リアルタイム Java、第 1 回: リアルタイム・システムに Java 言語を使用する

Mark Stoodley
Advisory Software Developer
IBM Toronto Lab

Mike Fulton (fultonm@ca.ibm.com)
Senior Technical Staff Member
IBM Toronto Lab

Michael Dawson
Advisory Software Developer
IBM Ottawa Lab

Ryan Sciampacone
Senior Software Developer
IBM Ottawa Lab

John Kacur
Software Developer
IBM Toronto Lab

2007年 4月 10日

リアルタイム Java™ に関する 5 回連載の第 1 回目となるこの記事で取り上げるのは、Java 言語を使用してリアルタイム・パフォーマンス要件を満たすシステムを開発する際にキーとなる課題です。まず全体像を取り上げ、リアルタイムアプリケーションを開発するという意味、そしてリアルタイム・アプリケーションの要件を満たすランタイム・システムの設計方法を説明します。5人の著者が紹介する実装は、標準ベースの技術を組み合わせてリアルタイム Java という難問に対処します。

リアルタイム・システムで Java 言語があまり使用されていないのには、大きな理由がいくつもあります。その 1 つとして、クラスの動的ロードといった Java 言語の設計に特有の処理や、ガーベッジ・コレクターやネイティブ・コード・コンパイルなどの Java ランタイム環境 (JRE) 自体に特有の処理のパフォーマンスの影響が確定しないということがあります。そんなJava 言語でリアルタイム・システムを構築する大きな可能性を開くオープン仕様が、RTSJ (Real-time Specificationfor Java) です。RTSJ を実装するには、オペレーティング・システム、JRE、そして JCL

(Java Class Library)でのサポートが必要となります。そこで、この記事では Java 言語を使用してリアルタイム・システムを実装する際の難題を取り上げ、それらの難題に立ち向かう開発キットとランタイム環境を紹介します。連載の今後の記事では、今回の記事で紹介するコンセプトと技術をさらに詳しく説明する予定です。

リアルタイム要件

リアルタイム (RT) とは、実世界のタイミング要件を持つアプリケーションを説明するために幅広く使用される用語です。一例として、反応が遅いユーザー・インターフェースは平均的なユーザーの汎用のRT 要件を満たしませんが、このようなアプリケーションはよく、ソフト RT アプリケーションとして説明されます。RT 要件をより明確に言えば、例えば「マウスのクリックに対して0.1 秒以内で応答するアプリケーション」と表現することもできます。この要件を満たさない場合は、ソフト障害ということになります。つまり、アプリケーションは続行し、ユーザーは満足できないながらも、引き続きそのアプリケーションを使用できるということです。それに対して、実世界のタイミング要件に完全に適合しなければならないアプリケーションは通常、ハード RT アプリケーションと呼ばれます。例えば、飛行機のラダーを制御するアプリケーションでは、いかなる理由であっても遅延は許されません。遅延が大惨事につながる恐れがあるからです。RT アプリケーションであるか否かは、主に、タイミング要件に適合しないという形の障害に対してアプリケーションがどれだけ耐えられるかによって決まります。

応答時間も、RT 要件の重要な側面です。ハードまたはソフト RT アプリケーションを作成するプログラマーにとって、応答時間の制約を理解することは必要不可欠となります。応答時間1 マイクロ秒というハード要件を満たすために必要な手法は、応答時間 100 ミリ秒のハード要件を満たす場合とは大幅に異なるからです。実際、数十マイクロ秒未満の応答時間を達成するには、おそらくオペレーティング・システムの層がない(あるいは極めて薄い) カスタム・ハードウェアとソフトウェアの組み合わせが必要になります。

最後に、設計者が応答時間の要件を満たす確実な RT アプリケーションを設計するには、通常は、定量化可能なレベルにパフォーマンスが確定するような特性が必要です。予測不可能なパフォーマンスの影響が、アプリケーションの応答時間要件を満たすためのシステム性能を左右するほど大きい場合、アプリケーションを適切に設計することが困難になったり、あるいは不可能になることさえあります。そのような理由から、たいていのRT 実行環境の設計者は、RT アプリケーションの考えられる限り広い範囲で応答時間のニーズを満たすため、パフォーマンスの影響が不確定な処理を減らすことにかなりの努力を払っています。

RT Java アプリケーションの場合の難題

汎用オペレーティングシステム上の汎用 JVM で動作する標準 Java アプリケーションで満たすことを望めるソフト RT 要件は、せいぜい数百ミリ秒といったところです。Java言語によって行われる基本的な側面には、スレッド管理、クラス・ロード、JIT (Just-In-Time) コンパイラーのアクティビティー、そしてガーベッジ・コレクション(GC) があります。これらの問題のいくつかはアプリケーション設計者が軽減できますが、それには多大な作業が必要にならせざるを得ません。

スレッド管理

標準 Java では、スレッド・スケジューリングやスレッドの優先順位が保証されていません。つまり、アプリケーションが一定の時間内にイベントに応答しなければならない場合でも、優先度の

低い別のスレッドがそれよりも優先度の高いスレッドの前にスケジュールされないようにする手立てはないということです。これを埋め合わせるため、プログラマーはアプリケーションをいくつかに区分し、オペレーティング・システムが異なる優先順位で実行できるようにする場合がありますが、このような区分化は、イベントのオーバーヘッドを増やし、イベント間の通信を一層困難にすることになります。

クラス・ロード

Java 準拠の JVM は、クラスがプログラムによって初めて参照されるまで、クラスのロードを遅延させなければなりません。クラスをロードする時間は、そのクラスをロードしてくる媒体(ディスクなど) の速度、クラスのサイズ、そしてクラス・ローダー自体によってもたらされるオーバーヘッドによって異なります。クラス・ロードの遅延は、通例10 ミリ秒に及びます。そのため、数十、数百のクラスをロードしなければならない場合、ロード時間自体が大幅な、そしておそらく予期しない遅延を引き起こす可能性があります。アプリケーションの開始時にすべてのクラスをロードするように慎重にアプリケーションを設計することもできますが、この設計は手作業で行わなければなりません。Java言語仕様では、JVM がこのステップを早い段階で実行できないためです。

