Performance Exploring Using Model Checking

A Case Study of Hard Disk Drive Cache Function

Takehiko Nagano[1,3], Kazuyoshi Serizawa[1], Nobukazu Yoshioka[2], Yasuyuki Tahara[3] and Akihiko Ohsuga[3]
[1]Research & Development Group, Hitachi, Ltd., Yokohama, Japan
[2]GRACE Center, National Institute of Informatics, Tokyo, Japan
[3]Graduate School of Information Systems, University of Electro-Communications, Chofu, Japan
e-mail: {takehiko.nagano.nr, kazuyoshi.serizawa.fz}@hitachi.com, nobukazu@nii.ac.jp, {tahara, ohsuga}@is.uec.ac.jp

***Abstract*—To avoid performance problems (e.g., execution delay), model-based development represented by model checking is used to improve performance quality. However, not so many studies have applied the model checking of performance to actual product development. Specifically, model checking has not been applied to performance exploring, so it is hard to say how effective model checking is. Furthermore, creating a new model for performance verification in addition to the usual development process greatly burdens developers. To reduce this burden, man hours for performance verification modeling must also be reduced. Accordingly, we embedded parameter deployment code to create a performance verification model and achieved performance exploration to ease performance optimization. Also, we developed a performance verification modeling method reusing existing product code to reduce modeling costs (man hours). In this paper, we report a case study in which the proposed method was applied to a Hard Disk Drive (HDD) cache emulation program. According to the results, the minimum cache capacity required processing was completed within the target time. We also show that 57.89% of cache emulation program codes were reused to create the new performance verification model. These results validated the proposed method.**

Keywords-performance; model checking; embedded system.

I. INTRODUCTION

Embedded computer systems acquire more advanced features and become more complicated every year, so the lines of code also increase. Therefore, the parameters that control the system increase, the combinations of the processing that attains performance become huge, and the performance prediction and exploring of the system are difficult. For example, in the database software case, although the tuning parameter is prepared, performance optimization is not carried out for each product. Thus, system engineers need to do performance tuning using the above parameter before product release. Therefore, the tuning documents and tools are prepared by the software vender [11]. Moreover, system engineers need to explore system performance including hardware controlled by software and other software packages. However, if performance tuning is not finished by the release deadline and products are released while still having performance problems, we may suffer damaged customer relations, business failures, income loss, additional project resources, reduced competitiveness, and project failure [2]. Complicated product exploring is difficult to fit in to the limited time of a product's release schedule. Compuware reported that 20% of computer systems have performance problems (e.g., execution delay) [13].

To solve these problems, usually two approaches have been taken. One is carrying out performance prediction and design at early phase of system development. The other is verifying, analyzing, and solving the performance problems at later phase of system development [1][2].

Specifically, at early phase of system development, we carry out system performance prediction using a mathematical model represented by queuing theory [3][4] and performance verification of an algorithm using model checking represented by UPPAAL [6][16][17]. At later phase of system development, we carry out implementation based on a design using the above techniques and performance evaluation, analysis, tuning, and redesign using test results [2]. These techniques have achieved positive results. However, it is difficult to evaluate and analyze performance comprehensively. Because, the parameters that control the system increase, and the combinations of the processing that attains performance become huge. In this paper, we focus on model checking from the viewpoint of comprehension. And we apply it to performance exploring.

The case studies of using model checking are reported [6], [7][8]. However, not so many studies have applied the model checking of performance to actual product development [16][17]. Specifically, model checking has not been applied to performance exploring, so it is hard to say how effective model checking is. Furthermore, creating a new model for performance verification in addition to the usual development is a big burden for developers. To reduce this burden, man hours for performance verification modeling must also be reduced.

In this paper, we propose the following two methods:

1) An easy performance exploring method embedding parameter deployment code used to create performance verification model;

2) A performance verification modeling method reusing existing product code to reduce modeling costs (man hours).

By method 1), performance exploring realizes a comprehensive verification mechanism of model checking. Moreover, by method 2), the C code embedded function of PROMELA is used for performance verification modeling [20]. Specifically, costs are reduced by using the actual product C code instead of new modeling by PROMELA.

Moreover, we report a case study in which the proposed method was applied to a cache emulation program.

In Section 2, we describe a performance problem and objective. In Section 3, we explain our proposed method. In Section 4, we present about our target, a HDD. Specifically, we present a cache emulation program and analysis results of its application. In Section 5, we discuss the effect of the proposed method. In Section 6, we detail our conclusions and future work.

II. Problem and objective

A. Performance problem and research scope

A purpose of this paper is to solve the execution delay problem of the embedded computer system. We assume that all programs are implemented in C language in this paper, because C is a major programming language in embedded systems. Particularly, a target of this paper is an embedded system in that software controls hardware, such as a storage system, a car engine controller and so on.

B. Related works

To solve these problems, many techniques have been proposed and applied. To overcome system performance problems, two approaches have been taken. One is carrying out performance prediction and design at early phase of system development. The other is verifying, analyzing, and solving the performance problem at later phase of software development. Below, examples of these approaches are presented.

1) Countermeasures against performance problems at early phase of system development

At early phase of system development, we carry out system performance prediction and performance verification of an algorithm. Performance prediction uses a mathematical model, typically queuing theory. Queuing theory has been applied in various fields, and many results have been reported [3][4]. Moreover, an example using the Markov model for the performance prediction model has also been reported [5].

Next, the prediction and verification using a design model are described. The modeling method consists of a mathematical model and a programmatic model. In the mathematical model, the model is created using timed-automata [9], Petri net [18], and so on. In the programmatic model, the model is created using UML extended by MARTE [1]. The performance design and verification using model checking is included here. UPPAAL using timed automata is a widely used model checking tool in this domain [6][16][17]. For example, UPPAAL is applied to time constraint verification of Audio/Visual protocol [6]. There are also other models checking tools like PRISM that can verify a statistical model [7].

