GRAMMATECH

AIC INDUSTRIAL SOFTWARE

マルチコア・マルチスレッド環境での静的解析ツールの応用

米GrammaTech社CodeSonarによるスレッド間のデータ競合の検出

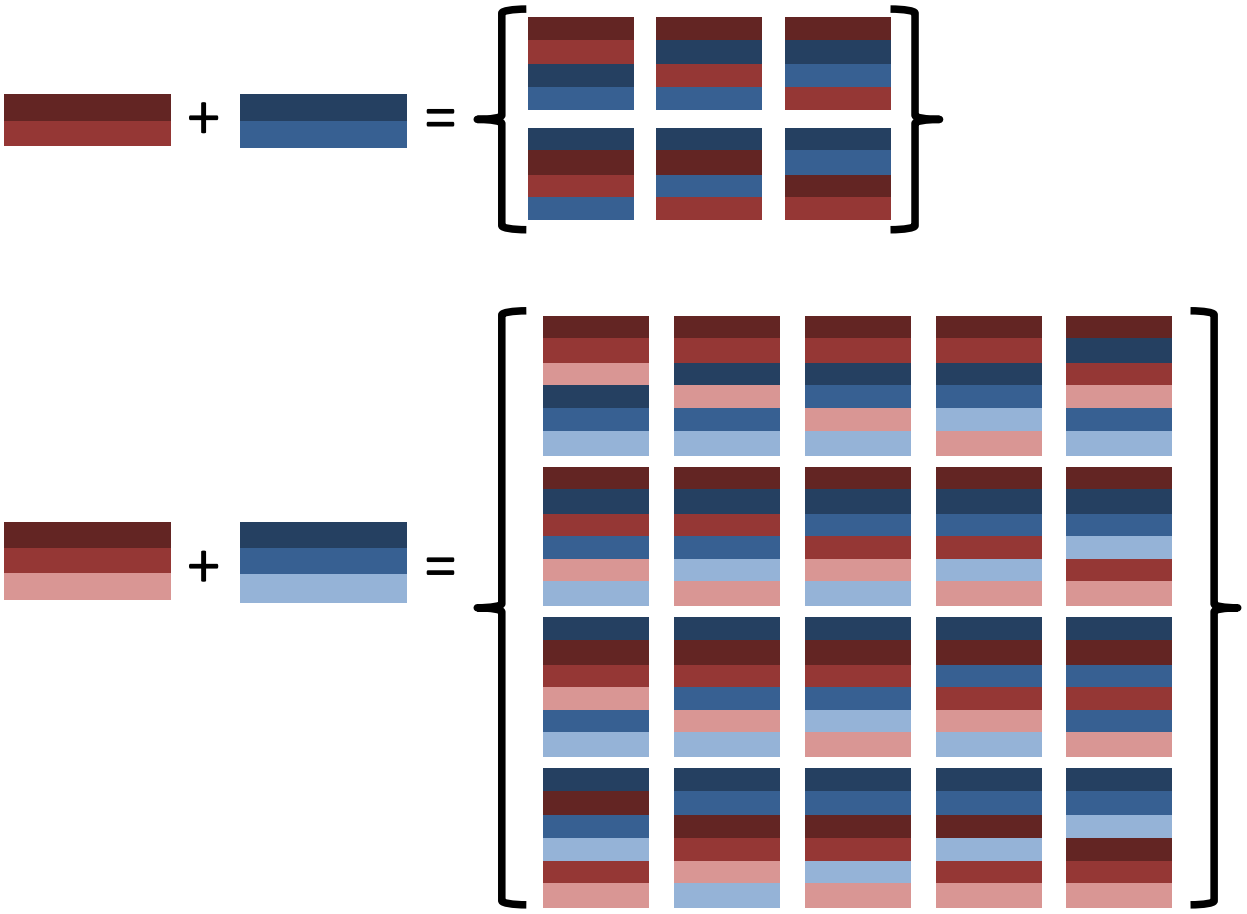
Agenda

- 並列実行に起因する不具合の摘出
  - なぜ、並列実行されるプログラミングは難しいのか
    - データの競合
    - デッドロック
  - どのようにして静的解析ツールで並列実行の問題を見つけるのか?
- コーディングの例
  - 遅延初期化(Lazy Initialization)
- まとめ

トレンド:シングルからマルチコアへ

- 伝統的な、性能向上のアプローチはそろそろおしまい
  - クロック周波数の上昇、メモリアクセスの高速化、そして賢い命令のスケジュールによる最適化は、もはや時代遅れ
- 開発チームはマルチコアプロセッサを使用したがっている
  - よりよいパフォーマンスへの近道
  - 消費電力でもメリット
- しかし、実際の適用には大きな障害があります
  - アプリケーションソフトウェアは、マルチコアを利用するために変更が必要
    - マルチコアプロセッサを導入しただけでは、もっている性能はフルに発揮できません
  - そして、マルチコアプログラミングは難しい

