マルチスレッドCプログラムのための高互換かつ高効率かつ高精度な競合検出法

荒堀喜貴[†1] 小宮常康[†1] 多田好克[†1]

データ競合の動的検出法は現在までに多数提案されている．それらのうち，Choi らが提案した手法（後に O'Callahan と Choi が拡張）は，実競合（actual race）と潜在的競合（feasible race）の両方を高精度かつ高効率に検出でき，対象コードの手動変更が不要な最も効果的な手法の一つである．しかし，この手法は，現実の C プログラムに適用した場合，対象コードとの互換性と競合検出精度を維持できない．我々はこの問題を解決する一連の実装技術を提案する．まず，競合検査コードと対象コードの互換性を維持するために，検査コードをオブジェクト表ベースで実装する（fat ポインタは使用しない）．次に，競合検出精度を改善するために，共有オブジェクト内で排他的にアクセスされる部分領域を動的に識別し，それらに個別のイベント履歴を関連付ける．最後に，シグナルハンドラに挿入された検査コードが深刻な互換性の問題（競合状態）を回避できるよう，シグナル処理中の競合検査をハンドラ終了後まで遅延させる．実験の範囲内で，これらの技術は効果的に機能しており，その有効性が明らかになった．

Backwards-Compatible, Efficient and Precise Datarace Detection for Multi-Threaded C Programs

Yoshitaka Arahori,[†1] Tsuneyasu Komiya[†1]
and Yoshikatsu Tada[†1]

There have been proposed a large amount of techniques for dynamic datarace detection. Among them, an approach proposed by Choi et al. (extended by O'Callahan and Choi) is one of the most effective ones that can detect both actual and feasible races precisely and efficiently without requiring manual source code changes. When applied to real C programs, however, this approach can maintain neither of the backwards compatibility nor the accuracy of race checking. We propose a collection of implementation techniques to solve this problem. First, we implement check code based on object tables (instead of fat pointers) so that race checks can maintain the compatibility with checked code. Second, we dynamically recognize the exclusively-accessed subregions of a shared object and associate such regions with separate event histories to improve the accuracy of race checking. Finally, we defer race checks during the handling of signals until the handling finishes so that the check code inserted into signal handlers can avoid serious compatibility problems (i.e. race conditions). Within our experiments, these techniques have worked well, showing their effectiveness.

1. 序　　論

1.1 背　　景

データ競合とは，マルチスレッドプログラムなどで複数のスレッドが誤った様式で共有データをアクセスする場合に発生するバグである．より正確には，データ競合は（1）同一メモリ位置に対して異なる 2 個のスレッドがアクセスを行い，(2) そのうちの少なくとも一つが WRITE アクセスであり，(3) それらのアクセス間に順序関係が強制されない，という 3 条件を満たすアクセスの組として定義されている[11)]．

データ競合はテストやデバッグが非常に困難である．この難しさはマルチスレッド実行の非決定性に起因する．毎回同じ入力を与えて実行しても，プログラムのスレッド群の振る舞い（どのスレッドがどのタイミングで共有データをアクセスするか）は実行ごとに異なる．したがって，データ競合を含むマルチスレッドプログラムであっても，テスト時にその競合が顕在化することは稀である．ある実行で運良く競合が顕在化したとしても，スレッド実行の非決定性により，その競合を再現することは難しい．

このようにデータ競合はテストやデバッグが困難であるため，プログラムの開発効率を著しく低下させる．また，実システム内で稼働中のプログラムが引き起こすデータ競合は，システム全体の信頼性やセキュリティにも悪影響を及ぼす．したがって，データ競合の検出は重要な課題として認識されている．

†1 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications

このような事情から，現在までに様々なデータ競合検出法が提案されている．検出法は静的手法と動的手法に分類できる．静的手法は，検査対象プログラムのソースコードを解析し，データ競合が発生し得る箇所を予測する[5),13),14),17),19)]．動的手法は，検査対象プログラムに検査コードを自動挿入し，プログラムの実行と同時に競合検査を行う[2),7),9),12),15),18)]．

これらの競合検出法のうち，Choi らが提案した手法[2)]（後に O'Callahan と Choi が拡張[12)]）は最も効果的な手法の一つである．彼らの手法は動的であり，実競合（actual race）と潜在的競合（feasible race）[11)]の両方を高精度かつ高効率に検出する．Java プログラムを対象とする場合，この手法を実現する検査コードは対象プログラムとの間で互換性[⋆1]を維持できる．

彼らの手法に基づく競合検出器は，検査対象プログラムの実行時に，各オブジェクトに対して 1 個のアクセスイベント履歴を関連付ける．検査対象プログラムが各オブジェクトをアクセスする際に，競合検出器はそのアクセスイベントを対象オブジェクトのイベント履歴に記録する．アクセスイベントの記録時に，競合検出器はイベント履歴を探索し，現在のイベントと競合条件を満たす過去のイベントを探す．そのような過去のイベントを履歴から発見できた場合，競合検出器はそのイベントと現在のイベントのペアをデータ競合として報告する．

この手法では，競合判定の結果がアクセスイベントの発生及び記録の順序に依存しないため，実競合と潜在的競合の両方が検出可能である．更に，イベント履歴の検索に伴うオーバヘッドを大幅に削減する最適化が可能である．これらの利点により，イベント履歴に基づく競合検出法は最も効果的な検出法の一つであると考えられている．実際，この手法に基づく検出器は，実用 Java プログラムのデータ競合を高精度かつ高効率に検出することに成功している[2),12)]．

1.2 問題点の要約

Choi らの競合検出法は Java プログラムを対象とする限り極めて有効に機能するが，実用 C プログラムに適用しようとすると次の三つの重大な問題点に直面する．

- **問題点 1（互換性の問題）**: C では，各オブジェクトとイベント履歴の関連付けの実現が困難である．fat ポインタ[⋆2] を使用したり構造体の定義を変更するなどの安直な関連付けの実装では，元のプログラムが正常に動作しなくなる場合がある．
- **問題点 2（検出精度の問題）**: 各オブジェクトに単一のイベント履歴を関連付ける方式が多量の誤検出（false positive）を誘発する．
- **問題点 3（互換性の問題）**: 非同期シグナルハンドラに挿入された検査コードが競合状態を引き起こす．

1.3 提案手法の要点及び貢献

本研究では，これらの問題点を解決する一連の実装技術を提案する．我々は，以下に要約する実装技術に基づき Choi らの競合検出法を拡張する．それにより，実用 C プログラムに対する高互換かつ高精度かつ高効率な競合検出法を実現する（本研究の貢献）．

- **要点 1**: 各オブジェクトとイベント履歴の関連付けをオブジェクト表で実現する．これにより，対象プログラムと検査コードの互換性を損なうことなく，各オブジェクトのイベント履歴を記録できる．
- **要点 2**: 共有オブジェクト内で特定のスレッドから排他的にアクセスされる部分領域を動的に識別し，それらに個別のイベント履歴を関連付ける．これにより，各オブジェクトに単一のイベント履歴を関連付ける方式の誤検出を防止できる．
- **要点 3**: 非同期シグナルハンドラ実行中の競合検査をハンドラ終了後まで遅延させる．これにより，ハンドラ内の検査コードの競合を回避しつつ，対象プログラムの競合検査を実行できる．

