情報処理学会論文誌　プログラミング　Vol. 4　No. 3　68–83 (June 2011)

バイトコード変換によるActionScript プログラムのマルチスレッド化

杉本卓哉[†1]　新城　靖[†1]　中井　央[†2]
板野肯三[†1]　佐藤　聡[†1]

ActionScript によるプログラミングでは C 言語や Java 言語などでのマルチスレッドプログラミングに慣れた開発者にとって制御フローの記述について制約をかかえている．ユーザ定義関数は以下の 3 つのケースにおいて，処理の途中で処理系にリターンする必要がある．3 つのケースとは，1 つはサーバからの応答を受け取る場合，1 つは画面の再描画の処理を行う場合，1 つは CPU 時間のかかる処理の際に Web ブラウザ全体のフリーズを回避する場合である．これらのケースにおいて処理の途中で処理系にリターンするために，ActionScript ではイベント駆動プログラミングが行われる．しかし，イベント駆動によるプログラミングでは単純な処理も煩雑なプログラムとなってしまう．本研究では，以上の問題を解決するために ActionScript でマルチスレッドプログラミングを可能にする．提案手法では，まず開発者は本手法において提供する API を用いてマルチスレッドのプログラムを記述し，それを標準のコンパイラでバイトコードにコンパイルする．次に，本手法において提供するコード変換器を用いて，バイトコード中の各関数に実行状態を保存して処理を中断するコードを挿入する．実行時には，本手法において提供するスレッド管理機構が最初の実行可能なスレッドの関数を呼び出す．呼び出された関数はそれぞれの処理を実行するが，挿入された中断コードが実行されるとスレッド管理機構にリターンする．スレッド管理機構は次の実行可能なスレッドの関数を呼び出す．提案手法を利用し，アニメーションやサーバからのファイルのダウンロードを行うプログラムをマルチスレッドで開発することができた．

Multithreading ActionScript Programs with Bytecode Translation

Takuya Sugimoto,[†1] Yasushi Shinjo,[†1]
Hisashi Nakai,[†2] Kozo Itano[†1] and Akira Sato[†1]

Programming in ActionScript has restrictions on coding control flows for developers who are familiar with mutithread programming in C, Java, and other languages. There are three typical cases when a user defines functions which must return to the runtime during a task. The first case is to receive a response from a remote server. The second case is to redraw the graphics. The third case is to prevent the Web browser from freezing in a CPU intensive task. In these cases, an event-driven programming style is used to return the control flow to the runtime. However, the event-driven programming style results in complex program codes even for writing simple procedures. To overcome these problems, we make it possible for developers to write multithread programs in ActionScript. In our scheme, first, a programmer writes a multithread program using our API, and compiles the program into bytecode using the standard ActionScript compiler. Second, our translator inserts code into each function in the bytecode program. This code will store the current execution state and suspend the execution of the function. At runtime, our thread manager calls the function of the first executable thread. This function executes its bytecode, and returns to the thread manager when the inserted code is executed. Our thread manager then calls the function of the next executable thread. With our scheme, we wrote several programs including animations and downloading files from servers using the multithread programming style.

1. はじめに

ActionScript は，Java や JavaScript と同様に，Web ブラウザの中で動作するプログラムを開発するための言語である．現在，多くのリッチインターネットアプリケーションが ActionScript の技術を利用している．Flash は ActionScript バイトコードの処理系を内蔵した Web ブラウザのプラグインである．Flash のコンテンツは SWF [1),⋆1] と呼ばれるフォーマットによって Web 上で配布され，HTML によって記述された Web ページに埋め込まれた形で表示される．開発者が，ActionScript を用いてユーザプログラムを記述することにより，インタラクティブ性や豊かな表現力を持ったリッチインターネットアプリケーションを実現することができる．

Web ブラウザの中で動作するプログラムでは，時間のかかる処理が必要になる．たとえば，遠隔のサーバに対して要求を送信した後，その応答を受け取るまで待つ処理が必要になる．こ

†1 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
Department of Computer Science, University of Tsukuba
†2 筑波大学大学院図書館情報メディア研究科図書館情報メディア専攻
Graduate School of Library, Information and Media Studies, University of Tsukuba

⋆1 SWF は Small Web Format，あるいは Shockwave Flash の略である．

© 2011 Information Processing Society of Japan

のとき，単純にブロックしてしまうと Web ブラウザ全体を停止させてしまう．ActionScript のプログラムにおいて全体のブロックを避ける場合，開発者はイベント駆動プログラミングを用いる．ActionScript のプログラムはその言語のみを習熟した開発者だけでなく，C 言語や Java 言語に習熟した開発者によって記述されることがある．C 言語や Java 言語では，開発者は全体のブロックを避ける処理をマルチスレッドスタイルで記述する．本論文では ActionScript によるプログラムの開発者を次の 2 つに分類する．

開発者 A　並行処理を記述する場合，マルチスレッドスタイルよりもイベント駆動スタイルで容易に記述できる開発者．

開発者 T　並行処理を記述する場合，イベント駆動スタイルよりもマルチスレッドスタイルで容易に記述できる開発者．

開発者 T にとって，ActionScript がマルチスレッドスタイルのプログラミングを支援していないことは問題となっている[2]．本研究で対象とする問題について，詳しくは 2 章で述べる．

このような問題を克服するため，本研究では，C 言語や Java において従来から用いられてきたマルチスレッドプログラミングのスタイルを ActionScript に導入する．これにより，開発者 T はイベント駆動スタイルを用いることなく，慣れ親しんだマルチスレッドスタイルを用いて ActionScript で並行処理を記述することを可能にする．本論文の貢献は，バイトコードレベルのコード変換を用いて ActionScript 言語におけるマルチスレッドプログラミング環境を実現したことである．提案手法では，開発者は ActionScript を標準のコンパイラでコンパイルしてバイトコードを得る．これに対し，本研究で提供するコード変換プログラムを適用する．コード変換プログラムは，バイトコード中の各関数に実行状態を保存して処理を中断し，後で再開することを可能とするバイトコードプログラムを挿入する．ランタイムライブラリはスレッド管理機構とライブラリ関数からなり，開発者が実装したプログラムにリンクして用いる．スレッド管理機構はスレッドを実現するために，コード変換プログラムによって変換された関数を呼び出す．

提案方式の利点として，まず，ActionScript の文法を変更しないので，既存のコンパイラや開発環境をそのまま利用できることがあげられる．次に，バイトコードのレベルで分岐命令やレジスタの操作が可能なので，効率良くスレッドのコンテキスト切替えが実現できることがあげられる．また，変換されたコードは手を加えていない標準の ActionScript 処理系で実行できるため，一般ユーザが利用しやすい．

本研究は ActionScript 3.0 を対象とする．以下，ActionScript 3.0 を単純に ActionScript と表記する[⋆1]．また，本研究で提案する手法ではマルチスレッドスタイルのプログラミングを利用するが，一方で開発者 A にとってはイベント駆動スタイルのプログラミングの方が自然である．このため，本研究は，開発者 T のみで構成された 1 人以上の開発者のグループを対象とする．開発者のグループの中に開発者 A が含まれている場合は，本研究の対象外とする．

本論文の構成を以下に示す．2 章では，ActionScript におけるプログラミングがかかえる制約について説明し，本研究が解決すべき問題を提示する．3 章では，関連研究について述べる．4 章では，ActionScript を用いる開発者にマルチスレッドスタイルのプログラミングを提案し，マルチスレッド化を実現するためのコード変換と実行時ライブラリについて述べる．5 章では，今回 ActionScript で行った本手法の実装について述べる．6 章では，本研究の有効性を検証する．最後に，7 章では，本論文をまとめ，残された課題について述べる．

2. ActionScript におけるイベント駆動プログラミング

ActionScript の処理系は，外部からの入力を受け取るなど，何かが起きるたびにイベントを送出する．**表 1** に，よく利用されるイベントを示す．ActionScript のプログラムは，これらのイベントに対応して呼び出されるユーザ定義関数の集合として記述される．このようなユーザ定義関数をイベントハンドラと呼ぶ．イベントハンドラを主に用いて行うスタイルのプログラミングは，イベント駆動プログラミングと呼ばれている．

ActionScript において並行処理を行うための機構としてイベント駆動が採用された理由としては，次のようなものがある[3]．

- 入出力でブロックを避けるために非同期入出力を用いるため．非同期入出力において，入出力の完了を知るためにイベントを用いる．

表 1　ActionScript でよく利用されるイベント
Table 1　Frequently used events in ActionScript.

