Java VM 用メモリー解析ツール Marusa

緒方 一則　河内谷 清久仁　三廻部 大　小野寺 民也

Marusa, a Memory Analysis Tool for Java VM

Kazunori Ogata, Kiyokuni Kawachiya, Dai Mikurube, and Tamiya Onodera

Java™ プログラム全体のメモリー使用量の内訳を解析するには，その実行環境である JVM™ 自体のメモリー使用量も同時に解析する必要がある．しかし，既存のツールで，JVM のデータ構造ごとのメモリー使用状況（JVM レベルのメモリー情報）と，OS のプロセス・メモリー管理情報（OS レベルのメモリー情報）を組み合わせ解析するものはなかった．筆者らが研究・開発している Java VM 用メモリー解析ツール Marusa は，これら二つのレベルのメモリー情報を統合して，JVM プロセスのメモリー使用量の詳細な解析や，物理メモリー使用量の解析が可能である．それによって，Java プログラムの動作の推測や JVM の設定変更の効果の確認に効果的に利用できる．

It is important to analyze the memory footprint of JVM™ itself, as well as that of Java program, for fully breaking down the memory footprint of a Java™ program. However, there was no such tool that analyzes both the memory usage in each of JVM data structures (JVM-level information) and the process management information from OS (OS-level information). Our memory analysis tool Marusa breaks down the footprint of JVM using the both levels of the information. It can also break down the amount of physical memory usage. These features are useful for detecting and analyzing problems caused by large footprint, and verifying the effect of changes in configuration parameters.

Words & Phrases：Marusa, Java, メモリー使用量解析, 解析ツール, JVM 構成パラメータ調整
Marusa, Java, memory footprint analysis, analysis tool, JVM configuration

1．はじめに

現在，多くの IBM ソフトウェアが Java で実装されている．Java プログラムは，Java 仮想マシン上で動作するので機種依存性が少ないという利点がある反面，Java 仮想マシン（JVM）や Just-in-Time（JIT）コンパイラーもメモリーを使用するので，メモリー使用量が C 言語で実装したプログラムより多くなることが多い．

Java プログラムの実行環境を構成する場合，Java ヒープ・サイズの設定には注意を払うが，JVM や JIT コンパイラーが使用するメモリーを意識して構成することは少ない．そのため，システム全体のメモリー使用量を測定すると，予想外に多い場合がある．その予想外に多いメモリー使用量が，Java プログラムの動作に必要なメモリーによるものか，あるいは，不適切な JVM の設定や Java プログラムの問題などにより浪費されているものかを判断し，適切なシステム構成や，プログラム修正によるメモリー使用量削減に役立てるには，何が多くのメモリーを使用しているかを調べる必要がある．

提出日：2008年5月12日　再提出日：2008年9月22日

JVM など仮想マシンのメモリー使用量解析に特有の難しさとして，仮想マシン上で動くプログラム（Java プログラムなど）のメモリー使用量と，仮想マシン自体が動作するために必要なメモリー使用量の両方を測定しなければならない点がある．例えば，JVM は，Java プログラムの実行に必要なデータ構造（クラス格納領域，Java ヒープ，Java スタック，JIT 生成コード領域および，JIT コンパイラーの作業領域）を確保すると同時に，JVM 自体のヒープやスタックも使用する［1］．

既存のメモリー使用量測定ツールには，JVM のデータ構造内のメモリー使用状況（JVM レベルの情報）を使用するものと，オペレーティング・システム（以下，OS）のプロセス・メモリー管理情報（OS レベルの情報）を使用するものがある．しかし，これらの情報を組み合わせて解析するものはなかった．JVM レベルの情報を使用するツールは Java ヒープの解析に特化したもの［2,3］が多く，Java オブジェクトのクラスやオブジェクト間の参照を解析することで，メモリーを浪費している Java コードの特定に使えるが，JVM や JIT コンパイラーが使用するメモリー使用量を測定

できない．一方，Windows® のタスクマネージャや，UNIX の ps コマンドなどの OS レベルの情報を用いるツールは，プログラム全体のメモリー使用量をもれなく測定できるが，JVM のデータ構造などを考慮しないために，何に使われているか分からないメモリーが多く残ってしまう．また，大規模なシステムでは JVM を複数立ち上げることが一般的であるが，JVM 間で共有可能なデータ量の解析なども行えない．より効率よいメモリー使用のためには，Java プログラムを対象とし，OS レベルの情報も利用してメモリー解析を行うツールが必要である．

筆者らのリサーチ・プロジェクトでは，これら二つのレベルの情報を統合して解析するメモリー解析ツール Marusa（Memory Analyzer for Redundant, Unused, and String Areas）を開発している．Marusa は，以下の解析の片方ないし両方を行い，メモリー使用量の多い部分を見つける．