Stop The World

昔から、ガーベッジ・コレクションはアプリケーション・プログラムの停止中に実行されます。これは、STW (Stop-The-World) という造語で呼ばれるプロセスです。コレクション中は、一連の「ルート」オブジェクト(静的フィールドによって指定されているオブジェクトや、スレッドのスタックに現在存続しているオブジェクトなど) から始めて、ライブ・オブジェクトを追跡し、未使用のメモリーを空きリストに戻して、以降の割り当て要求で使用できるようにします。

STW ガーベッジ・コレクターを使用する場合、アプリケーション・プログラムは GC をプログラム動作の一時停止として経験します。このような STW一時停止は期間に制限がなく、数百ミリ秒から数秒にまで至る非常に煩わしい時間となるのが通常です。一時停止の長さは、ヒープ・サイズ、ヒープに含まれるライブ・データの量、そしてコレクターがどれほど積極的に空きメモリーを再利用するかによります。

最近の多くのコレクターは、並行アルゴリズムやインクリメンタル・アルゴリズムなどの手法を使用して、この一時停止時間を短縮するようにしていますが、それでもGC による一時停止が予測できない時点で無期限に発生することには変りありません。

ガーベッジ・コレクション

GC がアプリケーション開発にとって有益であること (ポインターの安全、リークの回避、そして開発者がカスタム・メモリー管理ツールを作成しなくても済むことを含め)は十分に裏付けされていますが、Java 言語を使用するハード RT プログラマーにとってはフラストレーションの種です。ガーベッジ・コレクションは、Javaヒープが消耗されて割り当て要求を満たせなくなると自動的に発生するからです。また、アプリケーション自体がガーベッジ・コレクションをトリガーすることもあります。

GC は Java プログラマーにとっては素晴しいものです。メモリーを C や C++ などの言語で明示的に管理しなければならない場合に引き起こされるエラーのなかには、診断が極めて難しい問題もあります。アプリケーションをデプロイする時点でそのようなエラーがないことを証明するのも、根本的な難題です。Javaプログラミング・モデルの主要な長所の 1 つは、アプリケーションではなく JVM がメモリー管理を行うという点です。そのため、アプリケーション・プログラマーはこのようなメモリー管理の重荷から解放されます。

その一方で、従来のガーベッジ・コレクターは時折、長い遅延を引き起こします。これがどの時点で発生するかをアプリケーション・プログラマーが予測するのは事実上不可能です。また、このような遅延が数百ミリ秒になることも珍しくありません。この問題をアプリケーション・レベルで解決する唯一の方法は、再利用される一連のオブジェクトを作成し、これらのオブジェクトによってJava ヒープ・メモリーが決して消耗されないようにして GC の発生を防ぐことです。つまり、プログラマーはこの問題を解決するため、JVM によるメモリー管理というメリットを投げ捨てて自らメモリーを管理するということです。ですが実際には、この方法がうまくいくことはほとんどありません。その理由は、プログラマーがJDK や他のクラス・ベンダーによって用意されたクラス・ライブラリーの多くを使用できなくなることから、おそらく一時オブジェクトが作成され、結局はこれらの一時オブジェクトによってヒープがいっぱいになってしまうからです。

コンパイル

Java コードをネイティブ・コードにコンパイルすることによっても、クラス・ロードと同じような問題が引き起こされます。最近のほとんどの JVMでは、まず Java メソッドを解釈してから、頻繁に実行されるメソッドのみをネイティブ・コードにコンパイルします。コンパイルを遅延させることによって起動は迅速になり、アプリケーションの実行中に行われるコンパイルの量も減りますが、解釈されたコードでタスクを実行するのと、コンパイルされたコードでタスクを実行するのとでは、実行時間が大幅に変わってきます。ハードRT アプリケーションの場合、コンパイルがいつ発生するかを予測できないため、不確定な部分が大きすぎてアプリケーションのアクティビティーを効果的に計画できなくなります。この問題を緩和するには、クラス・ロードの場合と同じく、Compilerクラスを使用してアプリケーションの開始時にメソッドをプログラマムによってコンパイルするという方法があります。しかし、このようなメソッドのリストを維持するのは厄介な作業なので、結局はエラーの原因になるだけです。

Java リアルタイム仕様

RTSJ は、RT 実行環境で Java 言語が広く使われることを妨げる原因である Java 言語の制約の一部に対処するために作成されました。RTSJが対処する問題領域には、スケジューリング、メモリー管理、スレッド化、同期、時刻、クロック、そして非同期イベントの処理があります。

スケジューリング

RT システムでは、スレッドのスケジューリングを厳密に制御し、スレッドが確定的にスケジューリングされることを保証しなければなりません。つまり、スレッドは同じ一連の条件下では同じようにスケジューリングされるということです。JCLはスレッドの優先順位の概念を定義していますが、従来の JVM には優先順位付けを実施するという要件はありません。さらに、RT 以外の Java 実装でも、一般的にはスケジューリングの順序が予測不可能なラウンドロビンによるプリエンプティブ・スケジューリングの手法を使用しています。RTSJでは、真の優先度、そして優先度の継承をサポートする固定優先度のプリエンプティブ・スケジューラーを RT スレッドに対する要件としています。このスケジューリング手法は、最高の優先度を持つアクティブ・スレッドが常に実行中になり、そのスレッドが自発的にCPU を解放するか、あるいはそれよりも高い優先度を持つ別のスレッドによってプリエンプトされるまでアクティブ・スレッドが実行されることを確実にします。高い優先度を持つスレッドがそれよりも低い優先度のスレッドが保有するリソースを必要とする場合でも、優先度の継承により優先順位の逆転が確実に回避されます。優先順位の逆転は、RTシステムにとって重大な問題です。その詳細は、RT Linux® に記載されています。

