Boundary Checking Optimization についてのサーベイ

cocoa

September 17, 2001

1 Integer Programming についてのサーベイ

integer programming とは整数係数の等式 ($\sum a_i x_i = 0$) と不等式 ($\sum a_i x_i \geq 0$) の系を扱う分野を意味している。integer programming ではある系の解を求めることが問題である場合と、ある関数を最大にする解を求める場合がある。整数制約系の解を求めるアルゴリズムは NP-complete であることが知られている。boundary checking のために integer programming を利用する場合は必ずしも解を求める必要はない。配列の添字が配列のサイズの範囲内であることを証明できれば充分である。

1.1 Fourier-Motzkin elimination [5]

この方式は変数を一つずつ消去することで制約解を求める。変数が一つの時は制約を満たす解を簡単に求めることができる。ある変数 z の上限と下限が次のように制約されていたとする。

$$az \leq e_u$$
$$e_l \leq bz$$

この時に $ae_l \leq be_u$ という制約を追加し、z に関する制約を削除する。

この方式では変数を削除するごとに制約の数が $O(n^2)$ 個増加する。z の上限と下限がそれぞれ n, m 個あったとすると新しく nm 個の制約が加えられるためである。

このアルゴリズムは古い (1973) が DTAL でもこれを使っているので、気にしなくていいのかも。

1.2 Simple fixed point iteration [12]

この方式は変数の値を繰り返し計算することで解を求める。変数の上限と下限を一つ前の変数の値を利用して計算し、それを新しい上限と下限とする。例えば、次のような制約があり、ある時点で y の下限が 4 であることがわかったとすると

$$y + 1 \leq x$$

次の iteration では x の下限は 5 となる。これを fixed point に達するまで繰り返す。このアルゴリズムは変数の依存関係に循環があると止まらない可能性がある。例えば $x + 1 \leq y, y \leq x$ という制約があるとまずい。そのため制約に循環がある時を特別に扱う必要がある。(この場合 x は ∞ になる。)

2 Partial Redundancy Elimination についてのサーベイ

partial redundancy elimination [9] とはループ不変式の移動、共通部分式の削除などを包括するような最適化である。まず言葉の定義を行う。

式 e が **program point** p において **available** である 式 e の計算を control flow 上、p 以前の program point (p を含む) で既に行っており、かつ p で e を評価しても同じ値になること。

式 e が **program point** p において **anticipated** である control flow 上、p 以後に式 e の計算が行われ、かつ p で e を評価しても同じ値になること。

式 e の計算が **program point** p で **redundant** である p に到達するすべての control flow 上で e が available であること。

式 e の計算が **program point** p で **partially redundant** である p に到達する少なくとも一つの control flow 上で e が available であること。

redundant な計算は単純に消してしまって以前の結果を再利用することができる。一方、partially redundant な計算は単純に削除することはできない。なぜならその値を計算していない control path があるからである。したがって削除する代わりに、現在の地点より前で、その式が anticipated である地点にその計算を移動させる。これを partial redundancy elimination と呼ぶ。

redundancy elimination と partial redundancy elimination は実行中の再計算を減らすことができる、しかし以前の計算の結果を保存するために register pressure を高めることになるので register の少ないアーキテクチャでは再計算してしまった方がコストが安く付くこともある。この観点から、できるだけ register pressure が少なくなるように式を移動させる方式を研究したのが [6] である。

そこで述べられているアルゴリズムについて解説する。まず定義を行う。

$$\mathrm{used}(n,t) = \text{node } n \text{ において } t \text{ または } t \text{ の subterm が参照されていること}$$

$$\mathrm{transp}(n,t) = \text{node } n \text{ において } t \text{ または } t \text{ の subterm が代入されていないこと}$$

$$\text{d-safe}(n,t) = \begin{cases} \text{false} & \text{if } n = e \\ \mathrm{used}(n,t) \vee \mathrm{transp}(n,t) \wedge \prod_{m \in \mathrm{succ}(n)} \text{d-safe}(m,t) & \end{cases}$$

$$\mathrm{earliest}(n,t) = \begin{cases} \text{true} & \text{if } n = s \\ \sum_{m \in \mathrm{pred}(n)} \neg\mathrm{transp}(m,t) \vee \neg\text{d-safe}(m,t) \wedge \mathrm{earliest}(m,t) & \end{cases}$$

ただし s と e はそれぞれ関数の entry point と exit point を意味する。

直観的には d-safe とは、ある値の live range を表している。つまりそれは定義されており、かつ以後必ず使われる値を表している。したがって d-safe な node に t の定義を移動させても semantics は変わらず、本来するはずのなかった control flow 上で余計な計算を行うこともない。

earliest とは s と現在の node との間に d-safe ではない node が存在する control flow があることを意味している。つまりある node が earliest であればそれより以前に計算を移動させることができない。$\neg\mathrm{transp}(m,t)$ という条件が付いているのは定義を挟んで計算を移動できないからである。