イベント名	送出されるタイミング．
`complete`	何かの処理が完了した．
`click`	マウスのボタンがクリックされた．
`enterFrame`	定期的に送出され，画面の再描画が行われる．

⋆1 ActionScript と呼ばれるプログラム言語には，ほかに主に Flash Player 8 およびそれ以前で使われる ActionScript 1.0 や ActionScript 2.0 が存在する．ActionScript 2.0 およびそれ以前と ActionScript3.0 とでは，処理系の構造やバイトコードの形式が根本的に異なる．

© 2011 Information Processing Society of Japan

- デッドロックなどのスレッド間の同期にかかわる複雑性を排し，レビューを簡単にするため．
- ActionScript の処理系の実装を簡潔にし，スレッド実装にともなうオーバヘッドを避けるため．

ActionScript ではイベントハンドラで時間のかかる処理を続けることは問題がある．ActionScript の処理系は，ブラウザの中でシングルスレッドで動作している．1 つのイベントハンドラを実行しているときに別のイベントが生じたとしても，そのイベントの処理は，実行中のイベントハンドラがリターンした後に行われる．また，Web ブラウザの画面の再描画も，処理系に制御が戻ってきてから行われる．したがって，イベントハンドラがリターンしなかった場合，Web ブラウザ全体がフリーズしてしまう．

1 章で取り上げた開発者 A は，イベント駆動スタイルを用いて並行処理を行うプログラムを容易に記述することができる．しかし，開発者 T にとっては，イベント駆動による並行処理のプログラミングは次に示すような要因からプログラムの可読性が下がり，またプログラミングが難しくなる．

(1) スレッドを用いたプログラミングではブロックできるが，イベント駆動ではブロックできない．このため素直な実装では，イベント待ちを行う可能性のある部分で関数を分割する必要がある．たとえば，スレッドを使えば 1 つの関数で for 文を使ったループを記述しその中でブロックできるが，イベント駆動では複数の関数に分割する必要がある．また，関数の分割にともない，スレッドでは関数内の局所変数であった変数がイベント駆動では関数の外の大域変数やクラス変数になってしまう．

(2) スレッドを用いたプログラミングにおいては不要であった，関数をイベントハンドラとして登録し，登録解除を行う記述が，イベント駆動プログラミングにおいては必要になる．

これらの問題については，2.1 節，2.2 節，および 2.3 節で，具体的なプログラムの例をあげて説明する．

2.1　サーバとの通信を行うプログラム

図 1 は ActionScript を用いてサーバ通信を行うプログラムの実装例である．このプログラムは，ファイルを読み込み，ファイルの読み込みが完了した後にそれを画面に表示する．`Loader` オブジェクトは ActionScript に組み込みで用意されたオブジェクトであり，Web サーバに存在するファイルを読み込む機能を持つ．`Loader` オブジェクトの `load()` 関数は，Web サーバに存在するファイルを読み込む．`addChild()` 関数は，読み込まれた画像を画面に表示する．このプログラムは，開発者 A にとっては難しいが，開発者 T にとっては分かりやすい．

```
1 var loader:Loader;
2 function run():void {
3   loader = new Loader();
4   var request:URLRequest = new URLRequest("image.gif");
5   loader.load(request); //読み込みを実行する
6   addChild(loader.content); //読み込まれた画像を表示する
7 }
```

図 1　サーバとの通信を行う不完全なプログラム
Fig. 1　An incomplete program which communicates with a server.

```
 1 var loader:Loader;
 2 function run():void {
 3   loader = new Loader();
 4   var request:URLRequest = new URLRequest("image.gif");
 5   loader.contentLoaderInfo.addEventListener("complete", completeHandler);
 6   loader.load(request); //読み込みを開始する
 7 }
 8 function completeHandler(event:Event):void {
 9    addChild(loader.content); //読み込まれた画像を表示する
10 }
```

図 2　イベント駆動によるサーバとの通信を行うプログラム
Fig. 2　A program in event-driven style which communicates with a server.

図 1 のプログラムは，実際には期待された動作を行わない．理由は，`load()` 関数の動作にある．サーバからのファイルの読み込みには時間がかかる．単純にその完了を待っていたのでは，Web ブラウザ全体がフリーズしてしまう．これを避けるために，`load()` 関数はサーバに要求を送信すると応答の受信を待たずにリターンする．`load()` 関数を呼び出した後，ファイルの読み込みが完了すると処理系は `complete` イベントを送出する．ファイルの読み込みの完了を待って画像の表示を行うためには，`complete` イベントに対するイベントハンドラを登録する必要がある．

図 2 は，図 1 のプログラムにイベントハンドラを追加したプログラムである．`completeHandler()` 関数は，プログラムに追加したイベントハンドラである．このイベントハンドラを `Loader` によるファイル読み込みが完了した後に呼び出されるように登録するために，組み込みの `addEventListener()` 関数を用いている．このプログラムは，開発

© 2011 Information Processing Society of Japan

```
1 function run():void {
2   for (var i:int = 0; i < 100; i++) {
3     this.x += 5; // 表示オブジェクトを右に 5ピクセル動かす
4     redraw(); //再描画を行う
5   }
6 }
```

図 3　for 文によるアニメーションプログラム
Fig. 3　A program which displays an animation using the for statement.

```
1 function run():void {
2   addEventListener("enterFrame", enterFrameHandler);
3 }
4 var i:int = 0;
5 function enterFrameHandler(event:Event):void {
6   this.x += 5; // 表示オブジェクトを右に 5ピクセル動かす
7   if (++i >= 100)
8     removeEventListener("enterFrame", enterFrameHandler);
9 }
```

図 4　イベント駆動によるアニメーションプログラム
Fig. 4　A program which displays an animation in an event-driven style.

者 A にとっては分かりやすいが，開発者 T にとっては図 1 のプログラムよりも分かりにくくなっている．

この例では，イベント駆動によるプログラムに書き換えたことにより，1 つの処理を実装するために 2 つの関数を定義する必要が生じている．

2.2　アニメーションを行うプログラム

図 3 は for 文で記述されたアニメーションプログラムの実装例である．このプログラムは，表示オブジェクトを右に 5 ピクセルずつ，100 回動かしている．また，`redraw()` 関数は画面の再描画を行った後，次に画面の再描画が必要になるまで待つ関数である．しかし，実際の ActionScript には `redraw()` のような関数は存在しない．このプログラムは，開発者 A にとっては難しいが，開発者 T にとっては分かりやすい．

実際の ActionScript でアニメーションプログラムを実装するためには，イベント駆動プログラミングのスタイルを用いて，**図 4** に示すようなプログラムを記述する必要がある．この例では，`enterFrameHandler()` 関数が `enterFrame` イベントの発生により呼び出される．この際，プログラム中の表示オブジェクトが右に 5 ピクセルずつ動く．このプログラムは，開発者 A にとっては分かりやすいが，開発者 T にとっては図 3 のプログラムよりも分かりにくくなっている．

```
1 function run():void {
2   for (var i:int = 0; i < 10000; i++) {
3     heavyTask();
4   }
5 }
6 function heavyTask():void {
7   for (var i:int = 0; i < 10000; i++) {
8     //重い処理
9   }
10 }
```

図 5　CPU 時間を消費する処理を行うプログラム
Fig. 5　A program which consumes CPU time.

```
1 function run():void {
2   addEventListener("enterFrame", enterFrameHandler);
3 }
4 var i:int = 0;
5 function enterFrameHandler(event:Event):void {
6   heavyTask();
7   if (++i >= 10000)
8     removeEventListener("enterFrame", enterFrameHandler);
9 }
10 function heavyTask():void {
11   for (var i:int = 0; i < 10000; i++) {
12     //重い処理
13   }
14 }
```

図 6　イベント駆動による CPU 時間を消費する処理を行うプログラム
Fig. 6　A program which consumes CPU time in an event-driven style.