メモリー管理

一部の RT システムはガーベッジ・コレクターによって発生する遅延を許容できるものの、たいていの RT システムでは、このような遅延は許されません。GC割り込みを許容できないタスクをサポートするため、RTSJ では標準 Java ヒープを補完する永続メモリー領域とスコープ・メモリー領域を定義しています。これらの領域では、ガーベッジ・コレクターがヒープ内でメモリーを解放しなければならない場合でも、タスクはブロックを要求されることなく、メモリーを使用できます。永続メモリー領域に割り当てられたオブジェクトは、すべてのスレッドからアクセスできます。これらのオブジェクトに対しては、ガーベッジ・コレクションが行われることも決してないため、永続メモリーは限定されたリソースとして慎重に使用しなければなりません。一方、スコープ・メモリー領域の作成および破棄は、プログラマーが制御できます。最大サイズが割り当てられたそれぞれのスコープ・メモリー領域は、オブジェクトの割り当てに使用することができます。オブジェクト間の参照の完全性が確実になるように、RTSJではメモリー領域 (ヒープ、永続、またはスコープ) 間でのオブジェクト参照を管理するルールを定義しています。また、スコープ・メモリー内のオブジェクトをファイナライズするタイミングと、メモリー領域を再利用可能にするタイミングについて定義するルールも追加されています。このような複雑さにより、永続メモリーとスコープ・メモリーに推奨される用途は、GCによる一時停止を許容できないコンポーネントに限られます。

スレッド

RTSJ では、RT 動作をするタスクを実行するための基礎となる 2 つの新しいスレッド・クラス、`RealtimeThread` と `NoHeapRealtimeThread` (NHRT) のサポートを追加しています。この 2 つのクラスは、優先度、定期的動作、ハンドラーによる期限 (ハンドラーは、期限を過ぎた時点でトリガーできます)、そしてヒープ以外のメモリー領域の使用をサポートします。NHRTはヒープにアクセスできないため、他のタイプのスレッドとは異なり、大抵は GC によって割り込まれたり、プリエンプトされることはありません。RTシステムは通常、最も厳しい遅延要件を持つタスクに対しては優先度の高い NHRT を、ガーベッジ・コレクターで対応可能な遅延要件を持つタスクに対しては`RealtimeThread` を、これ以外のタスクに対しては通常の Java スレッドが使用されます。NHRT はヒープにアクセスできないことから、このスレッドの使用には十分な慎重さが求められます。例えば、標準JCL のコンテナー・クラスを使用する場合でさえも、コンテナー・クラスがヒープ上に不用意に一時オブジェクトまたは内部オブジェクトを作成しないよう慎重に管理しなくてはなりません。

同期

同期は、RT システム内で慎重に管理し、高い優先度のスレッドがそれよりも優先度が低いスレッドを待機しないようにする必要があります。RTSJ には同期が発生したときに同期を管理するための優先度の継承サポートが含まれているため、スレッドは同期を行わずに、待機なしの読み取り/書き込みキューを介して通信できるようになっています。

時刻とクロック

RT システムには、標準 Java コードが提供するクロックよりも分解能の高いクロックが必要です。新規 `HighResolutionTime` クラスと `Clock` クラスは、このようなタイム・サービスをカプセル化します。

非同期イベントの処理

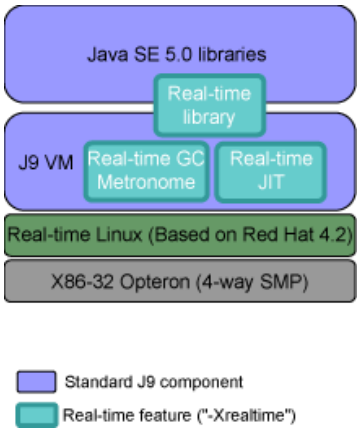
RT システムは多くの場合、非同期イベントの管理と非同期イベントへの応答を行います。RTSJ には、タイマー、オペレーティング・システムの信号、期限の超過やその他のアプリケーションが定義したイベントをはじめとする多数の要因によってトリガーされる非同期イベントの処理に対するサポートが含まれています。

IBM WebSphere Real Time

WebSphere Application Server とは異なり、WebSphere Real Time には Java EnterpriseEdition アプリケーション・サーバーは組み込まれていません。

RTSJ を実装するには、基礎となるオペレーティング・システム、そして JRE のコンポーネントによる広範なサポートが必要となります。2006年 8 月にリリースされた IBM® WebSphere® Real Time (「参考文献」を参照) は RTSJ に完全準拠します。さらに、RT システムのランタイム動作を改善し、アプリケーション設計者が RT システムを作成する際に必要となる作業を容易にすることを目的とした新規技術も組み込まれています。図1 に、WebSphere Real Time のコンポーネントの簡略表現を示します。

図 1. WebSphere Real Time の概要



RTSJ 実装の (小さな) 世界

Linux 上で実行する RTSJ 準拠の実装としては、他に TimeSys 社の RTSJ 参照実装と Apogee 社の Aphelionの 2 つがあります。Sun の Java SE Real-time (Java RTS) は、Sparc/Solaris 上で動作します。詳細は、「参考文献」を参照してください。

WebSphere Real Time は、IBM のクロスプラットフォーム J9 技術をベースとしています。Linux オペレーティング・システムに適用されたオープン・ソースのRT パッチは、RTSJ が規定する必須動作をはじめ、RT 動作のサポートに必要な基本的な RT サービスを提供します。大幅に拡張されたGC技術は、1 ミリ秒の一時停止時間をサポートします。完了させる必要のある優先度の高い作業がなくなるとコンパイルが発生するような比較的ソフトな RTシナリオには、JIT コンパイルを使用できます。一方、JIT コンパイルが適切でないシステムで比較的ハードな RT パフォーマンスを提供するために、新しいAOT (Ahead-Of-Time) コンパイル技術 (図 1 には記載されていません) も導

入されています。以降のセクションでは、これらの技術を紹介します。それぞれの技術がどのように機能するかについては、この連載の今後の記事で詳しく説明する予定です。

RT Linux

WebSphere Real Time は、カスタマイズされた完全なオープン・ソース・バージョンの Linux 上で動作します。RT Java対応の環境にするため、いくつかの変更が適用されています。これらの変更により、完全にプリエンプト可能なカーネル、スレッド化された割り込みハンドラー、高分解能タイマー、優先度の継承、そして堅牢なmutex が実現されています。

