スレッド局所性を利用したJavaロックの高速化

河内谷 清久仁[†] 古関 聰[†] 小野寺 民也[†]

Javaでは言語の性質上，オブジェクトに対するロック操作が頻繁に行われる．これを高速化することは，システム全体の性能向上に非常に重要である．オブジェクトがそれぞれどのスレッドにロックされているかに着目した調査を行ったところ，特定のスレッドにのみ頻繁にロックされているという「スレッド局所性」が見られることが分かった．この性質に着目し本論文では，各オブジェクトごとに特定のスレッドに「ロック予約」を与え，ロック処理を高速化する手法について述べる．予約を持っているスレッドは，従来よりも軽い処理でそのオブジェクトのロックを行える．具体的には，従来ロック処理に不可欠と考えられていた`compare_and_swap`などの不可分命令ではなく，単純なメモリアクセス命令でロックを獲得/解放できる．予約者以外のスレッドがロックを行った時点でロック予約が解除され，以後そのオブジェクトは従来の方式でロックが行われる．この予約ロック機構を，IBM Java VMとJITコンパイラに実装し，いくつかのベンチマークを走らせたところ，従来のロック手法に比べて最大で53%の性能向上が確認された．

Accelerating Java Locks by Utilizing Their Thread Locality

Kiyokuni Kawachiya,[†] Akira Koseki[†] and Tamiya Onodera[†]

In Java execution, lock operations are performed very frequently to realize exclusive operations among multiple threads. Therefore, accelerating the lock performance has been very important to execute Java-based applications faster. We investigated the lock behavior of Java programs, focusing on the relation of each object and threads acquiring the object's lock. It turned out that for many objects, the lock is acquired by only one thread specific to the object, even in multi-threaded Java programs. By utilizing the *thread locality*, this paper shows a novel ultra-fast locking technique for Java. The algorithm allows locks to be *reserved* for threads. When a thread attempts to acquire a lock, it can do without any atomic operation if the lock is reserved for the thread. Otherwise, it cancels the reservation and falls back to a conventional locking algorithm. We have implemented the lock reservation mechanism in IBM's production virtual machine and JIT compiler. The results show that it achieved performance improvements up to 53%.

1. はじめに

Java[1]の特徴の1つに，言語内に並列処理を記述する仕組みを持っていることがあげられる．これは具体的には，スレッドと呼ばれる実行主体を複数生成することで実現される．各スレッドは，Java処理系上で独立に実行されるが，それらの同期のための仕組みとして，Javaではモニタ機構[2),3)]を採用している．このモニタ機構は各オブジェクトに関連づけられており，1つのオブジェクトのモニタには同時点ではたかだか1つのスレッドだけが入る（ロックの獲得）ことができる．スレッドがモニタに入ろうとした際に，別のスレッドがそのオブジェクトのモニタにすでに入っていた場合，そのスレッドがモニタから出る（ロックの解放）まで次のスレッドは待たされる．モニタの出入りは，ロックの獲得/解放を個別に指定するのではなく，メソッドやブロックを`synchronized`と宣言することで指定される．

Javaのプログラム，特にライブラリは，マルチスレッド環境でも動作するよう「スレッドセーフ」に作成しなければならない．そのため，スレッド間で共有されうるデータ構造を扱うメソッドは，どうしても`synchronized`と宣言することになり，ロック処理が頻繁に行われてしまう．そのコストを下げることは，Java処理系の性能向上のために非常に重要であり，従来より様々な研究が行われてきている[4)~13)]．その結果，現在までに，衝突（contention）がない場合わずか数命令でロックを行える手法がいくつか開発されている[4)~7)]．しかし，それらのいずれの手法において

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan, LTD.

も，compare_and_swap に代表される「不可分命令」が用いられている．最近のプロセッサアーキテクチャでは，このような不可分命令の実装が重くなる方向にあり，ロック処理の新しいオーバヘッドになってきている．

不可分命令は，複数のスレッドが偏りなく 1 つのロックをとりあう場合には非常に有用である．しかし，ロックを行うスレッドに偏りがある場合はその限りではない．もし，あるオブジェクトを特定のスレッドだけが頻繁にロックすることが分かっていれば，そのスレッドになんらかの優先権を与えた非対称なロック処理を行うことで，コストをさらに下げられる可能性がある．この観点から，本論文では「予約ロック[14]」というJava ロックの高速化手法を提案する．この手法では，各オブジェクトのロックを特定のスレッドが予約することができる．予約を持っているスレッドがオブジェクトをロックする場合は，不可分命令なしで処理を行うことでコストを最小化する．一方，他のスレッドに予約されているオブジェクトをロックしたい場合は，まず予約の解除を行い，以後そのオブジェクトは従来方式でロックを行う．

以下，2 章では，Java プログラムにおけるロックの挙動を調べ，予約の有効性を探る．3 章で，我々の提案する予約ロックの詳細なアルゴリズムを述べ，4 章で各種の議論を行う．5 章では性能評価を行い，いくつかの可能な拡張についても示す．6 章で，関連する研究との比較を行い，7 章でまとめを示す．

2. Java ロックのスレッド局所性

我々が提案する予約ロックは，各オブジェクトのロックを特定のスレッドが予約することで処理の高速化を行う．これが有効であるためには，オブジェクトがそれぞれ特定の（予約を与えた）スレッドにのみ頻繁にロックされているという局所性が見られなければならない．本章では，この「スレッド局所性」についての調査を行う．

各オブジェクトごとに，生成から消滅までの間にロックを行ったスレッドの ID を調べ，そのオブジェクトの「ロック列」として記録する．1 つのオブジェクトのロック列に同じスレッドが連続して現れる場合，局所性があるということができる．ただし，すべてのスレッド局所性が高速化のために利用可能なわけではない．**図 1** に示す 2 つのオブジェクトのロック列を考えてみる．オブジェクト 1 は，スレッド B に対して局所性があるといえるが，ランタイムシステムが，実行途中でそれを予測し適切な予約を与えることは難しい．一方，オブジェクト 2 のように，最初にロックを行ったスレッドが引き続きロックを繰り返す場合は，このスレッドに優先権を与えることは比較的容易である．そこでまず，このような「最初のスレッドによるロックの繰返し」がロック操作全体のどの程度の割合を占めているかを調査する．

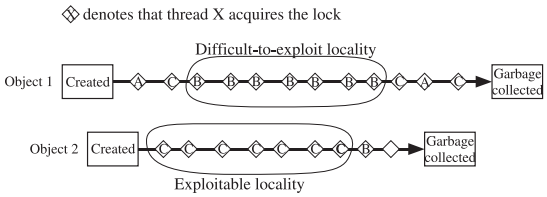

図 1 スレッド局所性の例
Fig. 1 Thread localities.