この例では，イベント駆動によるプログラムに書き換えたことにより，1 つの処理を実装するために 2 つの関数を定義する必要が生じている．ここでも，2.1 節の例と同じような問題が発生している．さらに，図 3 のように繰返し構造を for 文を用いて記述することはできなくなっているという問題も発生している．

2.3　CPU 時間を消費するプログラム

図 5 は CPU 時間を消費する処理を行う `heavyTask()` 関数を 10,000 回繰り返して呼び出すプログラムである．このプログラムは，開発者 A にとっては難しいが，開発者 T にとっては分かりやすい．このプログラムを Web ブラウザ上で実行すると，すべての計算が

© 2011 Information Processing Society of Japan

終わるまでの間 Web ブラウザがフリーズしてしまう．Web ブラウザのフリーズを避けるためには，**図 6** に示すような，イベントハンドラを用いたプログラムを記述する必要がある．図 6 のプログラムでは，`heavyTask()` 関数を 1 回呼ぶごとにイベントハンドラからリターンしている．このプログラムは，開発者 A にとっては分かりやすいが，開発者 T にとっては図 5 のプログラムよりも分かりにくくなっている．

ここでも，2.2 節に示した問題と同様の問題が生じている．

3. 関連研究

ライブラリにより ActionScript でイベントハンドラの登録・解除の煩わしさを軽減する手法がいくつか提案されている．PseudoThread は，2.3 節で述べたようなループを含む CPU 時間を消費するプログラムを簡単に記述するためのライブラリである[4]．PseudoThread ライブラリを利用する開発者は，ループ処理の 1 回分の処理を．関数として定義する．PseudoThread ライブラリは，この関数を定期的に呼び出す．ActionScript Thread Library（そうめん）は，PseudoThread よりも汎用性があり，ループ以外の処理も記述することができる[2]．そうめんライブラリを利用する開発者は，1 つの処理の途中で中断したい場合，その処理を複数の関数に手作業で分割して記述する．各関数は，末尾で次に実行すべき関数をそうめんライブラリに返す．そうめんライブラリは，指定された関数を実行したり，別の制御の流れに属する関数を実行したり，あるいは，ActionScript 処理系に制御を戻したりする．本研究の手法では，これらの手法とは異なり，開発者は 1 つの処理をライブラリ側の都合で複数の関数に分割する必要はない．また，バイトコードレベルのコード変換を用いている点が異なる．

JavaScript 言語は，Web ブラウザの上で動作する処理系である．文献 5) では，JavaScript におけるイベント駆動プログラミングスタイルに関連して同期処理を非同期処理で記述しなければならないという問題があることを指摘している．ActionScript にも類似の問題がある．JavaScript のプログラムに対してコード変換を行い，JavaScript によるマルチスレッドプログラミングを実現する試みとしては，研究 6) やライブラリ 7) がある．JavaScript ではプログラムのソースコードをプログラムから取得することが容易である．そのため，これらのシステムでは JavaScript におけるコード変換を実行時にソースコードレベルで行う．これに対して，本研究では ActionScript のプログラムを実行前にあらかじめバイトコードコードレベルで変換することによりマルチスレッドプログラミングを実現する手法を提案する．

Web Workers[8] は，JavaScript の処理系自体を改良し，マルチスレッドプログラミングを可能とする技術である．Web Workers により提供されるスレッドは本研究や研究 6) などが提供するコルーチンに基づくマルチスレッドではなく，OS レベルのプリエンプティブなスレッドである．このため，Web Workers によるスレッドは低オーバヘッドであり，マルチコアプロセッサの恩恵を受ける．一方，それぞれの Web Workers スレッドはオブジェクトを共有しておらず，スレッド間の通信は文字列で行う必要がある．他にも，`document` オブジェクトの操作を行うことができるスレッドを新たに生成することはできないなど，Web Workers が提供するスレッドにはいくつかの制限がある．本研究の手法が提供するマルチスレッドはコルーチンに基づくマルチスレッドであるが，すべてのスレッドがすべてのオブジェクトにアクセス可能である．

Java のバイトコードを変換することによってプログラムの機能を変更する試みとしては Javassist[9] が存在する．本研究では ActionScript のバイトコードを変換することによってプログラムの機能を変更する．バイトコード変換を実現するため，本研究では swfassist[10] と Tamarin[11] の ABC（ActionScript Byte Code）アセンブラと逆アセンブラを組み合わせて用いる．Javassist を用いたコード変換を制御するためには，Java の機能であるアノテーションを用いることができる．本研究においても，コード変換の対象となるプログラムを指定するためにアノテーションを用いる．本研究のコード変換は ActionScript を対象とする．

C 言語や C++言語で書かれたプログラムを ActionScript バイトコードにコンパイルする技術として，Adobe Alchemy[12] が存在する．Alchemy は LLVM[13] を利用することにより，Web ブラウザをフリーズさせることなく実行することができる ActionScript のバイトコードプログラムを生成する．本研究と Alchemy は，いずれも ActionScript の処理系の上に何らかのプログラムを動かすための機構を設ける．しかし，Alchemy は C 言語や C++言語で書かれたプログラムを対象とする．本研究は ActionScript で記述されたプログラムを対象とする．

Adobe が Flash Player の一部として提供している PixelBender[14] は，科学技術計算や画像処理など，CPU に負荷がかかる処理を Flash Player 上で並列に行うための仕組みである．PixelBender のプログラムは ActionScript とは異なる言語で記述される．また，専用の開発環境が用意されている．一方，PixelBender では ActionScript のオブジェクトにアクセスすることができず，できることは数値演算などに限られる．このことから，PixelBender は 1 章で述べた開発者 T がマルチスレッドスタイルのプログラムを記述するために利用す

© 2011 Information Processing Society of Japan

ることはできない．本研究の手法は，ActionScript を用いたマルチスレッドスタイルのプログラミングを行うためのものである．

文献 15) では，C 言語におけるユーザレベルのスレッド・ライブラリの実現に継続を利用している．また，様々なプログラミング言語でスレッドの実装に必要なコードをコード変換の過程で挿入する方法がすでに提案されている[16)–19)]．本研究では，ActionScript 言語におけるスレッドの実装にバイトコード変換を用いる方法を提案する．

4. 提案手法

4.1 概要と構成

ActionScript のイベント駆動プログラミングは，1 章で取り上げた開発者 T にとっては 2 章で説明したような問題をかかえている．一方で，C 言語や Java では，ライブラリや処理系が提供するスレッド API を用いてマルチスレッドプログラミングを行うことが可能である．マルチスレッドプログラミングでは，時間のかかる処理を別のスレッドで動かすことができる．ActionScript でスレッドを使うことが可能ならば，開発者 T は 2 章で述べたような簡単な記述で時間のかかる処理を行うプログラムを実装することが可能になる．

本研究は，ActionScript において時間のかかる処理をイベントハンドラを用いずに逐次的に記述できるようにすることを目的とする．目的を達成する手段として，本研究では，ActionScript にスレッドを導入して，マルチスレッドプログラミング環境を実現することを提案する．

図 7 は，ActionScript を用いる開発者が提案手法を利用して開発を行うワークフローである．まず，開発者は提案手法が提供する API を利用し，マルチスレッドプログラミングによってプログラムを記述する．次に，開発者は通常の ActionScript コンパイラを利用し，ソースコードから中間 SWF ファイルを得る．最後に，開発者はこの中間 SWF ファイルを提案手法が提供する**コード変換プログラム**によってコード変換し，変換後の SWF ファイルを得る．変換後の SWF ファイルが Web ブラウザで開かれると，SWF ファイルに組み込まれた**ランタイムライブラリ**はコード変換プログラムによって変換されたプログラムをマルチスレッド的に実行する．提案手法における開発者が記述するプログラムについては，4.2 節で説明する．

本手法では，バイトコードレベルの変換を採用した．これには以下の 3 つの利点がある．1 つ目の利点は，本手法では，ActionScript の文法には変更を加えないことである．このため，開発者は新たなプログラミング言語を覚える必要がない．また，開発者は既存の開発環境をマルチスレッドプログラミングにおいてもそのまま利用できる．さらに，標準の ActionScript コンパイラによってコンパイル時に型エラーをチェックすることができる．2 つ目の利点は，ActionScript の処理系に変更を加えないことである．このため，既存の Flash プラグインでマルチスレッドの ActionScript アプリケーションを利用できる．3 つ目の利点は，コード変換の際に ActionScript から直接扱うことができないバイトコード命令を利用できることである．たとえば，関数内の任意の位置へのジャンプを行うバイトコード命令である `jump` は，対応する ActionScript の言語要素が存在しない．このため，ソースコードレベルの変換では，関数内の任意の位置に直接ジャンプすることができない．

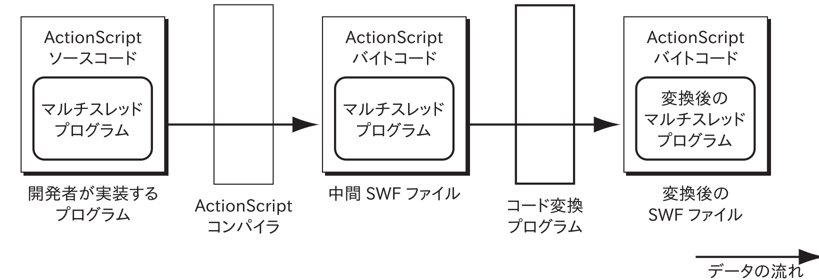

図 7　提案手法のワークフロー
Fig. 7　The workflow of the proposed method.

