Vol.2019-GN-106 No.27
Vol.2019-CDS-24 No.27
Vol.2019-DCC-21 No.27
2019/1/25

既存プログラムを再利用した効率的な性能検証法

長野岳彦[†1][†3]　吉岡信和[†2]　田原康之[†3]　大須賀昭彦[†3]

概要：組込みシステムをはじめとするコンピュータシステムは，システムの性能予測，性能最適化が困難になっている．その結果，コンピュータシステムの出荷後製品の 20%は何らかの性能問題を抱えている．このような性能問題に対し，設計段階でモデル検査を用いて性能検証をし，問題点を抽出・改善することで，性能品質の向上を図る取り組みがされている．しかしモデル検査を製品開発に適用するには，通常の開発に加え，性能検証用のモデルを開発しなくてはならないため，開発コストの増加に繋がる課題がある．また組込みシステムの多くは，短期開発を実現するため，既にあるソフトウェアを利用し，機能を追加・改造をする世代型開発が多くされている．そのため，モデル検査を用いた性能検証を導入するタイミングで既に存在するソフトウェア全体を考慮し，効率よく性能に関して必要な部分だけを抽出し性能検証用のモデルを作成する必要がある．ここで組込みシステム全体から性能検証用のモデルを作成するには，既存システムの構造解析を実施した後，性能モデル作成のための実行解析・実行時間取得ののち，検証モデルを作成，その妥当性を検証するという作業を進めるが，ここで我々は検証モデル作成を支援する手法やツールがなく，性能検証導入上の課題になっていると考えた．この課題に対して我々は，これまでに HDD のキャッシュエミュレーションを対象に性能検証モデルを作成し，そのモデルを用いて性能探索をする報告をしたが，その内容は発表時点での途中経過であったため，モデルを作成するための作業の流れ，作業時の課題，モデル作成時のプログラム再利用の指針，モデル作成の手続が明確になっていなかった．そこで本論文ではこれらの不明確な内容を明確にするため，既存のソースコードを再利用し，必要最低限のモデル化により性能検証を実現する手法を HDD のキャッシュ模擬プログラムを対象に検討し，評価したのでその内容について報告する．

キーワード：性能，モデル検査，組み込みシステム

Efficient Performance Verification Method Reusing Existing Program

TAKEHIKO NAGANO[†1]　NOBUKAZU　YOSHIOKA[†2]
YASUYUKI　TAHARA[†3]　AKIHIKO　OHSUGA[†3]

Abstract: Embedded computer systems become more complicated every year, therefore performance prediction and optimization of the system become more difficult. As a result, 20% of computer systems have performance problems (e.g., execution delay). To solve these problems, we are carrying out performance model checking at design phase and pointing out problems for improvement of performance quality. However, creating a new model for performance verification in addition to the usual development greatly burdens developers. Moreover, many embedded systems developer reuse existing programs to realize short-term development. Therefore, 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. In order to reduce above burden, it is necessary to extract only the necessary parts and create performance verification model. We thought which is a problem that there were no methods and tools to support creating performance verification model when creating models from embedded system program. Against this problem, we have performance modeling of HDD hard disk emulation program and reported.
However, as the report was in progress, the flow of model creation work, task at work, guidance on program reuse, and procedure for model creation were not clear. Therefore, in this paper, in order to clarify these unclear contents, we examined the method of realizing performance modeling reusing the existing HDD cache emulation program.

Keywords: Performance, Model checking, Embedded system

1. はじめに

コンピュータシステムの 20%は何らかの実行時間の未達成に代表される性能問題を抱えたまま出荷されている[10]．それらの幾つかには製品出荷後に性能遅延等の問題が発覚している．結果，出荷遅延による収入の損失やペナルティが発生し，プロジェクト中止や顧客との関係悪化に繋がっている[2].

特に組込みシステム開発では，機能の差別化や，仕向け地毎の対応などに基づく，顧客要望に対応した機能の変更がある．このような変更毎に性能を達成するチューニングをするため，そのつど追加工数が必要となる．この様な対策が求められる一方，コモディティ化などに対するための開発コストの削減が要求されている．結果，性能達成のための問題は難化している．この様な状況で，性能対策を実施し，要求性能を実現することは，難しさを増している．

この様な組込みシステムの性能問題に対し，我々は大きく別けて二つの対策を実施している．1)開発工程の前半にて性能の予測・検討・設計を実施するアプローチ．2)開発

†1 (株)日立製作所　研究開発グループ　システムイノベーションセンタ
Hitachi Ltd.　Research & Development Group　Center for Technology Innovation
†2 国立情報学研究所
National Institute of Informatics
†3 国立大学法人　電気通信大学
The University of Electoro-Communications.

Vol.2019-GN-106 No.27
Vol.2019-CDS-24 No.27
Vol.2019-DCC-21 No.27
2019/1/25

工程の後半で，性能の検証を実施し，問題を分析・改善をするアプローチの二つである[2].