2) Countermeasures against performance problems at later phase of system development

At later phase of system development, we carry out two main performance improvement measures. One is a performance analysis test of a developed system to evaluate whether the target performance is achieved. The other is performance tuning to analyze test results. After that, the system is redesigned, parameters are reconfigured, etc. [1][2]. These techniques have been applied to actual systems, and designs for next generation products have been reported [15]. Moreover, our company also applies these measures in many product developments. Furthermore, documents and tools needed to master a software package are prepared by the software vender [11].

C. Problems to solve

The countermeasure described in Section 2-B is implemented to prevent performance problems. And, these techniques have achieved positive results. However, it is difficult to evaluate and analyze performance comprehensively. Because, the parameters that control the system increase, and the combinations of the processing that attains performance become huge. In this paper, we focus on model checking from the viewpoint of comprehension. Also, we apply it to performance exploring.

Not so many studies have applied the model checking of performance to actual product development. Specifically, model checking has not been applied to performance exploring, so it is hard to say how effective model checking is. Moreover creating a new model for performance verification in addition to the usual development greatly burdens developers. Furthermore, to reuse old product code, it is necessary to create a performance verification model that also includes the past code. This recurrent work also becomes a big burden. To reduce the above burdens, man hours for performance verification modeling must also be reduced.

As a result of the above issues, the problem to solve is as follows.

Problem to solve: Enable performance exploring of complicated systems with advanced features.

To solve the above problem by model checking, we first do the following.

- Establish a method for applying model checking to performance exploring
- Develop an efficient performance modeling method

III. Performance exploring using parameter development and performance verification modeling reusing product code

There are various types of performance, such as execution time and throughput. In this paper, we define execution time as performance.

A. Outline of proposed method

Many modeling languages exist for design and verification. Modeling languages for design include UML, and modeling languages for verification include model checking such as PROMELA [20]. Furthermore, there are two types of language for verification. One is for functional verification such as PROMELA, and the other is for verification for real time systems such as UPPAAL [6]. In this paper, our target is a modeling language for functional verification such as

PROMELA. Because model checking is used, comprehensive verification is attained. Additionally, by applying model checking, performance exploring is achieved. From the above, we propose the following two methods.

1) Easy performance exploring using parameter deployment code
2) Performance verification modeling reusing product code

By method 1), we can apply model checking to performance exploring. Performance exploring is realized using the comprehensive verification mechanism of model checking. Moreover, by method 2), we can develop an efficient performance modeling method. We use the C code embedded function of PROMELA for performance verification modeling. Specifically, costs are reduced by using actual product C code instead of new modeling by PROMELA. Here, *FeaVer*, which generates the PROMELA model from the C code, exists as related research. However, *FeaVer* is not a performance verification model but only a functional verification model [9].

Moreover, we explain how to verify HDD performance using PROMELA/SPIN not aimed at real-time verification, unlike UPPAAL.

B. *Performance exploring using parameter deployment code*

In case that there are some parameters affecting to system performance, to find a set of the parameters to achieve required performance, performance exploring of the parameters needed to repeat until adequate set was found.
We propose a parameter exploring method for performance to let a model checker, like SPIN. For example, in selecting cache size, we want to choose the smallest cache that satisfies the target performance. In this case, after the cache size is changed, many tests must be performed and results evaluated. When a tester uses a simulation program, the program evaluates by creating a script as shown in Figure 1. In Figure 1, the caches sizes in the second line (4, 8, 16, 32, and 64MB) are inputted to the cache_simlator program, and all patterns are executed to calculate execution time.

```
1 #!/bin/sh
2 for CACHE in 4 8 16 32 64
3 do
4./cache_simulator workload_cmd_data.csv $CACHE >
result$CACHE.txt
5 done
```

Figure 1. Wrapping program

By using a model checking technique, SPIN deploys parameters for exploring. Furthermore, the machine was checked to see whether verification conditions were satisfied. To evaluate cache size, as shown in Figure 2, all cache sizes that can be taken in "if" sentences must be described. By this description, the verification machine (SPIN) verifies by exploring using all parameters. Thereby, to create a script as shown in Figure 1, performance test using an actual machine, analysis of the result log, etc. become unnecessary, and performance exploring efficiency improves.

```
if
:: CacheSize_MB = 4
:: CacheSize_MB = 8
:: CacheSize_MB = 16
:: CacheSize_MB = 32
:: CacheSize_MB = 64
fi;
```

Figure 2. Parameter deployment sample

C. *Performance verification modeling reusing product code*

1) *Reuse of whole processing*

The part that does not contain the conditional branch that influences performance reuses the original C code. The only thing necessary is to surround the function of C language with the c_code{}. An example is shown in Figure 3. In Figure 3, the function sorts a segment's structure by time using qsort of libc. To apply this technique, it is necessary to check whether the target function is processed atomically. This is because the inside of the processing surrounded by c_code{} is processed atomically by SPIN.

```
c_code{
 //compare function
 int comp_segment(const void *seg1,const void *seg2)
 {
   int Time1,Time2;
    SegmentUnit *Unit1 = *(SegmentUnit **)seg1;
    SegmentUnit *Unit2 = *(SegmentUnit **)seg2;

    Time1 = Unit1->Time;
      Time2 = Unit2->Time;

      return Time1 - Time2;
 }
}
```

Figure 3. Example of call function writing by C code

2) *Modeling of the part containing conditional branch that influences performance*

In this subsection, we describe modeling the part containing the conditional branch that influences performance. In the proposed method, the conditional branch (if, while, etc.), which has influence on performance need to be converted to conditional branch of PROMELA, and about expression of the condition, the original C code need to be surrounded with the c_expr{}.