1.4 実験結果の要約

我々は提案手法に基づく競合検出ツールを GCC[6)]の拡張及び実行時検査ライブラリとして実装し，実用 C プログラムへの適用実験を行った．実験の範囲内で，実用 C プログラムを対象とする高互換かつ高精度かつ高効率な競合検出法の実現に我々の提案手法が有効であることが分かった．

1.5 本稿の構成

以降，2 節で，Choi らの競合検出法を確認し，C プログラムへの適用時の問題点を明らかにする．3 節で，我々の提案手法を示す．4 節で，提案手法に基づく競合検出器の実験結果を示す．5 節で，関連研究を議論する．6 節で，結論と今後の展望を述べる．

⋆1 検査コードの挿入後も対象プログラムが元の正常な動作を維持できる場合，検査コードと対象プログラムは互換性がある（compatible）と言う．そうでない場合，互換性がないと言う．検査コードの挿入によって元のプログラムが動作しなくなる場合や誤動作を引き起こす場合，検査コードと対象プログラムは互換性がない．正常動作の維持の（相対的な）度合いに応じて，高互換，低互換とも表現する．

⋆2 ポインタのワードサイズを拡張し，通常のポインタ値の他に，ターゲットのオブジェクトに関するメタデータを持たせたもの．

2. Choi らの競合検出法

Choi らはイベント履歴に基づく動的競合検出法を提案した[2)]．Java プログラムを対象とする限り，この手法は，ソースコードの変更を要求せず，高精度かつ高効率な競合検出を実行できる．本節ではまず，この手法を確認する（2.1 節，2.2 節）．その後，この手法を実用 C プログラムに適用した場合に直面する問題点を述べる（2.3 節）．

2.1 データ競合の判定条件

本節では，Choi らの手法におけるデータ競合の判定条件を説明する．彼らは，検査対象プログラム内で発生するアクセスイベントを以下のように定義した．

定義 1 (アクセスイベント) アクセスイベント e は 5 つ組 (m, t, L, a, s) である．ここで，イベントの構成要素は以下の通り：

- m はアクセス対象のメモリオブジェクト
- t はアクセスを行うスレッドの ID
- L はアクセス時にスレッドが保持するロック集合
- a はアクセス種別（READ/WRITE）
- s はアクセス実行位置（ファイル名と行番号）

例えば，検査対象プログラムのスレッド `T` がロック `l` を獲得した後に共有配列 `a` に対して書き込みを行うというアクセスイベントを e とすると，$e = (\mathtt{a}, \mathtt{T}, \{\mathtt{l}\}, \mathtt{W}, ...)$ である（$e.s$ は競合判定に関与しないため省略した）．

2 個のアクセスイベント e_1，e_2 に対し，データ競合の判定条件 $IsRace(e_1, e_2)$ は次のように定義される．

定義 2 (データ競合の判定条件)

$$
\begin{aligned}
&IsRace(e_1, e_2) \\
&\Leftrightarrow (e_1.m = e_2.m) \wedge (e_1.t \neq e_2.t) \\
&\quad \wedge (e_1.a = \mathrm{WRITE} \vee e_2.a = \mathrm{WRITE}) \\
&\quad \wedge (e_1.L \cap e_2.L = \phi)
\end{aligned}
$$

すなわち，(1) 同一のメモリオブジェクトに対して異なるスレッドがアクセスを行い，(2) そのうちの少なくとも一つが WRITE アクセスであり，(3) 両アクセスが共通のロックで排他制御されない場合，それらのアクセスの組をデータ競合と判定する．

2.2 イベント履歴に基づく動的競合検出

Choi らの手法は，検査対象プログラムの実行時に各メモリオブジェクトに対して競合検出用のイベント履歴を関連付け（**図 1** (a)），実行中に発生したアクセスイベントを順次記録して行く（図 1 (b)）．この時，各記録の直前で，アクセス対象オブジェクトのイベント履歴を探索し，現在のアクセスイベントと条件 *IsRace* を満たす過去のイベントを探す．見付かった場合は，それらのイベントの組をデータ競合として報告し，見付からなければ，現在のイベントを履歴に記録する．図 1 の例では，`Debuggee` クラスのインスタンスの変数 `var`，配列 `a` の各々にイベント履歴が関連付けられている．`update` メソッド内で配列 `a` への代入イベント f が発生すると，検査コードは配列 `a` のイベント履歴を探索し，代入イベント f と競合条件 $IsRace(f, e*)$ を満たす過去のイベント $e*$ を探す．過去のあるイベント e_i について条件 $IsRace(f, e_i)$ が成立する場合，アクセスイベントの組 (f, e_i) をデータ競合として報告する．それ以外の場合はイベント f を配列 `a` のイベント履歴に記録する．

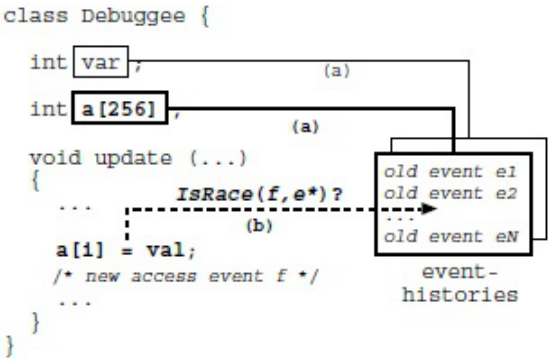


図 1 イベント履歴に基づく動的データ競合検出
Fig. 1 Dynamic Datarace Detection using Event Histories

Choi らの手法では，競合判定の結果がアクセスイベントの発生順序に依存しないため，実競合だけでなく潜在的競合も検出することができる．更に，実行進行に伴うイベント履歴のサイズ増加及び探索時間を抑制するために，競合検出に必要なアクセスイベントだけを選別し記録する最適化が可能である．実際，彼らは 2 個のアクセスイベント e_1，e_2 の間に競合に対する脆弱性の序列を表現する半順序関係 $e_1 \sqsubseteq e_2$ （weaker-than 関係）を定義し，より脆弱なアクセスイベントのみを履歴に残せば良いことを示した（weaker-than 定理）．更に，実行時に weaker-than 関係を効率的に判定するためのイベント履歴のデータ構造及び競合判定アルゴリズムを提示した．Java プログラムを対象とする実験の範囲内で，彼らの手法は実競合と潜在的競合を高精度かつ高効率に検出することに成功している．

2.3 問　　題

Java プログラムを対象とする限り，Choi らの競合検出法は極めて有効に機能する．しかしながら，彼らの手法を実用 C プログラムに適用しようとすると様々な問題に直面する．本節では，これらの問題を明らかにする．