本論文では，上記のうち，1)開発工程の前半における性能をモデル検査する際のモデル作成の難しさに着目する．

組込みシステムでは，性能の達成に寄与するプログラムの実装は，システム内に散在している．また，組込みシステムの多くが既存プログラムを再利用しているため，性能の検証にモデル検査技術を導入する時点で，既存プログラムと新規開発部全体を対象にモデルを作成しなくてはならない．しかし既存プログラムまで含めた性能検証用のモデル作成の工数は，低コストな製品開発要求に反するため，性能検証にモデル検査を導入する際の大きな障壁となる．

我々は，この課題を解決するため，検証用のモデル作成の作業の見直しをした．性能検証用のモデルを作成するには，既存システムの構造解析を実施した後，性能モデル作成のための実行解析・実行時間取得ののち，検証モデルを作成，その妥当性を検証するという作業を進める．これに対し検証モデルの自動作成技術は，実行時間などの性能に対応しておらず[9][17]，性能の検証を行う場合は，検証用モデルを新規に作るか，ツールで取得した検証モデルを性能検証用に作り直さなくてはいけない．我々はこの課題に対し，これまでに HDD のキャッシュエミュレーションを対象に性能検証モデルを作成し，そのモデルを用いて性能探索をする報告をしたが[18]，その内容は発表時点での途中経過であったため，モデルを作成するための作業の流れ，作業時の課題，モデル作成時のプログラム再利用の指針，モデル作成の手続きが明確になっていなかった．

そこで本論文では上記課題を解決し，不明確な内容を明確にするため，既存のソースコードを再利用し，必要最低限のモデル化により性能検証を実現する手法を HDD のキャッシュ模擬プログラムを対象に検討し，評価したので，その内容について報告する．

2. 従来の性能検証手法とその課題

本章では，これまで性能問題を解決するために実施された手法について述べる．その後，関連する研究について述べ，最後に，今回解決する課題について述べる

2.1 関連する研究とその課題

組込みシステムの性能問題に対し，1 章の通り，これまで二つのアプローチが取られてきた．以下，詳細を示す．

1) 開発プロセスの上流における性能対策

開発プロセス上流では，システム全体の性能予測や，アルゴリズムの性能検討を実施している．具体的に A)数理モデルを用いた性能の予測・検討と，B)設計モデルを用いた性能検証の実施である． A)数理モデルを用いた性能の検証では，待ち行列理論による統計モデルや[3]，Markov モデルがよく用いられる[4]．次に，B)設計モデルを用いた予測と検証では，プロセス代数や，ペトリネット[21]，UML の設計モデルに MARTE 等の拡張を施し，性能要件を記述したモデルを作成し，それを用いて性能の設計・検討を実施する例がある[1][12]．モデル検査を用いた性能設計・検討もここに含まれ，特に時間制約の検証に用いられる UPPAAL が有名で，実適用例も多い．UPAAL は時間オートマトン[7]を利用したモデル検査環境で，プロトコルの時間制約の検証等でも既に実適用されている[5]．他にも PRISM 等の統計モデルを扱える検証機もある[6].

2) 開発プロセスの下流における性能対策

開発プロセスの下流工程では，開発したシステムの性能が目標の性能を達成しているかをテストし，問題発生時に結果を分析，性能を改善するための再設計やチューニングなどをが行われる[1]．これらは，実際のシステムに対しワークロード分析を行い，次世代の予測を実施した例等も報告されている[11].

以上 1),2)の手法は個々に製品開発に適用されているが，未だに性能問題は発生している．そこで本論文では，これらの手法のうち，1)のモデル検査技術に着目し，議論を進める．

モデル検査は，検査技術自体，従来技術とのすり合わせの 2 点が難しいとされている．検査技術自体の難しさとしては，検査モデルや条件の作成，ツール操作方法の習得，反例解析の難しさなどがあげられる．従来技術とのすり合わせの難しさとしては，設計時に検証モデル作成の工程を追加するといった開発プロセスの変更と，検査技術習得までの人的リソース確保などが挙げられる[14].

本論文では，上記のうち，検査モデル作成の難しさ，特に性能を達成するための実装がプログラムの中に散在しているため，性能モデルの作成には，製品全体のモデル化が必要[15]な点に着目する．組込みシステムの多くは，開発コスト削減のため，既存ソフトウェア再利用した世代型開発が多くされている．そのため、新たにモデル検査を導入する際，システム全体を考慮した性能検証用のモデルを作成する必要がある．しかし，組込みシステムは開発コストの削減が望まれているため，性能検証用のモデル検査導入時には大量のモデリング対象を効率良くモデルを作成する必要がある．

性能検証用のモデルを，既存プログラムがある製品に組み込むためには，以下 4 つの作業の実施が必要となる．

（作業 1） 既存システムの分析・構造の把握
（作業 2） 既存システムの実行時間解析
（作業 3） 既存システムの性能検証用モデル作成
（作業 4） 作成した検証用モデルの妥当性検討

以下上記 4 作業と，既存の対策手法について説明する．まず，(作業 1)であるが，この作業では，既存システムの制御構造やデータ構造を可視化し，把握する．この作業の

Vol.2019-GN-106 No.27
Vol.2019-CDS-24 No.27
Vol.2019-DCC-21 No.27
2019/1/25

支援ツールとしては，OSS の，gcc のオプションである“—fdump-rtl-expand”と，可視化ツールである egypt や[19]，市販のソースコードの解析ツールである Understand[20]がある．これらを用いて処理フローを可視化し，構造を迅速に把握することが出来る．次に，（作業 2）であるが，この作業では既存システムを実行し，実行時間や実行回数を，取得する．この作業の支援ツールとしては，OSS の gprof や，gcc のオプションである“—finstrument-function”を使い，情報を取得する手法がある．これらにより，モデル作成に使う実行時間情報を取得し，モデル化可能になる．（作業 3）では，（作業 1），（作業 2）の結果を基に，性能検証用のモデルを作成する．この作業に関連するツールとして，プログラムのソースコードをモデル検査に直接利用するものがある．例えば Fever[9]や CBMC[17]といったツールにより，C 言語のプログラムから，モデル検査用のモデルを生成したり，直接モデル検査することが出来る．また SPIN では，C 言語のプログラムを呼び出すことが出来る[13]．最後に，（作業 4）では，処理の実行を組み込みシステム・モデル双方で実施し，組込みシステムの実行に要した時間と，検証用モデルと検証機で検証した結果出力された時間を比較し，妥当性を検証する．