調査は，IBM Developer Kit for Windows, Java Technology Edition, Version 1.3.1[15] を変更したもので行った．調査した Java プログラムは SPECjvm98[16] の 7 つのプログラム（それぞれアプリケーションモードで 3 回続けて実行），SPECjbb2000[17]（ウェアハウス数 8 で実行），Volano Mark[18] のサーバおよびクライアント（200 の接続を張り各 100 個のメッセージを送る）の 10 種類である．このうち，SPECjvm98 の _227_mtrt と，SPECjbb2000，Volano Mark はマルチスレッドプログラムである．なお，この調査は Java プログラム自体のロックの挙動を調べることを目的としており，JIT コンパイラによる他のロック最適化の影響を避けるため，JIT は使用していない．

調査結果を**表 1** に示す．左側の 3 つの桁は，各プログラムについて，行われたロック操作の総数と，そのうち「最初の繰返し」の中で行われたものの割合を示している☆．ほとんどのロック操作は，そのオブジェクトを最初にロックしたスレッドによる最初の繰返しの中で行われていることが分かる．これはシングルスレッドプログラムでは当然であるが☆☆，マルチスレッドプログラムでも，ロック操作の 75%以上が最初の繰返しの中で行われている．つまり，オブジェクトを最初にロックしたスレッドにロックの優先権を与えることで，全ロック操作の 75%以上を高速化できるということである．

なおここで，オブジェクトを最初にロックするスレッ

☆ 最初の繰返しが 1 つのロックだけで終わる場合もありうるが，これも 1 回からなる最初の繰返しとしてカウントされている．逆に，最初の繰返しが終わった後に，最初のスレッドによるロックが再び行われた場合は，カウントに含まれない．

☆☆ シングルスレッドプログラムでも割合が 100%になっていないのは，Java 処理系自体が生成するスレッドが一部のオブジェクトのロックを行うためである．

表 1 ロックの挙動調査
Table 1 Investigation of Java locks.

プログラム名	ロック操作の総数	最初の繰返しで行われた割合	オブジェクトの総数	うちロックされたもの	うち複数スレッドからロックされたもの
SPECjvm98					
_202_jess	14646978	99.993%	23999733	21278	187 (0.878%)
_201_compress	28895	97.211%	18586	2135	127 (5.948%)
_209_db	162117521	99.9998%	9883475	66592	52 (0.078%)
_222_mpegaudio	27168	98.108%	26456	1620	91 (5.617%)
_228_jack	38570415	99.998%	19334735	1635497	144 (0.0088%)
_213_javac	47062772	99.974%	19140558	1192734	1760 (0.148%)
_227_mtrt	3522926	99.557%	21622389	3020	114 (3.775%)
SPECjbb2000☆	102282147	79.392%	18511021	2077210	176318 (8.488%)
Volano Server	7244208	75.983%	719245	7279	1888 (25.94%)
Volano Client	10419671	84.270%	3957166	4102	1640 (39.98%)

ドは，そのオブジェクトを生成したスレッドとは限らないという点に注意が必要である．実際，Volano Mark の 2 つのプログラムは，1 つのスレッドがオブジェクトを生成し複数の作業スレッドに渡すという構造になっている．このため，オブジェクトを生成したスレッドにロックの優先権を与える手法では高速化が期待できない．

表 1 の右側の 3 つの桁は，ロックの挙動をオブジェクト数の観点からみた参考データである．ロックされたオブジェクトのうち，複数のスレッドからロックされるものの割合は，多いものでも 40%程度であることが分かる．すなわち，ロックされたオブジェクトの半分以上は，実は 1 つのスレッドからしかロックされていないということである．

3. 予約ロック

前章の調査で，Java のロックにはスレッド局所性がみられ，最初にロックを行ったスレッドに優先権を与えることで処理を高速化できそうだということが分かった．本章では，その具体的な手法である「予約ロック」について述べる．

この手法では，各オブジェクトのロックを特定のスレッドが予約することができる．スレッドがオブジェクトをロックしようとした際，予約の状態に応じて以下の処理が行われる．

(1) そのオブジェクトのロック予約を持っていた場合，ロック処理は不可分命令なしに高速に行われる．
(2) 他のスレッドがロック予約を持っていた場合，まずその予約を解除する処理が行われる．以後そのオブジェクトは従来の方式でロック処理が行われる．
(3) そのオブジェクトのロック予約がない（すでに解除されていた）場合，従来の方式でロック処理が行われる．

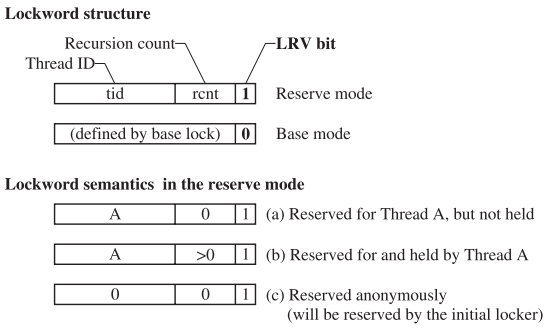

図 2 ロックワードの構造と意味
Fig. 2 Lockword structure and semantics.