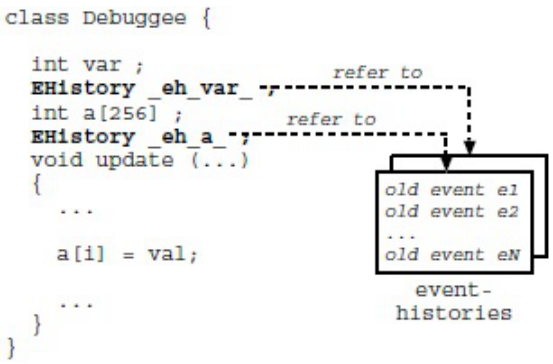


図 2 Java でのイベント履歴の関連付けの実現
Fig. 2 Implementation of Event History Association in Java

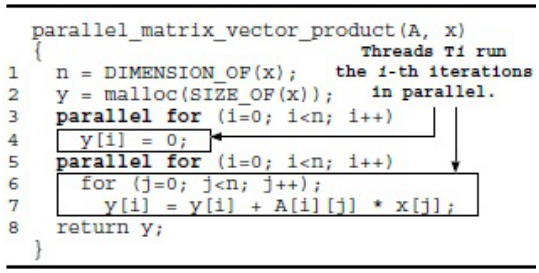


図 3 （イベント履歴の）粗粒度の関連付けが誤検出を起こす典型例
Fig. 3 A Typical Case where Coarse-Grained Association (of Event Histories) Incurs False-Positives

2.3.1 イベント履歴の関連付けの実現困難性

C では Java に比べ，メモリオブジェクトとイベント履歴の関連付けの実現が難しい．Java では，複数のスレッド間で共有され得るメモリオブジェクト[⋆1] は原則として何らかのクラスに属する．このため，各メモリオブジェクトとイベント履歴の関連付けは，各クラスの定義を編集するだけで容易に実現できる．具体的には，Java プログラムのコンパイル時（またはロード時）に，各クラスの定義を参照し，各インスタンス変数（またはクラス変数）の宣言に対してイベント履歴を保持する変数の宣言を追加すれば良い．例えば，図 1 の Java プログラムにおけるイベント履歴の関連付けは，**図 2** のように実現できる（`Debuggee` クラスのインスタンス変数 `var`，`a` に対し，各々のイベント履歴を保持する変数 `_eh_var_`，`_eh_a_` を追加している）．

一方，C では，メモリオブジェクトとイベント履歴の関連付けが困難である．C には Java のクラスに対応する機構が存在しないため，前述の関連付けは実行できない．構造体をクラスの一種と考え，各フィールドにイベント履歴保存用の変数を追加する方式も考えられる．しかし，この方式による関連付けは不完全である．なぜなら，構造体に属さないメモリオブジェクト（例えば，`int` 型のグローバル変数）も共有対象になり得るからである．また，特定の構造体のサイズやレイアウトに何らかの規約を設け，その規約を遵守することで動作するプログラムも存在する．この種のプログラムで構造体の定義を変更してしまうと，元の正常な動作は維持できない（互換性の問題）．同様の理由で，fat ポインタによる関連付けも互換性の問題を引き起こす．このように，C ではオブジェクトとイベント履歴の関連付けの実現が困難である．

2.3.2 粗粒度の関連付けに伴う誤検出

Choi らの手法は，各メモリオブジェクトに対して単一のイベント履歴を関連付ける．この方式を粗粒度の関連付け[⋆2] と呼ぶ．粗粒度の関連付けは，配列に対して各要素に個別のイベント履歴を関連付けるのではなく，全体に 1 個の履歴を関連付ける．Choi らの実験[2] 及び後続の O'Callahan らの実験[12] では，実用 Java プログラムに対し，粗粒度の関連付けが検査精度に悪影響を及ぼす例は報告されていない．

しかし，実用 C プログラムを対象とする場合，粗粒度の関連付けはしばしば多量の誤検出の原因となる．粗粒度の関連付けが誤検出を引き起こす典型的なマルチスレッド処理を**図 3** に示す．図 3 は，整数上の $n \times n$ 行列 $A = (a_{ij})$ と n 次元ベクトル $x = (x_j)$ の積を計算する疑似 C コード[⋆3] である．積は次の方程式を満たす n 次元ベクトル $y = (y_j)$ として求められる．

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j \quad \text{for } i = 0, ..., n-1.$$

この疑似 C コードは，2 行目で積を格納するベクトル（配列 `y` で表現）を確保し，3〜4 行目の並列ループで各要素を初期化する．続く 5〜7 行目の並列ループで，上記の方程式に基づき，各要素の値を計算する．並列ループでは，i 番目のイテレーションの実行を個別のスレッド $\mathtt{T}_i$ が担当する．すなわち，各スレッド $\mathtt{T}_i$ $(i = 0, ..., n-1)$ が要素 `y[i]` を個別に更新する．この時，各要素の更新はロックで保護されることなく実行されるが，単一スレッドによる排他的な更新であ

⋆1 コンストラクタやメソッド内でローカルに宣言される変数は共有候補（関連付けの対象）として扱わない．また，Java プログラムと JNI 経由で連動する他言語プログラムが生成するメモリオブジェクトも関連付けの対象外とする．

⋆2 これに対し，各メモリワードなどの（オブジェクト単位より）細かな記憶単位に対して個別のメタデータを関連付ける方式を本稿では細粒度の関連付けと呼ぶ．

⋆3 CLRS[3] の 27 章 Multithreaded Algorithms に掲載されている疑似コードを C の文法風に書き直したもの．

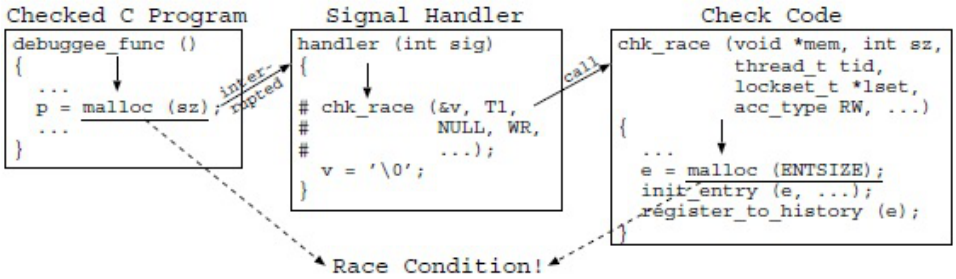


図 4 イベント履歴の更新が引き起こす競合状態
Fig. 4 A Race Condition caused by Event-History Update

るため，配列 y 上でデータ競合は起きない．

しかし，Choi らの競合検出法をこの疑似 C コードに適用すると，粗粒度の関連付けが原因となり，多量の誤検出が発生する．Choi らの手法では，配列 y に単一のイベント履歴しか関連付けない．その結果，各スレッドによる各要素の更新（ロックによる保護なし）を配列 y という単一のオブジェクト上でのデータ競合と誤判定してしまう．例えば，5〜6 行目の並列ループ実行のある時点で，スレッド $\mathtt{T}_k$ による要素 y[k] の更新イベント $e_k = (\mathtt{y}, \mathtt{T}_k, \phi, \mathtt{W}, ...)$ が発生し，その後，スレッド $\mathtt{T}_l$ による要素 y[l] の更新イベント $e_l = (\mathtt{y}, \mathtt{T}_l, \phi, \mathtt{W}, ...)$ が発生したとする．ここで，配列 y には全体に 1 個のイベント履歴しか関連付けていないため，競合検出器は各要素の更新イベントを全てその履歴に記録する．したがって，イベント e_l の記録時に履歴を探索した際に $IsRace(e_l, e_k)$ を満たすイベント e_k が見付かるため，検出器はアクセスの組 (e_l, e_k) をデータ競合として報告する（誤検出）．スレッド $\mathtt{T}_k$ と $\mathtt{T}_l$ の組に限らず，他の組でも同様の誤検出が起きるため，図 3 の疑似 C コードに対し，Choi らの検出法は多量の誤検出を報告してしまう．

