Vol.2012-EVA-38 No.1
2012/8/2

チェックポインティングとコード差分実行による時短シミュレーション法の提案

椎名 敦之[1]　大津 金光[1]　大川 猛[1]　横田 隆史[1]　馬場 敬信[1]

概要：近年，計算機の性能向上のために高性能な計算機アーキテクチャが開発されている．同様に，それらを活かすためにソフトウェアの最適化技術の重要性が増してきている．最適化技術の開発では，プログラムコード中の最適化する部分を異なるコード最適化手法を適用しながら繰り返しプロセッサシミュレーションを行う必要がある．しかし，一般にプログラム全体のシミュレーションの実行に時間がかかるため，繰り返し何度もシミュレーションを行うと評価に要する時間が長大なものとなる問題が生じる．本稿ではシミュレーションプロセス状態の保存と復元を可能にするチェックポインティングと変更部分に限定してシミュレーションを行うコード差分実行を利用し，シミュレーション時間を短縮する手法を提案する．シミュレーション中に評価対象コードの直前でチェックポインティングを行い，チェックポイントデータの改変後シミュレーションを再開することで，改変したコードごとに最初からシミュレーションし直す手間を省く．

キーワード：アーキテクチャシミュレーション，チェックポインティング，ソフトウェア最適化技術

Proposal of Evaluation Time Reduction Method using Checkpointing and Code Substitution

Shina Atsushi[1]　Ootsu Kanemitsu[1]　Ohkawa Takeshi[1]　Yokota Takashi[1]　Baba Takanobu[1]

Abstract: Recently, various high-performance computer architecture have developed for performance improvement. Similarly, importance of various software optimization techniques have increased to make use these architecture. In optimization technique development, we must repeat simulation with applying multiple candidate methods to parts of program codes. However, the simulation of entire program codes generally takes huge amount time. Therefore, a problem occurs that repeated simulations make the evaluation time enormously long. In this paper, we propose a method to reduce the evaluation time using checkpointing and code substitution. Checkpointing technique allows us to store and restore simulation process image, and we can limit the program execution for simulation to necessary parts of target program codes by code substitution. Our method can reduce the evaluation time by checkpointing the simulation process and by resuming the execution from checkpoint after the modification of the checkpointed process image.

Keywords: architecture simulation, checkpointing, optimization software technique

1. はじめに

近年，計算機の性能向上のために高性能な計算機アーキテクチャが開発されている．同様に，それらを活かすためのソフトウェアの最適化技術の重要性が増してきている．最適化技術の開発では，ループに対する展開数が異なるコード等，プログラム中の特定の部分に対して異なる最適化を施したコードを用いてアーキテクチャシミュレーションを行い，最適化を行った処理の実行サイクル数等を比較することにより評価を行っていく．しかしシミュレータ上での実行は実機で実行した場合と比較して数桁程度実行速度が遅く，シミュレーション対象のプログラムによっては数週間を要することもある．そのためシミュレーションによる評価全体の時間が長大になるという問題が生じる．

我々は，このような一部のコードが異なるプログラムの評価において必要な実行情報はプログラム全体ではなくプログラム中の一部のコードに関するものであり，処理の大

[1] 宇都宮大学大学院工学研究科情報システム科学専攻
Department of Information Systems Science, Graduate School of Engineering, Utsunomiya University

部分が前回のシミュレーションと同じであることに着目した．実行情報を比較する際，前回と同じ処理のコードの実行情報は関係ないため実行する必要はないが，評価対象の処理を実行するためにプログラムを最初から順次実行していかなければならないという問題が生じる．従来のシミュレーション時間の短縮方法として，インクリメンタルシミュレーション [1] が存在する．これは，過去のシミュレーション結果を保存しておき，変更のある部分のみをシミュレーションすることでシミュレーション時間を短縮するものである．