本手法では，ActionScript プログラムに対してコード変換を行い，コルーチンに基づくマルチスレッドを実現する．ネイティブにスレッドをサポートしない ActionScript 処理系においてスレッド間のコンテキスト切替えを可能とするために，本手法では以下の 4 つの要素を用いる．

中断可能関数　処理を中断して後で再開することができる関数である．
継続オブジェクト　中断可能関数の実行状態を保存するオブジェクトである．
スレッドオブジェクト　スレッドを表現するオブジェクトである．
駆動プログラム　現在実行中のスレッドオブジェクトを管理するプログラムである．

このうち，中断可能関数は開発者が実装したプログラムをコンパイルして生成されたバイトコードを入力として，本手法において提供されるコード変換プログラムによって生成される．残りの 3 つ，すなわち継続オブジェクト，スレッドオブジェクト，駆動プログラムは，本手法において提供されるランタイムライブラリの一部である．

図 8 に，変換後の SWF ファイルが Web ブラウザ内で実行されている様子を示す．駆動

© 2011 Information Processing Society of Japan

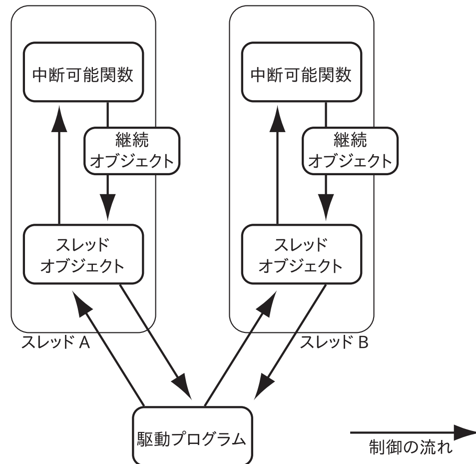

図 8　提案手法の構成
Fig. 8　The structure of the proposed method.

プログラムは実行可能なスレッドオブジェクトを選び，制御を渡す．スレッドオブジェクトは，制御を受け取ると保持している継続オブジェクトから中断可能関数を取り出して，それを呼び出す．中断可能関数は開発者が記述したとおりの処理を行うが，コード変換により挿入されたコードを実行すると，現在の関数の実行環境を継続オブジェクトに保存して，制御をスレッドオブジェクトに戻す．スレッドオブジェクトはこの継続オブジェクトを保存して，制御を駆動プログラムに返す．駆動プログラムは，ここで次の実行可能なスレッドオブジェクトに制御を渡す．

中断可能関数と継続オブジェクトの実現については，4.3 節で説明する．駆動プログラムとスレッドオブジェクトについては，4.4 節で説明する．

4.2　マルチスレッドプログラミング

開発者は以下の手順を踏み，提案手法を利用したマルチスレッド ActionScript プログラムを開発する．

提案手法では，スレッドとして実行可能にするためにバイトコード変換の対象となる関数にのみ [Suspendable] というアノテーションを付加する．アノテーションが付加されていない関数は，変換せずにそのまま出力する．特定の関数の並行処理のためのマークを付加することは，Cilk-5 言語[20] など，多くのプログラミング言語で用いられている手法である．

表 2　関数呼び出しの可否
Table 2　Semantics of function calling.

		呼び出し先	
		通常の関数	中断可能関数
呼び出し元	通常の関数	可	不可
	中断可能関数	可	可

```
1 new Thread(func, [arg1, arg2, ...]).start();
```

図 9　新しいスレッドを生成するプログラム
Fig. 9　A program to create a new thread.

また，Java 言語[21] においても，意味は異なるが synchronized という構文を導入している．すべての関数をバイトコード変換の対象とすることも考えられるが，本研究では特定の関数だけを変換の対象とした．その理由は，CPU 時間を消費する処理において，通常の関数と中断可能関数を組み合わせることで，オーバヘッドをともなう中断にともなう処理の頻度を調整しやすくするためである．この調整について詳しくは，6.2 節で述べる．

中断可能関数と通常の関数は関数の呼び出しに関して異なる性質を持つ（**表 2**）．中断可能関数と通常の関数は，直接呼び出すことができる関数が異なる．通常の関数は，通常の関数のみを直接呼び出すことができる．一方，中断可能関数は，通常の関数に加えて中断可能関数を呼び出すことができる．

中断可能関数を通常の関数から利用するためには，スレッドを生成する．スレッドを生成するためには，スレッドオブジェクトを作成したうえで中断可能関数とそれに渡す引数を指定し，スレッドオブジェクトの `start()` メソッドを呼び出す．この様子を**図 9** に示す．スレッドの生成は，通常の関数と中断可能関数の両方から行うことができる．

通常の ActionScript と異なり，中断可能関数内では for 文によるループの途中で定期的に制御が ActionScript の処理系に戻る．このため，開発者は CPU 時間を消費するループを細切れに分割することなく，for 文を利用して素直に記述することができる．逆に通常の関数が実行中の間，制御が処理系に戻ることはない．この性質を利用することでアトミックな処理を実現することができる．たとえば，アニメーションにおいて 1 つの図形をアトミックに作成したい場合，その処理を通常の関数として記述して，中断可能関数から呼び出せばよい．

本研究では，マルチスレッドプログラミングを行うことを助けるためのいくつかのライブ

© 2011 Information Processing Society of Japan

表 3　提案機構が提供するライブラリ関数
Table 3　Library methods offered by proposed method.

関数名	関数の説明
yield()	制御を処理系に戻す．
redraw()	画面を再描画するまでスレッドを一時停止する．
sleep()	指定した時間が経過するまでスレッドを一時停止する．
wait()	指定したイベントが送出されるまでスレッドを一時停止する．

```
1 [Suspendable]
2 function run():void {
3   var loader:Loader = new Loader();
4   var request:URLRequest = new URLRequest("image.gif");
5   loader.load(request); //読み込みを開始する
6   wait(loader, Event.COMPLETE);
7   addChild(loader.content); //読み込まれた画像を表示する
8 }
```

図 10　提案手法を用いたサーバとの通信を行うプログラム
Fig. 10　A program using the proposed method which communicates with a server.

ラリ関数を提供する．本研究で実装したライブラリ関数を**表 3** に示す．これらのライブラリ関数は中断可能関数から呼び出すことができる．ライブラリ関数 wait() は，入出力の完了などの標準のイベントのほかに，ユーザ定義のイベントも待つことができる．この機能を利用することで，C 言語（Pthreads）や Java 言語における条件変数を使った待ち合わせと似た処理を記述することができる．たとえば，ある条件が満たされるまで待ちたいスレッドがユーザ定義イベントを引数にして wait() を呼び出し，別のスレッドがそのユーザ定義イベントを発生させ待っているスレッドを起こすことができる．

以上に述べた相違点を除いて，開発者の目から見た中断可能関数と通常の関数は同じである．たとえば，開発者は通常の関数と同じ ActionScript の文法を用いて中断可能関数を記述する．また，通常の関数と中断可能関数はともに ActionScript の処理系が提供する組み込みの機能を利用することができる．

中断可能関数を実装することによって 2 章で述べた問題を解決できることを，4.2.1 項，4.2.2 項，および 4.2.3 項で示す．

4.2.1　サーバとの通信を行う中断可能関数

図 10 は，2.1 節で例示した処理と同一の処理を行う中断可能関数である．この例では，

```
1 [Suspendable]
2 function run():void {
3   for (var i:int = 0; i < 100; i++) {
4     this.x += 5; // 表示オブジェクトを右に 5ピクセル動かす
5     redraw(); //再描画を行う
6   }
7 }
```

図 11　提案手法を用いたアニメーションプログラム
Fig. 11　A program which displays an animation using the proposed method.

```
1  [Suspendable]
2  function run():void {
3    for (var i:int = 0; i < 10000; i++) {
4      heavyTask();
5    }
6  }
7  [Suspendable]
8  function heavyTask():void {
9    for (var i:int = 0; i < 10000; i++) {
10     //重い処理
11   }
12 }
```

図 12　提案手法を用いた CPU 時間を消費する処理を行うプログラム
Fig. 12　A program which consumes cpu time using the proposed method.