図 3 の例に限らず，実用 C プログラムでは，単一のメモリオブジェクトの各部分領域に対し，個別のスレッドがロックを用いず排他的にアクセスするケースが見受けられる．このようなプログラムに対して Choi らの手法を適用すると，図 3 の例と同様に，粗粒度の関連付けが原因で誤検出が多発する．

2.3.3 イベント履歴の更新に伴う競合状態

Choi らの手法では，Java プログラムのコンパイル時に，メモリアクセスを行う各式（変数参照や代入など）に対して競合検出のための検査コードを挿入する．この検査コードは実行時に，アクセス対象オブジェクトのイベント履歴を探索し（履歴の参照），また，履歴にアクセスイベントを記録すること（履歴の更新）により競合検出を行う．

Choi らの手法を基に C プログラムの競合検出器を実現する場合，イベント履歴の更新の実現には動的メモリ管理関数を使用することになる．具体的には，イベント履歴に新たなアクセスイベントを記録する際に malloc などの関数で新たなイベントエントリを割り当てる．イベント履歴の最適化等で不要になったエントリは free などの関数で解放する．

しかしながら，このようなイベント履歴の更新は，非同期シグナルハンドラを備えた実用 C プログラムの検査時に（検査コード内で）競合状態を引き起こす．なぜなら，履歴の更新に伴う動的メモリ管理は一般に非同期シグナル安全ではないからである．**図 4** にイベント履歴の更新が引き起こす競合状態の例を示す．この例では，検査対象の C プログラムが関数 malloc の実行途中で非同期シグナルによって割り込まれている．対象プログラムはシグナルを受信すると該当のシグナルハンドラを起動するが，このハンドラにはコンパイル時に競合検出のための検査コード（chk_race）が挿入されている．起動したシグナルハンドラが検査コード（chk_race）を呼び出すと，検査コードは内部で malloc を呼び出してイベント履歴のエントリを新たに割り当てる．ここで，malloc は非同期シグナル安全でないため，検査コード内の malloc の実行は（割り込まれた malloc との間で）競合状態を引き起こす危険性がある．

図 4 の例は，対象プログラムが malloc の実行中にシグナルに割り込まれる例であるが，他にも，内部で動的メモリ管理関数を呼び出す各種のライブラリ関数（例えば，printf や strdup など）が割り込まれた場合も，競合検出に伴うイベント履歴の更新が競合状態を引き起こし得る．

3. 提 案 手 法

本節では，Choi らの競合検出法の問題点を同時に解決する一連の実装技術を提案する．まず，各メモリ

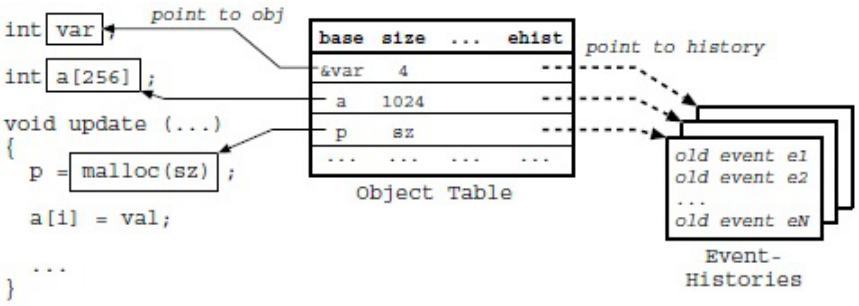


図 5 オブジェクト表を用いたイベント履歴の関連付け
Fig. 5 Event-History Association Using an Object Table

オブジェクトとイベント履歴の関連付けをオブジェクト表と呼ばれる heap 領域上の表で実現する方法を示す（3.1 節）．次に，イベント履歴の粗粒度の関連付けに伴う誤検出を低減する手法として，細粒度の関連付けの必要性を判定する状態機械及びそれに基づく粒度詳細化手法を提案する（3.2 節）．最後に，非同期シグナルハンドラ内の検査コードが引き起こす競合状態を回避する手法として，遅延検査と呼ぶ検査制御を導入する（3.3 節）．

3.1 オブジェクト表を用いた履歴の関連付け

本節では，マルチスレッド C プログラムにおけるオブジェクトとイベント履歴の関連付けの問題をオブジェクト表を用いて解決する方法を示す．

我々は対象 C プログラムのコンパイル時に，履歴関連コードと競合検査コードと呼ぶ 2 種類の検査コードを挿入する．以下では，このうちの履歴関連コードの実装を示し（3.1.1 節），オブジェクト表を用いたイベント履歴の関連付けが対象コードとの互換性を高度に維持できることを説明する（3.1.2 節）．

3.1.1 履歴関連コード

履歴関連コードは，各メモリオブジェクトとそのイベント履歴の関連付けをオブジェクト表と呼ばれる heap 領域上の表で管理する．履歴関連コードは，各オブジェクトの割り当て/解放の実行位置に挿入する．例えば，対象プログラム中の `malloc`/`free` の呼び出しに対し，次の挿入を行う（#で始まる行が履歴関連コード）：

```
  p = malloc(size);
# if (p != NULL) assoc_history(p, size);
  ...
  free(p);
# if (p != NULL) dissoc_history(p);
```

関数 `assoc_history` は，新たに割り当てられたオブジェクトのベースアドレス，サイズ，イベント履歴などからなるエントリをオブジェクト表に登録する．このエントリを履歴関連エントリと呼ぶ．関数 `assoc_history` の操作は，対象プログラムが割り当てたメモリオブジェクトに対してイベント履歴を関連付けることに相当する．履歴関連エントリは上記の属性の他に，オブジェクトの所有者や状態など，履歴関連付けの粒度を詳細化するための属性を保持する．これらの属性については 3.2 節で説明する．

一方，関数 `dissoc_history` は対象プログラムが解放したオブジェクトの履歴関連エントリをオブジェクト表から削除し，エントリ及びエントリが保持するイベント履歴を破棄する．この操作は，対象プログラムが解放したオブジェクトに対してイベント履歴の関連付けを解除することに相当する．