- OS レベルの情報を使用してプロセス・メモリーをもれなく集計し，さらに JVM のデータ構造に対応するアドレス範囲については，JVM レベルの情報を組み合わせて，より詳細な解析を行う
- OS の物理メモリー割り当て情報を使用し，複数プロセス間でのページ共有を考慮して，物理メモリー使用量の解析を行う

実現にあたっては，両レベルのメモリー情報を「メモリー属性マップ」を導入して統合し，データ収集とデータ集計の処理の分離によってデータ収集時のオーバーヘッドを軽減している．

Marusa を用いることで，Java プログラム実行環境のサイジングの検証や，JVM の設定変更の効果を確認できる．また，定期的にメモリー使用量を測定することで，運用中にメモリー使用量が漸増し将来メモリー不足が起きる可能性などを事前に発見しやすくなる．

本論文では，両レベルのメモリー情報を統合した解析手法を提案する．また，Marusa による解析結果を，Java プログラムの動作の推測や JVM の設定変更の効果の確認に利用する方法を示す．以下，2 章で JVM プロセスのメモリー構成を整理し，3 章で Marusa の解析手法を説明し，4 章で測定結果と考察を述べ，最後に 5 章で本論文をまとめる．

2. JVM プロセスのメモリー構成要素

現バージョンの Marusa は，IBM の商用 JVM である IBM SDK, Java Technology Edition, Version 5.0［4,5,6］（以下 IBM JDK）の，Linux® x86 32bit 版をサポートする．本章では，Marusa によるメモリー解析の前提として，Linux 版 IBM JDK 5.0 のプロセス・メモリーの構成を説明する．**表 1** に，OS レベルおよび，JVM レベルの，プロセス・メモリーの構成要素をまとめた．

2.1 OS レベルのプロセス・メモリー構成要素

Linux など UNIX 系 OS では，プロセスの仮想アドレス空間は，コード，スタック，共有メモリー領域，ヒープおよび，それ以外のプライベート領域に分類できる．共有メモリー領域は，shmget() および shmat() システム・コールを使用して割り当てる．ヒープ領域のサイズは malloc() および free() の呼び出しで増減し，それ以外のプライベート領域は，mmap() システム・コールによるメモリー割り当てなどで増減する．また，Linux 上の malloc() では，大きな領域を 1 回の malloc() 呼び出しで割り当てると，内部では mmap() を使用してメモリーを確保する場合もある．

メモリー使用量の問題の多くには，実際に使用している物理メモリー量の解析が必要である．仮想メモリー機構を持つ OS は一般にデマンド・ページングを採用しており，プログラムがメモリーを確保しても，そのメモリーを実際に読み書きするまで物理メモリーの割り当てを遅らせたり，アクセスがないページをディスクに書き出したりすることで，物理メモリーを効率的に利用する．この場合，物理メモリーの実使用量を知るには，OS 内の仮想メモリー機構に問い合わせる必要がある．

2.2 JVM レベルのプロセス・メモリー構成要素

JVM の動作に必要なデータ構造には，クラス・ファイルから読み込まれたデータ（コンスタント・プールやメソッド・エリアなど）を格納するクラス領域，Java オブジェクトを格納する Java ヒープ，Java プログラムのローカル変数を格納する Java スタックおよび，JVM 自体の作業領域とスタックがある．また，JIT コンパイラーも，コンパイラー作業領域，生成コードを格納する領域および，スタックを JVM のプロセス・メモリー内に割り当てる．スタック以外のこれらのデータ構造は，ヒープおよび，それ以外のプライベート領域から確保される．Marusa では，JVM のデータ構造ごとにメモリー使用量を集計するが，対

表 1．プロセス・メモリーの構成要素

OS レベルの構成要素	JVM レベルの構成要素
コード	JVM および，JIT コンパイラーのコード
スタック	スタック （JVM および，JIT コンパイラーのスタック）
共有メモリー	共有クラス領域
ヒープおよび， それ以外の プライベート領域	クラス領域（非共有部分）
	Java ヒープ
	Java スタック （Java プログラムのローカル変数領域）
	JVM 作業領域
	JIT 作業領域
	JIT 生成コード
	その他の領域

応付けができなかったメモリー領域は「その他の領域」にまとめて集計する.