さて最初に述べた partial redundancy elimination のアルゴリズムは「d-safe かつ earliest な node に計算を移動させる」という定式化を行うことができる。このアルゴリズムは、計算を移動させることによって本来するはずのなかった計算を行うことはないという意味で無駄はないが、計算をできるだけ外側に移動させるためレジスタに対する pressure を増大させる。そのため [6] で提案されているアルゴリズムでは、直観的には、それ以上計算を内側にいれると無駄が生じてしまうような限度までしか計算を外側に移動させない。

いくつか定義を行う。

$$\mathrm{delay}(n,t) = \text{d-safe}(n,t) \wedge \mathrm{earliest}(n,t) \vee \begin{cases} \text{false} & \text{if } n = s \\ \prod_{m \in \mathrm{pred}(n)} \neg\mathrm{used}(m,t) \wedge \mathrm{delay}(m,t) & \text{otherwise} \end{cases}$$

直観的には delay とは、まず naive な partial redundancy elimination を行った時に、s と n の間に計算を行う node が存在し、かつ値はまだ使われていない状況を示している。

delay である node の中で最も後にあるものを latest と呼ぶ。

$$\mathrm{latest}(n,t) = \mathrm{delay}(n,t) \wedge (\mathrm{used}(n,t) \vee \neg \prod_{m \in \mathrm{succ}(n)} \mathrm{delay}(m,t))$$

latest な場所まで計算を移動させれば良い、というのがこの論文 [6] で提案するアルゴリズムである。

3 Static Detection

3.1 A first step towards automated detection of buffer overrun vulnerabilities [12]

CERT advisories によると buffer overrun が原因による攻撃は全体の半数におよぶという。この手法は scalability を重視し、precision を犠牲にする。その結果、安全に見積もることもできず false alarm も多発する。解析は整数区間ベースの抽象解釈である。sendmail を解析するのに Pentium III (クロックは不明) で 15 分かかっている。コメントを除いて 10000 行前後の sendmail を解析するのに Pentium III で 15 分もかかるということはこれよりも精度の高い解析は現実のアプリケーションに対して使えないような印象を受ける。いわゆる model checking の手法では sendmail クラスのアプリを扱えないということだろうか。この解析アルゴリズムは Aiken の BANE を使って実装されているのだが、他の論文から受ける印象では BANE はどうもかなり遅いように感じる。例えば pointer analysis を実装した論文を読むと、他の論文と比べて圧倒的に遅く、メモリ使用量も多すぎる。

3.2 Symbolic bounds analysis of pointers, array indices, and accessed memory regions [10]

プログラムから変数の上限と下限に関する線形の制約を生成し、解く。それだけである。

3.3 Safety checking of machine code [14]

binary code を直接安全かどうかチェックできる。抽象解釈に基づいて命令を仮想実行する。再帰構造などを使うとポインタは一点に潰れてしまう。使うには次のような条件がある。

- 引数や参照するメモリについて型情報が分かっている必要がある。
- メモリや配列を使うと正確さが落ちる。
- subtyping を扱えない。
- setjmp/longjmp, signal, varargs とかについて書いていないがたぶん無理。
- また local に array があると out-of-bounds をチェックできない。
- ポインタに対して xor や bit mask などを使ってはいけない。

解析できたのは 9 割が 100 命令以下の関数。最大は 883 命令。null かもしれないポインタへのアクセス、out-of-bounds access を検出できた、と書いてあるが false alarm については書いていない。

4 Fully Dynamic Checking

4.1 Efficient detection of all pointer and array access errors [1, 2]

いわゆる Safe C の論文である。既に解説されているので省略。

5 Boundary Check Elimination

5.1 Elimination of redundant array subscript range checks [7]

この研究は Fortran を対象にしている。そのため、プログラムの解析は容易であり、ループの構造が見積もりやすいことを考慮する必要がある。この研究では次のような変換を行う。

```
int A[5..10];
if ( n * 2 < 5 ) TRAP;
if ( n * 2 > 10 ) TRAP;
A[n * 2] = 0;
if ( n * 2 - 1 < 5 ) TRAP;
if ( n * 2 - 1 > 10 ) TRAP;
A[n * 2 - 1] = 1;
```

(a)

```
if ( n * 2 < 5 ) TRAP;
if ( n * 2 > 10 ) TRAP;
A[n * 2] = 0;
if ( n * 2 - 1 < 5 ) TRAP;
A[n * 2 - 1] = 1;
```

