**算法基础** Thomas Würthinger

(Array Bounds Check Elimination for the Java HotSpotTM Client Compiler)

我们的数组边界检查消除算法，通过索引变量是否在给定索引的条件范围内决定是否消除。我们保持这种条件尽可能简单而不会显著降低平均Java程序中消除的检查数。与其他方法相比，我们没有构建不等式图。相反，算法保留每条指令x下面形式的约束条件：

*ilower* + *clower <*= *x <*= *iupper* + *cupper*

变量*ilower* 和*iupper*r关于HIR指令变量，而*clower*和*cupper*是整数常量。如果指令的边界条件是部分缺少的，则边界是编译时期常数。起初对边界条件是一无所知，但指令是有界限。

MIN *<*= *x <*= MAX

其中MIN和MAX表示最小和最大32位值。当两个以上变量的边界是已知的，算法试图计算它们的结合。例如，如果边界包含的值是编译时期常数，只会保存一部分信息。当算法知道x小于或等于对于a，同时小于或等于b，它只能保存一个这两个条件中的一个。这是信息的损失，但大多数情况下的实际简化。

1. **完全冗余检查**

我们的算法受益于HIR的SSA形式，其中每个变量仅在程序定义一次，其后不会改变。即使方法是SSA形式，变量范围条件在不同的方法上是不相同。控制流指令，如：虽然if不修改的值操作数，但边界(Bound)在后续的基本块是不同的。

该算法**以支配者为基础处理blocks并维护每个变量的条件堆栈bound，**其中最上面的堆栈元素是当前有效条件。block的支配者是总是在被执行的block之前执行。范围bound在一个支配者中持有的也在block本身中持有。它只能变得更强。因此，该算法使用支配者树的预遍历。当block处理时，**变量的条件为父block中变量的所有条件支配树联合**。使用这种方法，算法可以避免构建扩展SSA表单。如Listing1：

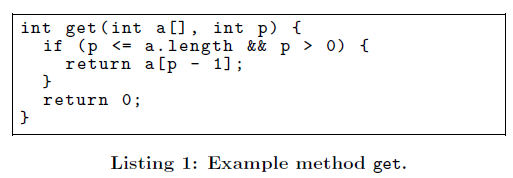


Figure2展示了Java方法的HIR如Listing1所示。该方法分为四个部分。block1的if指令检查是否p＜＝a.length，并且在block2中的if指令检查是否p＞0。如果两个条件都成立，数组加载在block3中执行并且返回该值。如果条件不成立，返回0。

在本例中，支配者树的根是block1,因为该块是该方法的开始。block2是block1的子块以及block3的父块。这个边界检查消除算法与处理block的编号顺序相同。

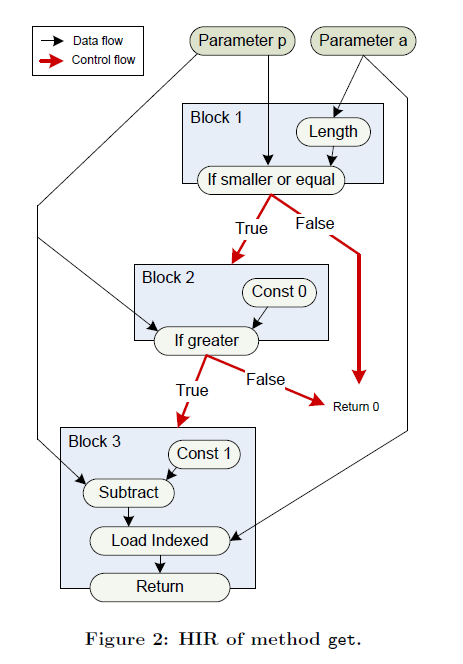
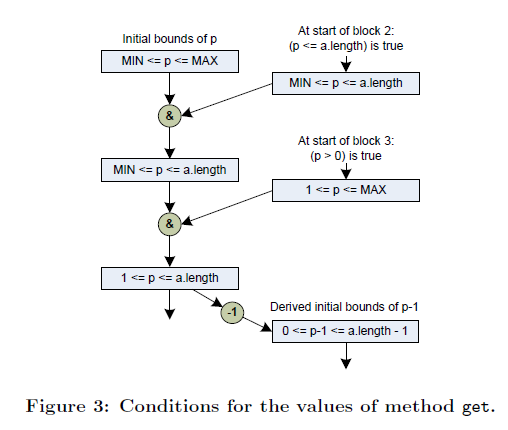