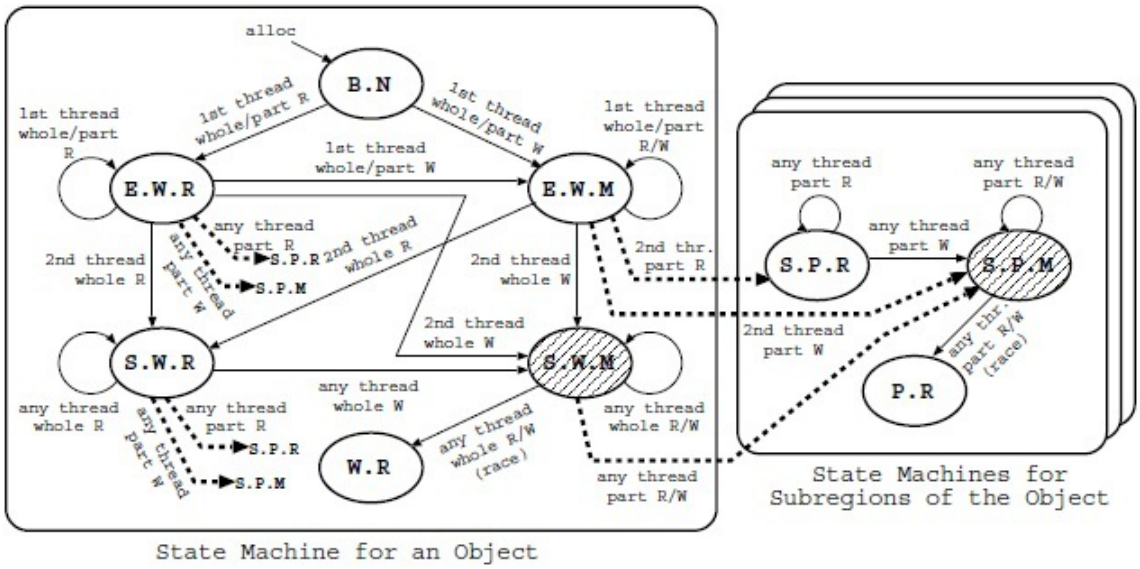
static 領域上のオブジェクトについても，履歴関連コードを挿入する：

```
  char g_buf[64];
# void init_global_objs(void) {
#   assoc_history(&g_buf[0],
#                 sizeof(g_buf));
# }
```

ここで，関数 `init_global_objs` はプログラムの開始直後に一度だけ呼ばれて各グローバル変数とそのイベント履歴の関連付けをオブジェクト表に登録する．stack 領域上のオブジェクトについては，複数スレッド間で共有されないと仮定し，競合検出及び履歴関連付けの対象外とする．

図 5 は，対象プログラムの使用する各メモリオブジェクトとイベント履歴の関連付けがオブジェクト表で実現された状態を表わしている．対象プログラムの実行進行とともに様々なメモリオブジェクトが割り当てられ解放されて行く．割り当てと解放に付随する履歴管理コードは各オブジェクトとイベント履歴の関連付けをオブジェクト表で一括管理する．

我々の検出器のプロトタイプ実装では，オブジェクト表を heap 領域上の splay 木[16] として表現し，各履



State Machine for an Object

図 6 有効メモリオブジェクトの状態機械
Fig. 6 State Machine for a Valid Memory Object

歴関連エントリをオブジェクトのベースアドレスとサイズの組で索引付けしている．splay 木を採用した理由は，(1) 実装が単純であり，かつ，(2) エントリアクセスの時間的局所性を利用してイベント履歴の取得に要する償却計算量を低く抑えられるからである．

3.1.2 検査コードと対象コードの互換性

オブジェクトとイベント履歴の関連付けをオブジェクト表で実現する事により，検査コードは対象コードとの互換性を高度に維持することができる．なぜなら，オブジェクト表に基づく関連付けは対象プログラムの構造体やポインタの内部表現（サイズやレイアウト）を一切変更しないからである．図 5 の関連付けの例からも分かるように，オブジェクト表に基づく検査コードは関連付けの情報を対象プログラムのデータ構造の外部で管理する．その結果，対象プログラムは検査コードの挿入後もデータ構造の相違による影響を受けることなく元の動作を高度に維持できる．

3.2 履歴関連付けの動的詳細化

本節では，粗粒度の関連付けに伴う誤検出の低減手法を提案する．2.3.2 節で述べた通り，各オブジェクトに単一のイベント履歴を関連付ける方式（粗粒度の関連付け）は，実用 C プログラムを対象とする場合，深刻な誤検出の原因となる．この問題に対処する最も単純な方法は，オブジェクト単位ではなくより細かな粒度（例えば，ワード単位）でイベント履歴の関連付けを行うことである．しかし，プログラムが使用する全てのメモリオブジェクトに細粒度の関連付けを行ってしまうと，空間的にも時間的にも大きなオーバヘッドが発生する．そこで，我々はオブジェクトの所有者・状態とアクセスの実行者・範囲・種別に応じて各オブジェクトの状態をより厳密に管理し，必要なオブジェクトにだけ細粒度の関連付けを行う手法を提案する．

3.2.1 オブジェクトの状態機械

図 6 に，細粒度の関連付けの必要性を判定するためのオブジェクトの状態機械を示す．各オブジェクトは所有者・状態という 2 種類の属性を持ち，アクセスイベントの発生時にアクセスの実行者・範囲・種別に応じて状態を遷移させる．

オブジェクトの所有者はそのオブジェクトにアクセスしたスレッド群を表す属性であり，図 6 の状態遷移において，アクセスの実行者との比較に利用する．我々は各オブジェクトの履歴関連エントリに所有者フィールドを設け，そのオブジェクトにアクセスしたスレッドの ID を記録する．複数のスレッドからアクセスを受けたオブジェクトの所有者フィールドには「共有（shared）」を意味する特別な ID [*1] を記録する．

オブジェクトの状態は図 6 の各状態に対応する属性であり，状態遷移を決定する入力の一つである．我々は各オブジェクトの履歴関連エントリに状態フィールドを設け，そのオブジェクトの状態 ID を記録する．状態 ID とは，後述する状態 B.N，状態 E.W.R，… な

⋆1 この ID は対象プログラムのどのスレッド ID とも一致しないものとする．

どの各状態を識別する値である．

アクセスイベントについては，実行者（アクセスを実行したスレッド），種別（READ/WRITE）という従来からの属性に加え，より詳細な状態管理のために範囲（whole/partial）という属性を新たに考慮する．具体的には，配列要素のアクセスイベントの範囲属性は「部分アクセス（partial）」として扱う．構造体フィールドのアクセスイベントも同様に扱う．プリミティブ型変数へのアクセスイベントの範囲は「全体アクセス（whole）」として扱う．