IBM JDK 5.0 から導入された「共有クラス」[7,8] や, Sun Microsystems® 社の HotSpot JVM [9] が実装している Class Data Sharing [10] のように, 複数の JVM プロセスでクラス領域を共有することで, 起動時間の短縮やメモリー使用量の削減を行う機能を持つ JVM もある. これらのクラス共有機構が使用するメモリー領域は, 通常のクラス領域とは別に OS の共有メモリー機構を使用して確保されるので, この領域も解析の対象に加える必要がある.

3. Marusa のメモリー解析

本章では,Marusa のメモリー解析手法と実装を説明する.

3.1 Marusa のメモリー解析手法

Marusa は, ユーザーが指定した瞬間のメモリー使用量を, OS レベルのメモリー情報と,JVM のデータ構造ごとのメモリー情報の両方を収集し解析する. 二つのレベルの情報を組み合わせることで, OS レベルのみ, あるいは JVM のデータ構造（特に Java ヒープ）のみの解析を行う従来技術より詳細な解析が可能になる.

メモリー使用量の解析は, データ収集とデータ集計の 2 ステップで行う. データ収集ステップは Java プログラムの実行中に行うため, 解析に必要なデータだけを少ないオーバーヘッドで収集するように実装する. 集計ステップは解析対象の Java プログラムとは独立した実行環境で行う. 集計用のプログラム（Marusa ポスト・プロセッサー）を用いて, 二つのレベルの収集データを統合し, JVM プロセスのデータ構造ごとにメモリー使用量を集計する.

データ収集と集計を別ステップに分離することの利点は, データ収集のオーバーヘッドを小さくできることと, 収集データ保存が容易なことである. Marusa のデータ収集のオーバーヘッドが発生するのは, malloc() の実行トレースを収集しなければ, ユーザーがデータ収集要求を出したときだけである. 収集データを保存することの利点は, プログラムを再実行することなく, いつでも解析できることである. 例えば, メモリー使用量を定期的に（毎週など）測定し, 収集データを保存しておけば, メモリー使用量の長期変化の調査に利用できるので, メモリー使用量が漸増し将来メモリー不足が起きる可能性などを事前に発見しやすくなる.

3.1.1 メモリー情報の収集

Marusa は, OS レベルのメモリー情報として, カーネル内での仮想アドレス管理単位（Linux では vm_area_struct で管理される領域）ごとの, アドレス範囲, アクセス権および, ファイルがマップされている場合はそのパス名を収集する. また, 仮想アドレス空間のページごとに, 割り当てられた物理メモリーの有無も収集する. JVM レベルのメモリー情報としては, JVM のデータ構造（Java ヒープやクラス領域など）ごとに, 割り当てられている領域および, 使用済みである部分のアドレス範囲を収集する.

3.1.2 メモリー情報の集計

異なるレベルのメモリー情報を統一して扱うために, Marusa 内部では, JVM プロセスの仮想アドレス空間内の各バイトがどのように使われているかを集計する「メモリー属性マップ」を使用している. すべての収集データが同一時点の情報であることと, 必ずアドレスを含むことから, アドレスをキーにして集計すれば複数の手段で収集したデータを統合できる. メモリー属性マップを用いて, JVM プロセス・メモリーのすべてのバイトについて, 例えば, このバイトは JVM 内のメモリー管理コンポーネントが呼び出した malloc() によって割り当てられ, クラス領域に使われ, 物理メモリーが確保されている, というような属性付けを行う.

メモリー属性マップ作成後, 同じ属性のメモリー領域を集計し, メモリー使用量をグラフで表示する. まず, 一つの属性で集計して大まかな分類を行い, それぞれを別の属性でさらに細分化して集計することで, さまざまな粒度での解析を行うことができる. イメージとしては, 表データをソート・キーを追加しながら詳細な集計を行う処理に近い.

3.1.3 実行トレース方式との比較

OS レベルの情報を利用する, 既存のメモリー解析ツールは, Sun Microsystems 社の DTrace※1 [11] のように, プログラムの実行トレースを解析するものが多い. このようなツールでは, 測定対象プログラムの実行中に解析に必要なトレースをフィルターで選択し解析することが多い. この方法では,

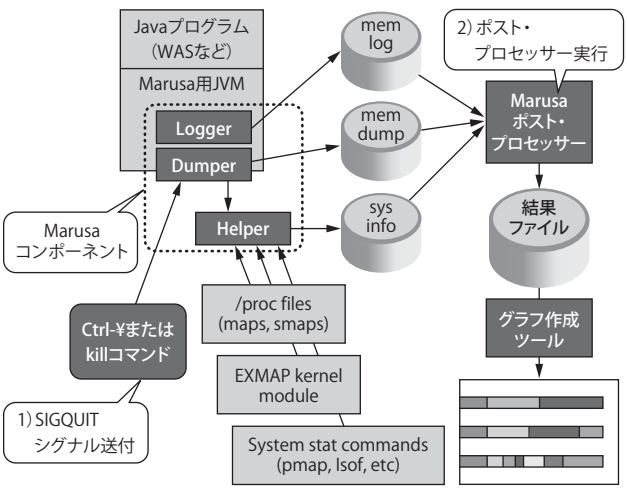


図 1. Marusa の構成と処理の流れ

※1 DTrace は, OS レベルの情報だけでなく, Java のオブジェクト生成やメソッド呼び出しなどもトレースできる. しかし, JVM 作業領域サイズの測定などはできない.