Figure 4 shows the original C code of the conditional branch, and Figure 5 shows an example in which it is PROMELA-ized. The control structure of C language can be mostly used by PROMELA: "if" sentence, "while" sentence, etc. Thus, we use it as shown in Figure 5.

```
if(LRUDumpTime ==0){
      SystemTime += TimeInterval;
}else{
      SystemTime += LRUDumpTime;
          LRUDumpTime = 0; }
```

Figure 4. Example of original C code

```
1  if
2  ::c_expr{ LRUDumpTime == 0} ->
3    c_code{
4         Pcache_main->SystemTime += TimeInterval;
5    };
6  ::else ->
7    c_code{
8        Pcache_main->SystemTime = += LRUDumpTime;
9        LRUDumpTime = 0;
10   };
11 fi;
```

Figure 5. Example of PROMELA model

For example, when the "if" sentence shown in Figure 4 is written by PROMELA, the whole code is surrounded by "if" and "fi" like in the first and eleventh lines in Figure. 5. Conditional sentences are written like the second and sixth lines. Moreover, we need the cross-reference of the variable declared within the model of the PROMELA portion and the variable declared in the C code portion. In this paper, the variable declared within the model of PROMELA is updated by the C code side and then used for PROMELA model control. For example, the fourth line in Figure 5 is equivalent to this processing. In this case, the variable "SystemTime" declared by PROMELA is updated by the C code side. If SPIN can be distinguished in the variable of the PROMELA process, SPIN cannot be renewed. In this case, "Pcache_main" describes the PROMELA process information. P represents a process, and cache_main represents the process name. By following this notation, SPIN can execute a name resolution so that an applicable variable can be referred to.

IV. HARD DISK DRIVE CACHE EMULATION PROGLAM AND ANALYSIS RESULT

In this section, we describe the analysis results for applying the technique of performance verification and exploring described in Section 3 to a HDD cache emulation program. Moreover, we describe the application of the technique using the analysis results.

Therefore, first, we describe the HDD cache emulation program used this time. Next, we describe the analysis results of the cache emulation program. Furthermore, we describe the modeling of reusing actual cache emulation program code. Finally, we evaluate the created model's validity.

A. HDD outline

Here, we describe performance verification of the cache function of HDD. The performance of HDD is influenced by the frequency of drive access. For example, while the drive head attainment time (seek time + wait time of revolution) is 16.53msec in the drive of 7200rpm, cache memory control processing needs μ sec order. This proves that time of drive access is dominant in the I/O time of HDD [10]. From this, HDD is equipped with the cache function to hold the accessed data in a memory in order to reduce the number of disk accesses. The utilization efficiency of the cache is improved, and the whole performance is demonstrated.

In this study, we explore and verify the performance of this cache function using model checking and show the results.

Moreover, for the processing time of a drive portion, a value is returned using the time it takes on the average to make data size uniform.

1) Composition of HDD and cache memory

The composition of HDD is shown Figure 6. HDD consists of software, represented by firmware (FW), and hardware, represented by the I/F controller, memory, disk drive, and other controllers.

Next, we explain the processing flow using write processing. First, the HDD receives a host command (workload data) from the I/F controller. Second, the I/F controller sends a command to FW. Third, the FW's cache controller module checks whether writable cache area remains. If it does not, the data on cache is written to the disk drive using a memory controller and drive controller, thus opening up writable space on the cache. Fourth, after writing, new command data is written on cache memory by FW.

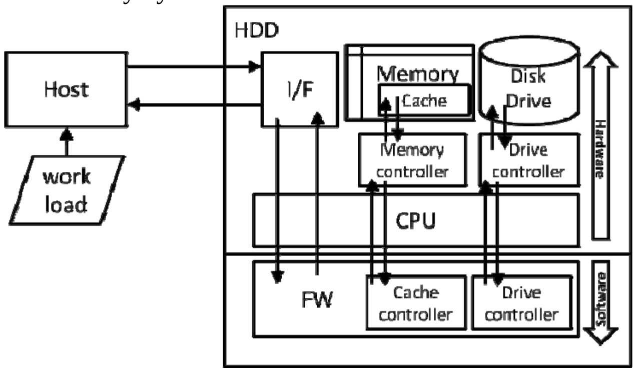


Figure 6. HDD Overview

2) Verification targets

In this paper, we verify the performance of the cache function. Here, performance is defined as execution time.

Based on the above definition, our verified targets define the time from the head command being accepted to the tail command being accepted.

Next, in the future, we plan to use verification results of actual product development. Hence, we plan to make time accuracy of verification results equivalent to the actual system. Therefore, we do not abstract time accuracy.

In this paper, we chose only write processing as the modeling target.

3) Parameters used for cache emulation

Here, we use parameters equivalent to an emulation program. These parameters' information is shown in Table I.

TABLE I. PARAMETERS FOR EVALUATION

Parameter	Meaning
Rotational speed	Revolution per minute
Sector Size	Subdivision area size of a track (512 or 4096 byte)
Cache Size	Total cache size
Average seek time	Head moving time to target
Max segment count	Subdivision area count of Cache memory
Max sector count	Max sector count per track

B. Cache emulation program

Cache processing outlines shown in Figure 7. Before Step 1, the cache program is checked to see if a command has arrived. If it has, cache program is checked to see if it still has easy-to-output data (Step 1). If it does, the cache program transfers the data from cache to a disk drive and opens up writable space in cache (Step 2). If it does not, cache receives a command from the I/F controller (Step 3). Next, the cache program judges whether the new caches used are to be bigger than cache capacity or not (Step 4). If cache overflows, the data chosen by the cache program using a policy (ex: LRU) is written to the disk drive (Step 5). After that, the cache program transfers the data held by I/F to cache memory (Step 6).