3.1 データ構造

本章で提案する予約ロックは，各オブジェクトのヘッダ内にロック処理用のワード（ロックワード）☆☆を用意しているロック手法であれば，容易に組み合わせて実装することができる．ロックワード内に，予約の有無を示す 1 ビット（Lock ReserVed: LRV ビット）を用意する．LRV ビットが 1 の場合を「予約モード」，0 の場合を「ベースモード」と呼ぶ．ロックワードの残りのビットは，予約モードでは予約ロックによって管理され，ベースモードでは元になる従来のロック手法（ベースアルゴリズム）によって管理される．

図 2 に，ロックワードの構造を示す．ロックワードが予約モード（LRV ビットが 1）の場合，残りの部分はさらにスレッド ID（`tid`）フィールドと再帰カウンタ（`rcnt`）フィールドに分割される．前者はそのロッ

☆ SPECjbb2000 の総ロック数および総オブジェクト数は，実行速度によって変動するためあまり意味がない．
☆☆ 正確にはロックワードとして 32 ビット全部が必要なわけではなく，同じワード内にロックと関係のないフラグ情報などを置くこともできる．しかしここでは説明を簡単にするため，1 ワード全部がロック処理に用いられるものとする．

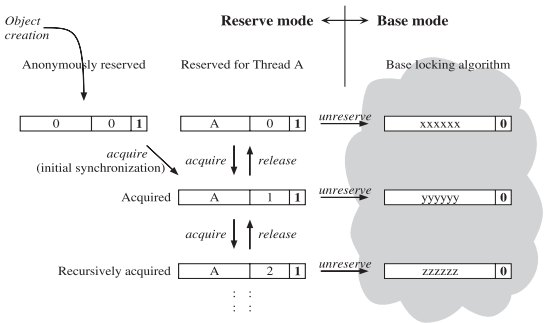

図 3 ロックワードの状態遷移
Fig. 3 Lock state transitions.

クを予約しているスレッド（予約スレッド）を示し，後者はロックの再帰獲得レベルを示す．

`rcnt` フィールドが 0 の場合（図 2 (a)），そのロックは予約されているが獲得されていない状態である．1 以上の場合（図 2 (b)）は，予約スレッドがロックを行っている状態で，値はその再帰獲得レベルを示している．次節で示すように，予約スレッドは `rcnt` フィールドを単純に増減することでロックの獲得/解放を行える．この際に不可分命令を用いる必要はない．

予約モードで，`tid` フィールドと `rcnt` フィールドがともに 0 の場合（図 2 (c)）は，「匿名予約」という特別な状態を示しており，そのオブジェクトを最初にロックしたスレッドに予約が与えられる（予約の確定）．オブジェクトが生成される際，ロックワードはこの状態で初期化される．

予約が解除される際に LRV ビットはクリアされ，ロックワードはベースモードとなる．この状態遷移は，予約スレッドを一時的にサスペンドし，予約モードのロックワードを，ベースモードの対応する状態へと置き換えることで行われる．ロックワードの残りの部分の構造は，ベースアルゴリズムが自由に定義してよい．

図 3 は，以上のロックワードの各状態間の遷移を示したものである．

3.2 アルゴリズム

図 4 が，予約ロックの具体的なアルゴリズムである☆．スレッドがオブジェクトのロックを獲得する際，Java 処理系内で `acquire()` 関数が呼び出される．この関数はまず，指定されたオブジェクトのロックワードを読み出し，予約ロックによる高速処理が可能かど

☆ このアルゴリズムにおいて，スレッド操作関数群 `thread_xxxx()` は，対象スレッドが存在しない場合は何もせず `FAIL` を返す．`thread_suspend()` は，同一スレッドに対して複数回行うことができ，`thread_resume()` が同じ回数だけ呼ばれた時点で対象スレッドの実行が再開される．

```
 1 : // Lockword structure in each object header
 2 : struct Object {
 3 :         :
 4 :   struct lockword {              // [tid:rcnt:R]
 5 :     unsigned int tid     : N;    // Thread ID of the owner
 6 :     unsigned int rcnt    : M;    // Recursion count
 7 :     unsigned int reserve : 1;    // LRV bit
 8 :   } lockword;
 9 :         :
10 : };
11 :
12 : int acquire(struct Object *obj)
13 : {
14 :   struct lockword l1, l2;
15 :   int myTID = thread_id();
16 :
17 :  retry_acquire:
18 :   l1 = obj->lockword;   // read the lockword   ---------(1)
19 :                                                        A
20 :   // check special cases                                |
21 :   if (l1.reserve == 0)     goto base_acquire;          |
22 :   if (l1.tid == 0)         goto make_specific;       unsafe
23 :   if (l1.tid != myTID)     goto unreserve_and_base;  region
24 :   if (l1.rcnt == RCNT_MAX) goto unreserve_and_base;    |
25 :                                                        |
26 :   // reserved for me, and rcnt < RCNT_MAX              |
27 :   l2 = l1; l2.rcnt++;                                  V
28 :   obj->lockword = l2;   // write the lockword  ---------(2)
29 :   return SUCCESS;
30 :
31 :  make_specific:
32 :   l2 = l1; l2.tid = myTID; l2.rcnt = 1;
33 :   if (compare_and_swap(&obj->lockword, l1, l2) != SUCCESS)
          goto retry_acquire;
34 :   return SUCCESS;
35 :
36 :  unreserve_and_base:
37 :   unreserve(obj, l1.tid, myTID);
38 :
39 :  base_acquire:
40 :   return base_acquire(obj);
41 : }
42 :
43 : int release(struct Object *obj)
44 : {
45 :   struct lockword l1, l2;
46 :   int myTID = thread_id();
47 :
48 :  retry_release:
49 :   l1 = obj->lockword;   // read the lockword   ---------(1)
50 :                                                        A
51 :   // check special cases                                |
52 :   if (l1.reserve == 0)     goto base_release;          |
53 :   if (l1.tid != myTID)     goto illegal_state;       unsafe
54 :   if (l1.rcnt == 0)        goto illegal_state;       region
55 :                                                        |
56 :   // reserved for and held by me                       |
57 :   l2 = l1; l2.rcnt--;                                  V
58 :   obj->lockword = l2;   // write the lockword  ---------(2)
59 :   return SUCCESS;
60 :
61 :  illegal_state:
62 :   return IllegalMonitorStateException;
63 :
64 :  base_release:
65 :   return base_release(obj);
66 : }
67 :
68 : void unreserve(struct Object *obj, int ownerTID, int myTID)
69 : {
70 :   struct lockword l1, l2;
71 :   struct Context context;
72 :
73 :   if (ownerTID == myTID) ownerTID = 0;
74 :   thread_suspend(ownerTID);
75 :
76 :  retry_unreserve:
77 :   l1 = obj->lockword;
78 :   if (l1.reserve == 0) goto already_unreserved;
79 :   l2 = base_equivalent_lockword(l1);
80 :   if (compare_and_swap(&obj->lockword, l1, l2) != SUCCESS)
          goto retry_unreserve;
81 :
82 :   // modify the owner's context if it's in an unsafe region
83 :   if (thread_get_context(ownerTID, &context) == SUCCESS) {
84 :     if (in_unsafe_region(context.pc)) { // (1)<NextPC<=(2)
85 :       context.pc = retry_point(context.pc);
86 :       thread_set_context(ownerTID, &context);
87 :   } }
88 :
89 :  already_unreserved:
90 :   thread_resume(ownerTID);
91 : }
```

図 4 予約ロックのアルゴリズム
Fig. 4 Algorithm of lock reservation.