本稿では，プログラム中の変更部分のみに限定して実行を行うコード差分実行に基づいたシミュレーション手法について提案し，評価時間の短縮を図る．プロセスの中断・再開を可能にするチェックポインティング技術を利用し，前回の評価コードのシミュレーションから変更がある部分のみを実行することで，評価に必要のない冗長な処理の実行時間を省く．

シミュレーション時間の短縮方法として，時間軸分割並列化による高速化 [2] が挙げられる．これはシミュレーションの過程を分割し複数の計算機ノードで実行することで時間短縮を図るものである．各シミュレータは割り当てられた範囲の開始点まで簡易シミュレーションを行い，以降は通常の詳細なシミュレーションを行う．手法を SPEC CPU95 での評価に適用したところ，8 並列での評価で通常の評価に対して最大 2.7 倍，平均して 1.9 倍の高速化を達成している．

本研究では上記手法と比較して，2 回目以降のシミュレーションでは最初に実行したシミュレーションのチェックポイントデータを利用するため，評価範囲までのコードをシミュレーションする必要がない．また，最初のシミュレーションを行った後は，チェックポイントデータを繰り返し利用し必要な部分のみシミュレーションを行うため，比較評価を行いたい種類が増えるほど評価時間の短縮が行えるという利点がある．

2. 時短シミュレーション手法

図 1 に従来のシミュレーションにおける，シミュレータ上で実行される評価対象プログラムの流れの例を示す．処理 A に対して処理 B では，評価対象範囲の処理を変更し，その実行にかかったサイクル数の総和を比較している．実際のシミュレーションでは全体の実行だけでなく，特定の部分に限定して実行に要したサイクル数やキャッシュヒット率等の情報を取得したい場合が多い．このようなシミュレーションでは，評価に必要となる情報はプログラム中の評価対象となる部分のみであり全体の実行情報は必要ない．しかし，評価部分を実行するためには最初から順次シミュレーションを行わなければならないため，一部分が異なる複数のコードを実行する場合，大半が同じ処理であっても全てプログラムの処理の最初からシミュレーションを行わなければならない．

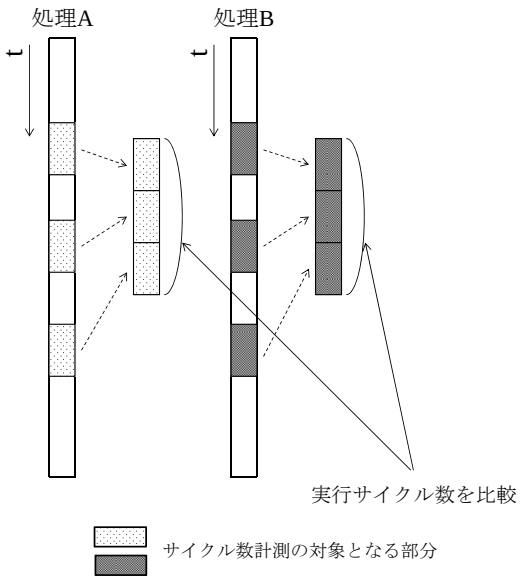

図 1 従来のシミュレーション評価

そこで我々は，複数回のシミュレーションにおいて前回のシミュレーションと大半の処理が同じということに着目し，前回のシミュレーション結果を利用し変更のあった部分のみのシミュレーションを行う．ここでシミュレーションにかかる時間を短縮する手法を提案する．提案手法を用いた場合の評価の流れを**図 2** に示す．提案手法ではまず，評価対象範囲の実行直前にチェックポイントを取り，シミュレーションプロセスを保存する．通常このチェックポイントファイルからプロセスが再開した場合には，チェックポイント以後の処理がそのまま実行される．ここでチェックポイントファイル内のシミュレーションプロセス上で実行されている評価対象プログラムのコードに修正を加えることで，再開後の処理を元とは異なるものに切り替える．通常のシミュレーションであれば，異なるコードを用意し最初から評価対象部分までを実行する必要があるが，提案手法では変更対象部分以前の実行を省くことが可能になる．さらに，一度チェックポイントデータを取得すると，何回でも同じプロセスを再開することができるため，いくつかの異なる評価コードを入れ替えて実行することも可能であり，比較評価を行いたいコードの種類が増えるほどより多くの時間を省くことができる利点も持つ．