完全にプリエンプト可能なカーネル

RT Java スレッドは、ファースト・イン・ファースト・アウトのスケジューリング・ポリシーを持つ固定優先度スケジューリング (静的優先度スケジューリングとしても知られています)が実装されています。標準 Linux カーネルが提供するのはソフト RT 動作です。優先度の高いスレッドが、優先度の低いスレッドがプリエンプトされるまで待機する時間の上限は保証されていませんが、この時間は10 マイクロ秒単位で概算できます。RT Linux では、ほとんどすべてのカーネル・アクティビティーがプリエンプト可能になるため、優先度の低いスレッドがプリエンプトされ、優先度が高いスレッドを実行するまでの所要時間が短縮されます。プリエンプトできない残りのクリティカル・セクションは、短時間で確定的に実行されます。RTスケジューリングのレイテンシーは 3 桁分改善され、現在は約 10 マイクロ秒単位で測定できるようになっています。

スレッド化された割り込みハンドラーによるレイテンシーの短縮

割り込みハンドラーはほとんどすべて、プロセスのコンテキストで実行されるカーネル・スレッドに変換されます。ハンドラーが、ユーザーによる構成とスケジュールが可能なエンティティーになり、他のプロセスとまったく同じようにプリエンプトされ優先順位がつけられるようになったことにより、レイテンシーの短縮と処理の確定性の向上が実現されています。

高分解能タイマー

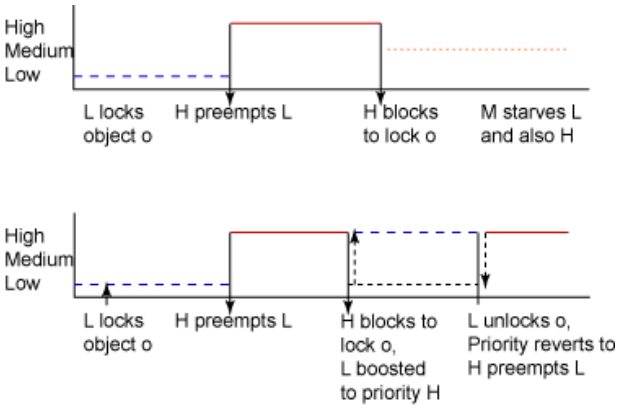
高分解能の時刻とタイマーにより、分解能と精度が向上しています。RT Java はこれらのフィーチャーを高分解能スリープと時間計測によるウェイトに使用しています。Linuxの高分解能タイマーは、高精度の 64 ビット・データ型で実装されています。従来の Linux では時刻とタイマーが低分解能のシステム・ティックに依存するため、タイマー・イベントの粒度が制限されてしまいますが、RTLinux では個別にプログラミングできる高分解能のタイマーイベントを使用しており、これらのタイマー・イベントは、数マイクロ秒以内で満了させることも可能です。

優先度の継承

優先度の継承は、優先順位の逆転という昔ながらの問題を回避するための手法です。図 2 の上の図に示しているのは優先順位の逆転の最も単純な例で、ここで用いられている3 つのスレッドの優先度はそれぞれ高 (H)、中 (M)、低 (L) となっています。最初、H と M は休止状態でイベントがトリガーされるのを待っている一方、Lはアクティブでロックを保持しているとします。その後、H がイベントを処理するためにウェイクアップすると、L をプリエンプトして実行を開始することになります。ここで、Lが保持するロックで H がブロックされるとどうなるかを考えてみてく

ださい。H は L がロックを解除するまで処理を進められないため、H はブロックされ、Lが再び実行を開始します。さらに、この時点で M がイベントによりトリガーされると、M は L をプリエンプトして必要なだけ実行を続けます。Hのほうが M より優先度が高いにも関わらず、M が H の実行を阻止できるというわけです。このような状況を、優先順位の逆転と呼んでいます。

図 2. 優先順位の逆転と優先度の継承の例



RT Linux は、図 2 の下の図に示した優先度の継承として知られるポリシー (優先度の貸与としても知られています) を使用して、優先順位の逆転を防ぎます。Hが L によって保持されるロックでブロックされると H はその優先度を L に渡します。これによって保証されるのは、H が必要とするロックをL が解除するまで、H より優先度の低いタスクが L をプリエンプトできないということです。ロックが解除されると同時に、L の優先度は元の値に戻るため、Hは L のことをそれ以上待機することなく処理を行えます。優先度の高いスレッドが優先度の低いスレッドが保持するリソースを必要とするような状況にならないよう、アプリケーション設計者が努力しなければならないことに変りはありませんが、この優先度の継承メカニズムは堅牢性を向上し、優先順位の逆転を防止します。

堅牢な mutex および rt-mutex

Linux の pthread mutex は、futex として知られる高速なユーザー空間 mutex によってサポートされます。futexはカーネルに依存せずに、競合しないロックを取得する時間を最適化します。つまり、カーネルの介入は競合するロックにのみ必要だということです。堅牢なmutex は、ロックを保持するアプリケーションがクラッシュした後に適切にロックをクリーンアップするという問題を解決します。さらに、rt-mutexは優先度の継承プロトコルを堅牢な mutex に拡張するため、RT JVM は pthread ライブラリーによる優先度の継承動作に依存できるようになっています。

確定的ガーベッジ・コレクション

RT Linux などの RT オペレーティング・システムが RT 動作の基本を提供することを考えれば、JVM の他の主要な部分も RT 動作を示すようにビルドできるはずです。JVMでは、GC が不確定動作の大きな原因の 1 つとなっていますが、この不確定性は、慎重な設計、そして RT Linux のフィーチャーに依存することによって軽減できます。

GC 一時停止による影響の不確定性は、RT アプリケーションが特定の期限でタスクを完了できるかどうかという能力を大きく左右します (「ガーベッジ・コレクション」を参照)。ほとんどの GC

実装は RT アプリケーションのレイテンシーの目標を妨げ、スケールが大きくタイミング要件が厳しくないタスク以外はGC 技術に依存できないというレベルにまで目標を引き下げてしまいます。この問題に対する RTSJ のソリューションは、永続メモリー領域とスコープ・メモリー領域、そしてNHRT を介してプログラマーが管理するメモリー割り当てを導入することです。ただし、このソリューションは Java アプリケーション設計者にとっては大きな頭痛の種になる可能性があります。