マルチコアはソフトウェア開発を複雑にする



2つの命令を持つ、2つのスレッドでは、組み合わせは6つ
3つの命令では、20の組み合わせとなります

現実世界でのスケジューリング

- 現実世界のソフトウェアは、前のスライドの例よりも、非常に多くの命令で構成されています
- インターリーブは、スケジューラーにより決定されます
  - 全てのパターンのインターリーブをテストすることは不可能です
  - 決められたパターンのインターリーブでさえも、テストすることは難しいです
- システムのスケジューリングは非常に複雑です
  - 決定することは難しい
  - 指定されたテストケースでも、多くの異なる振る舞いをします
    - 引数が同じ場合でも、同じ動作をするとは限らない
- 実行の正しさは、スケジューリングに依存します
  - 競合条件は、並列ソフトウェアの悩みの種

データの競合

- データの競合は以下の時に発生します:
  1. 複数の実行中のスレッドが共有しているデータをアクセスする場合
  2. 少なくともひとつのスレッドがデータの値を変更する場合
  3. アクセスが明確な同期を使用していない場合
- データの競合は、システムを矛盾した状態にします
- また、データの競合は、検出されることが難しく、極めて不可解な、稀な(困った)状況を引き起こします

データの競合の例

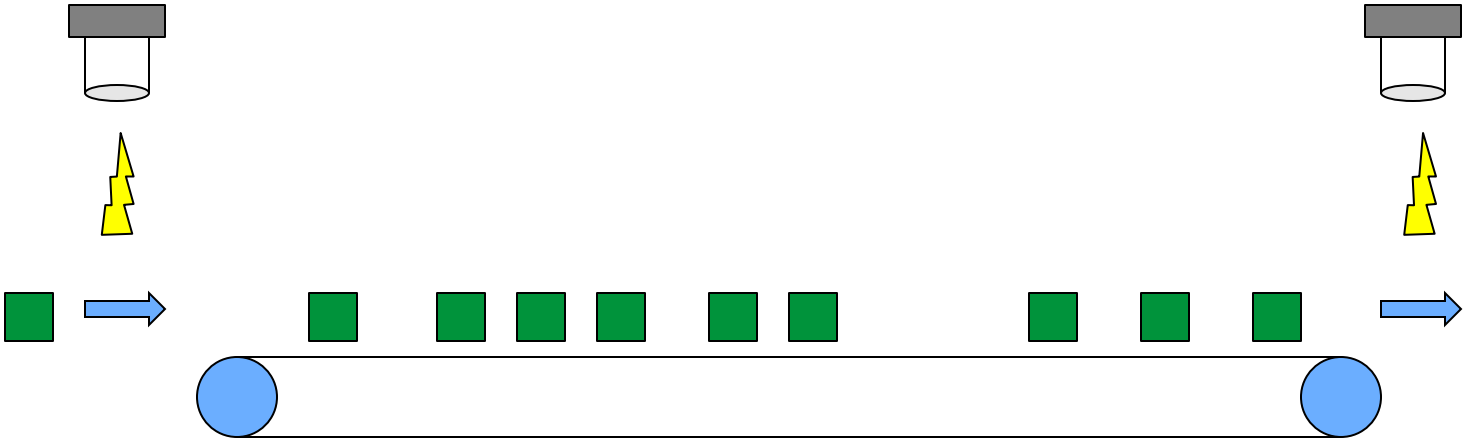
Thread 1

```
count := count + 1;
```

count: 1 9 0

Thread 2

```
count := count - 1;
```



```
count := count + 1;
```

メモリからレジスタに**転送**して
レジスタに1を**加算**して
レジスタからメモリに**書き戻す**

データの競合

Instruction	Register	Memory
load	**9**	9
increment	**10**	9
store	10	**10**
wait….		
load	**10**	10
decrement	**9**	10
store	9	**9**
wait….		
load	**9**	9
increment	**10**	9
load	**9**	9
decrement	**8**	9
store	10	**10**
store	8	**8**