解析もJavaプログラム実行中に行うので，同じマシン上で解析を行うとオーバーヘッドが大きく，Javaプログラムの実行速度が低下する．また，メモリー使用量の長期変化の調査に使う場合，実行トレースは膨大なので，そのまま保存することは現実的でない．一方，解析結果だけを保存すると，過去のデータに対して，後から別の解析を行うことができない．

3.2 Marusaの実装

図1に,Marusaの構成と処理の流れを示す．Marusaでは，表2に示すMarusaコンポーネントを調査対象のIBM JDKに組み込み,OSやJVMのメモリー使用情報を収集する.ユーザーが，調査対象JVMプロセスにSIGQUITシグナルを送ると,MarusaDumperとHelperが起動され,収集データをファイルに書き出す．Marusa Loggerは,JVMの実行中にメモリー割り当て・解放処理があると呼び出され，収集データを記録する．Javaプログラムの実行中，複数回SIGQUITシグナルを送ることで，何度でもデータ収集を行うことができる．

収集データのファイルを指定してMarusaポスト・プロセッサーを実行すると，解析結果が結果ファイルに書き出される．それをグラフ作成ツールに入力し，図2のような，メモリー使用量をJVMのデータ構造ごとに分類したグラフを表示する．

3.2.1 Marusaコンポーネントの説明

Marusa Dumperは，JVMのコマンド行オプション-XrunMarusaでロードされるヘルパー・ライブラリーである．JVMがSIGQUITシグナルを受け取るたびに呼び出され，GCを実行してから，JVM内部のデータ構造の情報をmemdumpファイルに書き出す．

Marusa HelperはDumperから起動され，OS依存データの収集を行う．JVMのコマンドラインと環境変数，プロセス一覧，OSレベルのメモリー情報などの情報をsysinfoファイルに書き出す．さらに，OSの物理メモリー管理情報が取得可能ならば,物理ページが割り当てられているアドレス一覧や，同一ページ内で共有するプロセス一覧などの情報を保存する．なお，物理メモリー管理情報が取得可能かどうかは環境に依存するので，この情報は必須ではない．

Linuxの場合,OSレベルのメモリー情報として，/proc/<pid>/mapsの内容を保存する．物理メモリーの管理情報は，OS標準機能ではページ単位の詳細な情報を取得できないので，オープンソース・ソフトウェアのexmap［12］に含まれているカーネル・モジュールを利用した．このカーネル・モジュールを利用しない場合は，/proc/<pid>/smapsが表示するアドレス範囲ごとに，物理メモリー使用量の平均値から概算する．

Marusa Loggerは，malloc()やmmap()などOSレベルのメモリー割り当て・解放APIの呼び出しごとに割り当て・解放されたメモリー領域のアドレスとサイズをmemlogファイルに記録する．

表2．Marusaで使用するモジュール一覧

モジュール	機能	必須？
Dumper	JVMのデータ構造の情報を保存	Yes
Helper	JVMプロセスのOS依存情報を保存	Yes
	物理メモリー管理情報の保存	No
Logger	JVMのメモリー割り当て・解放を記録	No

Loggerは，JVM起動時に有効化されると，Javaプログラムの実行中は常にトレースを記録する．そのため，メモリー割り当て・解放APIの呼び出し頻度が高いと実行速度が低下する．その頻度は，JVMの初期化時やクラス・ロード時に高く，定常状態になれば低下すると考えられるが，実際の頻度はプログラムに依存する．SPECjvm98［13］ベンチマークでの測定では，トレースによる実行速度の低下は，各ベンチマークの幾何平均で約1.6%であった．実行速度低下の問題があるので，Loggerの使用は必須としていない．Loggerを使用しない場合，一部の領域は詳細な解析ができなくなるが，JVMのデータ構造レベルの分別・解析は可能である．

3.2.2 メモリー使用量の解析と結果表示

Marusaポスト・プロセッサーは，収集データを読み込み，メモリー属性マップを作成する．memlogファイルはメモリー割り当て・解放のAPI呼び出しを記録したものなので，APIの動作を再現しながらSIGQUITシグナルが送られた瞬間のマップを再現し，各領域を最後に割り当てたAPI呼び出しの情報で属性付けを行う．