うかチェックする（21–24 行目）．このチェックをパスした場合（つまり，そのスレッドがロック予約を持っていた場合），単純に rcnt フィールドをインクリメントすることでロック処理が完了する（28 行目）．

高速処理が行えないのは以下のケースである．まず，すでに予約が解除されていた場合（21 行目），base_acquire() 関数が呼び出され，ベースアルゴリズムによる処理が行われる（40 行目）．ロックが匿名予約されていた場合（22 行目），compare_and_swap を用いて予約を確定する処理が行われる（33 行目）．ロックが他のスレッドに予約されていた場合（23 行目）は，まず unreserve() 関数を呼んで予約を解除する処理が行われる（37 行目）．rcnt フィールドがあふれインクリメントできない場合（24 行目）も，予約解除が行われる．

ロックを解放する際は，release() 関数が呼び出される．この関数もまずロックワードの状態チェックを行い（52–54 行目），スレッドがロック予約を持っていた場合は単純に rcnt フィールドをデクリメントすることで処理を終える（58 行目）．

release() 関数では，3 つの例外的なケースがチェックされる．まず，予約が解除されていた場合（52 行目）は，base_release() 関数が呼び出され，ベースアルゴリズムによる処理が行われる（65 行目）．残りの 2 つは，自分が獲得していないロックを解放しようとしたケース（53，54 行目）で，これらは Java の言語規約[1] に従い IllegalMonitorStateException として処理される（62 行目）．なお，我々のアルゴリズムでは獲得中のロックを他のスレッドが予約しているという状況は発生しないため，ロックの解放時に予約解除処理が行われることはない．

ロックを予約モードからベースモードに変換するのが unreserve() 関数で，獲得しようとしたロックを他のスレッドが予約中だった場合に呼び出される☆．この関数により，予約モードのロックワードが，対応するベースモードの値に置き換えられる（79，80 行目）．複数のスレッドがこの置き換えを行わないよう，この処理は compare_and_swap を用いて行われる．

ただし，予約スレッドは不可分命令を用いずロックワードにアクセスするので，データレースが生じないように配慮が必要となる．具体的には，acquire()，release() それぞれの処理において，(1) でロックワードを読み出した直後から (2) で書き込む直前までの間が「非安全区間（unsafe regions）」となる．unreserve() がロックワードを置き換えたときに予約スレッドがこの区間にいると，続く処理でロックワードが上書きされてしまい不整合な状態となる．

この問題を回避するため，unreserve() 処理では，まず予約スレッドをサスペンドして（74 行目）から，ロックワードをベースモードに置き換える．そして，予約スレッドの停止位置をチェックし，非安全区間で停止していた場合は，プログラムカウンタを対応する「リトライ点」（17 行目もしくは 48 行目）に強制移動させる（83–87 行目）．以上の処理が完了後，予約スレッドをリジュームさせる（90 行目）．非安全区間内には副作用が起こる処理がなくリトライ可能に設計されているので，この移動により整合性の問題が生じることはない．

4. 議　　論

続いて，前章で述べた予約ロックの実装について，いくつかの議論を行う．

4.1 正当性について

まず，アルゴリズムの正当性について検討する．我々のアルゴリズムでは，予約スレッドは通常のメモリ読み出しと書き込みでロックワードにアクセスする．そのため，この読み出しと書き込みの間にロックワードが変更されると，不整合な値を上書きしてしまうことになる．

しかし，確定した予約が存在している状態で，予約スレッド以外がロックワードを変更するのは，unreserve() でのロックワード置き換え処理（80 行目）だけである．この処理は，前章で述べたように，予約スレッドを一時停止させて行われ，非安全区間にいた場合，強制的にロックワード読み出し前のリトライ点に移している．これにより，すでに解除された予約に基づいてロックワードに書き込みを行うことが防げ，整合性が保証される．なお，匿名予約の確定（33 行目）や，予約解除時のロックワードの置き換え（80 行目）では，複数のスレッドが競合する可能性があるが，これらは compare_and_swap で行われるので，最終的に 1 つのスレッドだけが成功することが保証されている．

ロックワードがいったんベースモードになると，再び予約モードになることはない．そのため，予約解除後の正当性はベースアルゴリズムによって保証される．ただし，スレッドスケジューリングのタイミングによっては，予約解除後も，予約モードでのロックワードの書き換えが試みられることがありうる．しかし，この

☆ ほかに，rcnt フィールドがあふれる場合や，wait() メソッドが呼ばれた場合にも，unreserve() 関数が呼び出される．

うち匿名予約の確定（33 行目）と予約解除時のロックワードの置き換え（80 行目）は，compare_and_swap で行われるので，すでに予約解除されていた場合には必ず失敗する．また，予約下でのロックワード書き込み（28，58 行目）は，予約解除処理で予約スレッドが非安全区間から排除されるので，解除後にここが実行されることはない．

4.2 性能について

次に，このアルゴリズムの性能を定性的に述べる．まず，ロック予約が成功している場合，不可分命令が不要なので処理が高速化されると期待できる．これが，予約ロックの導入目的である．

一方，予約がない場合は，従来のロック処理が行われるので，追加オーバヘッドはほぼゼロである．厳密には，予約がないことのチェック（21，52 行目）が必要であるが，ベースアルゴリズムによっては，そこに元々存在するチェックとまとめて行うことができる．予約がない状態から予約のある状態に戻ることはないので，ベースアルゴリズム側に余計な追加処理は必要ない．

性能上問題となりうるのは予約解除処理で，この処理はロック処理に比べてかなり重いことが予想される．しかし，我々のアルゴリズムでは再予約を行わないので，予約解除は各オブジェクトの一生においてたかだか一度しか起こらない．そのため，恣意的に作成したプログラム以外でこれが性能上の問題となることはないと考えている．