このような性能検証用モデル作成の作業に対し，（作業 3）関連技術は性能（実行時間など）に対応しておらず，性能の検証を行う場合は，検証用モデルを新規に作るか，ツールで取得した検証モデルで性能を検証出来るように作り直さなくてはいけないという課題があった．

この課題に対し，我々は既存組込みシステムのプログラムから性能検証用のモデルを作成し，そのモデルを用いて性能検証をし，それを性能探索に応用した事例について報告してきた[18]．しかし左記の報告は発表時点での途中経過であったため，先に述べた性能検証用のモデルを作成するための作業の流れや，その課題の明確化がされていなかった．また，性能モデル作成においてプログラムを再利用するか，否かなどの判別の指針と，モデル作成の手続きなどが明確になっていなかった．

そこで本論文では上記課題を解決し，不明確な内容を明確にするため，既存のソースコードを再利用し，必要最低限のモデル化により性能検証を実現する手法を HDD のキャッシュ模擬プログラムを対象に検討し，評価したのでその内容について報告する．

3. ソースコードを再利用した性能検証用モデル作成手法

3.1 本論文における性能の定義と検証したい内容

性能には，実行時間・スループットという定義があるが[16]，本論文で以降性能は実行時間とする．また本論文にて実施する性能検証は，目標とする実行時間以内に，目的とする処理の実行が完了状態に到達することを検証するものとする．今回取り上げる HDD のキャッシュの検証の場合，ワークロードと呼ばれるコマンド列の最初のコマンドを HDD のキャッシュが受理してから，最後のコマンドを受理完了するまでの時間が，目標とする時間より短いかを検証する．

3.2 提案手法の概要

提案手法では，対象とするプログラムが，目標とする時間以内に実行完了することを検証可能にする．そのため，検証するモデルは，プログラムの実行開始から終了までの実行時間を積算し，目標時間内に完了したかを検証するモデルと検証式を決める．提案する手法は，既にソフトウェアが開発されており，新たにモデル検査を用いた性能検証を導入する場合において，性能検証に必要な部分をモデル化する手法である．ここで，検証用のモデル作成に関し，モデルを新規に作成する必要がある部分とそうでない部分を検討し，どのようにモデル化すればよいかを表 1 にまとめた．まず， (1)について説明する．性能検証のためのパラメータ定義，再利用する関数の定義，性能検証の開始，再利用するソースコードの呼び出し，検証式による判定，及び新規開発機能は，既存の製品コードには含まれないため，新たに検証用のモデルを作成する必要がある．次に(2)～(4)の既存の製品コードを再利用する部分について説明する．今回の事例から既存部がどのように性能に影響するかを分析したところ，以下 3 点から成ることが判明した．(2)実行時間が長く性能に直接影響する部分（例：ヘッドのシーク）(3)実行時間は短いが，その後の処理に影響する((2)の処理へ分岐する)部分（例：コマンド受信時処理）(4)その他の 3 点である．

以上の分類結果をもとに，必要な部分だけの実行時間のみを取得することを考慮すると，(2)に該当する部分は、既存コードの処理を流用しつつ，実行時間の算出をモデル化する必要がある．(3)に該当する部分は，(2)の出現パタンを網羅して実行時間を計算する必要があるため，制御構造を維持しつつモデル化する必要がある．(4)については性能の検証のため，処理の結果は必要であるが，それ自体をモデル化し検証する必要は無い．3.3 以降ソースコードの再利用方法について説明する．

表 1　モデル作成の指針

#	モデル化方針	モデル化対象機能・処理		説明箇所
(1)	新規作成	検証開始/終了処理 検証用パラメータ定義 再利用関数定義・呼び出し 性能検証部 新規開発機能		3.3.1
(2)	既存コード再利用	性能に直接影響する処理（実行時間が長い処理）		3.3.2
(3)		その他	(2)に分岐する処理	
(4)			その他	3.3.3

今回モデリング言語には，Promela を使い，検証器には SPIN を使うことにする． Promela/SPIN を使った理由は，

Vol.2019-GN-106 No.27
Vol.2019-CDS-24 No.27
Vol.2019-DCC-21 No.27
2019/1/25

C 言語の処理をそのまま呼び出すための I/F が提供されており，開発コード・開発済みのコードが再利用できるからである[13]．関連する研究として，C のソースコードから Promela のモデルを生成する *FeaVer*[9]や，C のソースコードを直接モデル検査の対象に出来る CBMC[17]がある．しかしこれらの検証対象は機能検証であり，これらを用いてモデルを作成しても，非機能要件である性能の検証は出来ない[9][17]．そのため，性能をどのようにモデル化し，検証するかを 3.3 節以降で説明する．

3.3 製品のソースコードを利用した性能検証用モデルの作成方法

3.3.1 新規作成部分のモデル化

本項では，3.2 に示した(1)新規作成部分に関して，モデル化する例を図 1 に示す．今回は簡単のため，reuse_func という関数を再利用し，実行時間を管理するパラメータ SystemTime が 100 以下で終了するか検証するモデルを例に説明する．