Figure3显示了表达式p的条件和p-1。在每一个if指令中，都有新的条件对于受影响的表达式使用和操作获得的条件。如果是双操作数加法或减法之类的运算操作数是常量，条件会修改以反映这个操作。例如，在Figure3中通过从中减去1，p被转换为p-1的条件下界和上界的常数部分。

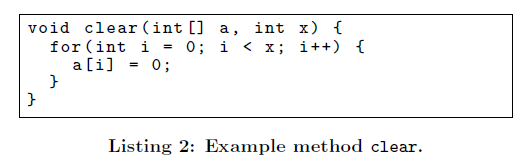


**使用索引表达式p-1的条件，它可以证明索引始终在有效范围内。这个下界为正，上界小于数组长度，因此数组边界检查是完全的冗余的并且可以省略。**

访问队列array的block不能消除边界检查。当从Java字节码构建基本块时，这样的指令会添加check标志，分析之前从支配者树中删除不需要处理的这类块。

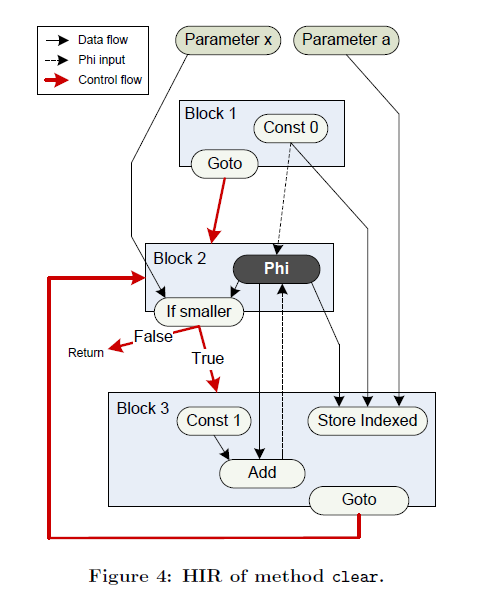
1. **Loop-Invariant Checks（不动点检查）**

Java中的完全冗余检查数程序并不高，在访问array前，会要求数组将长度与数组索引进行比较。相反，清单2中显示的方法clear使用频繁，在循环中并未显式检查数组长度。



循环变量用作数组索引,数组和变量x在循环中没有改变，即它们是**循环不变量**。在array访问点变量i的上界是已知的循环条件i<=x-1。可以推断下限从i的起始值为0并且i仅为增加。

图4显示了该方法的HIR。重要的证明i只增加的部分是phi函数。Phi函数在SSA形式中是必要的，以合并不同的控制流加入时相同变量的值。在里面例如，phi函数合并i的值来自两个前置block1和3。



当处理block2时，我们的算法检测到phi和add指令形成一个没有其他指令的循环,从这个结构中可以得出i的值总是在循环中增加。因此只有phi指令的上限必须设置为MAX，而下限是i在循环，即常数0。

在block3的开头，前面的if指令已经评估，因此phi指令的上限也是众所周知的。使用因此索引变量的结果条件在数组中，访问指令为

0 <= i <= x − 1

这种情况不足以完全消除检查，因为它没有包含实际的数组长度。这个索引的上界以及数组是参数，因此可以调用该方法，参数x为大于数组a的长度，并且该方法可以抛出异常。该检查不是完全冗余的，因此无法通过程序内分析消除。

**因为数组的长度和索引在循环中不更改，检查是部分的冗余的。它可以替换为循环前的检查。**当循环中的指令可能被多次执行，循环执行之前检查仅一次，新的边界检查是不必要的。

在本例中，变量x和常数0都是循环不变量。边界检查可以被替换为通过在循环之前检查x是否不超过数组长度。

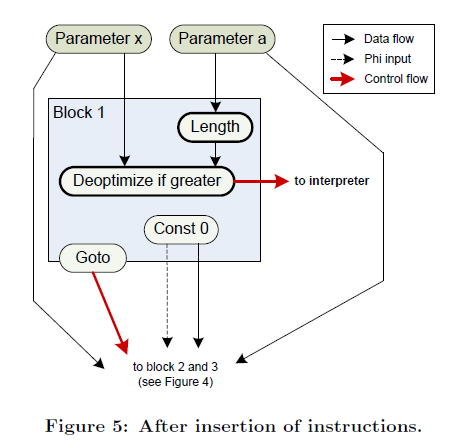
找出指令是否是循环不变量可以使用支配者树内定时完成。任何当前引用的指令必须在块中定义位于当前块和根之间的路径上支配者树的。当指令块位于循环头块和根块之间，则指令是循环不变的。