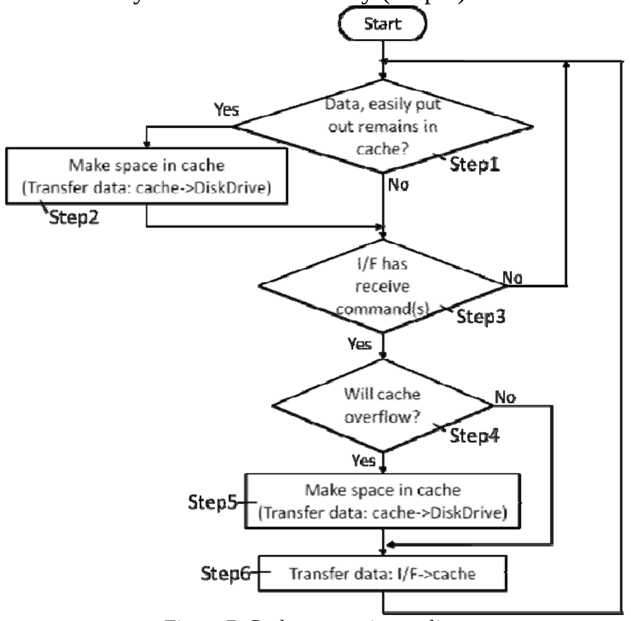

Start
Data, easily put out remains in cache?
Yes
No
Step1
Make space in cache (Transfer data: cache->DiskDrive)
Step2
I/F has receive command(s)
No
Yes
Step3
Will cache overflow?
No
Yes
Step4
Step5
Make space in cache (Transfer data: cache->DiskDrive)
Step6
Transfer data: I/F->cache

Figure 7. Cache processing outline

On the basis of the above process and in accordance with the modeling plan shown in Section 4-A, we created a verification model written in PROMELA from cache emulation program. Figure 8 shows the state transition diagram of cache emulation program with the object of performance modeling. The emulation program modeling this time does not have a host portion. The module of Host I/F reads the workload file and carries out emulation of cache.

Moreover, to calculate drive access time, we did not use an actual HDD. We use the virtual model that calculates average drive access time in this report.

States of the state transition diagram are as follows. The correspondence state in Figure 7 is shown inside of [].

q0: Workload check [before step1]
q1: Segment count check [step1]
q2: Create drive access list using cache data [step1]
q3: Judge existing access list [step1]
q4: Calculate drive access time and clear cache [step2]
q5: Check exist any drive access [step2]
q6: Set lapsed time by drive access [step2]
q7: Set interval time [before step3]
*q*8: Update system time [before step3]
q9: Obtain commands within update time [step3]
q10: Create new segment [step4]
q11: Modify hit segment [step4]
q12: Check cache size [step4]
q13: Decide destage segment [step5]
q14: Calculate drive access time and clear cache [step5]
q*15*: Transfer data from I/F to cache [step6]
q*16*: Finish

Next, we explain the flow of processing using Figure 8. When workload processing starts, the processing changes to q0: Workload check state. Then, the number of remaining commands of the workload is checked. If there are any remaining commands, the processing will change to q1, and if not, it will change to q16, finish emulation, and verify execution time. In q1: Segment count check state, segment count (Seg) in the cache is checked and whether to output cache contents to the drive or not is determined. If Seg > 1 (outputting cache contents to drive), processing changes to q2. If Seg <=1 (not outputting), then processing changes to state q5. In q2: Create drive access list using cache data state, a drive access list is created and processing changes to q3. In q3: Judge existing access list state, if an access list exists, processing changes to q4. If no list exists, processing changes to q5. In q4: Calculate drive access time and clear cache state, drive access time is calculated and acquired from head LBA address of the access list and the length of access data. After this step is completed, processing changes to q5. In q5: Check if any drive access state exists, check whether existing drive access (at q4 or q14) exists or not. If drive access exists, then processing changes to q6. If not, processing changes to q7. In q6: Set lapsed time by drive access state, drive access time is added to system lapsed time, and processing changes to q8. In q7: Set interval time state, configured interval time is added to system lapsed time, and processing changes to q8.

In q8: Update system time state, system time is updated using set lapsed time. After system time is updated, processing changes to q9. In q9: Obtain commands within update time state, the commands arrive within the updated time. If there are no commands, processing changes to q0. If commands exist, a cache is judged to be a hit or miss. If a command is judged to be a miss, processing changes to q10. If a command is judged to be a hit, processing changes to q11. In q10: Create new segment state, the new segment set up information is secured and processing changes to q12. In q11: Modify hit segment state, the updated information on hit cache segment is acquired and processing changes to q12. In q12: Check cache size state, updated cache size is judged to be bigger than the system cache or not. If it is bigger, processing changes to q13. If not, processing changes to q9. In q13: Decide destage segment state, the segment that is outputted to a disk drive or deleted is chosen by using a scheduling algorithm (ex. LRU), and processing changes to q14. In q14: Cache drive access time and clear cache state, cache segment information and clear segment are outputted and processing changes to q15. In q15: Transfer data from I/F to cache state, the command data which has reached I/F is transfer to cache. After this step is completed, processing changes to q 12.