2.1 コード差分実行

性能シミュレーション評価では，主に実行に要したサイクル数により評価を行う．プログラム全体に限らず，特定の部分の実行に要したサイクル数を調べることでも評価を行う．最適化手法の評価の例として，あるループに対して，逐次実行コードと並列実行コードを比べる，ループア

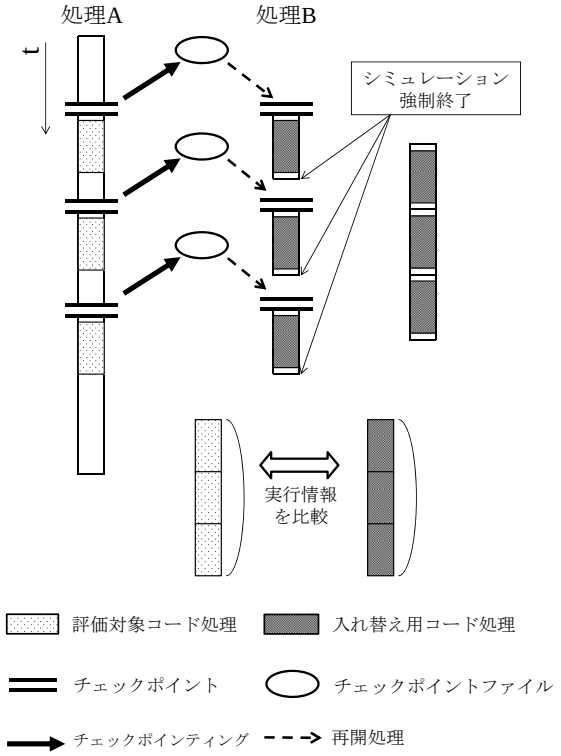

図 2 コード差分実行によるシミュレーション評価

ンローリングの展開数による実行サイクル数を比較する等が挙げられる．

このような場合，ループの開始点と終了点での累積サイクル数が分かればその差を求めることでループの実行サイクル数を求めることができる．従来の評価手法であればコードを変更した際にその都度最初から実行する必要があるが，必要な部分のみを実行しても同様の結果を得ることができる．我々はこの実行形態を“コード差分実行”と呼ぶ．本研究では，このコード差分実行の概念を用いてシミュレーション時間の短縮を図る．

2.2 実行可能バイナリファイルの書き換え

提案手法では，入れ替え用コードを格納する領域をあらかじめ実行可能バイナリファイル内に作成しておき，チェックポイントファイル修正の段階で入れ替え用コードを書き込む．これにより，入れ替え用コードが多数存在するような場合でも，元の実行可能バイナリファイルに対する修正は最低限で済む．

そのため，シミュレーションを実行する前に実行対象となる評価プログラムの実行可能バイナリファイルに修正を施す．**図 3**(a) に実行可能バイナリファイルの構造を示す．実行可能バイナリファイルは先頭に格納されているヘッダ情報と各セクション，シンボルテーブル等から構成されている．コンパイラによって変換された機械語命令は，それらのうちの.text セクションに格納される．.text セクションの内容はプログラム実行時にシミュレータ上の仮想的な