对于异常语义，出现了一个问题。我们知道边界在循环执行过程中的某个时刻，会检查失败。因此需要在循环之中插入正常边界检查。

此问题的常见解决方案使用循环版本控制（参见示例[11]）。循环的优化版本已知检查是不必要的，而未优化代码要执行边界检查，此解决方案有重复的代码，因此会出现膨胀备份代码出现。

为了避免这种情况，我们的算法插入了一条指令，如果循环前面的检查失败，则触发去优化，没有边界检查的优化机器代码被丢弃并在解释器中执行该方法；这将在正确的点引发异常。在该示例中，算法添加了一条触发如果参数x大于数组长度。

Figure5显示了插入去优化后的HIR指示如果条件为true，则编译机器代码被丢弃，解释器继续执行该方法。



要确定Java程序的变量是否在循环中总是递增到整数溢出。因此，我们需要证明加法量值大于或等于零，并且没有溢出。

然而，在大多数情况下，证明第二部分是不可能的案例。因此，我们需要确保去优化当循环变量可能溢出时调用。这可能在循环之前通过以下显式检查来完成。像在Listing2中定义，x表示循环变量，c是添加到每次迭代中的循环变量：

deoptimize if *x > MAX − c* +1 （去优化）

但是，当使用索引变量的条件时要消除边界检查，请执行以下操作指令已插入循环之前：

deoptimize if *x > a.length*

因此，该算法可以安全地假设循环变量,如果条件总是正确的。

*a.length <*= *MAX − c* + 1

如果变量可能溢出，则去优化无论如何都被调用了，所以我们不需要为此而烦恼。例子由于数组的长度必须适合32位带符号的整数值，如果c等于1，则此条件肯定成立。数组的最大长度也受堆大小除以单个数组元素的大小以字节为单位。因此，在大多数情况下，c的较高值也是可以接受的。

相反的情况是，当我们想要显示一个值总是在减少是更简单的。循环的下界通过取消优化检查变量是否大于或等于零。指示如果此检查成功，则不进行减法,任何正值都可能导致下溢。

1. **Grouping Checks (Range Checks)**

减少执行边界数量的另一种方法checks是将影响同一数组的多个检查分组变成一个单独的。我们将此优化应用于边界前面讨论的未删除的检查技术。为了简化分析，它仅限于检查发生在同一个基本块内，并且索引表达式只因常量而不同。清单3显示了方法有三个数组存储。边界检查**三个store中的一个可以进行优化**，同时需要进行去优化，以确保异常被抛出到正确的位置。

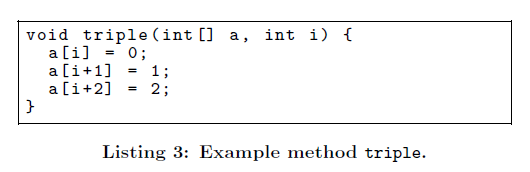
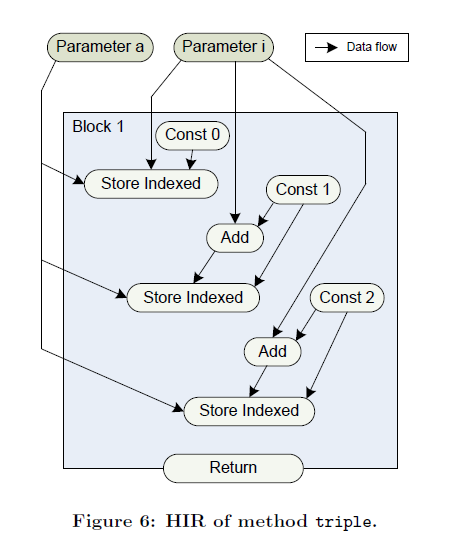