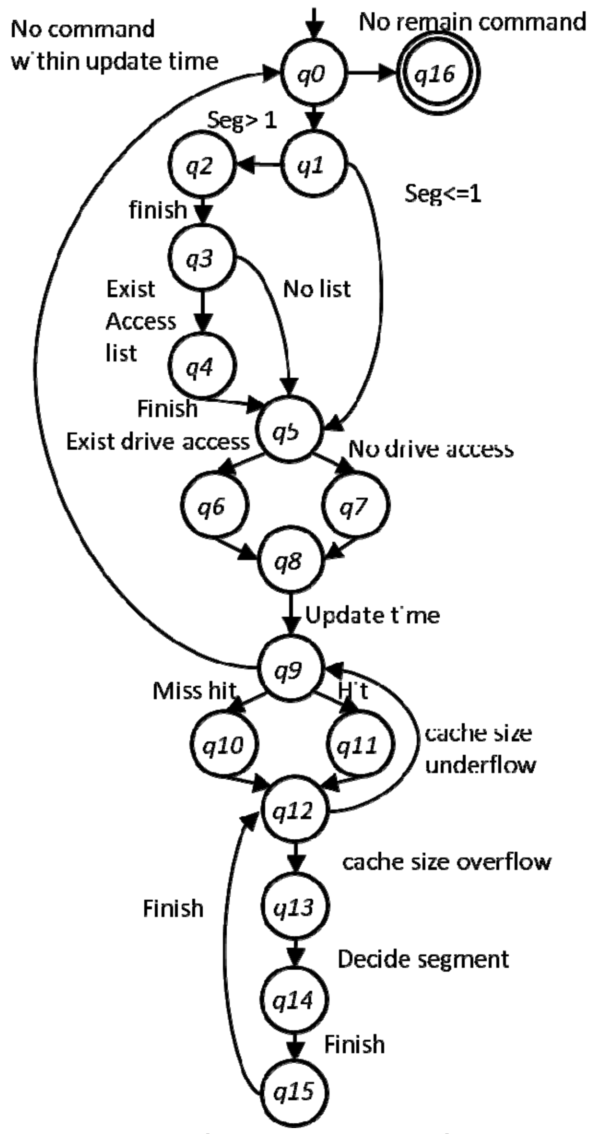


Figure 8. Cache program state transition diagram

The above is processing sequence of the target cache emulation program.

C. Analysis results of cache emulation program

This section describes the analysis result of a cache emulation program. This time, cache performance verification model is created reusing the existing cache emulation C program. Therefore, we describe how to judge whether to reuse the C program part or the new modeling part.

1) Analysis of the cache emulation program based on the contents of verification

Based on the verification contents described in Section 4-A-2, we analyzed the target cache emulation program. This subsection describes the analysis of results.

As described in Section 4-A the HDD I/O performance has dominant disk access time. Additionally, cache processing time does not influence system execution time. Thus, in this verification, addition of lapsed time was limited to the drive access part. However, the opportunity to generate drive access depends on command arrival time. Therefore, we decided to calculate lapsed time on the basis of the command arrival time. Moreover, as mentioned above, since a branch was required to judge the existence of drive processing and a branch accompanying command processing affected lapsed time, they were newly modeled by PROMELA.

Next, from the above-mentioned plan, in processing that determines the contents of drive access, only an execution result influences drive access time, so we thought that the process would not influence performance. Therefore, the processing model that determines the contents of drive access reused the cache emulation C program code. Furthermore, cache emulation program calculates drive access time using only access length, not an internal drive state. Thus, we chose the processing drive portion reusing cache emulation C program code.

From the results of the above analysis, we decided to determine the part that reuses cache emulation C program code and a new modeling part using PROMELA.

D. Development of performance verification model using cache emulation program

1) Create performance verification model

As opposed to the state transition diagram in Figure 8, on the basis of the analysis results in Section 4-C, we decided the part that reuses cache emulation C program code, the part that models using PROMELA, and the part that calculates time progress. The result is shown in Figure 9.

The parts enclosed in a dotted line reuse the existing code, and the parts enclosed in a solid line newly create a model using PROMELA. Time progress processing (to carry out drive access part) is in gray.

The example of modeling in Figure 9 already appeared in Figure 5. Figure 5 shows the same processing as the state diagram that consists of a tri-state of q5, q6, and q7. Lines 1, 2, 6, and 11 in Figure 5 show the same processing as q5. Lines 3 to 5 in Figure 5 show the same processing as q7. Lines 7 to 10 in Figure 5 show the same processing as q6. Finally, lines 3 to 5 and lines 7 to 10 are reused by inserting them into c_code. Other processing parts similarly create a model reusing C code or using PROMELA.

E. The validity check of created model

In this section, the verification model created in Section 4-D is verified using actual work load data. Results are described below.

1) Workload used for verification

In this verification, we use the workload in Table II.

TABLE II. WORKLOAD SPECIFICATIONS

Name	Value
Command count	6510
Command input time range (μ sec)	0~ 35529817
Start LBA range	95~1953512383
Data length (sector)	1~256

2) Parameters for verification

In this verification, we use following parameters shown in Table III.