表 1 に示す通り，新規作成部は検証開始/終了処理，検証用パラメータ定義，再利用関数定義，再利用コードの呼び出し，性能検証部，新規開発機能などをモデル化する．

検証処理の開始/終了は，検証モデルの主な処理になり，表 19 行目～16 行目の model_main()部が相当する．その中で，10 行目のような検証用のパラメータを定義する．そのパラメータを 4 行目～8 行目に示す再利用した関数 reuse_func を用いて，実行時間を演算，変更する．この再利用の仕方については，3.3.3 に示す．また再利用のための定義は，1 行目～3 行目が相当する．この定義の仕方についても，同じく 3.3.3 で示す．性能の検証は，15 行目の assert 文が行い．この場合は，実行時間 SystemTime が 100 未満であれば，検証は成功であり，100 以上であれば，検証が失敗し，反例が出力される．このほか，新規作成部をモデル化，適宜呼び出しをするようにする．13 行目の check_p は、reuse_func に引き渡すための変数で、このように定義することで、呼び出し先の C プログラムに Promela で定義した変数とその値を引き渡すことが出来る．

3.3.2 既存コードを再利用したモデル化（性能に直接影響する処理/または性能に直接影響する処理に分岐する処理）

本項では，3.2 にて説明した(2)性能に直接影響する部分と，(3)性能に直接影響する部分に分岐する処理部分の２つの方法について，Promela にてモデル化する例を示す．

図 2 に，1 例として 4 章でモデル化する HDD キャッシュ模擬プログラムのソースコードを示す．このソースコードは，キャッシュに登録されたデータがディスクドライブに出力された際の経過時間を取得するソースコードである．図 3 にそれをモデル化（Promela 化）したものを示す．図 2 において，3.2 の(2) 性能に直接影響する処理は 2，4，5 行目になる．また，(3)に該当する分岐は，1，3，6 行目になる．

```
1| c_decl{
2|   int reuse_func(int); //再利用する関数の定義
3| }
4| c_code{//関数の再利用
5|   int reuse_finc(int){
6|   //略
7|   }
8| }
9| proctype model_main(){ //開始
10| int SystemTime =0; //検証用パラメータ定義
11|   c_code{
12|     //再利用コード呼び出し
13|     Pmodel_main->SystemTime
= reuse_func(Pmodel_main->check_p)
14|   }
15| assert(SystemTime < 100); //性能検証部
16|}//終了
```

図 1　新規作成部の例

```
1| if(LRUDumpTime ==0){
2|     SystemTime += TimeInterval;
3 | }else{
4|     SystemTime += LRUDumpTime;
5|     LRUDumpTime = 0;
6| }
```

図 2　C ソースコード例

(2)性能に影響する部分をモデル化した例を図 3 に示す. 図 3 において，性能に影響のある部分をモデル化した個所は，3,4,5 行目と 7,8,9,10 行目となる．

```
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;
```

図 3　Promela モデルの例

今回の検証では，実行時間を検証するため，処理に応じて実行時間を計算するコードを検証モデルに追加するか，実行時間を計算する処理を再利用し，処理結果を参照出来るように既存コードをモデル化する必要がある．今回の例の場合は後者であり，図 2 の 2 行目及び 4 行目が，それぞれ処理が発生しない場合のインターバル経過時間の加算と，処理が発生した場合の時間経過の加算を表しているので，その処理を再利用する．したがって，処理自体は C 言語のまま実行するが，得た実行時間はモデル検査にて検証可能にする必要がある．そのため，図 3 の 3,5 行目，7,10 行目のように c_code{}にて処理自体を囲み，C 言語として処理を実施するが，のちに検証可能にするために，4,8 行目のように Pcache_main といったように，Promela プロセス cache_main の変数 SystemTime に結果を渡す処理を追加する．次に，(3)性能に直接影響する部分に分岐する処理については，図 3 の 1,2,6,11 行目がそれに相当する．if 等，C 言語に対応する制御構造は，Promela でもほぼ使用可能であり，それを使いモデル化する．例えば，if 文については，C 言語では図 4 の様に使用する場合，Promela では図 2 の 1,11 行目の様に if~fi;で囲んだ間に，2 行目と 6 行目の様に

Vol.2019-GN-106 No.27
Vol.2019-CDS-24 No.27
Vol.2019-DCC-21 No.27
2019/1/25

条件文を入れて使用する．条件式は，c_expr というプリミティブを用いて表現する．

3.3.3 既存コードを再利用したモデル化（処理全体の再利用）

本項では，3.2 の(4) その他の処理に関するモデル化手法について説明する．性能に関係ない部分は，システムの振る舞いを再現するためには必須であるが，性能検証のための時間経過に直接影響しない．そのため，処理結果だけを利用することを目的に，既存コードを丸ごと再利用する．その例を図 4，図 5 に示す．C 言語で記載された関数を丸ごと再利用する場合は，図 4　コード再利用の例に示すように，c_code というプリミティブで対象のソースコードをで囲めば良い．ヘッダの再利用の場合は，図 5　ヘッダ再利用の例に示すように c_decl というプリミティブで対象のソースコードを囲めばよい

```
1| c_code{
2|     //compare function
3|     int comp_segment(const void *seg1,const void *seg2)
4|     {
5|         int Time1,Time2;
6|           SegmentUnit *Unit1 = *(SegmentUnit **)seg1;
7|           SegmentUnit *Unit2 = *(SegmentUnit **)seg2;
8|
9|           Time1 = Unit1->Time;
10|          Time2 = Unit2->Time;
11|
12|          return Time1 - Time2;
13|    }
14|}
```