(b)

```
if ( n * 2 - 1 < 5 ) TRAP;
if ( n * 2 > 10 ) TRAP;
A[n * 2] = 0;
A[n * 2 - 1] = 1;
```

(c)

```
int A[1..10];
for ( i = 1; i < n*2; i++ ) {
    if ( i > 10 ) TRAP;
    ... A[i] ...
}
```

(A)

```
if (1<=n*2&&n*2>10) TRAP;
for ( i = 1; i < n*2; i++ ) {
    if ( i > 10 ) TRAP;
    ... A[i] ...
}
```

(B)

```
if (1<=n*2&&n*2>10) TRAP;
for ( i = 1; i < n*2; i++ ) {
    ... A[i] ...
}
```

(C)

この研究が使っているテクニックは以下のものである。

check motion check を partial redundancy elimination の手法により移動する。(b) → (c) で使用。

check strengthening 二つの check があり、片方が成り立てばもう片方も成り立つ時、後者を消去する。(a) → (b), (b) → (c), (B) → (C) で使用。

preheader insertion check の式が induction variable の線形の式で表すことができて、ループがすべて実行され、配列も必ず参照されるときに、check をループの外に移動する。(A) → (B) で使用。

科学技術計算ベンチマークを対象とした実験の結果は驚異的であり、61% から 99% の check (dynamic count) を除去することに成功している。

5.2 Optimizing bounds checking in Java programs [8]

この論文では 3 つの仮定をしている。

- Java の semantics を保つために正確な exception を実装する。
- 配列の添字はループ変数の線形結合として書ける。
- do-loop に限定する。上限と下限はループの最中に変わらない。すべての iteration は必ず実行される。

この論文では 4 種類の最適化法を述べている。基本となっているアイデアは次のようなものである。

```
for ( i = l; i <= u; i++ ) { ... A[i] ... B[j] ... }
```

というループがあった時に配列 A の上限と下限をそれぞれ up(A), lo(A) と置くと、次のようなコードに変換を行う。

```
L = max(l,lo(A));
U = min(u,up(A));
for ( i = l; i < L; i++ ) { ... raise exception ... check(j); B[j] ... }
for ( ; i <= U; i++ ) { ... A[i] ... check(j); B[j] ... }
for ( ; i <= u; i++ ) { ... raise exception ... check(j); B[j] ... }
```

もちろん for 文が一度も実行されないこともありえるし、それが静的に分かれば除去してもよい。また配列への参照は if 文の内側にあることもあるので、out of boundary access であることが分かってもループの body を実行しなくてよい訳ではない。

さてこの方式はループを 3 つのループに分割している。ループがネストしていた場合の扱い方に応じてこの論文では 2 通りの方法を提案している。

general method すべてのループを 3 つに分割する。n 段ループでは 3^n 個のループが生成される。
compact method 真中のループ (no check を含むもの) だけ 3 つに分割していく。n 段ループでは $2n+1$ 個のループが生成される。

また配列の添字は必ずしも静的には分からない。その場合のために次のような方法を提案している。まず n 次元配列に対して $n-1$ 次元の boolean 値の配列を用意する。この配列は innermost loop をチェックなして実行できるかどうか指示している。まずループ本体を実行する前にループの構造だけを実行し、動的に innermost loop を実行可能か調べ、boolean 配列を初期化する。ループ本体はチェックありバージョンとチェックなしバージョンに分かれており、boolean 配列の値に応じて適切な方を実行する。

この方式はループ内部で副作用のある式を実行していると適用できない場合がある。またループの構造を静的に決定できるときは boolean 配列を静的に構築できることもある。

array access intensive なプログラムを対象にテストした結果、通常の Java に対する速さは 3.7 倍から 14 倍となる。また Java で check code を生成しないバージョンに対するオーバヘッドは 1.08 % から 17% である。ちなみに Java 風に書いた C (具体的には良く分からん) に対するオーバヘッドは −14 % から 45% である。

5.3 ABCD: Eliminating array bounds checks on demand [3]

この論文の手法は [7] と比べて除去できる boundary check のクラスが特に変わるわけではないが、制約の表現法がとても賢い。

この手法では SSA (Static Single Assignment Form) [4] を拡張したプログラムの表現を使っている。そのため SSA について簡単に説明する。SSA とは RTL みたいなプログラムの表現であるが、同じ変数に対する代入が一度しかないように制限されている。それではループの中で同じ変数を increment する場合はどうするかというとデータフローがマージしていることを明示的に表す疑似命令を使う。たとえばあるループの body に入ってくる control flow が二つあるとする (initialization, iteration)。このときにループの body に次のような疑似命令を挿入しなければならない。

$$i_2 = \phi(i_0, i_1)$$

これを ϕ-関数と呼ぶ。ϕ-関数の引数の数は、入ってくる control flow の数を表している。引数は対応する変数を表している。この命令は、もし i_0, i_1, i_2 が同じレジスタに割り当てられたとすると実行する命令を持たない。