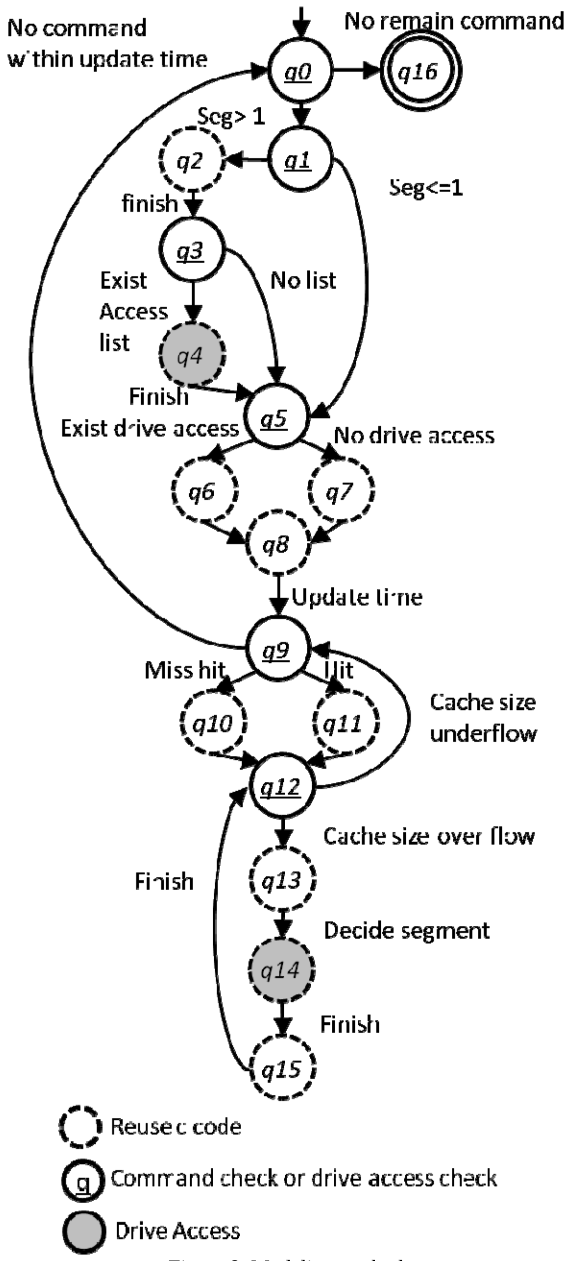


Figure 9. Modeling method

TABLE III. HARDDISK PARAMETERS

Parameter	Meaning
Rotational speed	7200 rpm
Sector Size	512 byte
Cache Size	4,8,16,32,64 MB
Average seek time	8.2 msec
Max segment count	2048
Max sector count	2048

3) PC used for verification

In this verification, we use the PC in Table IV.

TABLE IV. SPECIFICATIONS OF EXPERIMENT PC

Name	Dell Precision T1500
CPU	Intel(R)Core(TM)i7-860 2.8GHz
Memory	16GB DDR3 SDRAM(1066MHz)
Chip Set	Intel(R) H57

4) Using verification tool

In this verification, we use SPIN. The version of used verification tool is SPIN 5.2.5.

F. Verification of execution time

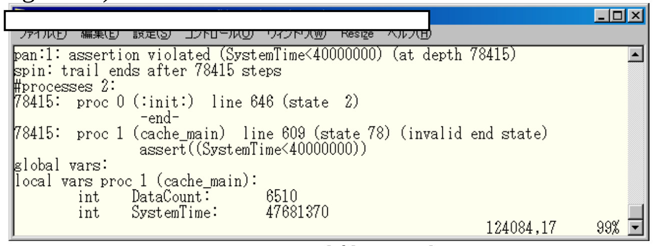
First, we explain the verification of execution time. After the input of the workload, the verification machine calculated execution time and verified whether it satisfied the conditional expression. Then, we verified whether the SystemTime for reaching q16: finish state in Figure 8 exceeded the requirement value. The used verification condition is assert (System Time < Target Time).

A [](System Time < Target Time) can also be used for the same verification.

In the results of this verification, the trail file was outputted when SystemTime exceeded the TargetTime. Thereby, the execution time was verified to satisfy the target or not.

Figure 10 shows an example case in which the above verification conditions were not satisfied.

When the cache size was 4MB and target time was 40,000,000 μ sec, processing took 47,681,370 μ seconds and System Time exceeded requirement time, so a trail file was outputted (Figure 10).



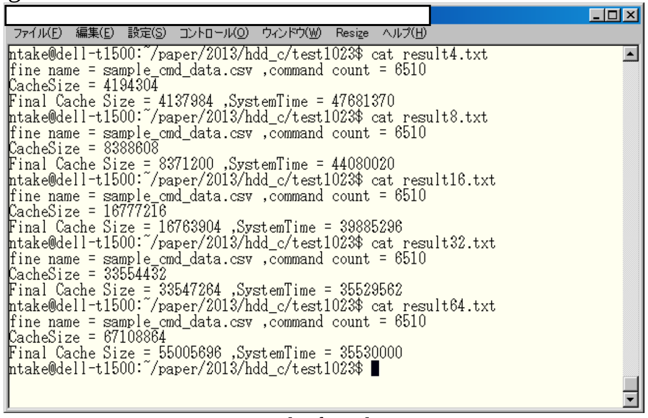
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) Resize ヘルプ(H)

```
pan:1: assertion violated (SystemTime<40000000) (at depth 78415)
spin: trail ends after 78415 steps
#processes 2:
78415:  proc 0 (:init:)  line 646 (state  2)
                -end-
78415:  proc 1 (cache_main)  line 609 (state 78) (invalid end state)
                assert((SystemTime<40000000))
global vars:
local vars proc 1 (cache_main):
        int    DataCount:       6510
        int    SystemTime:      47681370
                                                        124084,17      99%
```

Figure 10. Trail file example1

We acquired the execution results of the cache emulation program and compared them with the verification results of the created model.

The execution results of emulation program are shown in Figure 11.



ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) Resize ヘルプ(H)

```
ntake@dell-t1500:~/paper/2013/hdd_c/test1023$ cat result4.txt
fine name = sample_cmd_data.csv ,command count = 6510
CacheSize = 4194304
Final Cache Size = 4137984 ,SystemTime = 47681370
ntake@dell-t1500:~/paper/2013/hdd_c/test1023$ cat result8.txt
fine name = sample_cmd_data.csv ,command count = 6510
CacheSize = 8388608
Final Cache Size = 8371200 ,SystemTime = 44080020
ntake@dell-t1500:~/paper/2013/hdd_c/test1023$ cat result16.txt
fine name = sample_cmd_data.csv ,command count = 6510
CacheSize = 16777216
Final Cache Size = 16763904 ,SystemTime = 39885296
ntake@dell-t1500:~/paper/2013/hdd_c/test1023$ cat result32.txt
fine name = sample_cmd_data.csv ,command count = 6510
CacheSize = 33554432
Final Cache Size = 33547264 ,SystemTime = 35529562
ntake@dell-t1500:~/paper/2013/hdd_c/test1023$ cat result64.txt
fine name = sample_cmd_data.csv ,command count = 6510
CacheSize = 67108864
Final Cache Size = 55005696 ,SystemTime = 35530000
ntake@dell-t1500:~/paper/2013/hdd_c/test1023$ █
```

Figure 11. Result of emulation program

The file named result*.txt in Figure 11 is an execution result of an emulation program. The applicable numerical value at * shows the cache size. The result of Figure 10 and the result in cache size equals 4MB of Figure 11 are equivalent. All the results in Figure 11 became equal when a model is executed using the same conditions. From this, the created model was judged to have behavior equivalent to that of an emulation program from this result. As mentioned above, in this research, the created model was judged to be executed the same as an emulation program. Therefore, the created model is thought to be appropriate.