物理メモリー・ページがプロセス間で共有されている場合は，ページごとに使用量を算入するプロセスを決め，そのプロセスのみメモリー属性マップに物理メモリー割り当て済みの属性を付けることで，2重カウントを回避している．

メモリー属性マップ作成後，同じ属性のメモリー領域を集計し，図2のような積み重ね棒グラフを作成する．グラフのL0～L5は細分化のレベルを表す．レベルごとの分類方法を表3に示す．現バージョンでは既定の分類方法のみ可能だが，分類方法をユーザーが対話的に選択できるように改良する予定である．

4. Marusaによる解析の応用例

本章では,エンタープライズWebアプリケーションのメモリー解析結果と考察および，Marusaを利用した共有クラス領域サイズのチューニング手法を説明する．

4.1 単一スナップショットのメモリー解析

図2は，エンタープライズWebアプリケーションを15分間実行後の，JVMのメモリー使用量の解析結果である．Javaヒープ・サイズは，このアプリケーションのデフォルト値である「-Xms50m -Xmx256m」に設定した．

表 3. 細分化レベルの説明

細分化レベル	分類方法
L0	OS レベルのプロセス・メモリーの分類．仮想アドレス空間の使用量を DLL，mapped file，その他のプライベート・メモリーに分類する．
L1	L0 を，JVM 仕様で定義されているデータ構造で分類した表示．クラス領域，JIT 領域，JVM 作業領域，Java ヒープ，その他の領域に分類する．
L2	L1 を，JVM の内部データ構造に応じて細分化した表示．例えば，非共有クラスについては，クラス・ファイルからロードされた部分とそれ以外の部分を分けて表示する．
L3	IBM JDK では，メモリーをまとめて確保し，内部でメモリー管理を行う領域がある．その中の未使用部分のサイズを分けて表示する．
L4	Java ヒープを，GC 後のライブ・オブジェクト・サイズを分けて表示する．
L5	L4 を物理メモリー使用量で細分化した表示．割り当て済みの物理メモリー量とスワップ領域量および，仮想アドレス空間だけが確保されている量を分けて表示する．

L1 から，JVM のデータ構造ごとのメモリー使用量が分かる．メモリー使用量が大きいのは，Java ヒープとクラス領域であった．JIT（生成コードと作業領域）のメモリー使用量は少なかった．JIT コンパイラーは動的にコード生成することからメモリーを大量に消費するように思われがちだが，実行頻度の高いメソッドだけをコンパイルするので，生成コードが大量にメモリーを消費することはない．

一方，クラス領域のメモリー消費量は従来あまり注目されていなかったが，JIT コンパイラーよりずっとメモリー消費が多

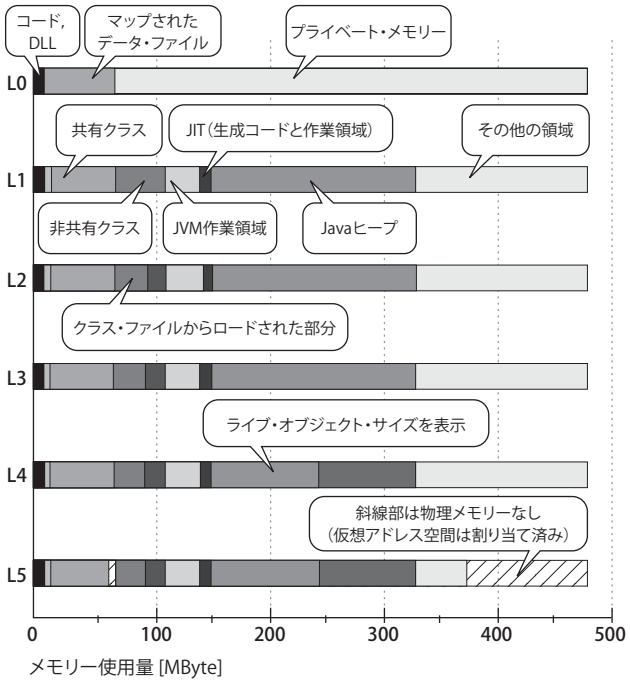


図 2. 15 分間実行後のメモリー解析結果

いことが分かる．この傾向は，筆者らが調査したほかの Java プログラムでも同様であった．

L4 から，GC 後のライブ・オブジェクトの総量が分かる．ライブ・オブジェクトの総量は Java プログラムに依存するので，1 回の測定で異常を発見できることは少ない．しかし，4.2 節に示すように，メモリー使用量の時間変化を測定すれば，異常発見につながる可能性もある．