インターリーブされていない命令では正しい結果が得られます

インターリーブされた命令では、誤った結果となる

…

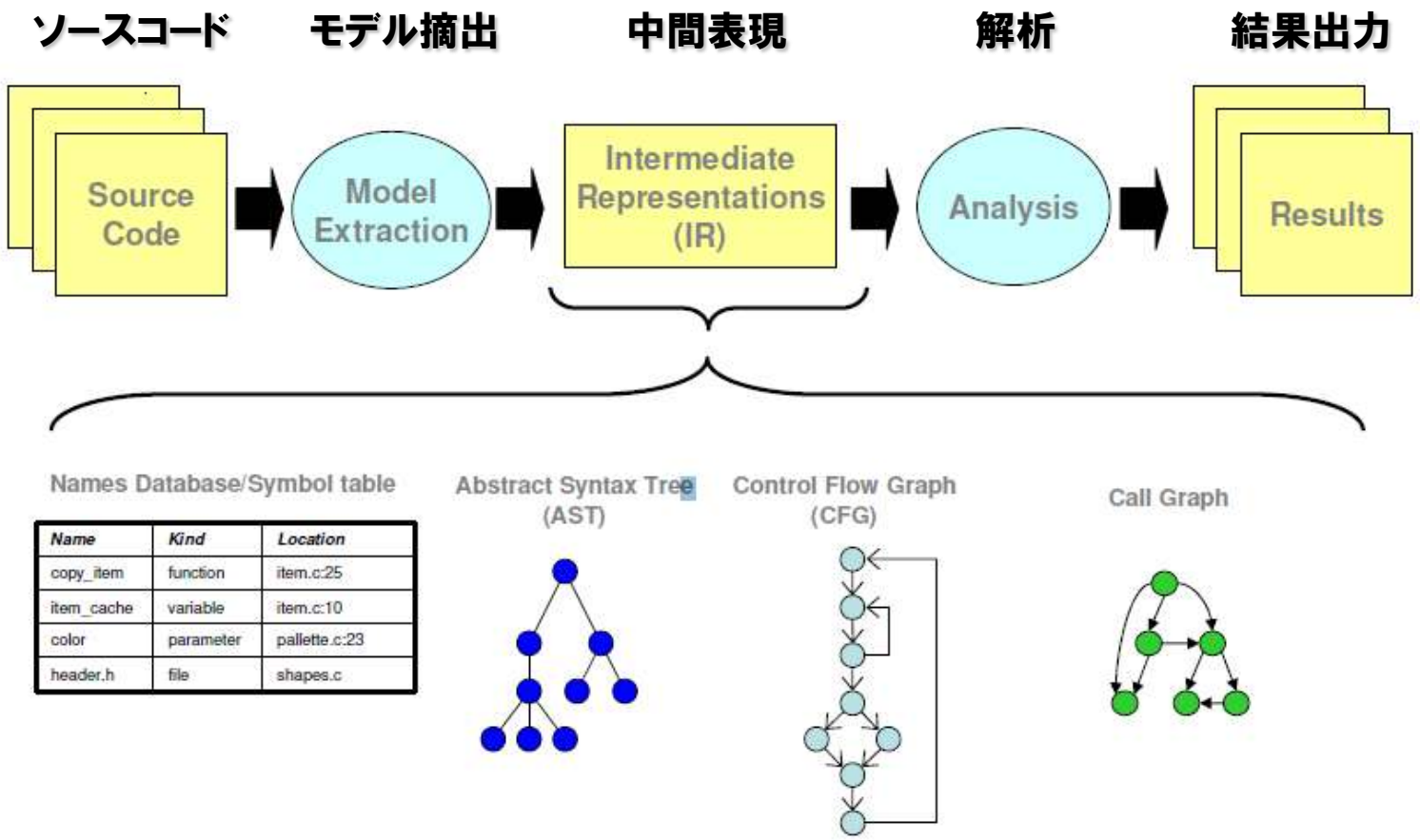
なぜデータの競合はデバッグが難しいのか

- 稀にしか発生しないということは、テスト中に発見するチャンスが少ないということ
- 診断は難しい
  - › 症状と原因が大きく離れていることがしばしばある
- 再現させることが最大の問題
  - › 結果が間違った実行のスケジュール再生が難しい
  - › スケジューリングは環境に敏感
    - デバッガの使用、コンパイラの最適化、I/O回数の違い、外部環境の違いで、スケジューリングは変化します
- 開発者はコードを読むときには、ひとつのスレッドしか一度に読むことができない
  - › スレッドの相互作用の影響は簡単に見落とされる

GrammaTech CodeSonar

- 重大な一般的な不具合とセキュリティの脆弱性をみつけます
  - プログラム全体でのパス構造解析での、シンボリック実行を使用
- どのように使用するか:
  - コードやビルドスクリプトの変更は必要なし
  - プログラムは通常通りビルドされ、CodeSonarがコンパイルコマンドラインを監視
  - ビルドの一方で、CodeSonarはコードを解析し、不具合を見つけます
  - 見つけた不具合はお客様のマシン上のデータベースに集められます
  - ユーザーのログもデータベースに収集されます
  - 解析のプロセスは全体で自動化可能です(例:毎日、夜に実行)