WebSphere Real Time では、プログラマーが望めば RTSJ メモリー領域に依存することもできますが、この手法が推奨されるのは極めて厳しいレイテンシーの要件を持つタスクだけです。約1 ミリ秒の GC 一時停止時間を許容できるタスクについては、IBM が確定性のある GC 技術を作成しています。この技術により、プログラマーは自動メモリー管理によるプログラミングの容易さを利用して、パフォーマンスを予測しながらタスクを管理できます。

IBM のが確定性のある GC 技術は、以下の 2 つの単純な前提に基づいています。

- 特定の上限時間を超える GC 一時停止が 1 つもないこと。
- 指定された時間枠内での GC 一時停止回数を制御し、その時間枠の一定のパーセンテージ以上が GC によって消費されないこと。

この 2 つの前提に留意して GC アクティビティーを管理すると、アプリケーションがその RT 目標を達成できる可能性が飛躍的に高くなります。

Metronome GC

WebSphere Real Time は Metronome GC を使用して、JVM で一時停止時間の短い GC 動作を実現しています (「参考文献」を参照)。Metronome GC が使用する時間ベースのスケジューリング・メソッドは、固定スケジュールでコレクターとアプリケーションをインターリーブします(GC 専門用語では、ミューテーターと呼ばれます。その理由は、ガーベッジ・コレクターから見ると、アプリケーションの動作によってライブ・オブジェクトのグラフが時間とともに変化するからです)。

割り当て率ではなく、時間を対象にスケジューリングを行う理由は、アプリケーションの実行中に割り当てが不均等になることが多いからです。割り当てに対する義務としてGC 作業を行うようにすると、GC 一時停止の分布が不均等になり、GC 動作における確定性のレベルが下がる可能性があります。MetronomeGC は時間ベースのスケジュールを使用して、一時停止の一貫性、確定性、そして時間制限を実現しています。さらに、既存のコードに対する言語拡張や修正は一切必要ないため、通常のJava アプリケーションが Metronome を透過的に利用して、その確定性という特性がもたらす恩恵を受けることができます。

Metronome は時間を一連の時間単位 (約 500 マイクロ秒。1 ミリ秒を超えることはありません) に分割し、それぞれの時間単位をGC作業専用またはアプリケーション作業専用にします。時間単位自体は非常に短いものですが、複数の時間単位が GC 作業専用になったとしたら、アプリケーションが経験する一時停止時間はもっと長くなり、それによってRT 期限が侵される可能性があります。Metronome は RT 期限を確実にサポートするため、アプリケーションに必要最低限の割合の時間が割り当てられるように、GC作業専用の時間単位を分配します。この割合は使用率と呼ばれるもので、ユーザーが指定するパラメーターです。いずれの時間間隔でも、アプリケーション

専用の時間単位の数が指定された使用率を下回ることはありません。デフォルトの使用率は70%で、つまり 10 ミリ秒の時間枠では、少なくとも 7 ミリ秒がアプリケーション専用となります。

使用率は、ユーザーがプログラムの起動時に設定できます。図 3 は、長期間でのアプリケーション使用率の例です。時々使用率が低下しているのは、ガーベッジ・コレクターがアクティブになっている時間単位に対応することに注目してください。図3 に示す全体的な時間枠では、アプリケーション使用率は指定された 70% (0.7) 以上に保たれています。

図 3. 使用率グラフの例

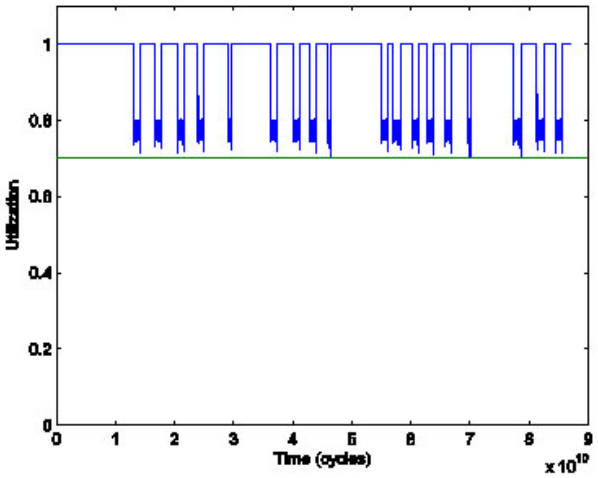
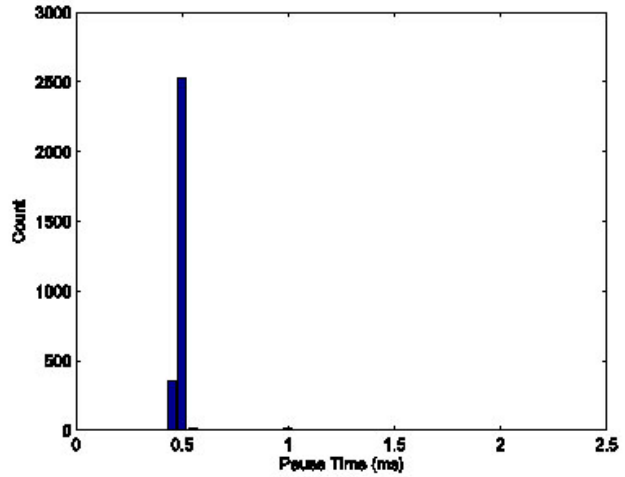


図 4 に、Metronome 技術を使用した場合の確定性のある GC 一時停止時間を示します。一時停止が 500 マイクロ秒を超えているのはほんの一部です。

図 4. GC 一時停止時間のヒストグラム



個別の GC 一時停止を短時間にするため、Metronome はヒープ内の書き込みバリアと関連メタストラクチャーを使用して、ライブ・オブジェクトおよび不要になった可能性があるオブジェクトを追跡します。ライブ・オブジェクトを追跡するには、存続させるオブジェクトと回収するオブジェクトを判断するための一連のGC 時間単位が必要です。この追跡作業はプログラムの実行とイ

ンターリーブされるため、アプリケーションがロードと保存を実行することによって「隠蔽」できる特定のオブジェクトについては、GCが追跡しきれなくなる可能性があります。

