# Loop Optimization for CCured

AJ Shankar

17 December 2002

## 1 Motivation

CCured is a source-to-source development tool for C that ensures runtime memory safety by inserting dynamic checks only where necessary. For the average C program the vast majority of memory reads and writes can been shown as safe statically, so relatively few runtime checks need to be inserted; as a result, runtime performance generally decreases by only 10-20%.

However, some C programs suffer a performance penalty of up to 1,000% when cured, rendering CCured an impractical way to ensure memory safety for these programs. Programs that manipulate certain data structures, like arrays and matrices, often use looping constructs to traverse the data:

```
for (i = 0 ; i < len ; i++) {
    ary[i] = 0;
}
```

To ensure that the program does not write out of bounds, CCured is obligated to check the memory write over every iteration:

```
for (i = 0 ; i < len ; i++) {
    CHECK_BOUND(ary_length, i);
    ary[i] = 0;
}
```

essentially doubling (or worse) the number of operations performed during each iteration. The situation degenerates further as more memory reads and writes occur in a loop. Simply swapping two adjacent elements in an array induces CCured to insert *six* bounds checks. Programs that use loops to access memory present a paradox: they cannot be cured without running woefully slowly, but they are exactly the kinds of programs (with off-by-one errors and so on) that would benefit from guaranteed runtime memory safety.

The loop optimization presented in this paper addresses this paradox by eliminating most of the runtime overhead caused by CCured bounds checks in loops.

## 2 Previous Work

There has been a lot of research in the area of hoisting loop-invariant computations. However, no standard algorithm would ever consider hoisting a function call such as CHECK_BOUND, especially one whose arguments are variable. This algorithm exploits the fact that we know exactly what the bounds check call does. However, we still impose many conservative requirements similar to the ones needed in normal loop-invariant hoisting, such as those found in [Appel98] and [Aho86].

ABCD [Bodik00] is a powerful yet simple bounds check optimizer. However, it is designed for Java, and does not have to face the direct memory stores that C allows; extending it to handle arbitrary memory writes would severely complicate its simple constraints system. It does, however, deal with complex control flow much better that the algorithm proposed in this paper; I am looking into implementing some of its concepts if time permits.

## 3 Optimization

The fundamental action of the optimization is to "hoist" a check: that is, move it from inside the loop to before the loop, and modify it in such a way that it still enforces memory safety. For instance, in the above example, the array index $i$ is being checked a linear number of times when in fact since the loop and $i$ are well-behaved (a notion we will define more precisely below), we need only insert one check before the loop:

```
CHECK_BOUND(ary_length, len-1);
for (i = 0 ; i < len ; i++) {
    ary[i] = 0;
}
```

This code is semantically equivalent to the original, since $i$ only ranges between 0 and $len - 1$, and now the check is executed just once. Not all checks can be hoisted in this way, and the primary difficulty in implementing this optimization is ensuring that it is conservative. Hoisting incorrectly can result in code that runs but should fail, and can even result in code that should run but fails!

## 3.1 Bad candidates for hoisting

As this optimization does not attempt to determine control flow, loops with certain control structures must be avoided. If a loop has a *goto* or multiple breaks, it is hard to determine under what conditions control will exit the loop, and so the optimizer skips it. If a check occurs within a conditional, then the optimizer skips it.

## 3.2 Requirements

If a check is determined to be potentially hoistable, certain requirements must be met to ensure that hoisting the check will not affect the execution of the program. Consider the following generic loop:

```
while(i < expr) {
    ...
    CHECK_BOUND(0, ary_length, i+2);
    ary[i + 2] = i;
    ...
    i += cexpr;
}
```

1. *expr* **must be invariant.** Since *expr* is essentially being substituted for the induction variable, $i$, in the hoisted check, it cannot change during the lifetime of the loop.

2. $i$ **must be invariant between the conditional and the check.** Again, since we are equating $i$ and *expr* in the hoist, $i$ cannot change its value once the conditional has been met. (This is not an absolutely necessary restriction; see Sample Improvements.)

