

taining high image quality. Figure 11 shows the structure of the 3D video playing system, which consists of a video dispatcher, a 3D depth map generator, and a pseudo display. The video dispatcher reads the decoded video and then splits it into several individual frames that are in order dispatched to the 3D depth map generator to produce the corresponding depth map information. Finally, the pseudo display collects all the information in order and completes the work.

3.2 Parallelization

We now explain how to parallelize the 3D depth map generator by using the proposed verification-aware design methodology. Our target is to leverage thread-pipelining parallelization to develop a parallel 3D depth map generator. According to the proposed design flow, we first start work in the behavior phase to construct a behavior-model program corresponding to the parallel 3D depth map generator. In addition to arranging the processed data and temporal buffers into suitable data packets, the workload distribution of the 3D depth map generator is analyzed in order to select an appropriate number of pipeline stages in the parallel program. After considering the issue of workload balance and the dependence relationships among operations in the generator, the generator is partitioned into five stages, $S_1 \sim S_5$, as shown in Fig. 12. Next, we implement the behavior-model program and perform behavior-level verification to ensure the behavior-model program can correctly mimic the behaviors of the parallel program. Figure 13 shows the timing diagram of the behavior-level verification for the 3D depth map generator. The behavior-model program, verifier, and original 3D depth map generator are concurrently executed through individual threads. After repeated modification and verification, we construct a certified behavior-model program. For a developer who is familiar with the proposed

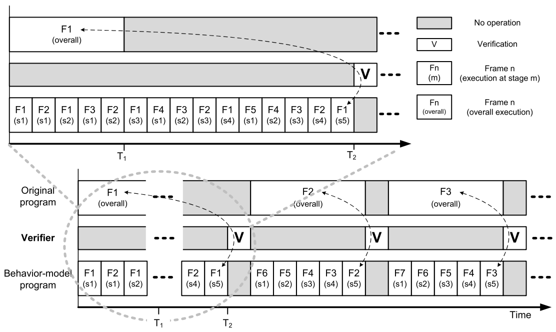
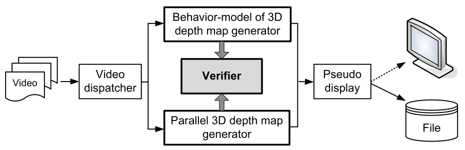


Fig. 13 Timing diagram of behavior-level verification for the 3D depth map generator.



Behavior-model of 3D depth map generator
Verifier
Parallel 3D depth map generator
Video
Video dispatcher
Pseudo display
File

Fig. 14 Concept of parallel-level verification for the 3D depth map generator.

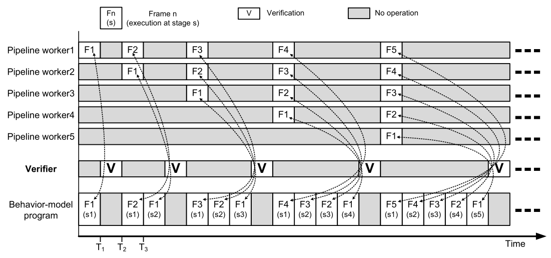


Fig. 15 Timing diagram of parallel-level verification for the 3D depth map generator.

verification-aware design methodology and the original 3D depth map generator program, it would take about 1 day to develop the certified behavior-model program. During the behavior phase, most of the coding time is spent modeling the pipeline behavior.

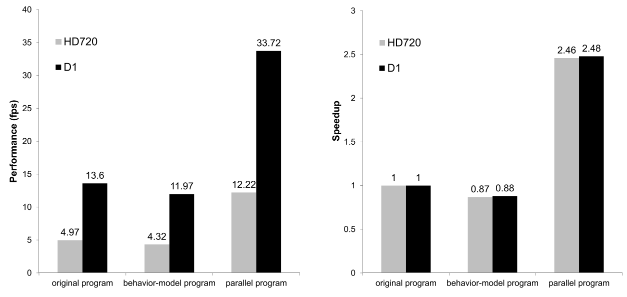
Next, we enter the parallel phase to develop the parallel 3D depth map generator. Figure 14 shows the concept of synchronous verification during parallel-level verification. The behavior-model program, verifier, and parallel program are concurrently executed to test the results generated from each pipeline stage. Figure 15 shows the timing diagram of the parallel-level verification for the development of the parallel 3D depth map generator. It takes 8 days to complete the parallel depth map generator from the certified behavior-model program. For the parallel phase, we spent most of the coding time handling the communications among pipeline stages. The design of the synchronization mechanism among threads is also time-consuming.