図 2 における `addEventListener()` 関数の呼び出しを `wait()` 関数で置き換えることにより，処理の流れを 1 つの関数にまとめて記述することができている．

4.2.2　アニメーションを行う中断可能関数

図 11 は，2.2 節で例示した処理と同一の処理を行う中断可能関数である．この例は，図 3 のプログラムに `[Suspendable]` の記述を付加したものである．この例では，処理の流れを 1 つの関数にまとめて記述することができているうえ，さらに繰返し構造を for 文を用いて記述することができている．

4.2.3　CPU 時間を消費する処理を行う中断可能関数

図 12 は，2.3 節で例示した処理と同一の処理を行う中断可能関数である．この例は，図 5 のプログラムに `[Suspendable]` アノテーションの記述を付加したものである．この例では，Web ブラウザをフリーズさせることなく，また処理を細かく分割することもなく CPU 時間を消費する処理を行うことが可能となっている．

© 2011 Information Processing Society of Japan

4.3　中断可能関数と継続オブジェクト

本研究では，バイトコードレベルのコード変換によって中断可能関数を生成する．コード変換では，関数の命令列中に継続オブジェクトを返す命令列を挿入する．ここで挿入される命令列を，本論文では**中断コード**と呼ぶことにする．中断コードは，中断可能関数の現在の実行状態を継続オブジェクトに保存してリターンするためのコードと中断可能関数が継続オブジェクトから実行状態を読み込んで再開するためのコードから構成される．

中断可能関数の性質については，4.3.1 項で述べる．継続オブジェクトについては，4.3.2 項で述べる．中断コードについては，4.3.3 項で述べる．

4.3.1　中断可能関数

中断可能関数は，次の 2 つのうち 1 つを引数としてとる．

(1)　本来の引数．

(2)　継続オブジェクト．

中断可能関数を最初に呼び出すときは，本来の引数を渡す．中断可能関数の実行を再開するときは，継続オブジェクトを渡す．中断可能関数は，このどちらを受け取ったかを判断するために，引数の型を調べる．

中断可能関数は，次の 2 つのうち 1 つを返値として返す．

(1)　本来の返値．

(2)　継続オブジェクト．

中断可能関数の実行が完了したときは，呼び出し元は本来の返値を受け取る．中断可能関数の実行が中断したときは，呼び出し元は継続オブジェクトを受け取る．中断可能関数の呼び出し元は，このどちらを受け取ったかを判断するために，受け取った返値の型を調べる．

4.3.2　継続オブジェクト

継続オブジェクトには以下の 4 つの値が格納される．

(1)　中断可能関数への参照．

(2)　中断可能関数の実行を再開する地点．

(3)　中断可能関数の実行状態．

(4)　次に実行する処理を表現する継続オブジェクトへの参照．

図 13 のようなプログラムを実行すると，複数の継続オブジェクトが生成され，それらが 1 つのリストにつながれる．この様子を**図 14** に示す．この場合，継続オブジェクトは 2 つ作成される．関数 `g()` は最初に作成された継続オブジェクトに実行状態を保存し，それを関数 `f()` に返す．関数 `f()` は別の継続オブジェクトを作成して実行状態を保存し，関数 `g()` から受け取った継続オブジェクトの次に実行する処理として登録する．関数 `f()` は関数 `g()` から受け取った継続オブジェクトを返値としてスレッドオブジェクトに返す．スレッドオブジェクトがこの実行状態を再開するときは，まず関数 `g()` の継続オブジェクトの実行状態を復元する．関数 `f()` の実行状態の復元は，関数 `g()` の実行が完了するまで遅延される．

```
1  [Suspendable]
2  function f():int {
3    return g();
4  }
5  [Suspendable]
6  function g():int {
7    yield(); //ここで中断処理が行われる
8    return 1;
9  }
```

図 13　中断可能関数から他の中断可能関数を呼び出すプログラム
Fig. 13　A program which calls a suspendable function from another suspendable function.

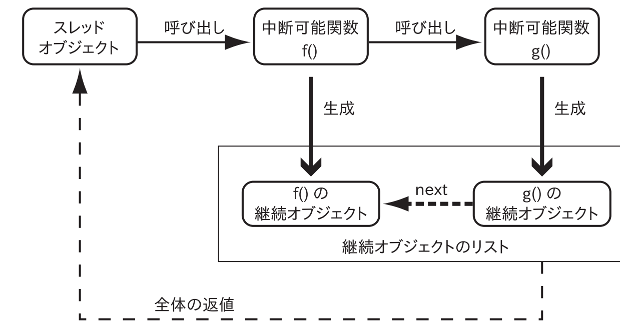

図 14　複数の中断可能関数の中断
Fig. 14　Suspending multiple suspendable functions.

4.3.3　中断コード

本研究では，3 種類の中断コードを用いた．

1 つ目は，中断可能関数内に記述された `yield()` 関数の呼び出しから変換される中断コードである．この中断コードでは，必ず中断処理が行われる．これは，本研究で提供するライブラリ関数のうち，`redraw()` 関数や `wait()` 関数を実装するために用いた．これらのライブラリ関数では，スレッドを待ち状態に移行したうえで `yield()` の処理を行う．

2 つ目は，中断可能関数内で関数呼び出しが行われている場所に挿入される中断コードで

© 2011 Information Processing Society of Japan

ある．この中断コードでは，呼び出された中断可能関数が継続オブジェクトを返した場合に，中断処理を行う．呼び出された関数が返値として継続オブジェクトではない値を返した場合は，そのまま処理を続ける．

3 つ目は，中断可能関数内のループ中に挿入される中断コードである．これは，for 文などによって中断可能関数内で CPU 時間を消費するループを実装した際に，ループの途中でいったん中断可能関数からリターンするためのものである．この中断コードについては，通過した回数をグローバル変数を用いて数え，グローバル変数が一定の値に達するごとに中断処理を行う．これは，短いループにおいて 1 回の繰返しごとに中断処理を行うことは効率が悪いためである．現在の実装では，この値として標準で 100 を用いている．

コード変換によって埋め込まれる中断コードについて述べる．実際のコード変換は ActionScript のバイトコードに対して行われる．ActionScript のバイトコードを直接示すのは煩雑である．代わりに ActionScript 風の擬似コードを用いる．

図 15 は，図 11 のプログラムを変換して生成された中断可能関数である．

ソース中の `Continuation` は継続オブジェクトの型であり，`c` は変換処理によって追加された引数である（図 15 の 1 行目）．関数の先頭には `c` が継続オブジェクトであった場合に継続オブジェクトから状態を再開するコードが挿入される（図 15 の 2–6 行目）．

ライブラリ関数 `redraw()` 関数は中断可能関数である．このため，`redraw()` 関数の返値の型を調べ，それが継続オブジェクトであれば中断処理を行うコードが挿入されている（図 15 の 9–16 行目）．ここで中断処理が行われると，4.3.2 項で述べたように，`redraw()` 関数が生成する継続オブジェクトの後にこの中断可能関数が生成する継続オブジェクトがつながれる（図 15 の 13 行目の `queue()` 関数）．この地点で中断処理が行われた場合，処理はこの中断コードの直後（図 15 の 15 行目）から再開される．

元の関数でリターンが行われていた場所では，呼び出し元に制御を移すためにコード変換を行う（図 15 の 24–31 行目）．ここでは，この中断可能関数が直接呼び出されたのか，あるいはスレッドオブジェクトから呼び出されたのかを調べる（図 15 の 24 行目）．この中断可能関数がスレッドオブジェクトから呼び出されていた場合，継続オブジェクトにつながれた続きの継続オブジェクトに返値を渡し，実行を再開する（図 15 の 26–27 行目）．続きの継続オブジェクトは，図 15 の 4 行目のような処理を行い，渡された返値を受け取る．この中断可能関数が呼び出し元の関数から直接呼び出された場合は，返値を直接リターンすることにより呼び出し元の関数に返値を渡す（図 15 の 32 行目）．