図 4　コード再利用の例

```
1| c_decl{
2|     int comp_segment(const void *, const void *);
3| }
```

図 5　ヘッダ再利用の例

3.4 性能検証用モデル開発の流れ

3.3 までに記載したモデル化手法を用いて，性能検証をするためのモデルの開発手順を図 6 に示す．

まず，1.既存コードの分析を行う．このステップで， 3.2 の（作業１）で説明した内容を行う．次に 2.処理/機能の分類を行う．このステップでは，ステップ 1 の理解を基に，既存コードの処理/機能を 3.2 で説明した性能面における影響に従い，分類する．次に 3.性能測定を行う．これは 3.2 の（手順２）の内容を実施する．次に 4.モデルの作成を行う．このステップでは，ステップ 2 の分類と，ステップ 3 の計測結果を用い，前節までに説明したモデリング手法に基づき，性能検証モデルを作成する．次に，ステップ 5 では，作成した性能検証モデルが，既存ソースコードでビルドされたプログラムと同じふるまいをするか，テストデータなどを作成し，検証する．ふるまいが異なる場合はステップ 3 に戻り，モデルを修正する．ふるまいが同じになったならば，ステップ 6 に進み，性能検証を実施する．ここで，本論文において「ふるまいが同じ」とは，3.1 に示した検証したい内容である「ワークロードと呼ばれるコマンド列の最初のコマンドを HDD のキャッシュが受理してから，最後のコマンドを受理完了するまでの時間」が，モデル作成の元となった対象（今回の場合，キャッシュエミュレーションプログラム）とまったく同じ時間になることと定義する．

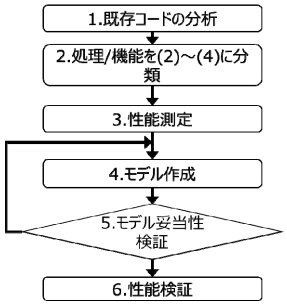

図 6　性能検証用モデル開発の流れ

4. 評価と分析

本章では，3 章で述べた性能検証・最適化手法を HDD キャッシュ機能プログラムに対して適用した結果と，性能最適化に必要な工数がどれだけ削減できたかについて述べる．

4.1 HDD の概要

今回は HDD(Hard Disk Drive)のキャッシュ機能の性能検証を対象とする．HDD の性能は，ディスクアクセス頻度に左右される[8]．そのため HDD はディスクアクセスを削減するために，アクセスしたデータをメモリに保持するキャッシュ機能を備えており，その利用効率を高めることで，HDD 全体の性能が発揮している．今回は，このキャッシュ機能を対象として，性能を探索・評価する際にモデル検査を利用した結果を示す．

4.1.1　モデル検査にて検証する内容

我々は，今回 HDD のキャッシュ機能に関連する性能について検証した． HDD の性能は，以下の通り定義されている．

・スループット：単位時間当たりのデータ転送速度(MB/s)
・実行時間（応答時間）：I/O 要求を受けてから応えるまで

本論文では，これらの性能のうち実行時間に絞って議論をする．また，今回作成するモデルの検証結果は，他社製品や，自社製品の先行モデルの実機を用いた性能の計測結果と比較することを目標とする．そのため，評価結果をそのまま比較に使える程度の模擬精度を求める．よって，時間精度に関しては抽象化の対象としない．

4.1.2 評価に伴うパラメータ

今回，評価に用いるパラメータを，以下表 2 に示す．

表 2　検証用のパラメーター一覧

パラメータ	内容
Rotational speed	1 分間あたりのプラッタの回転数
Sector Size	ﾄﾗｯｸの分割単位（512 or 4096 byte)
Cache Size	キャッシュメモリ容量
Average seek time	平均ヘッド移動時間する時間
Max segment count	キャッシュメモリの分割数
Max sector count	トラックあたりの最大セクタ数

Vol.2019-GN-106 No.27
Vol.2019-CDS-24 No.27
Vol.2019-DCC-21 No.27
2019/1/25

4.2 今回モデル化する HDD キャッシュ機構

4.1.2 に示した方針に従い，キャッシュ機能の模擬コードから Promela で記述したモデルを生成する． 図 7 に今回対象とするキャッシュ模擬プログラムの状態遷移図を示す．

動作の流れについて説明する．動作開始後，workload を読み出し，q0 に遷移してコマンドの残り数をチェックする．残りが有れば q1 へ，なければ q16 に遷移して実行時間を検証する．q1 ではキャッシュ内部の segment 数を見て，ドライブに出力するかを判定する．出力する場合は q2 へ，しない場合は q5 に遷移する．q2 では，ドライブアクセス用のキャッシュセグメントリストを作成し，q3 に遷移する．q3 では，ドライブアクセスリストの有無を検査し，ある場合は q4 へ．ない場合は q5 へ遷移する．q4 では，ドライブアクセスの時間を，アクセスリストの先頭 LBA の位置と，アクセスするデータの長さから計算し，取得する．アクセス処理が終了すると，q5 に遷移する．q5 では，（q4,q14 において）ドライブアクセスの有無を確認する．アクセスが有る場合は q6 へ，ない場合は q7 へ遷移する．q6 では経過時間にドライブアクセス時間の合計を設定し，q8 へ遷移する．q7 では，経過時間に設定済みの一定経過時間を設定し，q8 へ遷移する．q8 では設定された経過情報を元に，システム時刻を更新し，q9 へ遷移する．q9 では，q8 で更新されたシステム時間の範囲で，到着したコマンドを読み込む．コマンドが無ければ q0 に戻る．コマンドが有る場合は，キャッシュをチェックし，hit/miss hit の判定をする．miss hit の場合は q10 へ，hit した場合は q11 に遷移する．q10 では，新規にセグメントを確保する際のサイズを計算する，サイズ情報を引き渡し q12 へ遷移する．hit の場合は，hit した cache セグメントの更新情報を計算し，サイズ情報を引き渡し q12 に遷移する．q12 では，更新されたキャッシュ情報を元に，模擬するキャッシュサイズを超えるか否かを判定する．超えない場合は q9 へ，超える場合は q13 に遷移する．q13 では，キャッシュを更新するスケジューリング（例 LRU）を用いて，ディスクに出力，若しくはクリアするキャッシュセグメントを決定し，q14 に遷移する．q14 では，キャッシュのデステージを実行し，write キャッシュの場合にはドライブ出力を実施する．その後 q15 へ遷移する．q15 では，I/F 部に到着しているコマンドのデータをキャッシュへ転送する．転送完了後，q16 へ戻る．