4.2 メモリー使用量の時間変化の解析

図 3 は，4.1 節と同じ実行環境で，アプリケーション実行中に複数回 SIGQUIT シグナルを送り，メモリー消費量の時間変化を測定した結果である．測定は，WAS 起動直後，Web アプリケーションに最初にアクセスした直後，その後連続してアクセスしている状態で，30 秒後，1 分後，5 分後，10 分後および，15 分後に測定した結果の L5 を，時系列で並べている．なお，測定間隔が均一でないことに注意が必要である．

この測定では JVM のデータ構造ごとにメモリー使用量の時間変化が分かるので，変化を解析することで，Java プログラムの動作の推測や予期しない動作の発見の可能性がある．

例えば図 3 の結果では，クラス領域は最初のアクセス時に増加し，その後はほとんど変わらなかった．ここから，2 回目以降のアクセスで新たにロードされるクラスがほとんどなかったことが分かる．また，JIT 領域は，5 分後以降はほとんど増加していない．ここから，実行頻度の高いメソッドは，5 分以内にすべてコンパイルされたことが分かる．

今回測定した範囲では，Java ヒープとライブ・オブジェクトのサイズは時間とともに増加し続けていたが，このアプリケーションでは正常な動作と考えている．しかし，Java ヒープやライブ・オブジェクトの総量が予想以上に大きい場合は，MDD4J［2］など Java ヒープを対象とするツールを用いて詳細な解析を行い，予期しない動作がないことを確認することが望ましい．

4.3 共有クラス領域の最適化への応用

IBM JDK 5.0 から導入された「共有クラス」［7,8］は,ロードしたクラスを OS の共有メモリー上に保持することで，クラスのロードおよび初期化にかかる時間を短縮し，さらに複数の JVM プロセスを同一マシン上で起動する場合には，クラス領域を複数の JVM プロセスで共有することでメモリー使用量を削減する機能である．

IBM JDK の共有クラスでは,共有クラス領域に空きスペースがない場合は，新たにロードしたクラスを非共有クラス領域に格納する．共有クラスの効果を最大限に発揮するには，ロードしたすべてのクラスを共有クラス領域に格納することが望ましい．共有クラス領域にはクラス・ファイルからロードされた部分のみ格納できるので，非共有クラスのうちクラス･ファ