V. DISCUSSION

A. Source code reuse ratio and evaluation

In this paper, we attempted to create a model more efficient than the newly made model by reusing C source code. Then, we analyzed the ratio of the reused number of C codes close to the number of codes of the model.
The results of analysis are shown in Table V.

TABLE V. RESULTS OF CODE REUSE ANALYSIS

Name	Value
Model LOC	627 (comment lines are excluded)
Cache C code LOC	605 (comment lines are excluded)
C Line in model	363 (Number of C codes (reuse codes) in a model
Reuse rate	57.89% (vs. Model LOC)
Reuse rate	60.00% (vs. Cache C code)

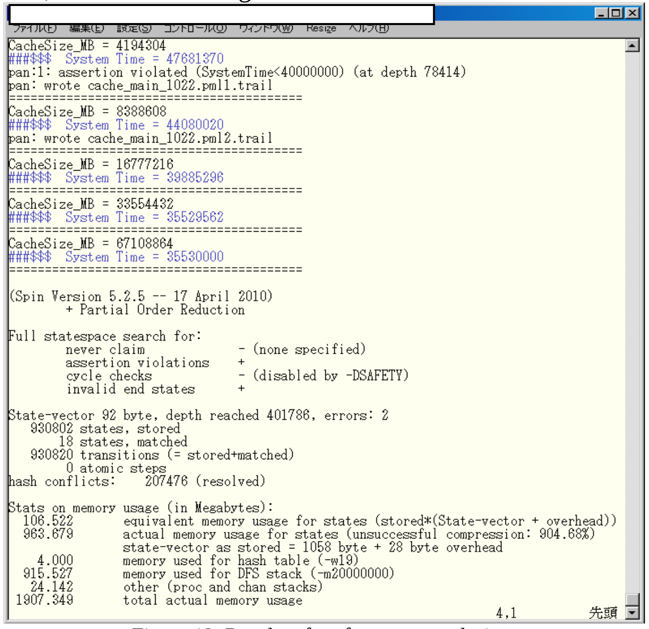
In the results, 60% of original source codes were reused. Moreover, the reuse ratio of the cache C code to a model became 57.89%.

B. Performance exploring using model checking

Next, we show the results of performance exploring using model checking. We used the same verification conditions as described in Ⅳ-F and the code shown in Figure 5, which distributes the cache sizes of 4, 8, 16, 32, and 64MB.

The target time was 40,000,000 μ sec like in Section 4-F, and we carried out performance exploring. In addition, this exploring was completed just to run the program once the pan file that the SPIN generated was executed. Creation of a program as shown in Figure 1 is unnecessary.

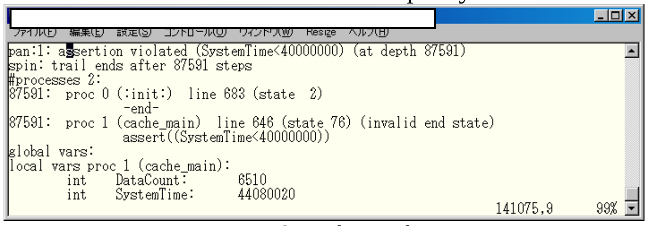
The results are shown in Figure 12. These results show that two cache sizes cannot fulfill the conditions, abnormalities occur, and a trail file is generated.


[redacted]

```
CacheSize_MB = 4194304
###$$$  System Time = 47681370
pan:1: assertion violated (SystemTime<40000000) (at depth 78414)
pan: wrote cache_main_1022.pml1.trail
=========================================
CacheSize_MB = 8388608
###$$$  System Time = 44080020
pan: wrote cache_main_1022.pml2.trail
=========================================
CacheSize_MB = 16777216
###$$$  System Time = 39885296
=========================================
CacheSize_MB = 33554432
###$$$  System Time = 35529562
=========================================
CacheSize_MB = 67108864
###$$$  System Time = 35530000
=========================================

(Spin Version 5.2.5 -- 17 April 2010)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 92 byte, depth reached 401786, errors: 2
   930802 states, stored
       18 states, matched
   930820 transitions (= stored+matched)
        0 atomic steps
hash conflicts:    207476 (resolved)

Stats on memory usage (in Megabytes):
  106.522       equivalent memory usage for states (stored*(State-vector + overhead))
  963.679       actual memory usage for states (unsuccessful compression: 904.68%)
                state-vector as stored = 1058 byte + 28 byte overhead
    4.000       memory used for hash table (-w19)
  915.527       memory used for DFS stack (-m20000000)
   24.142       other (proc and chan stacks)
 1907.349       total actual memory usage
                                                                         4,1         先頭
```

Figure 12. Results of performance exploring

The first pan file has the same contents as Figure 10, so an explanation is omitted. The results of having read the second pan file are shown in Figure 13. As Figure 12 shows, when cache size was 8MB, execution time became 44,080,020 μ sec, which did not satisfy verification formula. In the verification and results in Figure 11, when cache size was less than 8MB, verification showed that target performance could not be attained. It also turned out that 16MB attains target performance with the smallest cache capacity.