CodeSonarのコード解析原理



Name	Kind	Location
copy_item	function	item.c:25
item_cache	variable	item.c:10
color	parameter	pallette.c:23
header.h	file	shapes.c

CodeSonarはどのようにしてデータの競合を見つけるのか

- CodeSonarは症状ではなく、原因にフォーカスする
  - › コードを検証してどのスレッドでどのロックを使用しているかの情報を持つアブストラクトモデルを生成します。
  - › 自動的に、可能性のあるインターリーブを調査し、共有メモリへのアクセスパターンを検証します
  - › コードのどこに、問題を含むインターリーブがあるかを見つけます
- データの競合が認められた場合、CodeSonarは、関係する情報とともに、Data Race ウォーニングを通知します
- また、CodeSonarは同期構造により発生する問題も検出します
  - › 例: デッドロック、ロックの誤り

デッドロックする銀行処理の例

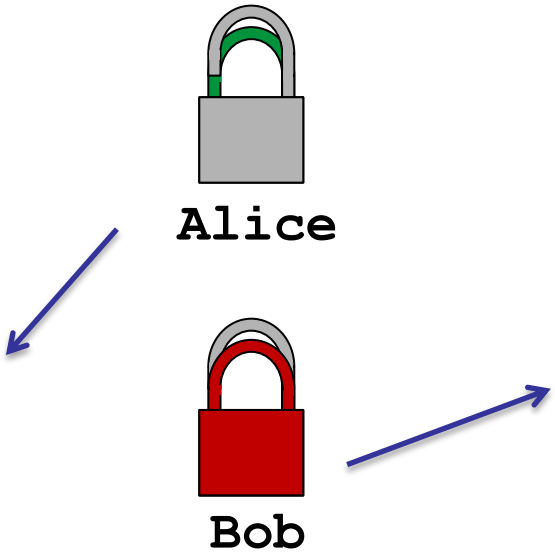
- アカウントXからアカウントYへ送金する手順
  - アカウントXをロック
  - アカウントYをロック
  - 指定金額をアカウントXから引き、Yに加算
  - アカウントYをアンロック
  - アカウントXをアンロック
- 2つのトランザクションを検討してみます
  - $100をアリスからボブへ送金
  - $50をボブからアリスへ送金

デッドロック

Thread 1

****
****
****
acquire lock for **`Alice`**
****
****
****
acquire lock for **`Bob`**

waiting on Thread 2



Thread 2

****
****
****
acquire lock for **`Bob`**
****
****
****
acquire lock for **`Alice`**

waiting on Thread 1

CodeSonar はどのようにしてデッドロックを見つけるのか

- リソースの順序を確認
  - Dijkstraが1965年に発表した「食事する哲学者たち」の問題
- Conflicting Lock Order ウォーニングは、異なるスレッドで、同じロックが異なる順序で発行されると、検出する
- CodeSonarは、さらにネストされたロックのチェックも行います
  - 複数のロックを使用している場合にチェックが行われます
  - 同期問題の難しさを強調します

コードの例

- **遅延初期化(Lazy Initialization)**

遅延初期化(Lazy Initialization) – コード例

```
float *singleton( void )
{
  static int is_initialized = 0;
  static float *actual_data = NULL;

  if( !is_initialized )
  {
    pthread_mutex_lock( &m );
    if( !is_initialized )
    {
      actual_data =
        (float *)malloc( sizeof( float ) );
      *actual_data = 4.2;
      is_initialized = 1;
    }
    pthread_mutex_unlock( &m );
  }
  return actual_data;
}
```