4.3 キャッシュ模擬プログラムの分析

今回は，C プログラムで作成された Cache 機能の模擬プログラムを再利用し，検証用のモデルを作成する．そのため，C プログラムを元に，再利用する箇所と，しない箇所をどの様に判定したかを以下に述べる

4.1 で述べた通り，HDD の I/O 性能は，ディスクアクセス時間が支配的となっているため，今回の実行時間の検証において，経過時間の計算はドライブアクセス箇所に絞り込んだ．しかし，ドライブアクセスが発生する契機については，コマンド到着時間に依存するため，コマンド到着時間の情報を基に，経過時間を計算することにした．また，上記から，ドライブ処理の状態の有無を判定するために必要な分岐と，コマンドの送受信に伴う分岐は，経過時間に影響を与えるため，C 言語のプログラムの呼び出し対象から外し，Promela のモデル化対象とした．

次に，上記の方針より，ドライブのアクセス内容を規定する処理については，実行結果のみがドライブのアクセス時間に影響があり，アクセス内容を生成する処理の過程は性能に影響が無いと考えた．よって，ドライブアクセス内容を決定する処理は C 言語のプログラム呼び出しの対象とした．

更にドライブ自体の処理については，今回は性能検証の対象がキャッシュ機能であることから，ドライブのデータ長に合わせ平均処理時間を返却するものとし，その処理自体を C 言語のプログラムの呼び出し対象とした．

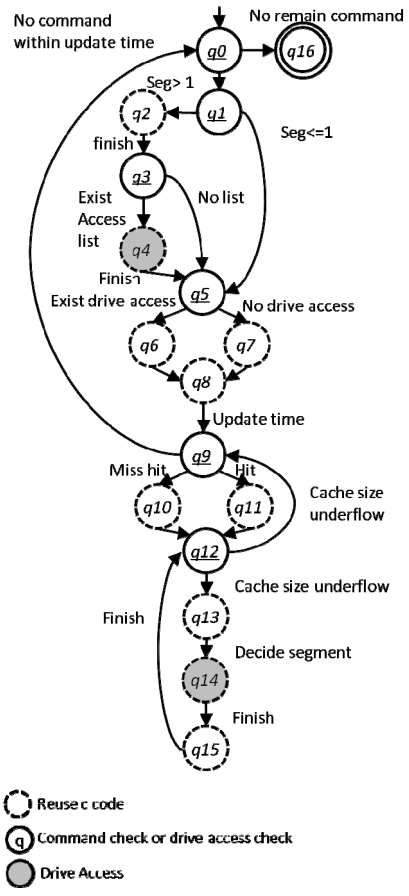

図 7 ｷｬｯｼｭ模擬ﾌﾟﾛｸﾞﾗﾑの状態遷移図/ﾓﾃﾞﾙ化方針

4.4 キャッシュ模擬プログラムを再利用した性能検証モデルの開発

4.4.1 性能検証モデルの作成

図 7 に，4.3 の分析結果を元に，表 1 に示した（2）～（4）の手法でモデル化する箇所を決定した結果を示す．点線で囲った箇所は表 1（4），3.3.3 に示した処理の再利用によってモデルを作成した．次に実線で囲った箇所は点

線箇所にて取得した結果を用い，表 1 の（2）、3.3.2 の手法を適用してモデル化を実施した．

図 7 の実装例の一部は，既出の図 3 である．図 3 は，q5,q6,q7 の 3 状態からなる状態遷移を示している．図 3 の 1,2,6,11 行は q5 を，3~5 行目は q7 を，7~10 行目は q6 を示している．そして 3~5,7~10 行目はそれぞれ c_code{}で処理が埋め込まれており，C 言語プログラムが再利用されていることがわかる．他の処理箇所も同様に 3.3 に記載した手法を用いてモデルの作り込を行った．

4.5 評価.