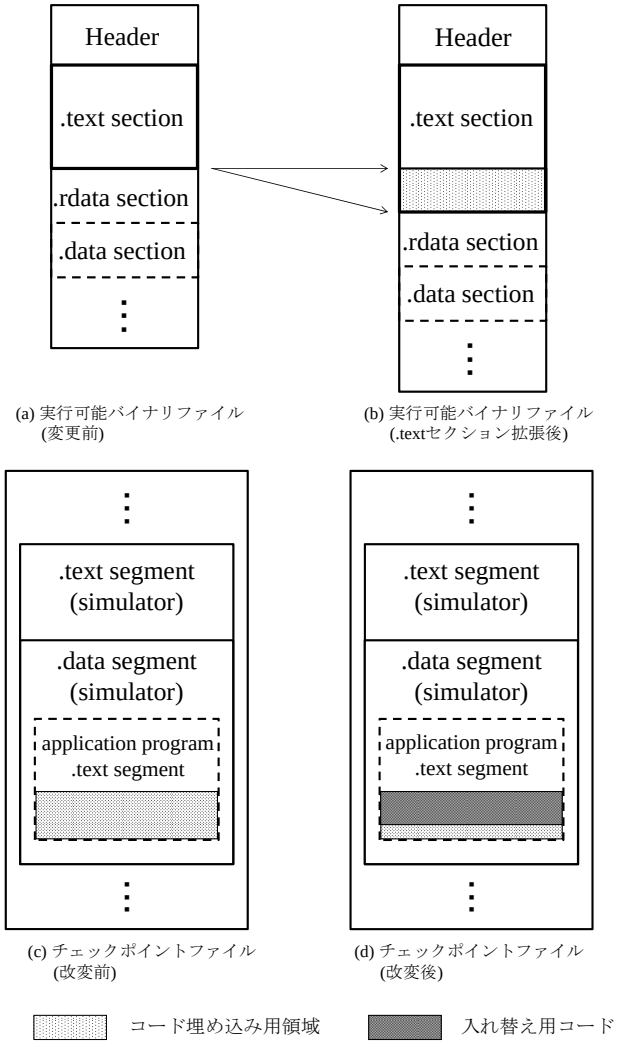

図 3 実行可能バイナリファイルとチェックポイントファイルの修正

メモリにロードされるため，シミュレーションプロセスのチェックポイントファイルにもそのまま保存される．

提案手法では，入れ替え用コードを格納する領域として.text セクションの空き領域を利用する．まず，.text セクションの後方に空き領域を作成するために.rdata セクション以降のデータを後方にずらす．さらに実行可能バイナリファイルとしての整合性を保つため，セクションヘッダのオフセット値やサイズ，シンボリックテーブルのオフセット値をずらした分だけ修正する．空き領域作成後の実行可能バイナリファイルのイメージを図 3(b) に示す．

この操作によって修正された実行可能バイナリファイルのチェックポイントを取ると，図 3(c) に示すように空き領域を含んだ状態でシミュレーションプロセスのチェックポイントデータが取得される．

2.3 シミュレーションのチェックポインティング

前段階で修正を施した実行可能バイナリファイルを使ってシミュレーション実行を行う．シミュレーション実行中，評価対象領域の命令が実行される直前にチェックポイ

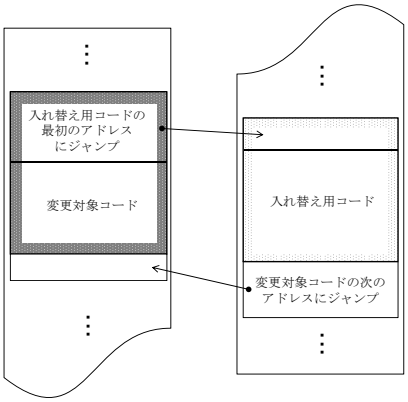

図 4 再開後処理の切り替え

ンティングを行い，チェックポイントデータを取得する．これを実現するため，シミュレータ上で実行されている評価プログラムの指定アドレスの実行時にチェックポイントを取る機能をシミュレータに追加する必要がある．シミュレータにチェックポイントデータを取得するための機能を追加するために，本研究では DMTCP (Distributed Multi-Thread CheckPoint)[3] を使用する．DMTCP はチェックポインティングのための機能ライブラリを提供しており，任意のプログラムにチェックポイント機能を容易に出来るようになっている．