我々は次の 6 種類の状態のオブジェクトに対して粗粒度の関連付けを行う．

B.N（Brand-New）： 新規に割り当てられ，まだどのスレッドからも読み書きされていない状態．

E.W.R（Exclusive-Whole-Read）： 割り当ての後，最初にアクセスしたスレッド（1st thread）からしかアクセスされておらず，かつ，これまでのアクセス全てが読み出しである状態．この状態のオブジェクトの所有者は最初のスレッド（1st thread）である．他のスレッドにはまだ共有されていないため，現時点では競合の対象外であると判断し，イベント履歴の記録や細粒度の関連付けは行わない．所有者以外のスレッド（2nd thread）からアクセスを受けると，オブジェクトの所有者を「共有（shared）」に変更し，アクセスの範囲・種別に応じた状態遷移を行う．

E.W.M（Exclusive-Whole-Mod）： 割り当ての後，最初にアクセスしたスレッド（1st thread）からしかアクセスされておらず，かつ，これまでに少なくとも 1 回は書き込まれている状態．この状態のオブジェクトも状態 E.W.R のオブジェクトと同様にスレッドローカルなオブジェクトである可能性を考慮し，イベント履歴の更新や細粒度の関連付けは行わない．所有者以外のスレッド（2nd thread）からアクセスを受けると，オブジェクトの所有者を「共有（shared）」に変更するとともに，アクセスの種別・範囲に応じた状態に遷移する．

S.W.R（Shared-Whole-Read）： 複数のスレッドからアクセスされており，それらが全て読み出しである状態．例外として，状態 E.W.M からの遷移があるが，この遷移は「オブジェクトが初期化された後に複数のスレッドから読み出し専用でアクセスされる」という典型的なケースに対応する遷移である．この状態のオブジェクトの所有者は「共有（shared）」であり，複数のスレッドから共有されてはいるものの，アクセスは全て読み出しである．したがって，本状態のオブジェクトは競合の対象外として扱い，イベント履歴の更新及び細粒度の関連付けは行わない．アクセスを受けると，オブジェクトの所有者は変更せずに，アクセス種別・範囲に応じた状態遷移を行う．

S.W.M（Shared-Whole-Mod）： 複数のスレッドからアクセスされており，少なくとも 1 回は書き込まれている状態．この状態のオブジェクトは複数のスレッドから共有され（所有者は「共有（shared）」），かつ，編集されている．したがって，本状態のオブジェクトは競合検査の対象として扱い，イベント履歴の更新を行う．ただし，イベント履歴の関連付けの詳細化は，オブジェクトの部分領域へのアクセスが発生するまで実施しない．本状態のオブジェクトが全体アクセスを受けると，競合条件が成立する場合は状態 W.R に遷移し，そうでない場合は本状態に留まる．

W.R（Whole-Race）： 全体アクセスによる競合が検出された状態．この状態のオブジェクトにはイベント履歴の更新及び関連付けの詳細化を行わない．

3.2.2 状態機械に基づく履歴関連付けの詳細化

我々は前節で示した 6 種類の状態から本節で述べる 3 種類の状態への遷移時（図 6 の破線矢印）にのみ，イベント履歴の関連付けを詳細化する．この遷移はアクセスイベントが次の 2 条件を満たす場合に限り許可される限定的な遷移である：

- **条件 1:** アクセスイベントの実行者がオブジェクトの所有者と異なる．
- **条件 2:** アクセスイベントの範囲が「部分アクセス（partial）」である．

言い換えれば，複数スレッドが共有する部分領域へのアクセスの発生時にのみ，その部分領域に個別のイベント履歴を関連付ける[⋆1]．

このように履歴関連付けの詳細化を限定的に実施することで，次の二つの効果を同時に得ることができる：

- **効果 1:** 粗粒度の関連付けに伴う多量の誤検出を抑制できる．
- **効果 2:** 細粒度の関連付けに伴う（競合検査の）空間的/時間的オーバヘッドを低減できる．

このうち効果 1 の実験結果を 4 節で述べる．本節の残りでは，詳細化後の部分領域の状態管理と関連付けの詳細化の保留についてそれぞれ説明する．

関連付けの詳細化が決定すると，対象の部分領域には個別のイベント履歴とともに個別の状態を割り当て

⋆1 状態 S.W.M のオブジェクトの部分領域に個別のイベント履歴を関連付ける場合に限り，オブジェクト全体のイベント履歴の内容を個別の履歴にコピーする．他の状態のオブジェクトでは，そもそも履歴が空であるため，このようなコピーの必要性は生じない．

る．部分領域は以下の 3 種類の状態をとる．

S.P.R（Shared-Partial-Read）： 該当の部分領域が複数スレッドに共有されており，かつ，現在までのアクセスが（一部の例外を除き）読み出しのみである状態．本状態の部分領域は，読み出し専用で利用されているものとして扱い，競合検査の対象外とする．したがって，本状態の部分領域のイベント履歴は更新しない．本状態の部分領域は書き込みアクセスを受けた場合に限り，状態 S.P.M に遷移する．

S.P.M（Shared-Partial-Mod）： 該当の部分領域が複数スレッドに共有されており，かつ，少なくとも 1 回以上書き込まれている状態．本状態の部分領域は競合検査の対象として扱い，イベント履歴の更新を行う．本状態の部分領域がアクセスを受けると，競合条件が成立する場合は状態 P.R に遷移し，そうでない場合は本状態に留まる．

P.R（Partial-Race）： 該当の部分領域に対する部分アクセス（または部分領域をカバーする全体アクセス）について競合が検出された状態．本状態には状態 S.P.M（または状態 S.W.M）から遷移可能である．本状態の部分領域にはイベント履歴の更新を行わない．

以上の 3 状態の説明からも分かる通り，我々の状態管理では，個別の状態及び履歴が関連づけられた部分領域の所有者は常に「共有（shared）」である．この事実は「共有されない部分領域には個別の状態及び履歴を割り当てない」という詳細化方針の顕れでもある．実際，状態 E.W.M のオブジェクトが所有者と同一のスレッドから部分アクセスを受けても，我々の状態機械は該当の部分領域に個別の状態及び履歴を関連付けることはしない（図 6 の状態 E.W.M からの同状態への遷移「1st thread part R/W」を参照）．状態 E.W.R のオブジェクトについても同様である．このように，我々の状態機械はスレッドローカルなオブジェクトに対し関連付けの詳細化を保留することで，細粒度の関連付けに伴う高オーバヘッドを回避している．

3.3 遅 延 検 査

本節では，シグナル処理中のイベント履歴の更新が引き起こす競合状態の回避策として，遅延検査[1] を導入する．遅延検査は割り込み処理に伴うデータ競合の検出を支援する一連の技術であるが，本研究では，これをイベント履歴の更新に伴う競合を回避する目的に利用する．遅延検査はシグナル処理中の検査の実行をシグナルハンドラ終了後まで遅延させる．したがって，シグナル処理中のイベント履歴の更新はハンドラ終了後に実行されることになり，2.3.3 節で述べた競合状態を回避することができる．