4.5.1 検証に用いたデータと環境

(1) 検証に用いたワークロード

今回の検証では， 表 3 に示す workload を使用した．

表 3 検証用ワークロードの仕様

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

(2) 検証に用いたパラメータ

今回の検証では表 2 に示すパラメータに対し、以下の値を設定した。Rotational speed 7200rpm，Sector Size 512byte，Cache Size は 4,8,16,32,64MB、Avg seek time は 8.2msec，Max segment/sector count はそれぞれ 2048 とした。

(3) 検証に用いた環境

今回の検証では，Spin Version 6.4.8 を用いた．また C コンパイラは gcc 4.8.4 を用いた. PC は Dell 社 Precision T1500 を使用した。CPU は Intel(R)Core(TM)i7-860 2.8GHz，メモリは 16GB DDR3 SDRAM(1066MHz)を用いた

4.5.2 提案手法妥当性の検証

今回我々は，モデル化対象のキャッシュ模擬プログラムと，そのプログラムを再利用し作成した性能検証用モデルのそれぞれに対し同一のワークロードを入力し，最初のコマンドを受理してから，最後のコマンドを受理するまでの処理時間を比較し，作成した性能検証モデルが妥当であるか検証した．具体的には，表 3 に示したワークロードを，入力した．また，その際に，4.5.1(2)の検証用パラメータのうちキャッシュサイズを 4MB，8MB，16MB，32MB，64MB と変更して検証した．入力したコマンドは pan –v –m10000000 である．ここで-v は Spin のバージョン表示をする，-mN は探索する深さの設定を示している．

まず，図 8 に性能検証用モデルに対し，4MB のキャッシュサイズを設定し，ワークロードを入力した際の結果を示す．1 行目の System Time が実行時間である．この例では 46,058,984 μ 秒を要した結果となった．

次に上記と同条件でキャッシュ模擬プログラムを実行した. その結果を図 9 に示す. 最終行に示した System Time が模擬プログラムの実行結果である．結果，図 8 の性能検証用モデルにて得られた実行時間と同じ 46,058,984 μ 秒となった．

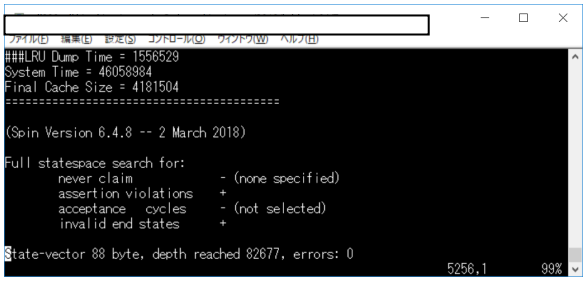

図 8 検証結果の例

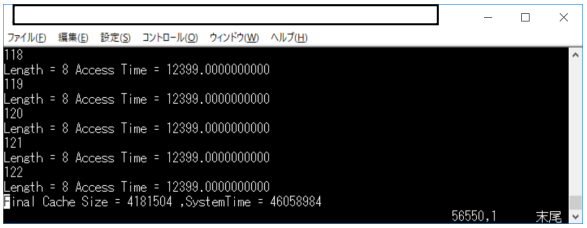

図 9 シミュレーションコード実行結果

上記の他, キャッシュサイズを 8MB, 16MB, 32MB, 64MB と変更した際も，以下表 4 に示す通り同じ実行時間となった．この結果より今回我々が作成したモデルは，実行時間の検証の観点で，キャッシュ模擬プログラムと同様にふるまい，性能検証をおこなうのに妥当であると判断した．

表 4 得られた実行時間

#	キャッシュサイズ	性能検証モデル(μ 秒)	模擬プログラム(μ 秒)
1	8MB	42,735,898	42,735,898
2	16MB	39,996,690	39,996,690
3	32MB	36,268,521	36,268,521
4	64MB	35,330,000	35,330,000

4.5.3 作成した検証モデルを用いた性能検証

次に，作成した検証モデルを用いて性能検証を行った例を示す．今回の検証では処理が目標時間内に終わるか否かを図 8 の q15:finish state に到達した際の SystemTime が，目標時間より小さくなるかで検証することにした．そのために，目標時間 Target Time を設定し，assert(System Time < Target Time)という検証条件をモデルに組み込み検証した．キャッシュサイズを 4MB に設定し， 4.5.2 同様に表 3 のワークロードを入力した．結果は図 10 に示す通り．実行時間は 40,000,000 μ 秒を超えるため反例が検出された。同様に他キャッシュサイズで検証した結果は 表 4 に示す通りで，16MB 以上の場合は反例が出されなかった．この結果から，提案手法により作成したモデルで性能検証が可能になったと判断した．

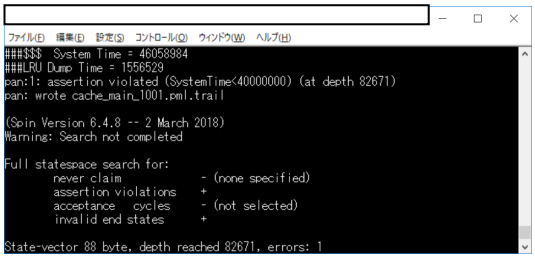

図 10 反例の検出結果例

4.5.4 性能検証モデル開発の効率改善に関する評価

今回我々は，既存コードを再利用できた分は，ゼロからモデルを作成する行為に対し，再コーディングが不要になる分モデル作成時の効率を改善出来る．そこで，モデルの全コード行数に対し，再利用したコードの行数から，性能検証用のモデル開発時の効率改善度を評価した．以下表 5 に評価結果を示す.

表 5 性能検証モデル開発効率改善結果

1	検証モデル行数	627 行
2	模擬プログラム行数	605 行
3	再利用した C コード行数	363 行
4	再利用率（1 に対する 3 の割合）	57.89%

今回の事例では，作成した性能検証モデルに対するコードの再利用率は 57.89%となった．.