上でもふれたように，我々のアルゴリズムでは，いったん予約が解除されると以後そのロックの処理はベースアルゴリズムで行われ，再予約されることはない．これは以下の理由からである．

(1) 前章の実験結果から，再予約なしでも十分な予約成功が見込まれる．
(2) 再予約により予約解除が余分に発生し，性能を低下させる危険性がある．
(3) 再予約を実現するアルゴリズムは複雑になると予想される．

4.3 非安全区間について

もし，ロックの獲得と解放がつねにランタイム関数 acquire() と release() を呼んで行われるのであれば，システム内には 2 つの非安全区間しか存在しないことになり，予約解除時（84 行目）には 2 つの範囲チェックを行うだけでよい．しかし，実際には JIT コンパイラがこれらのロック処理をインラインしたコードを生成し，多数の非安全区間ができてしまう可能性がある．この場合でも，それぞれの非安全区間とリトライ点の表を適切に管理し検索すれば問題は生じないが，Bershad らが提案した 2 ステージチェック手法[19]を用いて，大規模な表を使わず高速に検出することも可能である．これにはまず，ロック処理コードをインライン生成する際に，非安全区間もしくはその周辺に目印となるコードパターン（JIT コンパイラが他の部分には生成しないもの）を埋め込んでおく．ロック処理のコード列は確定しているので，非安全区間の各命令の内容と，対応する目印コードおよびリトライ点までの距離は固定的に定まる．予約解除時には，予約スレッドをサスペンドさせた後，停止位置の命令を読み出し，目印コードが存在するべき場所を求める（ステージ 1）．そこに実際に目印コードがあれば（ステージ 2），停止した位置は非安全区間内であるということになるので，対応するリトライ点に予約スレッドの処理を強制移動させる．

別の改良として，予約下でのロックワード書き込み（28，58 行目）を compare_and_swap() で行い，失敗した場合はリトライするようにすれば，非安全区間をなくすことができる．当然，この場合のロック処理の性能は低下するが，性能が要求されない部分ではこの方式を用いることで，非安全区間の数を減らすことができる．たとえば，高速性が要求されない Java VM インタプリタ内のロック処理関数ではこの方式を用い，高速性が要求される JIT コンパイルされたコード内でのみ，本来の（非安全区間の存在する）予約ロックコードを用いるという方法が考えられる．

Java VM の実装によっては，サスペンドしたスレッドが実行中の処理の種類を簡単に知ることができる場合がある．たとえば，次章の性能評価で用いる Java VM では，各スレッドが JIT コンパイルされたコードを実行中かどうかを示す「実行モード」を内部で保持している．このような場合，上の手法と組み合わせることで非安全区間チェックのコストを下げることができる．具体的には，以下のようにする．

```
  82 : // modify the owner thread's context if necessary
| 82a: if (get_exec_mode(ownerTID) == EXECUTING_COMPILED_CODE) {
  83 :  if (thread_get_context(ownerTID, &context) == SUCCESS) {
  84 :    if (in_unsafe_region(context.pc)) {
  85 :      context.pc = retry_point(context.pc);
  86 :      thread_set_context(ownerTID, &context);
  87 :  } }
| 87a: }
  88 :
```

非安全区間を JIT コンパイルされたコード内だけに制限し，82a 行目のチェックを追加することで，予約スレッドがそれ以外のコード（インタプリタや，イベント待ちなど）を実行中の場合はプログラムカウンタをチェックする処理が不要になる．

4.4 マルチプロセッサ環境での実装について

Java 言語規約[1] では，17 章で Java のメモリモデルについて規定している☆．これによれば，ロード操作はロックの獲得を越えて前に動かすことができず，ストア操作はロックの解放を越えて後に動かすことはできない．このため，緩和メモリモデル[22] を採用したマルチプロセッサシステム上で予約ロックを用いる場合は，適切なメモリバリアを張る必要がある．

なお，現実問題としては，予約モードではこのようなメモリバリアは不必要であると考えている．予約モードでロックの獲得/解放が成功した場合は，別のスレッドが同じクリティカルセクションを実行中であるということはありえないからである．複数スレッド間で必要となるメモリフラッシュなどの処理は，予約解除で予約スレッドがサスペンドする際にまとめて行うことができる．

5. 性能評価

次に，ここまでに述べた予約ロック機構を実際の Java 処理系上に実装し，評価を行う．

5.1 評価対象

評価のベースとして用いた処理系は，2 章の調査でも用いた IBM Developer Kit for Windows，Java Technology Edition，Version 1.3.1[15] である．付属の JIT コンパイラ[23),24)] も同時に変更している．

ベースアルゴリズムとしては，この処理系が採用している「Tasuki ロック[7]」を用いた．このロック手法では，ロックがスレッド間で衝突していない場合（「フラットモード」）は，`compare_and_swap` でロックの獲得，単純なメモリストア☆☆ で解放が行える．衝突が起こった場合は，ロックワードが一時的に「ファットモード」へと遷移し，待合せをサポートするモニタ機構によって処理が行われる．

予約ロックは，JIT コンパイルされたコードから呼ばれるランタイム関数として実装し，インタプリタで行われるロック処理は，4.3 節で述べた方法で非安全区間をなくしている．

以後の測定はすべて，2 個の 1.7 GHz Pentium 4[25] Xeon プロセッサと 1024 MB のメモリを搭載した IBM IntelliStation M Pro（6850）の Windows 2000 SP2 上で行ったものである．

☆ Java メモリモデルは，Pugh によりいくつかの欠陥が指摘されており[20]，改定が検討されている[21]．

☆☆ マルチプロセッサ環境では，続く衝突検出のためにメモリバリアが必要である．予約ロックでは，予約成功中は衝突していることがありえないのでこのバリアも省略できる．

表 2 ロック処理のコスト
Table 2 Synchronization costs.

ロックワードの状態	最外ロック	再帰ロック
予約モード	61.4 nsec	61.4 nsec
フラットモード	229.5 nsec	61.4 nsec
ファットモード	335.5 nsec	155.8 nsec
フラット（オリジナル）	228.9 nsec	62.2 nsec
ファット（オリジナル）	330.3 nsec	150.0 nsec

表 3 予約ロックの状態遷移コスト
Table 3 Costs of lock state transitions.