あらかじめチェックポイントとする命令アドレスを指定しておき，シミュレータ内の PC が指定されたアドレス値になった際にチェックポイント処理用の関数を呼び出す．これにより通常のシミュレーションと並行しつつ，指定した場所でチェックポイントを取ることができようになる．

2.4 再開後の処理切り替え

生成されたチェックポイントファイルに対して，処理切り替えのための修正を施す．まず，図 3(d) のように，2.2 節で述べた空き領域に対して入れ替え用コードを書き込む．続いて処理を切り替えるために，評価対象範囲の最初の命令を入れ替え用コードの最初の命令へのジャンプ命令に書き換え，入れ替え用コードの最後に変更対象範囲の次のアドレスへジャンプする命令を追加する．**図 4** に処理切り替えの操作を示す．

これにより，チェックポイントからの実行再開後，評価対象範囲に実行が移った際に入れ替え用コードの処理が実行される．入れ替え用コードの実行後には，2 つ目のジャンプ命令によって変更対象範囲の次の命令が順次実行されていく．

3. 提案手法による評価支援システム

提案手法は実行可能バイナリファイルのヘッダ情報の書き換えやバイナリコードの挿入等，バイナリコードレベルでの操作を必要とする．そのため，作業の負荷を軽減する

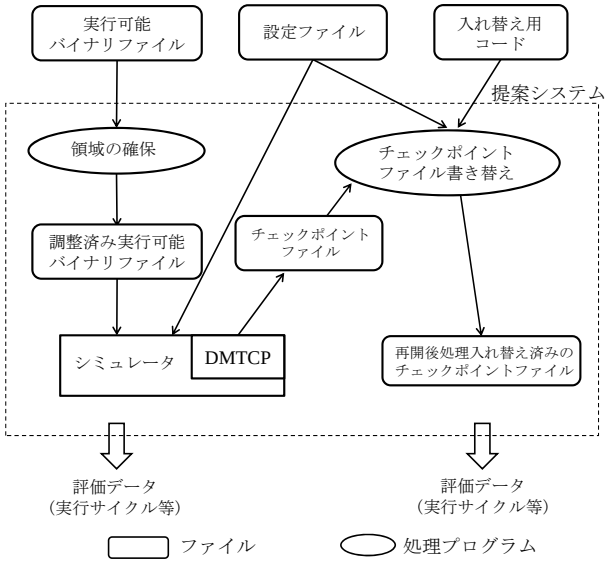

図 5 提案システムの全体図

評価支援システムを開発する．これは実際のシミュレーションで提案手法を実現するための各処理をユーザに代わって自動で行うものである．動作するプラットホームは UNIX 系 OS での動作を想定する．

ユーザはシステムを利用するにあたって，シミュレーションの対象となるプログラムの実行可能バイナリファイル，入れ替え用コードを記述したファイル，アドレス単位で評価対象範囲が記述されている設定ファイルの 3 つを用意する．入れ替え用コードはアセンブリファイルとして入力し，書き換え時にシステム側がバイナリファイルに変換する．

提案システム全体の動作を**図 5** に示す．本システムは初回のシミュレーション時と 2 回目以降のシミュレーション時とで異なる処理を行う．初回のシミュレーション時には，実行可能バイナリファイルに対しての空き領域生成とチェックポインティングを行う．入力された実行可能バイナリファイルは空き領域を作成され，シミュレーションが行われる．さらに指定されたアドレスの命令が実行されると，DMTCP によるチェックポインティングが行われる．この時生成されたチェックポインティングファイルが初回シミュレーションにおける提案システムの出力となる．また，1 回目の実行をしながらチェックポインティングを行う．