3.3 Evaluation and Discussion

Table 1 lists the configuration of the experimental environment. The experiments were evaluated on a multi-core platform with eight processors, each of which is an

Table 1 Configuration of the experimental environment.

	Item	Value
Hardware	CPU	8 quad-core processors, and each processor is AMD Opteron quad-core running at 2.3 GHz
	Cache	L1: 64 KB/core, L2: 512 KB/core, L3: 2 MB/processor
	Memory	64 GB
Software	Operating system	Linux (kernel version: 2.6.31)
	Native C compiler	GCC 3.4.6 with "-O2" option
	Thread library	POSIX thread library



(a) Performance comparison (b) Speedup comparison

Fig. 16 Comparison of performance and speedup for the evaluated configuration.

AMD Opteron quad-core processor running at 2.3 GHz. The 3D depth map generator was verified on 303 tested cases that contain images and videos with resolutions of HD720 (1280x720) and D1 (720x480). The tested cases are randomly selected and fed into the 3D depth map generator during the verification.

Figure 16 (a) compares the performance of the various programs for the evaluated configuration, and Fig. 16 (b) shows speedup values with the original program as a baseline for comparison. The original 3D video playing system can achieve 4.97 fps in HD720 resolution and 13.60 fps in D1 resolution. The behavior-model program of the 3D video playing system sequentially simulates the pipeline execution, and several control statements are added in the program to maintain the behavior of the pipeline execution. Therefore, it can only achieve 4.32 fps in HD720 resolution and 11.97 fps in D1 resolution, and the speedup is 0.87 in HD720 resolution and 0.88 in D1 resolution. With the help of thread-pipelining parallelization, the processing speed of the parallel system can achieve 12.22 fps in HD720 resolution, and 33.72 fps in D1 resolution, corresponding to speedup of the parallel system compared with the original program of 2.46 and 2.48, respectively.

In addition to evaluating the original, behavior-model, and parallel programs, we also test the performance of the 3D video playing system during the verification, in order to understand the impact of the synchronous verification overheads on performance. The behavior-level verification uses three threads to respectively execute the original, behavior-model, and verifier programs. The parallel-level verification uses five threads to execute the parallel program which is parallelized by a five-stage thread pipelining. Besides, it also uses one thread to execute the behavior-model pro-

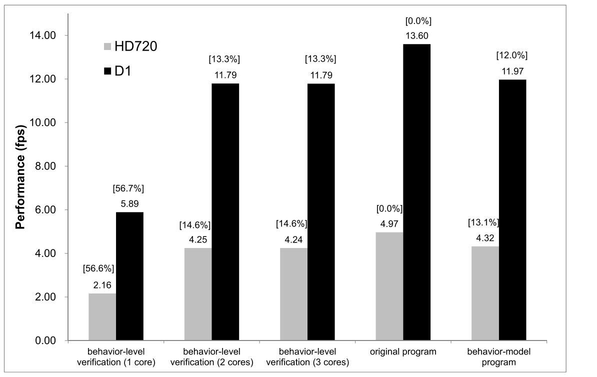


Fig. 17 Performance comparison of the behavior-level verification.