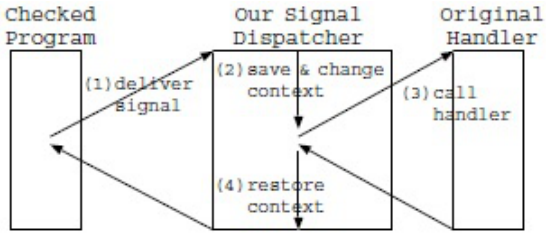


図 7 シグナルディスパッチ
Fig. 7 Signal Dispatching

上記の目的に向けて，マルチスレッドプログラムを扱えるように改変した遅延検査の実装を以下に要約する：

- 検査対象プログラムの各スレッドの実行コンテキスト（シグナル処理中か否か）を追跡管理する．この追跡管理はシグナルディスパッチ（後述）によって実現する．
- 各スレッドが検査コード（検査関数 `chk_race` の呼び出し）に到達した時に，実行コンテキストに応じて検査（検査関数の本体）の実行タイミングを制御する：

  Case 1: コンテキストが “シグナル処理中” の場合，検査実行をハンドラ終了後まで遅延させる．具体的には，検査関数は本体（イベント履歴の更新を含む）を実行せずに引数をスレッド固有のバッファ（検査バッファ）に保存してリターンする．

  Case 2: “シグナル処理中” でない場合，遅延中の検査を実行し，その後，現在の検査も実行する．具体的には，検査関数は検査バッファに保存された一連の引数を検査関数に与えて実行し，その後，自身の本体を実行する．

この実装は従来の遅延検査の実装とは異なり，実行コンテキストをスレッドごとに個別に追跡し，更に，検査バッファをスレッド固有領域上で実現している．

本研究の遅延検査により，二つの効果が得られる．まず，**Case 1** の処理により，各スレッドはシグナル処理中に検査コード内でイベント履歴を更新すること（及び非同期シグナル安全でない関数を呼ぶこと）を回避できる（検査コードの競合状態の回避）．次に，**Case 2** の処理により，シグナルハンドラのメモリアクセスをハンドラ終了後に検査できる（ハンドラ内の競合検査の実行）．

各スレッドの実行コンテキストの追跡管理は，シグナルディスパッチで実現する．我々は各スレッドに個別のコンテキスト変数を割り当て，同変数の値[⋆1] を

⋆1 説明の便宜上，コンテキスト変数が保持する値として “シグナル

以下の手順で管理する：

- **Step 1:** シグナルが発生した時，それを元のシグナルハンドラではなく（検出器の）シグナルディスパッチャに配送する（**図 7**（1））．
- **Step 2:** ディスパッチャは現在のコンテキスト変数の値を保存した後，変数値を “シグナル処理中” に変更する（図 7（2））．
- **Step 3:** ディスパッチャは受信したシグナルに対応する元のハンドラを呼び出す（図 7（3））．
- **Step 4:** 元のハンドラが完了した後，ディスパッチャはコンテキスト変数を元の値に復元し，リターンする（図 7（4））．

シグナルディスパッチにより，各スレッドの実行コンテキストがコンテキスト変数に反映される．上記の各ステップは従来の遅延検査と同様，シグナルハンドラ登録関数（`signal` や `sigaction` など）のラッパやディスパッチ表によって実現する．

4. 実　　験

本節では，我々の提案する一連の実装技術のうち，3.2 節で述べた「履歴関連付けの動的詳細化手法」に焦点を絞り，同手法の誤検出低減効果の予備評価結果を報告する．本研究の一連の提案手法は GCC4.3.2[6] の拡張及び実行時検査ライブラリとして実装した．実験環境には Intel Core2 Duo 1.33GHz × 2 及び 2GB RAM を搭載した Linux 2.6.24 ワークステーションを用いた．我々の競合検出器を実用 C プログラムに適用し，競合検出を行った結果を表 1 に示す．

表 1 履歴関連付けの動的詳細化による誤検出低減効果
Table 1 Effect of Dynamic History-Association Refinement on the # of False-Positives

Program	# Lines	Helgrind	RT-	RT
aget-0.4.1	1.1K	2/4	4/23	4/4
ctrace-1.2	1.2K	2/2	2/2	2/2
smtprc-2.0.2	3.1K	2/2	2/7	2/2

表中の列 Program は検査対象プログラムとそのバージョンを表し，列# Lines はそのソースコード行数を表す．列 RT は我々の提案手法に基づく競合検出器（RT）の適用結果を示す．RT-は RT から「履歴関連付けの動的詳細化」の実装を除去した競合検出器（RT-）であり，列 RT-はその適用結果を示す．列 Helgrind は Valgrind[10] 付属の競合検出器 Helgrind[18] の適用結果である．セル中の x/y は検出器による競合報告箇所が y 個あり，そのうちの x 個が正しい検出（true positive）であったことを意味する．したがって，$y-x$ が誤検出の数を表す．

aget は軽量な HTTP クライアントであり，ctrace はマルチスレッドプログラムの実行をトレースするデバッグ用ライブラリであり，smtprc は SMTP の中継を検査するプログラムである．これらのプログラムは LOCKSMITH[13] や LP-Race[17] でも検査結果が報告されており，それらは我々の競合検出結果の正しさを判断するための一つの基準となった．

aget-0.4.1 の検査では，ローカルウェブサーバから 101KB の HTML ファイルを 3 個のスレッドで並列ダウンロードした．その処理に対し，検出器 RT-は 23 箇所で競合を報告した．しかし，我々の確認では正しい検出はそのうちの 4 個だけであった（誤検出が 19 個）．この結果からも，Choi らの競合検出法が実用 C プログラムの検査において多量の誤検出を引き起こすことが分かる．原因を調査してみると，2.3.2 節で挙げた例に類似する処理が誤検出の引き金となっていた．具体的には，aget-4.0.1 では，各スレッドがスレッド ID をインデクスとして使用し，共有配列の異なる要素をロックを用いず排他的にアクセスしていた．2.3.2 節での指摘通り，Choi らの手法は配列全体に 1 個のイベント履歴を関連付けるため，各スレッドの各要素への排他的アクセスを配列全体へのアクセスの競合として誤検出してしまった．

一方，我々の履歴関連付けの動的詳細化手法に基づく検出器 RT は，上記の aget-4.0.1 の処理に対し，誤検出を報告しなかった（19 個の誤検出の削減に成功）．更に，合計 4 個の競合は従来通り正しく検出できた．ctrace-1.2 の検査では，付属のサンプルプログラム（`ctrace-1.2/examples/foo`）を実行する処理に対して競合検出を行った．smtprc-2.0.2 の検査では，ローカル SMTP サーバ経由のローカル mbox への中継検査の処理に対して競合検出を行った．その結果，ctrace-1.2 に対しては，RT-と RT ともに誤検出を報告することなく 2 個の競合を検出できた．また，smtprc-2.0.2 に対しては，RT-が 5 個の誤検出と 2 個の正しい検出を報告し，RT が誤検出なしで 2 個の競合を正しく検出した．

以上の結果から，サンプル数が 3 個と少ないものの，我々の履歴関連付けの動的詳細化手法は，(Choi らの手法に代表される）粗粒度の関連付け方式が引き起こす誤検出の低減に有効に機能することが分かった．