このようなライブ・オブジェクト隠蔽の原因が、悪意のあるアプリケーション・コードであるとは限りません。アプリケーションはガーベッジ・コレクターのアクティビティーを認識しないため、これはよくあることです。コレクターに見落とされるオブジェクトが1 つもないようにするため、GC と VM は協力して、アプリケーションが実行する保存操作でオブジェクトが作成、破棄される時点でオブジェクト間のリンクを追跡します。この追跡を行うのが書き込みバリアで、これはアプリケーションが保存操作を行う前に実行されます。書き込みバリアの目的は単に、アプリケーションの保存操作によってライブ・オブジェクトが隠しオブジェクトになった可能性がある場合、オブジェクトのリンク方法に対する変更を記録することです。これらの書き込みバリアによって表されるパフォーマンスおよびメモリー・フットプリント両方のオーバーヘッドが、確定性のある動作に対するニーズのバランスを取ります。

大きなオブジェクトの割り当ては、多くの GC ストラテジーにとって厄介な問題です。配列などの単一の大きなオブジェクトに対応するため、ヒープが過剰にフラグメント化されていることは珍しくありません。大規模な割り当て要求に対応するには、ヒープをデフラグ(または圧縮) し、多数の小さな空きメモリー領域を大きなメモリー領域に合体しなければならないため、長い一時停止が発生します。そのため、Metronomeでは、配列を対象とした新しい 2 レベル・オブジェクトモデル、arraylet を使用しています。arraylet は大きな配列を小さな配列に分割し、大きな配列の割り当てでもヒープをデフラグすることなく簡単に対応できるようにします。arrayletオブジェクトの最初のレベルはスパインと呼ばれ、このレベルには、配列の分割された各部分 (リーフ) へのポインターのリストが含まれます。それぞれのリーフは同じサイズなので、配列に含まれる特定の要素を見つけるための計算は単純化されます。また、コレクターが適切な空きスペースを見つけて各リーフを割り当てるにも容易です。このように、配列を小さな連続していない部分に分割することによって、通常ヒープで発生する多数の小さな空き領域内に、配列を圧縮せずに割り当てることを可能にしています。

従来の STW ガーベッジ・コレクター実装では、GC サイクルのコンセプトでガーベッジ・コレクションの開始と終了を表現しますが、Metronomeはアプリケーションの存続期間全体で GC を連続プロセスとして実行します。このため、アプリケーションの存続期間中、アプリケーション使用率が保証されるというわけです。アプリケーション使用率は、大量のGC 作業が必要ない場合には最小値より高くなる可能性があります。また、空きメモリーの量はコレクターが空きメモリーを検出してアプリケーションに戻す過程で上昇、下降します。

RT の場合のネイティブ・コードのコンパイル

最新の JVM は大抵、解釈とコンパイル済みコードの実行を組み合わせて使用します。解釈のパフォーマンス・コストはかなりのものになりますが、JITコンパイラーは多くの場合、このコストをなくすために実行済みのコードを CPU のネイティブ命令に直接変換するという方法を採っています。Java言語の動的特性により、このようなコンパイラーは (C++ や Fortran などの言語での場合のように) プログラムの実行前に発生するステップとしてではなく、プログラムの実行と並行して動作するのが一般的です。JITコンパイラーは、コンパイルするコードについては選択するので、コンパイルの所要時間はコードのパフォーマンスがどれだけ改善されているかによって決まる可能性が高いです。この動的コンパイル動作に加え、従来のJIT コンパイラーは各種の推測による最適化を採用しています。これは、実行中プログラムの動的特性を利用する最適化で、特定の

プログラムが実行している間のある時点では推測が当てはまるとしても、実行期間全体を通して当てはまるとは限りません。このような最適化は、この特性についての前提が後になって当てはまらなくなった場合には「取り消し」可能です。

RT 以外の従来の環境では、プログラムの実行中でもコードのコンパイルは上手くいきます。コンパイラーのアクションはほとんどアプリケーションのパフォーマンスに透過的だからです。一方、RT環境では JIT コンパイラーが予測不可能なランタイム動作を引き起こすため、最悪の場合の実行時間についての分析を台無しにしてしまいます。それでも、一層複雑なタスクを短時間で完了できることから、RT環境ではコンパイル済みコードがもたらすパフォーマンス上のメリットは重要です。

WebSphere Real Time は、異なるトレード・オフ・ポイントで上記の 2 つの要件のバランスを取る、2 つのソリューションを導入しています。一方のソリューションは、RT以外の低い優先度で動作する JIT コンパイラーを採用することです。このコンパイラーは推測による最適化を積極的に実行する回数が少なくなるように変更されています。RT以外の優先度で動作することにより、オペレーティング・システムはコンパイラーが RT タスクの実行を妨げないことを保証できます。それでもなお、コード・パフォーマンスは時間とともに変化するという事実は影響は不確定的であるため、このソリューションはハードRT 環境よりもソフト RT 環境に適しています。

ハードよりの RT 環境の場合、WebSphere Real Time はアプリケーション・プログラム用に AOT コンパイルを導入しています。このソリューションでは、JARファイルに保存された Java クラス・ファイルを、単純なコマンドラインによって JXE (Java eXEcutable) に事前コンパイルできます。アプリケーションのクラス・パスで元のJARファイルの代わりに JXE ファイルを指定することで、JIT コンパイラーによって解釈されたバイト・コードでもコンパイルされたネイティブ・コードでもなく、AOTコンパイルで処理されたコードが実行されるようにアプリケーションを起動することが可能です。最初の WebSphere Real Time リリースでのAOT コードの使用は、すなわち JIT コンパイラーが存在しないということを意味しますが、これには 2 つの主要な利点があります。まず、メモリー消費量が少なくなること、そしてJIT コンパイル・スレッドや、頻繁に実行するコードを識別するサンプリング・スレッドによる動的パフォーマンスへの影響がないということです。