2 回目以降のシミュレーションでは，1 回目のシミュレーション結果を元にコード差分実行を行う．入れ替え用コードはアセンブリコードとして入力され，アセンブル，さらに.text セグメントの開始アドレスを指定し入れ替え用コードのオブジェクトファイルを生成する．その内の.text セクション内の機械命令を入れ替え用コードとして取り出し，1 回目のシミュレーション時に生成されたチェックポイントファイル内の空き領域に書き込む．続いて，チェッ

クポイントファイルの情報とユーザが指定した評価対象範囲の情報を元にジャンプ命令を書き込む．終了処理を追加すし，DMTCP によって修正後のチェックポイントファイルからシミュレーションを再開する．これにより，1 回目とは異なる新しいコードが実行され，その時の評価結果を得ることができる．

評価対象範囲が複数存在する場合はその都度チェックポイントファイルが生成されることになる．システムはチェックポイントファイルごとに処理の切り替えとシミュレーションの再開，終了を行い，それぞれで得られた評価結果を統合して最終結果として出力する．

4. 従来のシミュレーションへの手法適用結果

4.1 提案手法の正当性・等価性

実際の評価において提案手法を適用し，従来の評価との簡単な比較を行った．評価に用いるシミュレータはスレッドパイプラインモデルのシミュレータである SIMCA[4] をベースに独自の機能拡張を行ったものを使用した．また，評価対象のプログラムには SPEC CFP2000 の 171.swim を使用し，入力データセットに train を使用した．

今回の評価では，(a) オリジナルのコード，(b) 関数 calc3_ 内の基本ブロック 97 の前後に現在のサイクル数を表示する処理を追加したコードの，ブロック 97 の処理が異なる 2 種類のコードを評価対象プログラムとして実行した．**図 6** に 171.swim 内の関数 calc3_の制御フローと入れ替えの操作を示す．図中の数字は基本ブロックの番号を示し，細い矢印は基本ブロック間の処理の遷移を表している．提案手法による評価と従来のシミュレーションによる評価とで，実行結果とシミュレーションの時間を比較する．

提案手法では，まず 1 回目の (a) のシミュレーションの際に関数 calc3_が呼び出された直後にチェックポイントを取る．今回の実験では関数 calc3_は合計 8 回呼び出されるため，チェックポイントファイルは 8 個生成される．それぞれのチェックポイントファイルに対し，ブロック 95 からブロック 98 を評価対象範囲とし，(b) のブロック 95 からブロック 98 のコードを切り替え用コードとして新たに埋め込み，処理の切り替え操作を行った．また，評価範囲であるブロック 98 が終了した時点でシミュレーションを終了する処理を追加した．本来であれば，処理の入れ替え範囲は (a) と (b) とで異なる処理を行うブロック 97 のみで問題ないが，今回の実験では終了コードを追加する都合でブロック 97 を含むループ全体を入れ替えた．従来のシミュレーションは (a) と (b) とで単純に 2 回シミュレーションを行った．

シミュレーションに要した時間を**表 1** に示す．提案手法の結果より，ブロック 97 の実行時間は全体の 2 割に満たないことが分かる．従来手法と比較して，提案手法を使用することで 1.634 倍の時間短縮が達成できた．同様の評価を