图6显示了该方法的HIR表示。该算法需要对的指令进行一次遍历每个基本块。对于所有数组和索引变量，它维护最小和最大常数在访问数组时添加到索引变量。相反在边界检查中，有两条去优化指令插入。他们检查索引变量加上最小常数和索引变量加上最大值常量在数组的范围内。



当此条件成立时，所有其他涉及的数组访问相同的索引变量和相同的数组是安全的。检查具有不同常量索引的多个数组访问被单个去优化指令所取代检查最大常数是否小于数组长度。

分组边界检查只有在超过两个检查可以组合在一起。这是因为Java数组的下限总是0，因此一个正常的边界检查可以用一个无符号的比较指令。对于分组边界检查，但是，下和上界是必要的。在本例中边界检查被以下两个去优化取代基本块开始时的指令：

deoptimize if *i <* 0

deoptimize if *i* + 2 *>*= *a.length*

i+2的计算可能导致溢流，然而，这种情况可以在没有任何额外费用的情况下处理通过，使用无符号的>=比较来降低成本，因此当i+2溢出时调用去优化。比较对于＜0的第一个去优化指令也可以通过对>=a.length进行无符号比较来执行。

1. **Supporting Optimizations**

应用数组边界检查消除算法作为HIR转换前的最后优化步骤之一至LIR。在我们之前应用的其他优化该算法可以消除更多的边界检查。客户端编译器类常量的内置优化折叠和全局值编号有助于识别索引变量上的更强条件。

为移动检查在循环之外，我们添加了一个简单形式的循环不变代码常量表达式移出循环loop-invariant code motion。对于算术运算，另外两个操作数都需要是循环不变量。

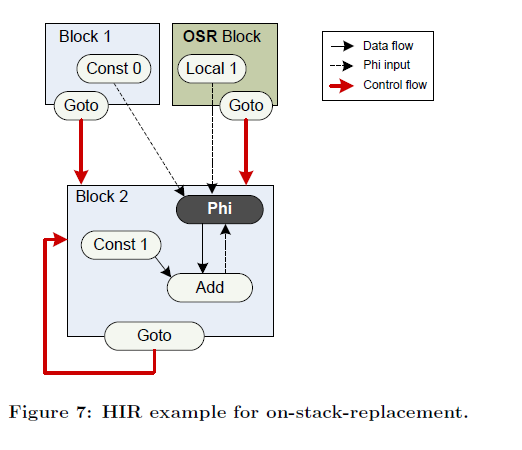
由于可能存在混叠效应，我们需要保守当移动字段加载或array访问时。A字段只有当没有字段时，负载才会移出循环存储到循环中相同类型的字段。数组当数组和索引表达式是循环不变的，并且的数组没有数组存储循环中的相同类型。

移动指令时，我们需要注意例外情况发生在正确的代码位置。现场负载或数组加载可能导致NullPointerException。我们添加了一个指令标志，用于标记不能抛出的指令立即出现异常，但将deoptimization改为实现异常语义。

1. **On-Stack-Replacement**

Java HotSpotTM VM通常编译一个方法,仅当它频繁执行时，即当它的调用计数器达到某个阈值，一个方法可以从在方法运行时被解释为已编译的代码。这被称为堆栈替换（OSR）[8]。此类方法使用一个跳转的附加入口点进行编译直接循环到编译所在的点已触发。解释器的局部变量和操作数堆栈被转移到OSR中的机器代码入口块。

图7显示了当方法使用OSR编译。在这种情况下，phi指令for循环头中的循环值有三个输入。第三个来自OSR块，表示在解释器中执行的循环迭代的结果。当OSR为在嵌套循环中发生堆栈上替换时，所有外部也需要额外的phi指令循环。



当我们的算法应用于没有任何更改，它都只能删除少数数组边界检查。它不能对即将到来的值假设任何条件从OSR入口块。因此他们需要被清除。

我们增强了算法，使其受益于OSR条目的值来自方法在解释器中的执行。这已经完成了通过在更新phi指令的条件。该算法可以假设来自解释器的值的条件与phi指令本身相同。当边界检查移出循环时必须执行其具有OSR。去优化指令必须插入两次：一次像往常一样在循环之前，另一次OSR块结束时的时间。

void RangeCheckEliminator::process\_access\_indexed(BlockBegin \*loop\_header, BlockBegin \*block, AccessIndexed \*ai) {

if (ai->check\_flag(Instruction::NeedsRangeCheckFlag)) {

Bound \*index\_bound = get\_bound(ai->index());