状態遷移	Time
匿名予約の確定	89.0 nsec
予約解除（速いケース）	6741 nsec
予約解除（遅いケース）	18986 nsec

5.2 マイクロベンチマーク

まず，マイクロベンチマークにより，予約ロックの基本的性能を測定した．用いたプログラムは以下の 2 つである．

第 1 のプログラムは，ロック処理自体のコストを測定するためのものである．同期用のオブジェクトを 1 つ作り，そのロックワードが，予約モード，フラットモード，ファットモードになっている状態で，以下の計測を行う．

- `synchronized` セクションを n 回実行し時間を計測する（最外ロックのコスト測定）．
- 別の `synchronized` セクション内で同じ計測を行う（再帰ロックのコスト測定）．

別途用意した，ロック処理で何も行わない特別な Java VM での実行結果との差分から，各状態でのロック処理のコストを求める．比較のために，予約ロックの場合とオリジナルの Tasuki ロックの両方を測定した．

表 2 がその結果で，各状態で 1 回のロック獲得＋解放に要した時間を示している．予約がある状態では，最外ロックのコストが 70%以上削減されていることが分かる．これによる儲けは 1 つ 1 つでは些細なものであるが，2 章でも示したように，プログラムによっては大量にロック処理が行われているため，それらで予約が成功すれば性能差として現れてくることが期待できる．一方，予約がない状態では，オリジナルとほぼ同等の性能が発揮できている．

第 2 のプログラムは，予約ロックでのロックワードの状態遷移のコストを測定するためのものである．同期用のオブジェクトを n 個作り，以下の計測を行う．

- 各オブジェクトについて 1 回ずつ `synchronized` セクションを実行し，時間を計測する（匿名予約確定のコスト測定）．

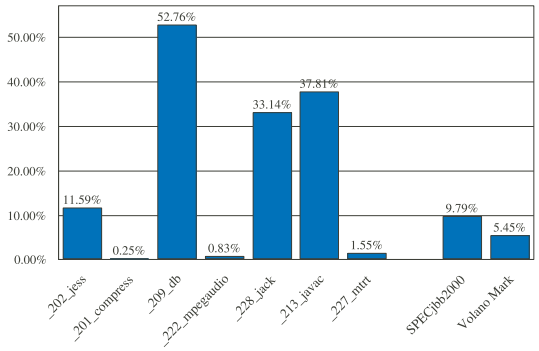

図 5 予約ロックによる性能向上
Fig. 5 Performance improvements by lock reservation.

表 4 ロック処理の統計情報
Table 4 Lock statistics.

プログラム名	ロック操作の総数	高速化された率	予約解除された率
SPECjvm98			
_202_jess	14585409	99.289%	0.00125%
_201_compress	29150	31.547%	0.419%
_209_db	162079177	99.963%	0.0000296%
_222_mpegaudio	27480	35.837%	0.313%
_228_jack	35207339	91.947%	0.000395%
_213_javac	43510883	99.402%	0.00403%
_227_mtrt	3523262	99.035%	0.00284%
SPECjbb2000	335718621	58.544%	0.0535%
Volano Server	6862014	79.755%	0.0248%
Volano Client	10381000	84.333%	0.0138%

- 続いて，別のスレッドで 1 回ずつ `synchronized` セクションを実行し，時間を計測する（予約解除のコスト測定）．

この結果と，状態遷移のない通常の予約モードでの実行結果との差分から，各状態遷移のコストを求める．

表 3 に結果を示す．なおこの数値にはロック自体に要した時間は含まれていない．まず，匿名予約確定のコストは，無視できる程度の小ささである．予約解除のコストは，予約スレッドをサスペンドするだけですんだ場合と，コンテクストの取得が必要だった場合で大きく異なるため，両方の数値を示した．いずれにしても，予約解除のコストはかなり大きく，ファットモードでのロック処理コストと比べても 20〜60 倍を要している．これは，Windows では，特にスレッドのコンテクストを取得する処理（`GetThreadContext()`）がかなり重いためで，より良い実装を検討していく必要がある．

5.3 マクロベンチマーク

続いて，実際のプログラムでの効果を測定した．測定に用いたのは，2 章で調査に用いたものと同じベンチマーク群である．オリジナルの Tasuki ロックと，予約ロックが導入された場合とで，それぞれ数回ずつ測定し，ベストスコア同士を比較している．なお，この測定は，JIT コンパイラが提供する様々な最適化はすべてオンになった状態で行っている．

図 5 がその結果で，予約ロックによる性能向上率を示している．元々ロックの回数が少ない `_201_compress` と `_222_mpegaudio` で有意な差がみられなかったほかは，すべてのテストで性能が向上している．特に，ロックの回数が多い `_209_db`，`_228_jack`，`_213_javac` では効果が大きく，30%以上の性能向上が得られている．これにより，SPECjvm98 のスコア（各ベンチマークの相乗平均で求められる）も，18.13%向上している．また，マルチスレッドプログラムである SPECjbb2000 と VolanoMark についても，5〜10%程度の性能向上が観測できている．

各ベンチマークの実行中に，予約による処理高速化と予約解除がどれくらいあったかを別途測定した結果を**表 4** に示す．JIT コンパイラによる最適化が行われている状況でも，依然として多くのロック処理が行われており，そのうち多くが予約ロックにより高速化できていることが分かる．なお，この表は実際にロックの高速化が成功したものの割合を示している．予約が成功していても，インタプリタ内でロックされたものや，再帰的なロックは高速化成功率には含めていない．特に，`_201_compress` と `_222_mpegaudio` で高速化成功率が低いのは，これらのプログラムのロックが，あまり実行されないコード内にあり，JIT コンパイルされないためである．この 2 つを除いた，ロックが頻繁に行われるプログラムでは，58%以上のロックが高速化できている．

5.4 可能な拡張

マイクロベンチマークの結果が示すように，予約ロックでは，予約が成功している場合は従来の方式に比べて性能が向上し，予約が存在しない場合でも従来とほぼ同等の性能となる．唯一問題となるのは，予約を保持しているスレッド以外がロックを行った場合の予約解除処理である．ただし，表 4 から分かるように，少なくとも本論文で測定したベンチマークのうちロックが頻繁に行われるものでは，予約解除は全ロック処理の 0.05%以下しか起こっておらず，性能上の足かせとはなっていない．