本手法では，関数呼び出しを行う際に，それが通常の関数であるか中断可能関数であるかを区別しない．そのため，中断可能関数の中ではすべての関数呼び出しについて，このような中断処理を行うコードを挿入する．また，ループの末尾にも中断コードを挿入する（図 15 の 17–22 行目）．このコードは，ループを通過した回数を数え，それが一定の値に達するごとに中断処理を行う．

```
1  function run(c:Continuation = null):Continuation {
2    if (c) {
3      i = c.locals[1];
4      c1 = c.returnValue;
5      goto c.label;
6    }
7    for (var i:int = 0; i < 100; i++) {
8      this.x += 5; // 表示オブジェクトを右に 5ピクセル動かす
9      var c1:* = redraw();
10     if (c1 is Continuation) {
11       if (c == null) c = new Continuation();
12       c.func = this.run; c.label = $L1; c.locals[1] = i;
13       c1.queue(c);
14       return c1;
15       $L1:
16     }
17     if ( ++loop_counter >= max_loop_counter ) {
18       if (c == null) c = new Continuation();
19       c.func = this.run; c.label = $L2; c.locals[1] = i;
20       return c;
21       $L2:
22     }
23   }
24   if (c) {
25     if (c.next) {
26       c.next.returnValue = null;
27       return c.next;
28     } else {
29       return null;
30     }
31   }
32   return null;
33 }
```

図 15　コード変換後のプログラム
Fig. 15　Pseudo code of a program after code translation.

4.4　駆動プログラムとスレッドオブジェクト

本手法において提供される駆動プログラムはすべてのスレッドオブジェクトを管理する．

スレッドオブジェクトは `start()` メソッドが呼ばれると自身を駆動プログラムに登録して実行可能状態に移行する．実行可能状態のスレッドオブジェクトは駆動プログラムから制

© 2011 Information Processing Society of Japan

御を受け取るたび，自身に登録された継続オブジェクトから中断可能関数を取り出し，それに継続オブジェクトを渡す．継続オブジェクトを渡された中断可能関数は実行を再開する．中断可能関数が継続オブジェクトを返した場合，スレッドオブジェクトはそれを保持し，制御を駆動プログラムに渡す．中断可能関数が継続オブジェクト以外の型を持つ値を返した場合，スレッドは動作を終了する．

スレッドは，ライブラリ関数 `redraw()`，`sleep()`，`wait()` が呼び出されると，待ち状態に移行する．待ち状態のスレッドは，イベントハンドラで対応するイベントを受信したときに実行可能状態にする．

5. 実　　装

5.1　ActionScript Byte Code と ActionScript Virtual Machine

ABC のプログラムは SWF に格納されている．ABC を実行することができる処理系は，ActionScript Virtual Machine，または縮めて AVM と呼ばれている．AVM は Flash Player の一部である．

AVM は ABC のプログラムを読み込むと，バイトコードを実行する．AVM における関数フレームには以下の 3 つのメモリ領域が存在する．

引数領域　引数を格納する．

スコープ領域　名前解決を行う対象のオブジェクトを格納する．

ローカルレジスタ　バイトコードプログラムが自由に利用することができる．

これらの領域の大きさは関数ごとに異なる．これらの値は ABC の中に記述されている．

また，AVM はバイトコードプログラムの実行に先立ち，ABC データが範囲外のメモリをアクセスしたり，不適切な型変換を行ったりしてしまうなどの危険な挙動を行わないよう，ABC データに対してコード検証を行う機能を持つ．AVM はコード検証の際，バイトコードに含まれる命令列を抽象的に実行し，起こりうる実行の流れをシミュレートする．コード検証では，引数スタックやスコープスタックの深さが深くなりすぎないことを，また引数スタックやローカル変数に含まれる値の型が矛盾を起こさないことをチェックする．

5.2　ABC のコード変換

本研究では，SWF ファイルに含まれるデータのうち，ABC についてコード変換を行い，それ以外のデータをそのまま保持する．今回の実装では，これを ActionScript のプログラムとして実現した．SWF および ABC はバイナリフォーマットであるため，直接コード変換を行う ActionScript のプログラムを実装することは困難である．このため，本手法では

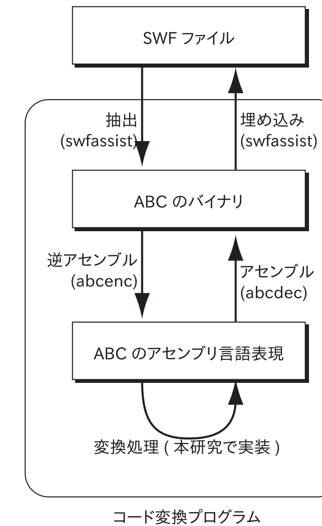

図 16　コード変換プログラムの構造
Fig. 16　The structure of the code translator.

SWF ファイルから ABC のバイナリを抽出したうえで，ABC を逆アセンブルし，得られたアセンブリ言語表現に対してコード変換を行う．本手法のコード変換プログラムは，ABC に含まれる関数のうち，`[Suspendable]` アノテーションの付加されたものを変換する．最後に，アセンブリ言語表現をアセンブルし，得られた ABC のバイナリを SWF ファイルに埋め込む．

ActionScript からローカルのファイルシステムに対する読み書きを行うためには，AIR ランタイム[22] を用いた．本手法において用いるコード変換プログラムの実装にあたっては，**図 16** に示すソフトウェアを利用した．SWF ファイルから ABC によるバイトコードプログラムを抽出し，また変換されたバイトコードプログラムを再度 SWF ファイル中に埋め込む処理は，swfassist[10] を利用して行った．アセンブラと逆アセンブラについては，Tamarin[11] に付属する `abcenc` プログラムと `abcdec` プログラムを用いた．`abcenc` プログラムが用いるアセンブリ言語は JSON フォーマットを用いている．これを ActionScript のオブジェクトとして読み込み，プログラムから操作するために，as3corelib[23] 内の JSON パーサに手を加えて利用した．

© 2011 Information Processing Society of Japan

```
 1 class Continuation {
 2   var func:Function;
 3   var label:int;
 4
 5   var operands:Array;
 6   var scopes:Array;
 7   var locals:Array;
 8   var returnValue:*;
 9
10   var next:Continuation = null;
11
12   function queue(c:Continuation):Continuation;
13   function resume():*;
14   function resumeWithReturnValue(value:*):*;
15 }
```

図 17　継続オブジェクト
Fig. 17　Continuation Object.

5.3　継続オブジェクトの実装

4.3.2 項で述べた継続オブジェクトの ActionScript における実装を**図 17** に示す．最初の 6 つの変数は，中断可能関数の実行状態を保存するための変数である．最後の 3 つの関数は継続オブジェクトを使いやすくするためのユーティリティ関数である．

中断可能関数への参照は `func` に保存される．中断可能関数の実行を再開する位置は `label` に保存される．`label` は中断可能関数内の命令のメモリアドレスを間接的に表現する整数値である．継続オブジェクトから関数の実行状態を復元する際は，まず実行関数を呼び出し，次に `label` の値に応じて最後に処理を中断した中断コードの直後に制御を移す．図 17 の，`operands`，`scopes`，および `locals` は，関数フレームに含まれる引数領域，スコープ領域，ローカルレジスタの 3 つを保存するための ActionScript の配列である．図 17 に存在する `next` は，4.3.3 項で述べた，継続オブジェクトのリストを表すためのフィールドである．

継続オブジェクトのユーティリティ関数について述べる．`queue()` 関数は中断可能関数の実行が中断されるときに呼び出され，`next` フィールドによって形成されるリンクリストの末尾に継続オブジェクトを追加する．`resume()` 関数と `resumeWithReturnValue()` 関数は中断可能関数の実行を再開する．このうち，`resume()` 関数はスレッドオブジェクトから呼び出される．`resumeWithReturnValue()` 関数は 4.3.3 項で述べたような場合に用いられる．