図 5 に、AOT コードを使用した場合に、Java コードが WebSphere Real Time でどのように実行されるかを示します。

図 5. AOT コードの使用方法

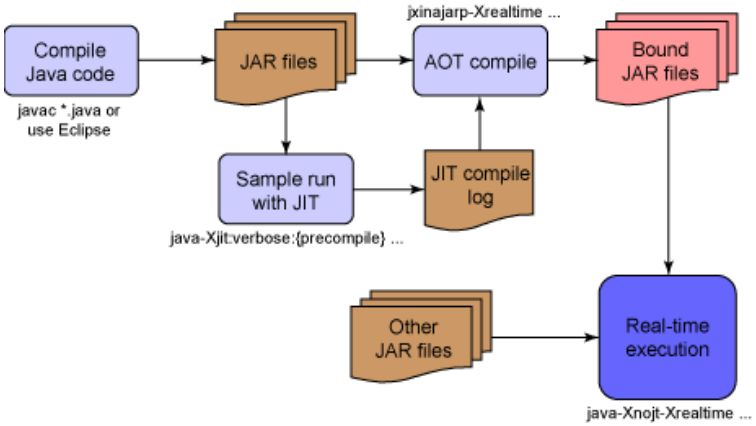


図 5 の左上から説明すると、すべての Java 開発プロジェクトの場合と同じように、開発者が Java ソースコードをクラス・ファイルにコンパイルします。クラス・ファイルはJAR ファイルにバンドルされ、次にこれらの JAR ファイルが `jxeinajar` ツールによって AOT でコンパイルされます。このツールは、JAR ファイル内のすべてのクラスに含まれるすべてのメソッドをコンパイルすることも、コンパイル対象となる最も重要なメソッドを識別するサンプルJIT ベースのプログラムを実行することによって生成された出力に基づいて、一部のメソッドを選択してコンパイルすることもできます。`jxeinajar` ツールは、JAR ファイルのメソッドをコンパイルして JXE ファイルを作成します。JXE ファイルに含まれるのは、元の JAR ファイルの内容とAOT コンパイラーによって生成されたネイティブ・コードの両方です。プログラムが実行されると、JXE ファイルはそのまま JAR ファイルの代わりとなります。`-Xnojit` オプションを使用して JMV を呼び出すと、クラス・パス上の JXE ファイルに含まれる AOT コンパイル済みコードが (Java 言語のルールに従って)ロードされます。プログラムの実行中は、JAR ファイルからロードされたメソッド、あるいは JXE ファイルからロードされたコンパイルされていないメソッドが解釈されます。JXEからロードされたコンパイル済みメソッドは、ネイティブ・コードとして実行されます。図 5 の `-Xrealtime` コマンド・ライン・オプションは RT VM の呼び出しを指定するためにも必要です。このコマンド・ライン・オプションは、WebSphere RealTime でのみ使用できます。

AOT コードの欠点

AOT コードによって一層確定的なパフォーマンスが可能になりますが、これには欠点もいくつかあります。まず、AOT コードの保存に使用される JXEは通常、クラス・ファイルを保持する JAR ファイルより遥かに大きくなるという点です。ネイティブ・コードは、クラス・ファイルに保存されるバイト・コードに比べ、密度が少ないのが一般的だからです。また、ネイティブ・コードを実行するには、例えばコードをJVM にバインドする方法や例外をキャッチする方法などを記述する各種の補足データも必要になります。2 つ目の欠点は、AOT でコンパイルされたコードは、解釈されたコードよりも高速に処理されますが、JITでコンパイルされたコードと比べると速度が大幅に落ちるという点です。最後に、解釈されたメソッドからコンパイルされたメソッドへの遷移、あるいはその逆の遷移は、解釈済みのメソッドを別の解釈済みメソッドから呼び出したり、コンパイル済みメソッドを別のコンパイル済みメソッドから呼び出すよりも時間がかかるという欠点もあります。JITコンパイラーがアクティブな JVM であれば、このような代償は、遷移の回数がパフォーマンスに影響を与えないほど少なくなるまでコンパイル対象のコードを「粗削りに」コンパイルすることで解消されます。AOTでコンパイルされたコードがある一方、JIT コンパイラーがない JVM では、JXE にコンパイルされた一連のメソッドによって遷移の回数が決まります。このような理由から、私たちが通常推奨しているのは、アプリケーションが依存するJava ライブラリー・クラスだけをコンパイルするのではなく、アプリケーション全体を AOT でコンパイルすることです。上記で述べたように、コンパイルされたメソッドの数を増やすとメモリーのフットプリントに影響が出ますが、一般的にはパフォーマンスに対するメリットの方が、フットプリントの増加より重要です。

AOT コードが JIT コードより遅いのは、Java 言語自体の性質が原因です。Java 言語では、プログラムが初めてクラスを参照する時点でクラスを解決することを要件としています。一方AOT コンパイラーは、プログラムを実行する前にコンパイルすることによって、コンパイル対象のコードが参照するクラス、フィールド、メソッドに関しては処理がされません。AOTでコンパイルされたコードが大抵の場合、JIT でコンパイルされたコードより遅くなるのは、JIT では実行中のプログラムによってこれらの参照の多くを解決した後でコンパイルを行うという利点があるためです。ただしコンパイル時間はプログラムの実行時間に加算されるため、JIT コンパイラーも同じく慎重

に、プログラムのコンパイルにかかる時間のバランスを取らなければなりません。このため、JIT コンパイラーは同じ最適化レベルですべてのコードをコンパイルすることはしません。AOT コンパイラーにはこのような制約がないため、より積極的なコンパイル手法を適用することが可能で、場合によってはJIT でコンパイルされたコードより優れたパフォーマンスをもたらすこともあります。さらに、AOT でコンパイルするメソッドの数は、JIT コンパイラーがコンパイル対象として決定するメソッド数より多くなるので、JITコンパイルではなく AOT コンパイルを使用した方がパフォーマンスの向上につながる場合もあります。それでもなお、AOT でコンパイルしたコードの方がJIT でコンパイルしたコードよりも遅いというのが通例です。