処理中” と “シグナル処理中でない” だけを記す．ただし，実際の検出器の実装では，現在処理中のシグナルの番号やスレッド ID や獲得済みのロック集合などの情報も保持する．

5. 関 連 研 究

データ競合の検出を目的として，現在までに多数の手法が提案されている．競合検出法は動的手法と静的手法に大別できる．本節では，それらのうち，実用 C プログラムに適用可能な動的検出法に焦点を当て，我々の手法と比較する．

動的手法は対象プログラムに検査コードを挿入し，プログラムの実行と同時に競合検出を行う．動的手法は一般に，大規模なプログラムの検査を現実的な時間で完了できる反面，網羅的な検査を行う事が難しい．Eraser[15] はバイナリコードに検査コードを挿入し，各メモリ位置の状態をメモリアクセスの属性を入力とする単純なステートマシンで管理しつつ，lockset 解析を行う．Eraser は lockset 解析に基づく初期のスレッド競合検出ツールであり，後の動的検出手法[2),7),9),12),18)] に影響を与えた．Helgrind[18] は Valgrind[10] の付属ツールであり，Eraser と同様，バイナリコードを対象に検査を行う．Helgrind は対象プログラムを解釈実行しながら各メモリ位置の状態管理と lockset 解析を行いスレッド競合を検出する．Helgrind は，スレッドの `fork/join` 操作や条件変数を用いた同期によって強制されるアクセス順序を Lamport の happens-before 関係[8] を用いて解析し，誤検出を低減する．Muhlenfeld らの手法[9] は Helgrind の改良であり，アノテーションと fake segment を用いて条件変数による同期に関する誤検出と検出漏れを低減する．Jannesari らの手法[7] は Helgrind の各メモリ位置の状態管理用ステートマシンを改良して検出精度を向上させる．

上記の手法は全て，各メモリ位置を保護する lockset が一意に決まるという仮定に基づき，各メモリ位置に 1 個の lockset を関連付ける（細粒度の関連付け）．これに対し，我々の手法は粗粒度の関連付けを必要に応じて動的に詳細化する．したがって，関連付けの粒度に起因する（競合検査の）空間的/時間的なオーバヘッドは我々の手法の方が小さい．

一方，Java プログラムを対象とする競合検出法も多数提案されている．Choi らの手法[2] は各メモリオブジェクトに 1 個のイベント履歴を関連付け，各メモリアクセスを記録する．O'Callahan らの手法[12] は Choi らの手法に happens-before 解析を導入し，検出精度を改善している．Elmas らの手法[4] は lockset 解析に基づく検出法であるが，Goldilocks と呼ばれる高度な lockset 解析によって happens-before 解析も同時に実現し誤検出を低減している．

これらの Java プログラムを対象とする手法は，C プログラムに適用しようとすると本研究で指摘した問題に直面する．我々の提案手法はそれらの問題を一連の実装技術によって解決している．

6. 結論と今後の展望

本稿では，Java プログラムを対象とする既存の高精度かつ高効率な競合検出法を C プログラムに適用する際に直面する問題点を明らかにし，それらを解決する一連の実装技術を提示した．我々は（1）各メモリオブジェクトとイベント履歴の関連付けの問題をオブジェクト表ベースの実装で解決し，(2）粗粒度の関連付けに伴う誤検出の問題を履歴関連付けの動的詳細化によって解決し，(3）シグナル処理中の履歴の更新による競合状態の問題を遅延検査によって解決した．

提案手法に基づく競合検出器の実用 C プログラムへの適用実験では，履歴関連付けの動的詳細化が誤検出の低減に有効であることを確認した．

今後の展望として，静的プログラム解析の併用等により競合検出の精度と性能を向上させること，及び，競合検出器を広範な実用 C プログラムに適用し提案手法をより厳密に評価することを計画している．

参 考 文 献

1) Arahori, Y., Gondow, K., Maejima, H.: Dynamic Interrupt-race Detection for C Programs, *IPSJ Journal*, Vol.51, No.9, pp.1816–1831 (2010).
2) Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V. and Sridharan, M.: Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'02)*, New York, NY, USA, ACM Press, pp.258–269 (2002).
3) Cormen, T.H., Leiserson, C.E., Rivest, R. and Stein, C.: Multithreaded Algorithms, *Introduction to Algorithms (third edition)*, MIT Press, pp.772–812 (2009).
4) Elmas, T., Qadeer, S. and Tasiran, S.: Goldilocks: A Race and Transaction-Aware Java Runtime, *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'07)*, New York, NY, USA, ACM Press, pp.245–255 (2007).
5) Engler, D. and Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks, *Proceedings of the 2003 ACM Symposium on Operating Systems Principles*

(SOSP'03), New York, NY, USA, ACM Press, pp.237–252 (2003).
6) Free Software Foundation (FSF): *GCC, the GNU Compiler Collection: http://gcc.gnu.org/*.
7) Jannesari, A., Bao, K., Pankratius, V. and Tichy, W.F.: Helgrind+: An Efficient Dynamic Race Detector, *Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, Los Alamitos, CA, USA, IEEE Computer Society, pp. 1–13 (2009).
8) Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Commun. ACM*, Vol.21, No.7, pp.558–565 (1978).
9) Muhlenfeld, A. and Wotawa, F.: Fault Detection in MultiThreaded C++ Server Applications, *Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, New York, NY, USA, ACM Press, pp.142–153 (2007).
10) Nethercote, N. and Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, pp.89–100 (2007).
11) Netzer, R. H.B. and Miller, B.P.: What Are Race Conditions?: Some Issues and Formalizations, *ACM Lett. Program. Lang. Syst.*, Vol.1, No.1, pp.74–88 (1992).
12) O'Callahan, R. and Choi, J.-D.: Hybrid Dynamic Data Race Detection, *Proceedings of the 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, New York, NY, USA, ACM Press, pp.167–178 (2003).
13) Pratikakis, P., Foster, J. S. and Hicks, M.: LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection, *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'06)*, New York, NY, USA, ACM Press, pp.320–331 (2006).
14) Qadeer, S. and Wu, D.: KISS: Keep It Simple and Sequential, *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'04)*, New York, NY, USA, ACM Press, pp. 14–24 (2004).
15) Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T.E.: Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs, *ACM Trans. Comput. Syst.*, Vol.15, No.4, pp.391–411 (1997).
16) Sleator, D. and Tarjan, R.: Self-adjusting binary search trees, *Journal of the ACM*, Vol.32, No.3, pp.652–686 (1985).
17) Terauchi, T.: Checking Race Freedom via Linear Programming, *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'08)*, New York, NY, USA, ACM Press, pp.1–10 (2008).
18) Valgrind-project.: Helgrind: a data-race detector (2007). *http://valgrind.org/docs/manual/hgmanual*.
19) Voung, J.W., Jhala, R. and Lerner, S.: RELAY: Static Race Detection on Millions of Lines of Code, *Proceedings of the 2007 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'07)*, New York, NY, USA, ACM Press, pp.205–214 (2007).