3. *ary* **must be invariant.** Clearly if *ary* were to change length within the loop, the check could not be hoisted.

4. **Check must involve just $i$ and invariants.** If the check involved any other variables, we would not be able to substitute for them accurately outside the loop.

5. $i$ **must be monotonically increasing.** Since the conditional is $i < expr$, $i$ must always stay the same or make progress towards *expr*. (Of course, if the conditional were $>$, then $i$ would have to be monotonically decreasing.) Otherwise lower bounds checks might be necessary inside the loop. To this end, *cexpr* must either be constant or an unsigned int.

### 3.2.1 Invariance

Invariance is one of the trickiest conditions to establish in loop optimization. In traditional C, any time a value is written to memory, any variable might change value, even if its address was never taken, thanks to pointer arithmetic. If this were the case, then this optimization would always fail, since it is trying to remove checks for array accesses, many of which might themselves be memory writes! Luckily, CCured enforces proper pointer behavior (that is, pointers cannot write outside the bounds to which they were initially assigned), and so we can use the following conservative rules to determine invariance:

1. If a variable is assigned, it is not invariant.

2. If a function other than CHECK_BOUND is called, all variables whose addresses have been taken are not invariant.

3. A function call itself is never invariant.
4. If a value is written to memory, all variables whose addresses have been taken are not invariant.

All other variables can be considered invariant.

### 3.3 Hoisting

If a check meets all the requirements, it can be hoisted. To do this, we simply syntactically replace the induction variable $i$ in the check with its extreme value in the loop. If the conditional is $i <= expr$ or $i >= expr$, then this value is just *expr*. If the conditional is $i < expr$ then it is $expr - 1$; $i > expr$ yields $expr + 1$. For instance,

```
while(i < 2*j) {
    CHECK_BOUND(0, ary_length, i+2);
    ary[i + 2] = i;
    i++;
}
```

becomes

```
CHECK_BOUND(0, ary_length, ((2*j)-1)+2);
while(i < 2*j) {
    ary[i + 2] = i;
    i++;
}
```

## 4 Implementation Details

CCured is written in O'Caml and uses CIL, a source-to-source C transformer that canonicalizes and simplifies programs, for instance by removing side-effects from expressions. The loop optimizer is written in O'Caml as well, and navigates the AST of a program after CIL and CCured are through with it. CIL provides convenient methods for traversing ASTs and its standardization of C code means that the optimizer has to handle many few cases than it would for a normal C program (for example, for and while loops look the same); the one downside is that CIL occasionally introduces bits of code that trick the optimizer, such as

```
tmp = v;
v = tmp;
```

In this case, if *tmp* and $v$ are of the same type, $v$'s value does not change but the optimizer thinks it does.

In this paper I have simplified the checking abilities of CCured into one operation, CHECK_BOUND, whereas in reality there are many different kinds of checks, depending on how a block of memory has been allocated (statically or with malloc) and how it is being accessed. Different checks require slightly different conditions to be satisfied; currently the optimization supports 4 kinds of checks, although I am implementing a framework that will allow it to support many more.

As I have not run the optimizer on large programs, I have been unable to assess how fast it is. Currently it visits each node in a loop's AST 3-4 times for each check it is considering hoisting, and of course it can be disabled as necessary. If it proves to be useful, I will spend more time optimizing it.

## 5 Results

I tested the loop optimization on three programs that used many loops and experienced marked slowdowns when cured. I timed each program under four conditions:

1. Cured, with no optimizations
2. Cured, with the existing optimizations[1]
3. Cured, with the existing optimizations and the loop optimization
4. Uncured

All programs were compiled with gcc 2.95; the average of three executions on manju was taken.