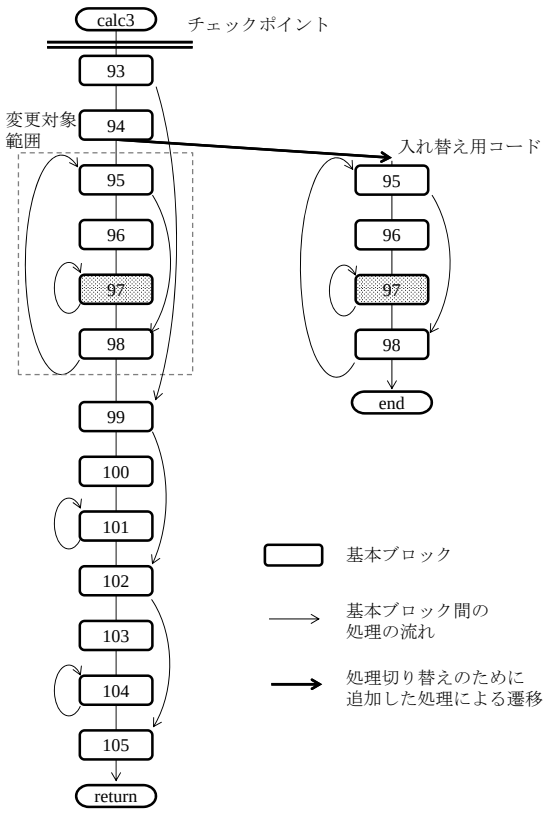

図 6 171.swim 内の関数 calc3_の制御フローと入れ替え

3 種類のコードを入れ替えながらシミュレーションを実行する場合では，約 2 倍の短縮が見込めると考えられる．また，両手法の 1 回目のシミュレーションにかかった時間を比較すると，提案手法の方が約 15 秒長く時間がかかっている．これはチェックポインティングの際のオーバーヘッドが原因である．

表 1 シミュレーションの実行時間

	提案手法による評価	従来手法による評価
コード (a)	662 秒	629 秒
コード (b)	121 秒	652 秒
計	783 秒	1281 秒

次に各手法でのシミュレーションの結果を比較した．提案手法ではチェックポイントからの実行再開を 8 回行ったが，どのシミュレーションにおいても最終的に得られる結果は従来手法の場合と同じであることを確認した．また，算出されたサイクル数に関しても同様の結果が得られた．以上により，提案手法により途中で処理を切り替えた場合でもその入れ替え用コードを含んだプログラムを最初から実行している場合と同じ評価を行うことが可能であるという等価性が示された．

4.2 実用化に向けての課題

現時点での提案システムにおいて改善すべき課題について述べる．

課題の一つは，チェックポイントファイルの管理である．今回の例では，関数 calc3_は 8 回実行されるため合計 8 個のチェックポイントファイルが生成されるが，仮にブロック 97 の開始点をチェックポイントに設定した場合には 1 万個近くのチェックポイントファイルが生成される可能性がある．今回の実験例ではチェックポイントファイルのサイズが 10MB であるため，トータルで 100GB 以上の大容量の記憶装置が必要になる．また，チェックポインティングの際のオーバーヘッドも非常に大きくなり，チェックポイントファイルを生成するための実行に非常に長大に時間を要することになる．

その解決案として，各変更対象範囲ごとにチェックポイントデータを取得するのではなく，ある程度広い間隔でチェックポイントデータを取得することで生成されるチェックポイントファイルの数を減らす方法が考えられる．提案手法で示したようなアドレスによるチェックポイントの指定では，命令単位でチェックポイントデータを取ることが可能だが，複数の評価対象範囲ごとにチェックポイントを取ることはそのままでは難しい．これに対して，チェックポイントのタイミングの指定方法として一定の時間間隔ごとにチェックポイントデータを取得するよう指定する方法が考えられる．一定間隔でチェックポイントを取った例を**図 7** に示す．この方法では 1 つのチェックポイントデータから再開後に評価コードまでに実行される処理が多くなり，シミュレーションの時間が長くなる．そのために提案手法と簡易シミュレーションを組み合わせてシミュレーションを行う．簡易シミュレーションは最低限の処理のみのシミュレーションであり詳細な実行情報が得られない欠点があるが，通常のシミュレーションと比較して高速に実行をすることができる．コード差分実行においては変更対象範囲は通常のシミュレーションを，変更対象範囲以外の前回と重複した処理のシミュレーションには簡易シミュレーションを用いることで時間短縮を図る．