[redacted]

```
pan:1: assertion violated (SystemTime<40000000) (at depth 87591)
spin: trail ends after 87591 steps
#processes 2:
87591:  proc 0 (:init:)  line 683 (state  2)
                -end-
87591:  proc 1 (cache_main)  line 646 (state 76) (invalid end state)
                assert((SystemTime<40000000))
global vars:
local vars proc 1 (cache_main):
        int     DataCount:      6510
        int     SystemTime:     44080020
                                                                141075,9      99%
```

Figure 13. Trail example 2

As mentioned above, in model checking, parameters are explored by using the code for parameter deployment, the code for selection of an algorithm is similarly embedded, and a user becomes able to optimize performance easily.

VI. CONCLUSION AND FUTURE WORK

In this paper, to enable performance exploring for embedded computer systems, which acquire more advanced features and become more complicated every year, we decided to achieve the following objectives for model checking.

- Establish a method for applying model checking to performance exploring
- Develop an efficient performance modeling method

To meet the above objectives, we proposed the following two methods.

1) Easy performance exploring using parameter deployment code

2) Performance verification modeling reusing product code

Moreover, the proposed techniques were applied to a HDD cache emulation program, and we verified whether processing could be completed within a target time and confirmed its validity.

Furthermore, we embedded parameter deployment code to create a performance verification model and achieved performance exploring, and then we the determined that minimum cache capacity required processing was completed within the target time. We also showed that 57.89% of cache emulation program codes were reused to create the new performance verification model. From these results, we validated the proposed technique.

For future work, we need to evaluate whether the proposed technique reduces the man hours in an actual product development.

Moreover, although reuse of code was considered to improve the efficiency of modeling this time, the used part of code will be processed atomically. From the characteristic of HDD, since the criterion of judgment of atomizing was created, it is necessary to also examine the criterion of judgment in the case of applying the proposed technique to other products.

Finally, the performance was defined as execution time and verified in this paper. However, since the throughput is similarly important as an index of performance, it will need to be considered too.

References

[1] M. Woodside, G. Franks, and C. Petriu, “The Future of Software Perfo-mance Engineering” in Proc. Future of Software Engineering 2007, May. 2007, pp. 171-187.

[2] C. Smith, L. Williams, “Performance solutions” Addison-Wesley Publishers, 2001.

[3] K. Trivedi, “Probability and Statistics with Reliability” Queuing, and Computer Science Applications. Wiley, 2001.

[4] L. H. Henry, “Software performance and scalability” Wiley, 2009.

[5] Q. Qinru, M. Pedram, “Dynamic power management based on continuous-time Markov decision processes” in Proc. of Design Automation Conference, New Orleans, LA, June 21-25. 1999, pp.555-561.

[6] K. Havelund, A. Skou, K. G. Larsen, and K. Lund, “Formal Modelling and Analysis ofan Audio/Video Protocol: An Industrial Case Study using UPPAAL” in Proc. the 18th IEEE Real-Time System Symposium, Dec 1997, pp 2-13.

[7] T. Nagaoka, A. Ito, K. Okano, and S. Kusumoto, “QoS Analysis of Real-time Distributed System Based on Hybrid Analysis of ProbabilisticModel Checking” IEICE Transactions on Information and Systems, Vol.E94-D, No.5, pp.958-966, May 2011.

[8] K. Moonzoo, K. Yunho, “Automated Analysis of Industrial Embedded Software” in Proc. 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011, pp. 51-59.

[9] R. Alur, D. Dill, “A theory of timed automata,” Theoretical Computer Science 126:183-235, April 1994, doi:10.1016/0304-3975(94)90010-8

[10] B. Jacob, N. W. Spencer, D. T. Wang, “Memory Systems Cache, DRAM, Disk” Morgan Kaufmann Publishers,2008

[11] Oracle(R), “Database Performance Tuning Guide 10g Release2” http://docs.oracle.com/cd/B19306_01/server.102/b14211/toc.htm. [Accessed: Sep 24, 2015]

[12] G. J. Holzmann, M. H. Smith, “Software model checking: extracting verification models from source code Formal Methods for Protocol Engineering and Distributed Systems” in Proc. (FORTE/PSTV99) October 1999,pp.481-47.

[13] Compuware, “Applied Performance Management Survey”, Oct 2006.

[14] V. Tiwari, S. Malik, and A. Wolfe, “Power Analysis of Embedded Software: A First Step Towards Software Power Minimization” IEEE Transactions on VLSI Systems, Vol2, pp. 437-445, Dec. 1994, doi:10.1109/92.335012

[15] S. Barber, “Creating Effective Load Models for Performance Testing with Incomplete Empirical Data,” in Proc. 6th IEEE Int. Workshop on Web Site Evolution, 2004, PP. 51-59.

[16] A. David, K. Larsen, K. Legay, M. Mikucionis, D. Poulsen, and S. Sedwards, “Runtime Verification of Biological Systems,” ISOLA, LNCS, Springer, Vol7609, 2012, pp 388-404.

[17] G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. Vaandrager, M. Voorhoeve, S Smet, and L. Somers, “Formal Modeling and Scheduling of Datapaths of Digital Document Printers.” Proceedings FORMATS'08, Saint-Malo, France, September 15-17, 2008. LNCS 5215, pp. 170-187.

[18] R. Hamadi, and B. Boualem, “A Petri net-based model for web service composition,” Proceedings of the 14th Australasian database conference-Vol17. Australian Computer Society, Inc., 2003, pp 191-200.

[19] Object Management Group, “UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems,” http://www.omg.org/spec/MARTE/, [Accessed: Sep 24, 2015]

[20] G. J. Holzmann, “The model checker SPIN ,” Software Engineering, IEEE Transactions, Vol23(5), 279-295., May 1997, doi: 10.1109/32.588521