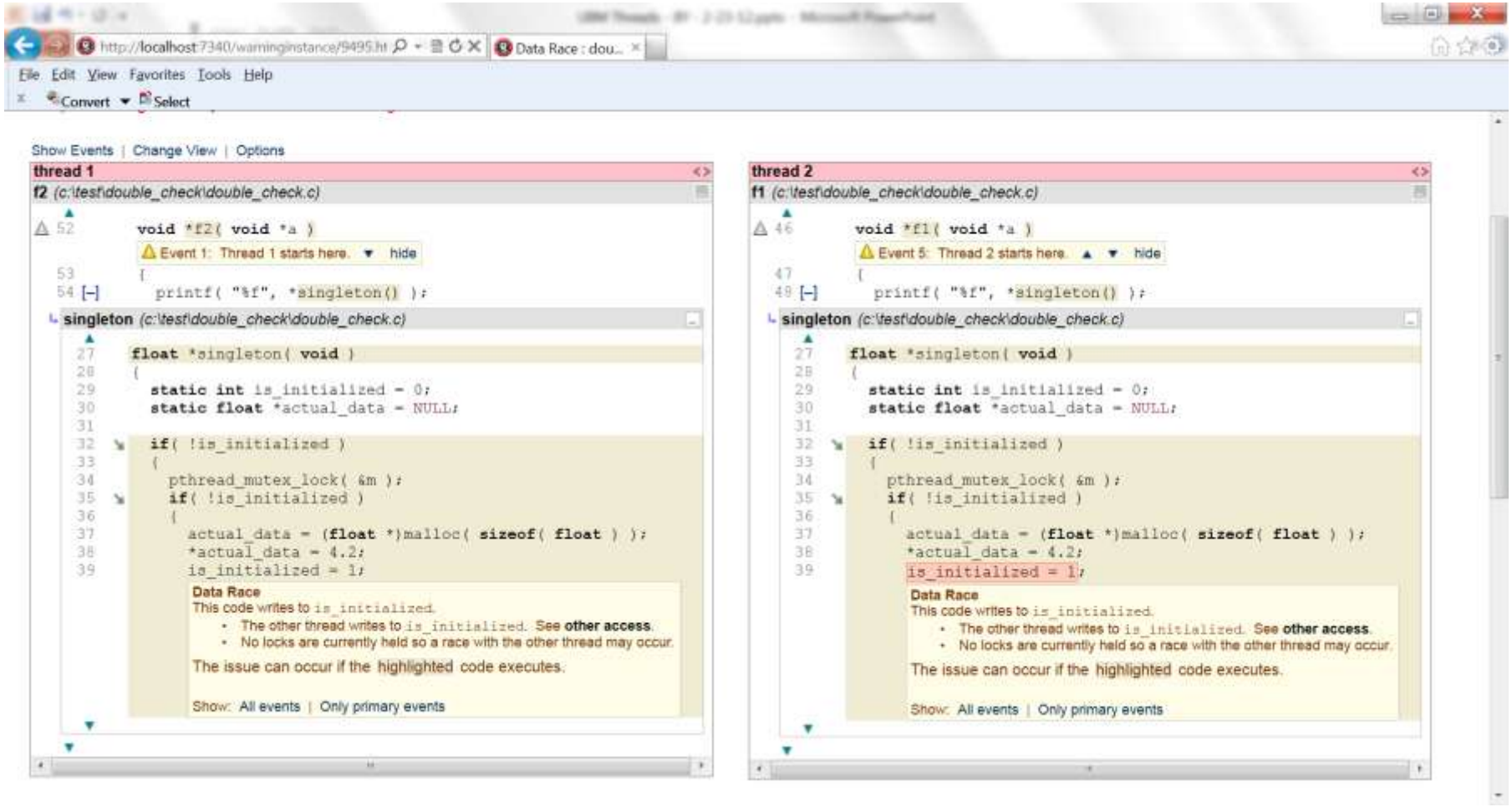
```
void *f1( void *a )
{
  printf( "%f", *singleton() );
  return a;
}

void *f2( void *a )
{
  printf( "%f", *singleton() );
  return a;
}
```

遅延初期化(Lazy Initialization) – 単一の関数

遅延初期化(Lazy Initialization) – 単一の関数

遅延初期化(Lazy Initialization) – 並べて比較



GRAMMATECH

INDUSTRIAL SOFTWARE

まとめ

- マルチコアの開発は避けられない
- 同時実行に伴うエラーは潜んでいる
  - 再現、診断、除去は難しい
  - 見たところ問題がないと思われるデータの競合でさえ、場合によっては突然悪い結果をもたらすことがある
- 静的解析ツールで、不具合の検出を助けます

CodeSonar の先進的な静的解析

- プログラム全体の解析 (モジュール間も含む)
- 非常に大規模なコードでの使用が可能なスケーラビリティー
- 重要な不具合の検出
  - バッファオーバラン
  - Nullポインタの参照
  - 0での割り算
  - 初期化されていない変数
  - ヒープ以外の変数の解放
  - 解放後メモリの使用
  - 競合条件
  - メモリ/リソースのリーク
  - ユーザー定義のチェック
  - その他多くの不具合

問い合わせ窓口

GrammaTech 日本総代理店

エーアイコーポレーション

http://www.aicp.co.jp

※お問い合わせ、資料請求、評価版申し込みはWebの
「お問い合わせ・資料申込」のフォームをご利用ください