しかし，プログラムによっては，頻繁に予約解除が発生してしまうものもあるかもしれない．今後の研究・改良課題としては，予約解除処理のより低コストな実装方法と，予約解除が発生する率を下げるロック予約

の初期割当方式の洗練があげられる．たとえば，予約解除の動向をモニタしておき，特定のクラスやコード位置で生成されたオブジェクトで頻繁に予約解除が発生することが分かった場合，以後生成されるオブジェクトには最初から予約を与えないように動的に挙動を変更するという手法が考えられる．また，トレース情報やプログラム解析などで，ロックを行うスレッドが予想できる場合，オブジェクト作成時にそのスレッドに対して予約を与えるという方法も考えられる．

最後に，本論文の方式では，いったん予約がはずれると以後の処理はベースモードで行われる．しかし，もし予約解除のコストを十分低くできれば，予約がはずれた場合に解除してしまうのではなく，新たなスレッドに予約を引き渡すという方式も検討する価値があるだろう．

6. 関 連 研 究

最後に，いくつかの切り口から，関連する研究をあげ，予約ロックの比較と位置づけを行う．

6.1 Java ロックの高速化

1 章でも述べたとおり，Java プログラムではロックが頻繁に行われる．そのため，Java の誕生以来様々なロック処理の高速化技法が提案されてきている．

初期の Java 処理系では，OS が提供するモニタ機構がそのままロックのために用いられており，非常に処理コストが高かった[26),27)]．Bacon らは，Java ではロックはほとんど衝突しておらず，待合せをサポートする重いモニタ機構は不要な場合が多いということを発見し，「Thin ロック」を提案した[5)]．この手法により，ロックがスレッド間で衝突していない間は，ロックワードを `compare_and_swap` で書き換えるだけでロックを獲得できるようになり，性能は大きく向上した．5 章の性能評価でベースアルゴリズムとして用いた Tasuki ロックは，これを改良したものである[7)]．

同様に，衝突していない場合を高速化するロック手法はいくつか提案されており[4),6)]，いずれの手法も，衝突がない状態では 1 つあるいは 2 つの不可分命令（と付随する数命令）でロック処理が行えるようになっている．本論文で述べた予約ロックは，この，最後まで残された不可分命令のオーバヘッドを取り除く試みである．衝突しないロックの挙動に着目し，多くのものが実は特定のスレッドからしかロックされていないという性質（Java ロックの「スレッド局所性」）を発見した．それを利用するために，ロックに「予約」という概念を導入し，予約スレッドは不可分命令なしでロックを行える手法を提案した．

6.2 Java ロックの除去

ロックのコストを下げるための別のアプローチとして，ロック処理自体をなくしてしまうというものがある．

Java において最もメジャーな手法は，脱出解析（Escape Analysis）[28)] により，生成スレッドからしか見えないオブジェクトを発見し，そのオブジェクトに対するロック処理をすべて省略してしまうというもので，多くの手法が提案されてきている[8)~13)]．この手法は，プログラム全体の挙動をコンパイル時に解析できる静的な処理系ではかなり有効であるが，Java は実行中のクラスロードなどが可能な動的言語であり，その環境下では脱出解析にも限界がある．

予約ロックでは，脱出解析で除去されうるロックはすべて予約が成功し高速化できる．それに加えて，脱出しているオブジェクトであっても，1 つのスレッドからしかロックされていない間は高速化できる．さらに，実行前の解析が不要なので，インタプリタ環境にも適用可能である．

ロックをなくすその他の手法として，再帰ロックを除去するというものがある．たとえば，JIT コンパイラが `synchronized` メソッドに別の `synchronized` メソッドをインラインしたときに，同期の対象が同じオブジェクトであることを確定できる場合は内側のロック処理を除去することができる．同様の最適化は Java プログラムの作成時にも可能で，`synchronized` メソッドから，同じオブジェクトに対する別の`synchronized` メソッドを呼ばないように工夫することで，無駄な再帰ロックを減らすことができる．Java の最近の標準クラスライブラリでは，この手法が幅広く使われている．予約ロックは，これらの手法では除去できない最外ロックを高速化するものであり，併用することでより高い効果が期待できる．

6.3 その他のロック改良

実は，`compare_and_swap` や `test_and_set` のような不可分命令を用いないロック手法は，すでにいくつか提案されている．

Bershad らは，OS 内のスケジューラを修正し，ロック処理のクリティカルセクション内でスレッドがプリエンプトされた場合は，再スケジュール時にクリティカルセクションの入口から処理を再開するという手法を提案している[19)]．彼らの定義した「restartable atomic sequence」は，我々の予約ロックにおける非安全区間と非常に似た発想であるといえる．

Java 処理系がユーザーレベルスレッドスケジューラ上に実装されていた場合，同様なロック手法が利用

可能である．たとえば，CACAO[29]やLaTTe[30]といったJava処理系では，ロック処理のクリティカルセクションでのスレッド切替えを禁止することで不可分命令なしにロックを実現している．しかし，このようなスケジューラを利用したロック手法は通常，ユニプロセッサシステムでしか利用できない．また，OSが提供するものとは別のスレッド機構を用いるため，JNIプログラムが使用するスレッドと整合性をとるのが困難であるという問題もある．一方，我々の予約ロックは，マルチプロセッサやプリエンプティブスケジューラの環境下でも問題なく動作する．

7. おわりに

本論文では,「予約ロック」という新しいJavaロック手法について提案した．これは，Javaでは多くのオブジェクトがそれぞれ特定のスレッドだけから頻繁にロックされているという「スレッド局所性」を利用した高速化手法である．

予約ロックでは，各オブジェクトのロックを特定のスレッドが予約することができ，予約スレッドがオブジェクトをロックする場合は，不可分命令なしで処理を行うことでコストを最小化できる．一方，他のスレッドが予約しているオブジェクトをロックしたい場合は，まず予約の解除が行われ，以後そのオブジェクトは従来方式でロックを行う．

提案する予約ロックを商用のJava処理系に実装して測定したところ，予約が成功している場合，ロックのコストを70%以上削減できた．また，実際のJavaプログラムにおいても，全ロック処理の58%以上が高速化され，その結果，Javaプログラムの実行速度が従来のロック方式の場合と比べて最大53%向上することを確認できた．