5.1 節で述べたとおり，AVM は ABC 読み込み時にコード検証を行い，関数フレームに格納された値の静的な型情報に一貫性があることを保証する．これは配列から関数フレームに値を書き戻す際に問題になる．ActionScript の配列には任意の型の値を格納することができるため，AVM はコードの検証に失敗する．この問題を防ぐため，今回行った実装ではオブジェクトを `*` 型にキャストするバイトコード命令を挿入した．この処置によって関数の行う処理は変化しないが，実行時の型検査によるオーバヘッドが発生する．

6.　評　　価

6.1　記　述　性

4.2 節では，本手法を用いることにより，従来の ActionScript においてイベント駆動によって記述する必要があった処理をマルチスレッドスタイルで記述することが可能であることを示した．マルチスレッドプログラミングには，ActionScript におけるイベント駆動プログラミングには存在しなかった，以下の利点がある．

(1)　本手法でスレッドを用いたプログラミングでは関数の途中でブロックできる．特に for 文を使ったループを記述しその中でブロックすることができることは，記述性を大きく高める．イベント駆動では，イベント待ちのために 1 つの処理を複数の関数に分割しなければならないことがあったが，本手法ではそのための分割は不要である．関数を分割する必要がないので，関数内の局所変数が使いやすい．

(2)　イベントハンドラを登録・解除する処理を削減することができる．

本手法では，通常の関数から中断可能関数（[Suspendable] アノテーションが付加されている）を呼び出すことはできない．この呼び出しを自動的に検出することは容易ではなく，現在実装されていない．したがって，コードレビューにおいてこの点を確認する必要があり，コードレビューの手間が増大する．また，他のマルチスレッドを採用した言語と同様に，デッドロックなどのスレッド間の同期に関わる問題にも直面するすることになる．

6.2　オーバヘッド

本手法を利用することによるオーバヘッドは，その発生源によって 2 つを考えることができる．1 つは，本手法のコード変換プログラムによりコードが変更されることによって生じるオーバヘッドである．もう 1 つは，スレッド管理機構を駆動することによって生じるオーバヘッドである．

これらのオーバヘッドを計測するために，本研究ではループや関数呼び出しを含む ActionScript プログラムを通常の関数と中断可能関数を用いて実装し，実行時間を測定した．実行時間の測定に用いた環境を，**表 4** に示す．

© 2011 Information Processing Society of Japan

表 4　実行時間の測定に用いた環境
Table 4　The environment for the benchmark test.

OS	Windows 7 Professional 32 bit
プロセッサ	Intel Core i7 2.67 GHz
メモリ	6 GB
Flash Player	バージョン 10.1.102.64

表 5　提案機構のオーバヘッド
Table 5　Overhead of proposed method.

操作	1 回あたりの実行時間	
	通常の関数	中断可能関数
for 文による 1 ループ	2.3 ns	11.4 ns
通常の関数の呼び出し	3.8 ns	21.9 ns
中断可能関数の呼び出し	–	35.6 ns
コンテキスト切替え	–	252.7 ns

通常の関数と中断可能関数の実行時間の差から，提案機構が持つオーバヘッドを計算した．その結果を**表 5** に示す．

まず，for 文によるループを中断可能関数に変換すると実行時間が 1 ループあたり 9.1 ナノ秒増加する．これは，今回の実装で，5.3 節で述べたように，コード中にキャストを行う処理を挿入したためである．次に，通常の関数における空の関数呼び出しの実行時間は 1 回あたり 3.8 ナノ秒であるのに対し，中断可能関数における空の通常の関数呼び出しの実行時間は 1 回あたり 21.9 ナノ秒と，通常の関数からの関数呼び出しと比べて 18.1 ナノ秒増加した．これは，4.3.3 項で述べたように，中断可能関数が他の関数を呼び出す際には，返値の型を調べて条件分岐を行う必要があるためである．中断可能関数における空の中断可能関数呼び出しの実行時間は 35.6 ナノ秒であり，中断可能関数から通常の関数を呼び出した場合と比べて 13.7 ナノ秒増加している．これは，中断可能関数が引数の型を調べて条件分岐を行う必要があるためである．スレッドのコンテキスト切替えには 1 回あたり 252.7 ナノ秒を要した．

ここまで述べたように，提案機構を用いたコード変換にはオーバヘッドが存在する．これが問題となるのかは，アプリケーションに依存する．

4.2.1 項で述べた，サーバからのファイルのダウンロードを行うようなアプリケーションにおいては，このオーバヘッドは実用上の問題とならない．これは，サーバからの応答の受信においては，通信遅延がミリ秒から秒オーダと，提案機構のオーバヘッドと比べて大きいためである．

```
 1 [Suspendable]
 2 function run():void {
 3   //ループ回数を変更し，処理の粒度を調整する
 4   for (var i:int = 0; i < 1000; i++) {
 5     heavyTask();
 6   }
 7 }
 8 //重い処理を，中断可能関数ではなく通常の関数で実装する
 9 function heavyTask():void {
10   //ループ回数を変更し，処理の粒度を調整する
11   for (var i:int = 0; i < 100000; i++) {
12     //重い処理
13   }
14 }
```

図 18　提案手法を用いた CPU 時間を消費する処理を行うプログラムのチューニング
Fig. 18　Tuning a program which consumes cpu time using the proposed method.

4.2.2 項で述べた，アニメーションを行うアプリケーションにおいては，このオーバヘッドは問題となる場合とならない場合がある．たとえば，毎秒 30 回の頻度で画面を更新する場合，およそ 33 ミリ秒ごとに 1 枚の画像を作る必要がある．つまり，ActionScript の実行時間とランタイムが描画処理を行う時間の合計が 33 ミリ秒より短ければ，提案機構のオーバヘッドは問題とならない．逆に，変換処理によってこの合計が 33 ミリ秒よりも長くなってしまう場合は，以下に述べる CPU 時間を消費するアプリケーションと同じ問題が生じる．

4.2.3 項で述べた，CPU 時間を消費するアプリケーションにおいては，このオーバヘッドは問題となる．たとえば，図 12 に示すようにすべての処理を中断可能関数で行うプログラムを記述した場合，実行時間が著しく増大してしまうことがある．

提案機構を利用して CPU 時間を消費するアプリケーションを記述する開発者は，この問題を回避するために**図 18** に示すようなチューニングを行うことができる．図 18 のプログラムでは，図 12 のプログラムのうち 3 カ所を修正している．うち 1 カ所は，一部の関数の通常の関数へと変更することである（図 12 中の 7 行目）．これにより，中断可能関数を呼び出すオーバヘッドを抑えることができる．残りの 2 カ所は，ループ回数の変更である（図 18 中の 3 行目と 10 行目）．これにより，1 回の中断処理あたりに行う処理の粒度を変更し，高コストである中断処理が実行される回数を調整することができる．

6.3　コードサイズの増加

本手法は，ActionScript のバイトコード中にプログラムを挿入する．このため，本手法

© 2011 Information Processing Society of Japan

表 6　コード変換によるバイトコード命令数の増加
Table 6　Code size increase by bytecode translation.

プログラム	コード変換前	コード変換後	比率
サーバとの通信（図 10）	38	241	6.3
アニメーション（図 11）	32	169	5.3
CPU 時間消費（図 12）	45	269	6.0

を利用すると，プログラムのサイズが増加し，SWF ファイルをダウンロードするための通信時間が増大する．4.2 節で取り上げたプログラムを対象としてどのくらい増加するかを調べた．結果を**表 6** に示す．このように，通常の処理（計算や分岐）が少なく，中断可能に関する処理が多い関数では，コード変換により，コードサイズは 5.3 倍から 6.3 倍になった．

我々は，このようなコードサイズの増加は実際の局面では大きな問題にならないと考えている．第 1 の理由は，プログラム全体の中でコードサイズが増加するのは中断可能関数のみであり，通常の関数は変化しないからである．たとえば，中断可能関数の割合が全体の 10%であれば，それが 6 倍になったとしても，プログラム全体サイズは 1.5 倍にしかならない．さらに，挿入されるコードも類似のものが多く，データ圧縮の効果が大きいと期待される．第 2 の理由は，Flash のアプリケーションが多くの場合，プログラム以外に画像や音声などのデータを内部に含むか，後にサーバからダウンロードするからである．そのような場合，大きなデータにまぎれてプログラムの部分の増加を利用者が気がつくことは少ないと思われる．