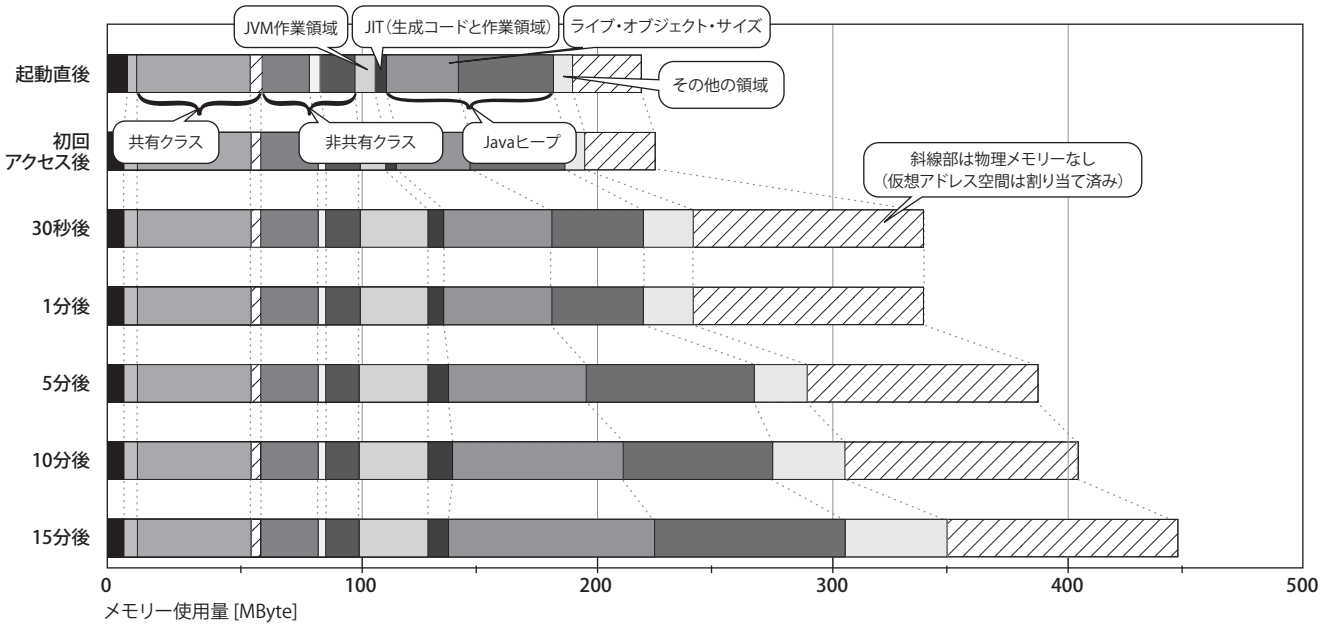


図 3. メモリー使用量の時間変化

イルからロードされた部分のサイズが最小になるように，共有クラス領域のサイズを設定すべきである．

共有クラス領域のサイズは IBM JDK のコマンド行オプション -Xscmx で増減できる．今回の測定に用いた Web アプリケーションでは，ミドルウェア部分のクラスがちょうど入るようにサイズが設定されていた．従って，このミドルウェア上に導入されたアプリケーションが多数のクラスを使用し，さらに複数の JVM を起動して運用している場合は，共有クラス領域のサイズを増やすことでマシン全体のメモリー消費量を減らせる可能性がある．

図 4 は，共有クラス領域をデフォルトの 50MByte と，100MByte に増やして実行した場合の L5 の結果である．最適化により非共有クラスのうちクラス・ファイルからロードされた部分が大きく減少し，共有クラスが増加したことが分かる．また，今回の測定環境では 63.1MByte 分しか物理メモリーが割り当てられなかったことから，最適な共有クラス領域のサイズは，70MByte 程度であることが分かる．

5. おわりに

Java プログラムは，JVM 自体もメモリーを使用するので，一般にメモリー使用量が多い．しかし，それが不適切な設定などの問題によるものかどうかを調べるための，JVM レベルと OS レベルの情報を組み合わせてメモリー使用量を調査するツールはなかった．筆者らが研究・開発している Marusa は，これらの情報を組み合わせて，JVM プロセス全体のメモリー消費量を JVM の内部データ構造の単位で詳細に解析するツールである．

本論文では，メモリー属性マップを使用して両レベルのメモリー情報を統合し，データ収集と集計を分離することでデータ収集のオーバーヘッドを削減するメモリー解析手法を提案した．また，Marusa によるメモリー解析結果から，クラス領域が Java ヒープに次いで多くのメモリーを消費することと，JIT 領域のメモリー使用量は少ないことを示した．さらに，JVM のデータ構造ごとのメモリー使用量を解析することで，プログラムの動作を推測したり，共有クラス領域サイズの最適化に応用したりできることを示した．

これらの解析を用いることで，Java プログラム実行環境のサイジングの検証や，JVM の設定変更の効果を確認できる．また，定期的にメモリー使用量を測定することで，運用中にメモリー使用量が漸増し将来メモリー不足が起きる可能性などを事前に発見できる場合もある．

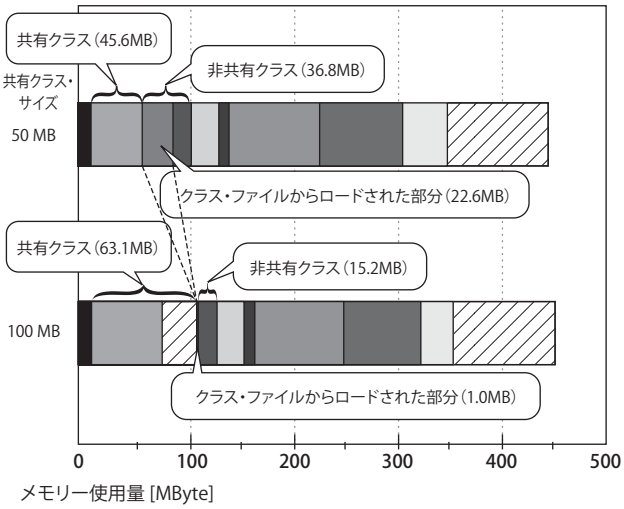


図 4. 共有クラス・サイズのチューニング

なお，本プロジェクトでは，Javaプログラムのメモリーの浪費を軽減するために，Javaヒープの大きな部分を占める文字列データの無駄を解析・削減する研究も行っている．この研究の詳細は，河内谷らの論文［14］で述べられている．

今後の予定としては，クラス単位やメソッド単位でのメモリー消費量など，さらに詳細な解析を行うことと，malloc()呼び出しやJavaオブジェクト生成など，メモリー割り当て時のコール・スタックを用いて，メモリーを要求したコードを探す手法を研究したい．