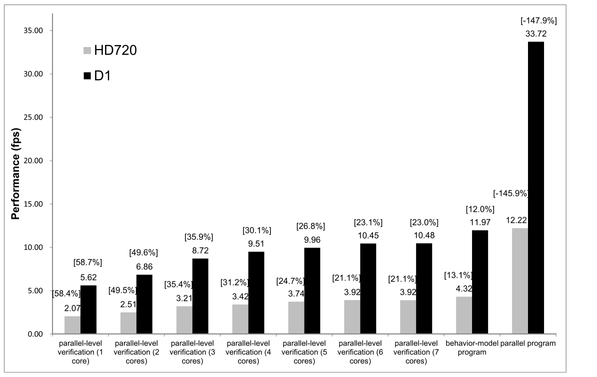


Fig. 18 Performance comparison of the parallel-level verification.

gram and one thread to execute the verifier program. Figure 17 and Fig. 18 show the performance comparison of the behavior-level and parallel-level verifications. In the figures, the labels "behavior-level verification (*X* cores)" and "parallel-level verification (*X* cores)" on the x-axis represent the performance during the behavior-level and parallel-level verifications, respectively. *X* represents how many cores are used to execute the behavior-level and parallel-level verifications. The y-axis represents the performance in fps. The number in the square brackets indicates the performance degradation with the original program as a baseline for comparison, while the number below the square brackets indicates the performance.

Consider the relationship between the performance of behavior-level verification and the number of cores available to the verification, as shown in Fig. 17. For the case that three cores available to execute the behavior-level verification, the performance degradation caused by verification is 14.6% in HD720 resolution and 13.3% in D1 resolution. The performance of behavior-level verification executed on two cores is slightly slower than that on three cores. The reason is the execution time of the verifier program is much smaller than the original or behavior-model programs. Consider the relationship between the performance of parallel-level verification and the number of cores available to the verification, as shown in Fig. 18. For the case that seven cores available to execute the parallel-level verification, the performance degradation caused during the

Table 2 Comparison of code size.

program	original program	behavior-model program	parallel program
code size (bytes)	654,978	705,621	710,303

Table 3 Description of programs bugs identified during verification.

	Description of errors	Difficulty
Behavior phase	Forget to give the data-buffer-related variables initial values.	Easy
	Read/write wrong data buffers due to coding error.	Easy
	Declare wrong data type during data packetization.	Medium
	Data access error due to access an invalid address.	Medium
Parallel phase	Deliver data packets to wrong threads.	Medium
	Synchronization error due to two threads concurrently access the same data buffer.	Hard
	Errors due to globally synchronize among threads (pipeline workers).	Hard

parallel-level verification is 21.1% and 23% for HD720 and D1 resolutions, respectively. The results indicate that the verification overhead brought by the proposed synchronous verification mechanism is affordable and reasonable. The evaluation also shows the performance of parallel-level verification is proportional to the number of cores available to the verification. Table 2 lists the code size of static linked executables for the original, behavior-model, and parallel programs.

Several program bugs were identified during the verification. Table 3 provides a description of the serious errors that arose during system development. In the behavior phase, most of the errors resulted from the process of rewriting the original 3D depth map generator to the corresponding behavior-model program. Because most of the work on data packetization and task partitioning was performed in the behavior phase, most of the errors in the parallel phase arose from synchronization among threads (pipeline workers). The term "Difficulty" in Table 3 is used to represent the efforts of finding the errors. Compared to the behavior phase, most of the synchronization errors in the parallel phase are difficult to be found. The reason is that the occurrence of such errors may depend on some execution order of threads and the execution of the threads may proceed out of order.