本論文の寄与点としては，以下の3点があげられる．

- Javaロックのスレッド局所性の発見
  Javaでは，特定のスレッドにしかロックされていないオブジェクトがかなり存在することの発見．
- 予約ロックの提案
  上記の性質を利用し，特定のスレッドに優先権を与える非対称型のロック手法の提案．
- 実装と性能評価
  予約ロックの，オーバヘッドが少なく従来の手法と親和性の高い実装，および性能向上の確認．

謝辞 普段より有用な意見をいただいている，IBM東京基礎研究所・ネットワークコンピューティングプラットフォームグループの皆様に感謝いたします．

参考文献

1) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison Wesley (1996).
2) Buhr, P.A., Fortier, M. and Coffin, M.H.: Monitor Classification, *ACM Computing Surveys*, Vol.27, No.1, pp.63–107 (1995).
3) Hoare, C.A.R.: Monitors: An Operating System Structuring Concept, *Comm.ACM*, Vol.17, No.10, pp.549–557 (1974).
4) Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y.S. and White, D.: An Efficient Meta-lock for Implementing Ubiquitous Synchronization, *Proc. ACM OOPSLA'99*, pp.207–222 (1999).
5) Bacon, D.F., Konuru, R., Murthy, C. and Serrano, M.: Thin Locks: Featherweight Synchronization for Java, *Proc. ACM PLDI'98*, pp.258–268 (1998).
6) Dice, D.: Implementing Fast Java Monitors with Relaxed-Locks, *Proc. USENIX JVM'01*, pp.79–90 (2001).
7) Onodera, T. and Kawachiya, K.: A Study of Locking Objects with Bimodal Fields, *Proc. ACM OOPSLA'99*, pp.223–237 (1999).
8) Aldrich, J., Chambers, C., Sirer, E.G. and Eggers, S.: Static Analyses for Eliminating Unnecessary Synchronization from Java Programs, *Proc. 6th Int'l Static Analysis Symposium* (*SAS'99*), pp.19–38 (1999).
9) Blanchet, B.: Escape Analysis for Object-Oriented Languages: Application to Java, *Proc. ACM OOPSLA'99*, pp.20–34 (1999).
10) Bogda, J. and Hölzle, U.: Removing Unnecessary Synchronization in Java, *Proc.ACM OOPSLA'99*, pp.35–46 (1999).
11) Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V.C. and Midkiff, S.: Escape Analysis for Java, *Proc. ACM OOPSLA'99*, pp.1–19 (1999).
12) Ruf, E.: Effective Synchronization Removal for Java, *Proc. ACM PLDI'00*, pp.208–218 (2000).
13) Whaley, J. and Rinard, M.: Compositional Pointer and Escape Analysis for Java Programs, *Proc. ACM OOPSLA'99*, pp.187–206 (1999).
14) Kawachiya, K., Koseki, A. and Onodera, T.: Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations, *Proc.ACM OOPSLA'02*, pp.130–141 (2002).
15) IBM developerWorks: Java Technology Zone. http://www.ibm.com/developerworks/java/
16) Standard Performance Evaluation Corpora-

tion: SPEC JVM98 Benchmarks.
http://www.spec.org/osg/jvm98/
17) Standard Performance Evaluation Corporation: SPEC JBB2000.
http://www.spec.org/osg/jbb2000/
18) Volano LLC: Volano Benchmarks.
http://www.volano.com/benchmarks.html
19) Bershad, B.N., Redell, D.D. and Ellis, J.R.: Fast Mutual Exclusion for Uniprocessors, *Proc. ACM ASPLOS V*, pp.223–233 (1992).
20) Pugh, W.: Fixing the Java Memory Model, *Proc. ACM Java Grande'99*, pp.89–98 (1999).
21) Java Community Process: JSR 133: Java Memory Model and Thread Specification Revision.
http://jcp.org/jsr/detail/133.jsp
22) Adve, S.V. and Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial, *IEEE Computer*, Vol.29, No.12, pp.66–76 (1996).
23) Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H. and Nakatani, T.: Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler, *Proc. ACM Java Grande'99*, pp.119–128 (1999).
24) Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H. and Nakatani, T.: Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal*, Vol.39, No.1, pp.175–193 (2000).
25) Intel Corporation: *IA-32 Intel Architecture Software Developer's Manual Vol.1–3*.
http://developer.intel.com/design/Pentium4/manuals/
26) Yellin, F. and Lindholm, T.: Java Runtime Internals, Presentation in JavaOne'96 (1996).
http://java.sun.com/javaone/javaone96/pres/Runtime.pdf
27) Armstrong, E.: HotSpot: A New Breed of Virtual Machine (1998).
http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html
28) Park, Y.G. and Goldberg, B.: Escape Analysis on Lists, *Proc. ACM PLDI'92*, pp.116–127 (1992).
29) Krall, A. and Probst, M.: Monitors and Exceptions: How to Implement Java Efficiently, *Proc. ACM Workshop on Java for High-Performance Network Computing*, pp.15–24 (1998).
30) Yang, B.-S., Lee, J., Park, J., Moon, S.-M., Ebcioglu, K. and Altman, E.: Lightweight Monitor for Java VM, *ACM SIGARCH Computer Architecture News*, Vol.27, No.1, pp.35–38 (1999).

(平成 15 年 2 月 18 日受付)
(平成 15 年 4 月 23 日採録)

河内谷清久仁（正会員）

1963 年生．1987 年東京大学大学院理学系研究科情報科学専門課程修士課程修了．同年日本アイ・ビー・エム（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マルチメディア処理システム，携帯情報システム，Java 処理系ランタイム等の研究に従事．現在，同研究所専任研究員．1994 年情報処理学会全国大会奨励賞受賞．ACM 会員．

古関　聰（正会員）

1969 年生．1998 年早稲田大学大学院理工学研究科電気工学専攻博士課程修了．同年日本アイ・ビー・エム（株）入社．以来，同社東京基礎研究所において，Java Just-In-Time コンパイラの開発に従事．工学博士．ACM 会員．

小野寺民也（正会員）

1959 年生．1988 年東京大学大学院理学系研究科情報科学専門課程博士課程修了．同年日本アイ・ビー・エム（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．現在，同研究所シニア・テクニカル・スタッフ・メンバー．第 41 回（平成 2 年後期）全国大会学術奨励賞，平成 7 年度山下記念研究賞，各受賞．理学博士．日本ソフトウェア科学会，ACM 各会員．