5. まとめと議論

我々は，既にソフトウェアが存在する組込み製品の性能検証にモデル検査を導入する際のモデル作成に着目し研究を進めてきた[18]．しかし，用のモデル作成時の作業の流れやその課題，プログラムを再利用するか，否かなどの判別の指針と，モデル作成の手続きなどが明確になっていなかった．本論文では上記課題を解決し，不明確な内容を明確にするため，既存のソースコードを再利用し，必要最低限のモデル化により性能検証を実現する手法を HDD のキャッシュ模擬プログラムを対象に検討し，性能検証モデルを作成，評価した．また，成したモデルを用いて性能検証を実施し，目標時間内に処理が完了するかを検証，の有効性を示した．更に，上記性能検証用のモデル作成時の工数を，既存コードの再利用率から評価した．結果，開発した性能検証用モデルの行数のうち，57.89%は，既存のキャッシュ模擬プログラムを利用して作ることに成功し，モデルを新規に作るよりも開発効率を高めることが出来る目処がたった．

5.1 提案手法の適用範囲と課題

提案手法の今後の適用先としては, HDD 同様短期開発が求められ，既存製品の改善開発がされており，性能要求が求められるテレビや自動車 ECU のソフトウェアへなどが考えられる．

次に，課題について述べる．モデル検査の課題の一つに状態爆発がある．今回提案した手法は，システム全体のモデル化を，既存コードを再利用して行う．そのため，作成されたモデルに多くの既存コードに由来する条件分岐，パラメータが含まれる．結果，検証時の状態数が多くなり，検証時に調整の工数が発生することが考えられる．特に今回は，モデル検査技術の有効性を検証するため，これまで検証してきたキャッシュ模擬プログラムと比較したが，モデル化対象であったキャッシュ模擬プログラム自体が決定的であったため，動作タイミングの変化や、順序の変更に伴う性能変化に伴う状態爆発に関する検証が未実施である。今後はそれらの検証を通して提案手法の限界を明確にする必要がある。更に，今回は性能を実行時間と定義し，検証を実施した．しかし，スループットは実行時間同様重要な性能評価の対象である．今後その対応をする必要がある．

参考文献

[1] Woodside, M. Franks, G. and Petriu, C.: The Future of Software Performance Engineering, Proc. Future of Software Engineering(FOSE ’07) , IEEE Computer Society ,pp. 171-187(2007).

[2] Smith, C. and Williams, L.: Performance solutions:A Practical Gude to Creating Responsive, Scalable Software, Addison-Wesley Publishers(2001).

[3] Liu, H.:Software performance and scalability, Wiley(2009).

[4] Qiu, Q. and Pedram, M.: Dynamic power management based on continuous-time Markov decision processes, Proc. Design Automation Conference, IEEE, pp.555-561(1999).

[5] Havelund, K. Skou, A. Larsen, K, et al.: Formal Modelling and Analysis ofan Audio/Video Protocol: An Industrial Case Study using UPPAAL, Proc. Real-Time System Symposium, IEEE(1997).

[6] Nagaoka, T. Ito, A. Okano, K. et al.: 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(2011)

[7] Alur, R. and Dill, D.: A theory of timed automata, Theoretical Computer Science, Vol.126, pp183-235, (1994)

[8] Jacob, B. Ng, S. and Wang, D.: Memory Systems Cache, DRAM, Disk, Morgan Kaufmann Publishers(2008)

[9] Holzmann, G. and Smith, M.: Automating software feature verification, Bell Labs Technical Journal, Vol5, Issue2, ,pp.72-87(2000).

[10] Compuware, Applied Performance Management Survey, Oct 2006.

[11] Barber, S.: Creating Effective Load Models for Performance Testing with Incomplete Empirical Data, Proc. 6th IEEE Int. Workshop on Web Site Evolution, IEEE Computer Society, pp. 51-59(2004).

[12] Object Management Group.: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems,(online), available from< http://www.omg.org/spec/MARTE/>(accessed 2018-09-11)

[13] Holzmann, G.: The model checker SPIN, IEEE Transactionsions on software engineering ,Vol 23, No 5, IEEE, pp279-295(1997).

[14] 池田信之，今村紀子，高田沙都子：実用化に向けたモデル検査適用手法の開発，東芝レビュー, Vol.62, No.9，pp46-49(2007)

[15] 鵜林尚靖：組み込みソフトウェアの設計モデリング技術， 情報処理，Vol45，No 7，pp682-689(2004).

[16] Patterson, D. Hennessy, J. 成田光彰(訳) ：コンピュータの構成と設計第 3 版（上），日経 BP 社(2006).

[17] C arnegie Mellon University.: Carnegie Mellon CBMC Homepage(online), available from <https://www.cprover.org/cbmc/>(accessed 2018-09-11)

[18] Nagano, T. Serizawa, K. Yoshioka, N. Tahara, Y. and Osuga, A.: Performance Exploring Using Model Checking A Case Study of Hard Disk Drive Cache Function, Proc. The Tenth International Conference on Software Engineering Advences(ICSEA2015), IARIA, pp31-39(2015).

[19] Andreas Gustafsson: egypt – create call graph from gcc RTL dump, (online), available from < https://www.gson.org/egypt/egypt.html>(accessed 2018-09-11)

[20] TechM@trix(R): scitools Understand, (online), available from < https://www.techmatrix.co.jp/product/understand/index.html>(accessed 2018-09-11)

[21] Hamadi, R. and Benatallah,B.: A Petri net-based model for web service composition, Proc. 14th Australasian database conference, pp191-200,Australian Computer Society, Inc(2003).