7. おわりに

本研究では，ActionScript の処理系の上でコルーチンに基づくマルチスレッドを実現し，開発者が ActionScript を用いてマルチスレッドプログラミングを行うことを可能とした．マルチスレッドの実現のために，本研究では開発者が記述した関数に対してバイトコードレベルのコード変換を行って中断可能関数を生成するためのコード変換プログラムを実装した．中断可能関数は，継続オブジェクトに自身の実行状態を保存することによって，実行を中断した後に再開することができる関数である．また，本研究ではランタイムライブラリを実装し，中断可能関数をマルチスレッド的に利用することを可能とした．提案機構を用いることにより，開発者は複数のイベントハンドラを定義することなくサーバとの通信，アニメーション，CPU を消費する処理を記述することができる．また，本研究では提案機構の実装を行い，提案機構を利用することによって生じるオーバヘッドを測定した．実験環境では，ループ 1 回あたり 9.1 ナノ秒，通常の関数呼び出しで 18.1 ナノ秒のオーバヘッドがあった．中断可能関数の呼び出しでは，さらに 13.7 ナノ秒のオーバヘッドがあった．このようなオーバヘッドは，サーバと通信する処理を行うアプリケーションでは問題にならない．CPU 時間を消費するアプリケーションでは，問題になる．この問題は，中断可能関数において 1 回の処理の粒度を調整することで解決できることを述べた．

今後の課題としては，ランタイムライブラリの改良があげられる．特に，以下の 2 点について改良を行いたいと考えている．1 つは，中断可能関数がループ文の中で中断処理を行う頻度の最適化である．現在の実装では，中断処理を行う頻度は実行されるループ 100 回あたり 1 回と固定である．システム時刻からスレッドの実行時間を計測することにより，この頻度を適切に調整する機能を追加する．もう 1 つは，スレッド間の通信や同期などを行うための API の整備である．また，既存の大きなアプリケーションを再実装し，提案手法の有用性を示す．

謝辞　本論文をまとめるにあたり，筑波大学システム情報工学研究科コンピュータサイエンス専攻前田敦司准教授には，貴重な助言とコメントをいただいた．ここに，心から感謝の意を表す．

参　考　文　献

1) Adobe Systems Incorporated: *SWF File Format Specification Version 10* (2008).
2) 新藤愛大：ActionScript Thread Library 1.0（そうめん）ドキュメント，http://www.libspark.org/htdocs/as3/thread-files/document (2008).
3) Moock, C.: *Essential actionscript 3.0*, O'Reilly (2007).
4) Harui, A.: Threads in Actionscript 3 Alex's Flex Closet, http://blogs.adobe.com/aharui/2008/01/threads_in_actionscript_3.html (2008).
5) 原　悠，鵜川始陽，湯淺太一，八杉昌宏：タプル空間によるブラウザ間通信を備えた Scheme 処理系の開発，情報処理学会論文誌：プログラミング（PRO），Vol.1, No.2, pp.85–99 (2008).
6) 牧　大介，岩崎英哉：非同期処理のための JavaScript マルチスレッドフレームワーク，情報処理学会論文誌：プログラミング，Vol.48, No.SIG 12 (PRO34), pp.1–18 (2007).
7) Oni Labs: Oni Labs: StratifiedJS, http://onilabs.com/stratifiedjs (2010).
8) Hickson, I.: Web Workers, http://www.whatwg.org/specs/web-workers/current-work/ (2010).
9) 千葉　滋，立堀道昭：Java バイトコード変換による構造リフレクションの実現，情報処理学会論文誌，Vol.42, No.11, pp.2752–2760 (2001).
10) 新藤愛大：yossy/swfassist — Spark project, http://www.libspark.org/wiki/yossy/

© 2011 Information Processing Society of Japan

swfassist (2008).
11) Mozilla Foundation: Tamarin Project, http://www.mozilla-japan.org/projects/tamarin/ (2006).
12) Adobe Systems Incorporated: Adobe Labs — Alchemy, http://labs.adobe.com/technologies/alchemy/.
13) Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *IEEE/ACM International Symposium on Code Generation and Optimization*, Vol.0, p.75 (2004).
14) Adobe Systems Incorporated: Pixel Bender Technology Center — Adobe Developer Connection, http://www.adobe.com/devnet/pixelbender.html (2008).
15) Dean, R.W.: Using continuations to build a user-level threads library, *Proc. 3rd conference on USENIX MACH III Symposium*, pp.137–151 (1993).
16) Alpern, B., Attanasio, C., Cocchi, A., Lieber, D., Smith, S., Ngo, T., Barton, J., Hummel, S., Sheperd, J. and Mergen, M.: Implementing jalapeño in Java, *Proc. 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (*OOPSLA 99*), pp.314–324 (1999).
17) 八杉昌宏，馬谷誠二，鎌田十三郎，田畑悠介，伊藤智一，小宮常康，湯淺太一：オブジェクト指向並列言語 OPA のためのコード生成手法，情報処理学会論文誌：プログラミング（PRO），Vol.42, No.SIG11(PRO12), pp.1–13 (2001).
18) 五嶋宏通，笹田耕一，三好健文，稲葉真理，平木　敬：Ruby 用仮想マシンにおける AOT コンパイラ，情報処理学会論文誌：プログラミング（PRO），Vol.2, No.1, pp.21–21 (2009).
19) 芝　哲史，笹田耕一，卜部昌平，松本行弘，稲葉真理，平木　敬：実用的な Ruby 用 AOT コンパイラ，情報処理学 SWoPP 2010 (SIG-PRO) (2010).
20) Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp.212–223 (1998). Proceedings published ACM SIGPLAN Notices, Vol.33, No.5 (May 1998).
21) Gosling, J., Joy, B., Steele, G. and Bracha, G.: *The Java Language Specification*, Addison Wesley (2005).
22) Adobe Systems Incorporated: rich Internet applications — Adobe AIR, http://www.adobe.com/products/air/.
23) Adobe Systems Incorporated: mikechambers/as3corelib — GitHub, http://github.com/mikechambers/as3corelib (2008).

(平成 22 年 12 月 17 日受付)
(平成 23 年 3 月 29 日採録)

杉本　卓哉（学生会員）

1985 年生．2008 年筑波大学第三学群情報学類卒業．2011 年筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士前期課程修了．在学中は，プログラミング言語およびその処理系に関する研究に従事．修士（工学）．

新城　　靖（正会員）

1965 年生．1988 年筑波大学第三学群情報学類卒業．1993 年筑波大学大学院工学研究科電子・情報工学専攻博士課程修了．同年琉球大学工学部情報工学科助手．1995 年筑波大学電子・情報工学系講師，2003 年同助教授，2004 年同大学院システム情報工学研究科助教授．2007 年同准教授．オペレーティング・システム，分散システム，仮想システム，並行システム，情報セキュリティの研究に従事．博士（工学）．ACM，IEEE，日本ソフトウェア科学会各会員．

中井　　央（正会員）

1968 年生．筑波大学第三学群情報学類卒業，同大学大学院工学研究科修了．1997 年 10 月図書館情報大学助手，2001 年 8 月同総合情報処理センター講師，2002 年 8 月同助教授．2002 年 10 月の筑波大学との統合により，筑波大学図書館情報メディア研究科助教授（学術情報メディアセンター勤務）．2007 年同准教授．コンパイラの研究に従事．博士（工学）．日本ソフトウェア科学会，ACM，ACM-SIGMOD-JAPAN 各会員．

© 2011 Information Processing Society of Japan

板野　肯三（正会員）

1948 年生．1977 年東京大学大学院理学系研究科物理学専門課程単位取得後退学．1993 年筑波大学電子・情報工学系教授．2004 年同大学院システム情報工学研究科教授．計算機アーキテクチャ，分散システム，プログラミングシステム等に関する研究に従事．理学博士．日本ソフトウェア科学会，電子情報通信学会，ACM，IEEECS 各会員．

佐藤　　聡（正会員）

1967 年生．1991 年筑波大学第三学群情報学類卒業．1996 年筑波大学大学院工学研究科単位取得退学．同年広島市立大学情報科学部助手．2001 年筑波大学システム情報工学研究科講師．現在，同大学学術情報メディアセンター勤務．キャンパスネットワークの企画管理運用，ネットワークデータベース，言語処理等の研究に従事．博士（工学）．電子情報通信学会，ACM-SIGMOD-JAPN 各会員．

© 2011 Information Processing Society of Japan