不確定性のあるパフォーマンスの影響を避けるため、JIT コンパイラーにしても、WebSphere Real Time 付属の AOT コンパイラーにしても、最近のJIT コンパイラーが通常適用している積極的な推測による最適化は適用していません。このような最適化を行う一般的な理由は、大幅なパフォーマンスの改善を実現できるためですが、これはRT システムには当てはまりません。さらに、RTSJ のさまざまな特徴と Metronome ガーベッジ・コレクターをサポートすることによって、従来のコンパイラーでは実行する必要のないコードのコンパイルにオーバーヘッドが生じます。このような理由から、RT環境を対象にコンパイルされたコードは、RT 以外の環境を対象にコンパイルされたコードよりも遅くなるのが一般的です。

今後の展開

RT Java 環境を予測可能なパフォーマンス、そしてそのままのスループットという点で高速化するためにできることはまだあるはずです。Java言語が RT アプリケーションの分野で成功するためには、以下の 2 つのキーとなる領域での前進が必要だと思います。

- RT 技術を、従来のオペレーティング・システムを実行しながら予測可能性を向上させたいというユーザーに提供すること
- この技術をもっと使いやすくすること

ソフト RT への動き

WebSphere Real Time の多くのフィーチャーは、従来のオペレーティング・システムをターゲットとするプログラマーにとって有益なものです。インクリメンタルGC と優先度ベースのスレッドは、たとえハード RT の保証条件を満たすことができず、ソフト RT のパフォーマンスしか提供できなくても、明らかに多くのアプリケーションで役立ちます。予測不可能なGC 遅延のない予測可能なパフォーマンスを提供するアプリケーション・サーバーは、多くの開発者にとって魅力的なアイデアです。同様に、アプリケーションが優先度の高いJava 正常性監視スレッドを妥当なスケジューリング・ターゲットで実行できるようにできれば、Java サーバーの開発は簡単になるでしょう。

RT の簡易化

Java 言語を使用するという利点を RT システムの作成プロセスに持ち込むだけでも、開発者にとっては多大なメリットになります。ただし、改善の余地は常にあり、私たちは絶えず、RTプログラミングをさらに簡易化できる新しいフィーチャーを評価しています。IBM alphaWorks サイトにアクセスして、私たちが進めているリアル・タイム・スレッド調査技術を試してみてください。この技術は、変化や遅延に対する許容度が極めて低く、とても頻繁に発生するイベントを開発者が

管理できるようにするためのものです(「参考文献」を参照)。このツールは、イベントを処理するコードを事前にロード、初期化、コンパイルしてから、ガーベッジ・コレクターとは独立してコードを実行することにより、RTSJでの NHRT の場合より数も煩雑さも少ない制限で確定性の高い動作を実現します。また、TuningForkという名前のツールもあります。これは、オペレーティング・システムからJVM を介し、アプリケーションへのパスを追跡し、詳細なパフォーマンス分析を実行しやすくするためのツールです。

著者について

Mark Stoodley

Mark Stoodley は、2001 年に University of Toronto でコンピューター・エンジニアリングの博士号を取得し、その翌年に、IBM Toronto Lab で開発された Java JIT コンパイル技術に携わるため同研究所に入社しました。2005 年初期からは既存の JIT コンパイラーをリアルタイム環境で動作するように適応させることによる、IBM WebSphere Real Time 向け JIT 技術に取り組んでいます。現在は Java コンパイル制御チームのリーダーとして、ネイティブ・コード・コンパイルの有効性をその実行環境で改善するという作業を行っています。余暇には、自宅の改装を楽しんでいます。

Mike Fulton

Mike Fulton は、1989 年、カナダ・ブリティッシュ・コロンビア州の Simon Fraser University でコンピューター・サイエンスの学位を取得しました。専攻はコンパイラー技術です。彼は卒業してから今に至るまでの 18 年間、IBM Toronto Lab で コンパイラー分野でのテスト、コード開発、ドキュメンテーション、サービス、アーキテクチャー、パフォーマンス分析に携わっています。扱った製品は C、C++、Java プログラミング、パーサー技術、デバッガー・プロファイラーなど広範囲に渡り、過去 7 年間は JIT Java コンパイルを対象としたコンパイラー最適化に取り組んでいます。2005 年には焦点をリアルタイム Java ソリューションの開発に移しましたが、研究所に入社して以来取り組んできた IBM zSeries 技術には引き続き携わっています。1999 年から、バンクーバー近郊の小さな町、Maple Ridge で在宅勤務しています。

Michael Dawson

Michael Dawson は、1989 年、University of Waterloo でコンピューター・エンジニアリングの学士号を取得した後、Queens University で暗号法を専攻し、1991 年に同大学を電気工学の修士号を取得して卒業しました。卒業後、セキュリティー・コンサルティング業に携わった彼は EDI セキュリティー製品を開発し、続いて新規事業の開発リーダーとして、さまざまなプラットフォームでセキュリティー製品を開発しました。それ以来、e-commerce アプリケーションを開発し、EDI 通信サービス、クレジット・カード処理、オンライン・オークション、電子請求書などのサービスとして実現する上で指導的役割を果しています。経験のある技術は C/C++ からJava および J2EE プラットフォーム、そして広範なオペレーティング・システムのコンポーネントに至るまで多岐に渡ります。 2006 年に IBM に入社してからは、J9 JVM および WebSphere Real Time に取り組んでいます。

Ryan Sciampacone

Ryan Sciampacone は 1997 年に Carleton University でコンピューター・サイエンスの学士号を取得して以来、コア VM 実装、JNI API 層、AOT コンパイルをはじめとする仮想マシン開発のすべての面に携わってきました。2002 年からは、J9 JVM 向けガーベッジ・コレクションの技術リーダー兼主任アーキテクトとして、JSE 実装で使用可能なスケーラブル・コレクター・スイートならびに Metronome コレクターと ME 構成コレクターに取り組んでいます。仕事以外では、ホッケー、ヨガ、そしてサイクリングを楽しんでいます。

John Kacur

John Kacur は Acadia University で美術の学士号を取得した後、Brock University でコンピューター・サイエンスの理学士号を取得しました。ウクライナでロシア語を学び、ドイツで英語教師を経験した後、2000 年からは IBM Toronto Lab で Java JIT コンパイラーに取り組んでいます。彼は、Linux オペレーティング・システムの熱烈なファンとして有名です。JIT の他、Linux プロファイラーに取り組んだ彼は、2005 年からリアルタイム・ Java プロジェクトに参加しています。