では SSA にすると何がうれしいのだろうか。それは、プログラムを flow insensitive に扱うことができるようになる点である。たとえばある変数 x を inlining できるかどうか知りたいとしよう。普通ではまずデータフロー解析をして x に到達可能な x の定義を調べる必要がある。しかし SSA ならば $x = 5$ という文があれば間違いなく x の値は 5 である。このような理由から SSA はさまざまな最適化アルゴリズムや verification などに利用されている。

この論文では if 文などから得られる情報を活用するために SSA を拡張している。

```
if ( i1 <= 10 ) {
```

```
if ( i1 <= 10 ) {
    i2 = pi(i1);
```

```
}
else {                              else {
                                        i3 = pi(i1);

}                                   }
                                    i4 = phi(i2,i3);
```

左側のような if 文があるとそれを右側のように書き換える。pi は ϕ-関数のような疑似命令である。このように書き換えることで変数が使われる場所を考慮することなしに $i_2 \leq 10, i_3 > 10$ を仮定することができる。

この論文での boundary check を消去する方法を説明する。boundary check を消去するためには変数の上限と下限を求めればよい。ここでは上限を求める方法だけ説明する。下限はその dual である。プログラムを拡張された SSA に変換した後で、変数同士の制約を重みつき有向グラフで表現する。グラフの node は変数、重みは変数の差分、グラフの edge は代入の方向を表している。この有向グラフは変数の上限を求めるために使われる。プログラムから有向グラフを生成するには次のような規則を利用する。

statement	constraint	edge
$v := c$	$v \leq c$	$c \xrightarrow{0} v$
$v := w + c$	$v \leq w + c$	$w \xrightarrow{c} v$
if $v \leq c$ then $w = \pi(v)$	$w \leq \min(v, c)$	$c \xrightarrow{0} v, w \xrightarrow{0} v$
$v := \phi(w, u)$	$v \leq \max(w, u)$	$w \xrightarrow{0} v, u \xrightarrow{0} v$

この規則によって生成されるグラフでは path の重みと最初の値を加えたものが上限を表している。まず、単純な代入文に関しては、自明である。SSA になっているので同じ変数に対する代入は一つしかない。問題は重みの合計が負になる閉路がグラフに存在する場合である。重みが負の閉路が意味しているのはプログラムにループが存在して、そのループを実行するごとに、変数の上限が低下していく状況を表している。これが問題を起こさない理由はループが作られるときには必ず ϕ-node が作られるからである。ϕ-node に関しては上限の max を取るように定義されており、したがって上限に関する情報が劣化しないようになっている。また重みの合計が正である閉路はその中に min が挟まっていない限り上限を発散させる。(したがって bounds check を除去できない。)

この制約グラフを満たす可能なすべての解に対して配列の index がサイズよりも小さいと証明できるのならば bounds check を除去することができる。解を求めるアルゴリズムは exhaustive なアルゴリズムを使っても求めることができるが、この論文では変数から有向グラフを逆方向に depth first search をすることで求めている。この方式では同じ node を何度もたどる必要が生じるため最悪の計算量を増大させるが、practical にはよい結果を出すという。ちなみに、タイトルの on demand はここで exhaustive にやらないことを意味している。

有効性は SPECjvm98 を含むアプリを対象にして dynamic count で 45% の check instruction の削除に成功している。

5.4 Dependent types [11, 13]

値に依存する型とはサイズ付きの配列型 (int[10], float[32])。関数の型に配列のアクセスに関する制約を付ける。もしそれが満たされていることを静的に証明できるなら、boundary check を行わない。そうでないならば runtime check を挿入する。というアイデアである。

参考文献

[1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of all Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of all Pointer and Array Access Errors. *SIGPLAN Notices*, 29(6):290–301, 1994.

[3] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, June 2000.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[6] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy Code Motion. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 224–234, June 1992.

[7] Priyadarshan Kolte and Michael Wolfe. Elimination of Redundant Array Subscript Range Checks. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[8] S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing Array Reference Checking in Java Programs. *IBM Systems Journal*, 37(3):409–453, 1998.

[9] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[10] Radu Rugina and Martin Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[11] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A Type System for Certified Binaries. Technical report, Yale University, 2001.

[12] D. Wegner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed Systems Security Symposium NDSS*, February 2000.

[13] Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[14] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety Checking of Machine Code. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.