4. Conclusion

We proposed a verification-aware design methodology that provides developers with a systematic and reliable approach to performing thread-pipelining parallelization on sequential programs, especially large or complicated programs. In contrast to the traditional design flow, a behavior-model program is constructed before parallelization as a bridge to help developers gradually leverage the technique of thread-pipelining parallelization. The characteristics of pipelining mean that construction of the behavior-model program is straightforward and much less error-prone than other design patterns of parallel programs. To verify the behavior-model program, a synchronous verification approach is proposed, whereby the original program is concurrently executed with the behavior-model program to automatically verify the results. After creating a certified behavior-model program, a parallel program can be constructed based on the behavior-model program. Employing an approach similar to synchronous verification, the behavior-model program is concurrently executed with the parallel program to test the correctness of parallel program. This strategy has a high likelihood of identifying bugs. To demonstrate the practicality of the proposed methodology, it was applied to parallelize a 3D depth map generator using thread pipelining. The parallel 3D depth map generator was further integrated into a 3D video playing system for evaluation of the verification overheads of the proposed methodology and the system performance. The results show the parallel system can achieve 33.72 fps at D1 resolution and 12.22 fps at HD720 resolution through a five-stage pipeline. When verifying the parallel program, the proposed approach of synchronous verification keeps the performance degradation within 23% and 21.1% for D1 and HD720 resolutions, respectively.

The proposed methodology can help developers to rectify bugs and to overcome the complexity of developing parallel programs by employing a verification-aware design flow. For leveraging the thread-pipelining execution model, this technique not only reduces the risks of problems arising when directly parallelizing a sequential program, it also provides a standard operation procedure (SOP) for designing a reliable parallel program on multicore systems.

Acknowledgement

This work was sponsored by the National Science Council of Taiwan under grants NSC-98-2220-E-194-014 and NSC-99-2221-E-194-048. The authors are grateful to the National Center for High-Performance Computing for computer time and facilities.

References

[1] T. Mattson, B. Sanders, and B. Massingill, Patterns for parallel programming, first ed., Addison-Wesley Professional, 2004.

[2] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," Proc. 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, Washington, DC, USA, pp.356–369, 2007.

[3] P. Pacheco, An Introduction to Parallel Programming, 1st ed., Morgan Kaufmann Publishers, San Francisco, CA, USA, 2011.

[4] G.R. Andrews, "Parallel programs: proofs, principles, and practice," Commun. ACM, vol.24, pp.140–145, March 1981.

[5] R.M. Keller, "Formal verification of parallel programs," Commun. ACM, vol.19, pp.371–384, July 1976.

[6] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[7] D. Drusinsky, Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking, Newnes, 2006.

[8] J. Hatcliff and M.B. Dwyer, "Using the bandera tool set to model-check properties of concurrent java software," Proc. 12th International Conference on Concurrency Theory, CONCUR '01, pp.39–58, London, UK, 2001.

[9] G.J. Holzmann, "The model checker spin," IEEE Trans. Softw. Eng., vol.23, no.5, pp.279–295, May 1997.

[10] D. Das, P.P. Chakrabarti, and R. Kumar, "Functional verification of task partitioning for multiprocessor embedded systems," ACM Trans. Design Automation of Electronic Systems (TODAES), vol.12, pp.44:1–44:53, Sept. 2007.

[11] P.A. Hsiung, S.W. Lin, Y.R. Chen, C.H. Huang, C. Shih, and W.C. Chu, "Modeling and verification of real-time embedded systems with urgency," J. Systems and Software, vol.82, pp.1627–1641, Oct. 2009.

[12] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.P. Joo, "Peace: A hardware-software codesign environment for multimedia embedded systems," ACM Trans. Design Automation of Electronic Systems (TODAES), vol.12, pp.24:1–24:25, May 2008.

[13] P. Mishra and N. Dutt, "Specification-driven directed test generation for validation of pipelined processors," ACM Trans. Design Automation of Electronic Systems (TODAES), vol.13, pp.42:1–42:36, July 2008.

[14] C.H. Huang, P.A. Hsiung, and J.S. Shen, "Uml-based hardware/software co-design platform for dynamically partially reconfigurable network security systems," J. Systems Architecture, vol.56, pp.88–102, Feb. 2010.

[15] C.Y.R. Huang, Y.F. Yin, C.J. Hsu, T.B. Huang, and T.M. Chang, "Soc HW/SW verification and validation," Proc. 16th Asia and South Pacific Design Automation Conference, ASPDAC '11, Piscataway, NJ, USA, pp.297–300, 2011.