この方法では，チェックポイントを何処に設定するかが課題となる．図 7(a) のように適切にチェックポイントデータを取ることができた場合，生成されるチェックポイントファイルが減り，かつ修正・再開するチェックポイントファイルの数も減らすことができるという利点がある．図 7(a) の場合では，2 番目と 4 番目のチェックポイントファイルは次のチェックポイントまでの間に評価対象コードが存在しない．そのため，これらのチェックポイントファイルに処理切り替えの修正を施す必要がない．

また，図 7(b) で示すように，チェックポイント同士の間隔が大きい場合では，生成されるチェックポイントファイルは少なくなるが，再開後に実行される評価対象以外の処理を実行しなければならないため図 7(a) の場合と比較してシミュレーション時間が長くなる可能性が高い．一方で図 7(c) のように間隔を小さくした場合でも，チェックポイントの位置によっては修正・再開する必要のあるチェックポイントファイルが多くなり，結果としてシミュレーション時間が長くなる可能性がある．

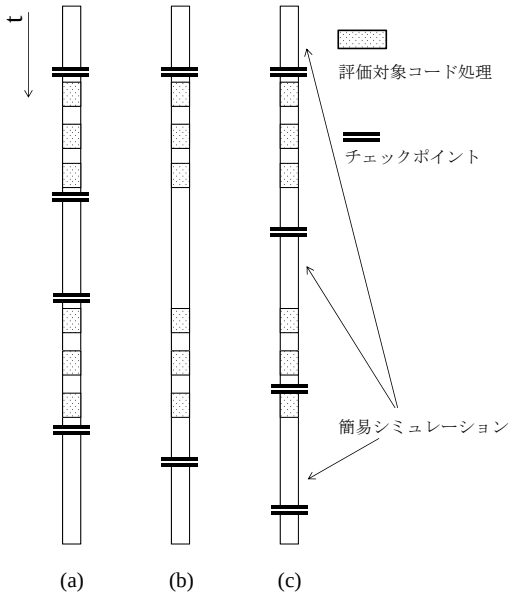

図 7 チェックポイント位置の違いによる実行の違い

シミュレーション実行中の評価対象コードの実行がどのように分布するのかは実行するまで分からない．

5. おわりに

本稿では，ソフトウェアの最適化技術の開発におけるアーキテクチャシミュレーションでの評価時間を短縮する手法について述べた．プログラムコードの一部を変更しながら繰り返し行われるシミュレーションでは処理が異なるのは一部のみで大部分は過去のシミュレーションと同じである点に着目し，変更のあった部分のみを実行するコード差分実行とプロセスの中断再開を行うチェックポインティングを用いたシミュレーションの時間短縮を目的とした評価手法を提案した．また，提案手法を利用した評価支援システムについて示し，実際の評価に提案手法を適用しその正当性と等価性，有効性を示した．

今後の課題は，提案システムを完成させることである．また，チェックポイントの位置の設定基準を確立することで生成されるチェックポイントファイルの数を減らし，データ量が膨大になるという問題の解決を図る．そして提案手法の有効性を示していくとともに，提案システムの実用化を目指す．

参考文献

[1] Kiyoung Choi, Sun Young Hwang, Tom Blank: “Incremental-in-time algorithm for digital simulation”, DAC ’88 Proceedings of the 25th ACM/IEEE Design Automation Conference, pp.501-505, 1988.
[2] 高崎透, 中田尚, 津邑公暁, 中島浩: “時間軸分割並列化に

よる高速マイクロプロセッサシミュレーション”, 情報処理学会論文誌, コンピューティングシステム 46, pp.84-97, 2005.

[3] Jason Ansel, Kapil Arya and Gene Cooperman: “DMTCP:Transparent Checkpointing for cluster computations and the desktop”, IEEE International Parallell and Distributed Processing Symposium, IPDPS 2009, pp.1-12, 2009.

[4] Jian Huang: “The SImulator for Multi-thread Computer Architecture Release 1.2”, Technical Report No: ARCTiC-00-05, 2000.