参考文献

［1］ T. Lindholm and F. Yellin: The Java Virtual Machine Specification, Prentice Hall PTR, ISBN0201432943, http://java.sun.com/docs/books/jvms/ (1999).
［2］ I. Poddar and R. John Minshall: “Memory leak detection and analysis in WebSphere Application Server: Part 1: Overview of memory leaks”, http://www.ibm.com/developerworks/websphere/library/techarticles/0606_poddar/0606_poddar.html (2006).
［3］ N. Mitchell and G. Sevitsky: “The Causes of Bloat, the Limits of Health”. Proc. OOPSLA '07, pp. 245-260 (2007).
［4］ C. Bailey: “Java Technology, IBM Style: Introduction to the IBM Developer Kit: An overview of the new functions and features in the IBM implementation of Java 5.0”, http://www.ibm.com/developerworks/java/library/j-ibmjava1.html (2006).
［5］ N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan: “Java Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications”. Proc. VM '04, pp. 151-162 (2004).
［6］ L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodly: “Inlining Java Native Calls At Runtime”. Proc. VEE '05, pp. 121-131 (2005).
［7］ “Class data sharing between JVMs”, IBM SDK for Linux platforms, Java Technology Edition SDK and Runtime Guide, http://www.ibm.com/ developerworks/java/jdk/linux/6/sdkandruntimeguide.lnx.html#classdatasharing (2008).
［8］ B. Corrie: “Java technology, IBM style: Class sharing”, http://www.ibm.com/developerworks/java/ library/j-ibmjava4/ (2006).
［9］ “Java SE HotSpot at a Glance”, http://java.sun.com/javase/technologies/hotspot/
［10］ “Class Data Sharing”, http://java.sun.com/j2se/1.5.0/ docs/guide/vm/class-data-sharing.html
［11］ B. Cantrill, M. Shapiro, and A. Leventhal: “Dynamic Instrumentation of Production Systems”. Proc. USENIX'04, pp. 15-28 (2004).
［12］ J. Berthels: “EXMAP”, http://www.berthels.co.uk/exmap/
［13］ Standard Performance Evaluation Corporation. SPEC JVM98, http://www.spec.org/jvm98/ (2004).
［14］ K. Kawachiya, K. Ogata, and T. Onodera: “Analysis and Reduction of Memory Inefficiencies in Java Strings”, Proc. OOPSLA '08, to appear (2008).

日本アイ・ビー・エム株式会社
東京基礎研究所
インフラストラクチャー・ソフトウェア
スタッフ・リサーチャー

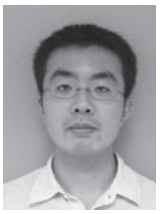
緒方 一則 Kazunori Ogata

［プロフィール］
1990年日本IBM（株）入社．IBM PCのシミュレーターの開発などを経て，現在，同社基礎研究所にて，Java仮想マシンおよびJITコンパイラーの高速化，高機能化および，信頼性能向上の研究に従事している．ACMおよび情報処理学会会員．
http://www.trl.ibm.com/people/ogata/

日本アイ・ビー・エム株式会社
東京基礎研究所
インフラストラクチャー・ソフトウェア
シニア・リサーチャー

河内谷 清久仁 Kiyokuni Kawachiya

［プロフィール］
1987年日本IBM（株）入社.以来,同社東京基礎研究所にて,オペレーティング・システムやプログラミング言語に関する研究に従事.Javaに関しては,Linux用JITコンパイラーの開発,同期処理の高速化,起動の高速化,使用メモリー解析と削減,仮想化などの研究を行っている.ACMおよび情報処理学会会員,日本ソフトウェア科学会理事.博士.
http://www.trl.ibm.com/people/kawatiya/



日本アイ・ビー・エム株式会社
東京基礎研究所
インフラストラクチャー・ソフトウェア
リサーチャー

三廻部 大 Dai Mikurube

［プロフィール］
2007年日本IBM（株）入社．以来，同社東京基礎研究所にて，セキュリティーやプログラミング言語処理系の研究に従事する．また，オペレーティング・システムとその仮想化技術,ユーザー・インターフェースなどにも興味を持つ．ACM会員．
http://www.trl.ibm.com/people/mikurube/

日本アイ・ビー・エム株式会社
東京基礎研究所
インフラストラクチャー・ソフトウェア
シニア・テクニカル・スタッフ・メンバー

小野寺 民也 Tamiya Onodera

［プロフィール］
1988年日本IBM（株）入社．以来，同社東京基礎研究所にて，オブジェクト指向言語の設計および実装の研究に従事．理学博士．ACM Senior Member．情報処理学会，日本ソフトウェア科学会，各会員．
http://www.trl.ibm.com/people/onodera/