[16] C.A. Chien, C.Y. Chang, J.S. Lee, J.H. Chang, and J.I. Guo, "Low complexity 3D depth map generation for stereo applications," Proc. IEEE International Conference on Consumer Electronics (ICCE), ICCE '11, pp.185–186, Jan. 2011.

Guo-An Jian was born in Taichung, Taiwan, in 1980, and received the B.S. and M.S. degrees in Computer Science from National Chung-Cheng University, Chia-Yi, Taiwan, in 2003 and 2005, respectively. He is currently a Ph.D. candidate in the Department of Computer Science and Information Engineering at National Chung-Cheng University. His research interests include software optimization and parallel computing for video technology.

Cheng-An Chien was born in Taipei, Taiwan, in 1983, and received B.S. and M.S degrees in the Department of Computer Science and Information Engineering at National Chung-Cheng University, Chia-Yi, Taiwan, in 2006 and 2008, respectively. He is currently working on his Ph.D. degree in the same department, focusing on his research interests of video processing algorithms, 3D display algorithms, VLSI architecture design, and digital IP design.

Peng-Sheng Chen was born in Tainan, Taiwan, in 1973, and received a B.S. degree in Computer Science from National Tsing-Hua University, Taiwan, in 1995, an M.S. degree in Computer Science and Information Engineering from National Cheng-Kung University in 1997, and a Ph.D. in Computer Science from National Tsing-Hua University in 2005. He is currently an Assistant Professor in the Department of Computer Science and Information Engineering, National Chung-Cheng University, Chia-Yi, Taiwan. His research interests include parallel programming, optimizing compilers, and computer architecture.

Jiun-In Guo was born in Kaohsiung, Taiwan, in 1966, and received B.S. and Ph.D. degrees in Electronics Engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1989 and 1993, respectively. He is currently a Processor in the Department of Electronics Engineering, National Chiao Tung University after serving as Processor in the Department of Computer Science and Information Engineering, National Chung-Cheng University, Chia-Yi, Taiwan, from 2003 to 2011. During this period, he joined the System-on-Chip (SOC) Research Center, becoming involved in several Grand Research Projects on low-power, high-performance processor design, and multimedia IP/SOC design. Dr. Guo was the Chairman of the Department of Computer Science and Information Engineering, National Chung-Cheng University, from 2009 to 2011, and was the Director of the SOC Research Center, National Chung-Cheng University, from 2005 to 2008. He was the Research Distinguished Professor of National Chung-Cheng University from 2008 to 2010, and held the post of Associate Professor in the Department of Computer Science and Information Engineering, National Chung-Cheng University, from 2001 to 2003, as well as an Associate Professor in the Department of Electronics Engineering, National Lien-Ho Institute of Technology, Miaoli, Taiwan, from 1994 to 2001. From 1996 to 1999, Dr. Guo held the post of Chairman of the Department of Electronics Engineering, National Lien-Ho Institute of Technology. His research interests include images, multimedia, digital signal processing, VLSI algorithm/architecture design, digital SIP design, and SOC design. He is the author of over 160 technical papers on low-power and low-cost algorithm and architecture design for DSP/Multimedia signal processing applications. Dr. Guo was the recipient of the National Science Council (NSC) Research Award in 1996 and 2001, and was the recipient of the 2003 MXIC Young Professor Award for his contributions to low-power Multimedia/DSP silicon IP design. He was also the recipient of the 2004 Chinese Institute of Electrical Engineering (CIEE) Outstanding Youth Electrical Engineer Award, the 2008 Chinese Institute of Electrical Engineering (CIEE) Tai-Chung Section Outstanding Engineering Professor Award, and the 2010 Chinese Institute of Electrical Engineering (CIEE) Outstanding Electrical Engineering Professor Award in recognition of his excellent contributions to research and development, and his service in the field of Electrical Engineering. He was also the recipient of the 2006 Outstanding Research Award of National Chung Cheng University. His research team has won over 36 awards in IC-related student design contests from 2003 to 2011.