Bound \*array\_bound;

if (ai->length()) {

array\_bound = get\_bound(ai->length());

} else {

array\_bound = get\_bound(ai->array());

}

if (in\_array\_bound(index\_bound, ai->array()) ||

(index\_bound && array\_bound && index\_bound->is\_smaller(array\_bound) && !index\_bound->lower\_instr() && index\_bound->lower() >= 0)) {

**//完全冗余检查,优化边界检查**

remove\_range\_check(ai);

} else if (\_optimistic && loop\_header) {

**//是否为不动点**

Instruction \*array\_instr = ai->array();

if (!loop\_invariant(loop\_header, array\_instr)) { return; }

// Lower instruction

Value index\_instr = ai->index();

Value lower\_instr = index\_bound->lower\_instr();

if (!loop\_invariant(loop\_header, lower\_instr)) { return; }

if (!lower\_instr && index\_bound->lower() < 0) { return; }

// Upper instruction

Value upper\_instr = index\_bound->upper\_instr();

if (!loop\_invariant(loop\_header, upper\_instr)) { return; }

// Length instruction

Value length\_instr = ai->length();

if (!loop\_invariant(loop\_header, length\_instr)) {

// Generate length instruction yourself!

length\_instr = NULL;

}

**//生成去优化代码**

BlockBegin \*pred\_block = loop\_header->dominator();

assert(pred\_block != NULL, "Every loop header has a dominator!");

BlockEnd \*pred\_block\_end = pred\_block->end();

Instruction \*insert\_position = pred\_block\_end->prev();

ValueStack \*state = pred\_block\_end->state\_before();

if (pred\_block\_end->as\_Goto() && state == NULL) state = pred\_block\_end->state();

assert(state, "State must not be null");

// Add deoptimization to dominator of loop header

if (!is\_ok\_for\_deoptimization(insert\_position, array\_instr, length\_instr, lower\_instr, index\_bound->lower(), upper\_instr, index\_bound->upper())) { return; }

**//因为数组的长度和索引在循环中不更改，检查是部分的冗余的。它可以替换为循环前的检查。**

insert\_deoptimization(state, insert\_position, array\_instr, length\_instr, lower\_instr, index\_bound->lower(), upper\_instr, index\_bound->upper(), ai);

remove\_range\_check(ai);

}

}

}

void RangeCheckEliminator::calc\_bounds(BlockBegin \*block, BlockBegin \*loop\_header) {

......

// Pushed stack for conditions

IntegerStack pushed;

// Process If

BlockBegin \*parent = block->dominator();

if (parent != NULL) {

If \*cond = parent->end()->as\_If();

if (cond != NULL) {

**//本质上为更新变量的条件堆栈bound,确保id的bound链top为最新的bound。**

process\_if(pushed, block, cond); }

}

......

Instruction \*cur = block;

while (cur) {

// Ensure cur wasn't inserted during the elimination

if (cur->id() < this->\_bounds.length()) {

// Process only if it is an access indexed instruction

AccessIndexed \*ai = cur->as\_AccessIndexed();

if (ai != NULL) {

**//完全冗余检查、不动点冗余检查**

process\_access\_indexed(loop\_header, block, ai);

accessIndexed.append(ai);

if (!arrays.contains(ai->array())) {

arrays.append(ai->array());

}

Bound \*b = get\_bound(ai->index());

if (!b->lower\_instr()) {

// Lower bound is constant

update\_bound(pushed, ai->index(), Instruction::geq, NULL, 0);

}

if (!b->has\_upper()) {

if (ai->length() && ai->length()->type()->as\_IntConstant()) {

int value = ai->length()->type()->as\_IntConstant()->value();

update\_bound(pushed, ai->index(), Instruction::lss, NULL, value);

} else {

// Has no upper bound

Instruction \*instr = ai->length();

if (instr == NULL) instr = ai->array();

**//更新变量的bound,确保id的bound链的堆栈top为最新的bound，看参考Figure3进行理解。**

update\_bound(pushed, ai->index(), Instruction::lss, instr, 0);

}

}

}

}

cur = cur->next();

}

// Do in block motion of range checks

in\_block\_motion(block, accessIndexed, arrays);

// Call all dominated blocks

for (int i=0; i<block->dominates()->length(); i++) {

BlockBegin \*next = block->dominates()->at(i);

if (!next->is\_set(BlockBegin::donot\_eliminate\_range\_checks)) {

// if current block is a loop header and: - next block belongs to the same loop

// or - next block belongs to an inner loop then current block is the loop header for next block

if (block->is\_set(BlockBegin::linear\_scan\_loop\_header\_flag) && (block->loop\_index() == next->loop\_index() || next->loop\_depth() > block->loop\_depth())) {

calc\_bounds(next, block);

} else {

**//变量的bound为父block中变量的所有条件支配树联合,递归调用体现了这种思想，参考Figure3。**

calc\_bounds(next, loop\_header);

}

}

}

// Reset stack

for (int i=0; i<pushed.length(); i++) {

\_bounds[pushed[i]]->pop();

}

}

// In block motion. Tries to reorder checks in order to reduce some of them.

// Example:

// a[i] = 0;

// a[i+2] = 0;

// a[i+1] = 0;

// In this example the check for a[i+1] would be considered as unnecessary during the first iteration.

// After this i is only checked once for i >= 0 and i+2 < a.length before the first array access. If this

// check fails, deoptimization is called.

void RangeCheckEliminator::in\_block\_motion(BlockBegin \*block, AccessIndexedList &accessIndexed, InstructionList &arrays) {

InstructionList indices;

// Now iterate over all arrays

for (int i=0; i<arrays.length(); i++) {

**//对同一array的访问index进行排队比较，即group checks**

int max\_constant = -1;

AccessIndexedList list\_constant;

Value array = arrays.at(i);

// For all AccessIndexed-instructions in this block concerning the current array.

for(int j=0; j<accessIndexed.length(); j++) {

AccessIndexed \*ai = accessIndexed.at(j);

if (ai->array() != array || !ai->check\_flag(Instruction::NeedsRangeCheckFlag)) continue;

Value index = ai->index();

Constant \*c = index->as\_Constant();

if (c != NULL) {

int constant\_value = c->type()->as\_IntConstant()->value();

if (constant\_value >= 0) {

if (constant\_value <= max\_constant) {

// No range check needed for this

remove\_range\_check(ai);

} else {

max\_constant = constant\_value;

list\_constant.append(ai);

}

}

}

.......

**//典型的Group check s，为方便理解，假设block里面有a[2]、a[3]、a[5]、a[7]等几个索引访问，对同一array即a，只要对a的队列长度和7进行判断并去优化，其余的均可以优化。而对于变量索引，思想和此类似。**

if (list\_constant.length() > 1) {

AccessIndexed \*first = list\_constant.at(0);

Instruction \*insert\_position = first->prev();

ValueStack \*state = first->state\_before();

// Load max Constant

Constant \*constant = new Constant(new IntConstant(max\_constant));

NOT\_PRODUCT(constant->set\_printable\_bci(first->printable\_bci()));

insert\_position = insert\_position->insert\_after(constant);

Instruction \*compare\_instr = constant;

Value length\_instr = first->length();

if (!length\_instr) {

ArrayLength \*length = new ArrayLength(array, state->copy());

length->set\_exception\_state(length->state\_before());

length->set\_flag(Instruction::DeoptimizeOnException, true);

insert\_position = insert\_position->insert\_after\_same\_bci(length);

length\_instr = length;

}

// Compare for greater or equal to array length

**//对队列访问索引中，最大常量进行判断去优化**

insert\_position = predicate(compare\_instr, Instruction::geq, length\_instr, state, insert\_position);

for (int j = 0; j<list\_constant.length(); j++) {

AccessIndexed \*ai = list\_constant.at(j);

remove\_range\_check(ai); **//除了最大常量后的其余常量进行边界检查消除**

}

}

}

// Clear data structures for next array

for (int i = 0; i < indices.length(); i++) {

Instruction \*index\_instruction = indices.at(i);

\_access\_indexed\_info[index\_instruction->id()] = NULL;

}

indices.clear();

}

}