More information on the algorithms used in these programs can be found in the References section. The rough performance metric I devised is

$$\frac{existing - loop}{existing - uncured}$$

[1]CCured already has a number of optimizations that attempt to remove redundant checks

| | Cured | Existing | Loop | Uncured |
|---|---|---|---|---|
| sorting | 4.76 | 4.10 | 0.81 | 0.40 |
| sieve | 0.97 | 0.96 | 0.59 | 0.57 |
| queens | 3.69 | 3.70 | 3.47 | 0.72 |

Table 1: Optimization results

That is, how close does the loop-optimized version get to perfection – no overhead at all – relative to the previously-optimized code? A ratio of 1 indicates optimality.

### 5.1 Sorting

This program sorts 5,000 numbers by bubble sort, and 5,000 numbers by selection sort. Both of these algorithms involve nested loops and the swapping of array elements, provoking worst-case behavior from CCured: a slowdown of over 1,000%. The loop optimization hoists roughly two-thirds of the checks, including the most costly ones, resulting in a speedup ratio of .89.

### 5.2 Sieve

This program finds all the primes under 8,000,000 using the Sieve of Eratosthenes. The sieve method also involves nested loops but far less array manipulation than sorting does. The loop optimization successfully hoists all of the checks, although some remain nested in other while loops, giving a speedup ratio of .95.

### 5.3 Queens

This program solves the queens problem for a 20x20 chessboard. While its heavy use of loops and arrays might suggest that it is a prime candidate for hoisting, the loop optimization provides only a marginal improvement in execution time. The reason is that the program uses two-dimensional arrays, and often just a single loop to iterate over them:

```
while (i < 20 && j < 20) {
    board[i][j] = ...;
    if (cond)
        i++;
    else
        j++;
}
```

rather than two nested loops, one for $i$ and one for $j$, since it traverses a linear rather than quadratic number of board spaces. The optimizer bails when it finds two breaks on different variables in the loop, and therefore fails to hoist most checks; the speedup ratio is an unimpressive (but still noticeable) .08.

### 5.4 Comments

It is important to note that with the exception of queens, these tests illustrate the best-case conditions for the loop optimizer: programs in which the vast majority of the slowdown incurred by curing is in the form of hoistable bounds checks. I have not been able to run the optimizer on larger applications because I am not convinced of its soundness and therefore do not want to run benchmarks unless I can hand-verify each hoist, but my guess is that large-scale applications will not benefit nearly as significantly from this optimization. After all, the optimizer can only reduce the amount of time spent in loops, and if not much time is spent in loops in the first place, any speedup must necessarily be marginal.

However, as evidenced by these results, there is a class of programs that rely on memory accesses within loops that require the loop optimization if they are to be cured for any practical use, and it is at this class that the loop optimization is aimed, at least in its initial stages of development.

## 6 Future Work

My primary goal is to show the soundness of the optimization: that it does not change the behavior of the cured program by hoisting checks incorrectly. I would also like to expand its scope by handling more types of checks (CCured has many), and by being more liberal in determining whether it is okay to hoist a given check. Furthmore, I would like to investigate how useful this optimization actually is across a broad range of programs by benchmarking various test suites and real-world applications.

(The reason that I have not made much progress in these areas since my presentation is that I am in the process of rewriting the entire optimizer to attain better modularity, maintainability and efficiency.)

# 7 Sample Improvements

## 7.1 Nested Loops

A potential improvement involves nested loops; a check should be hoisted as high as possible, instead of just out of its immediate loop:

```
for (i = 0 ; i < len ; i++) {
    for (j = 0 ; j < len ; j++) {
        ary[i][j] = 0;
    }
}
```

becomes, after optimization:

```
CHECK_BOUND(ary_length, len-1);
for (i = 0 ; i < len ; i++) {
    CHECK_BOUND(ary_length, len-1);
    for (j = 0 ; j < len ; j++) {
        ary[i][j] = 0;
    }
}
```

but here the second check is still being tested a linear number of times. It should really be:

```
CHECK_BOUND(ary_length, len-1);
CHECK_BOUND(ary_length, len-1);
for (i = 0 ; i < len ; i++) {
    for (j = 0 ; j < len ; j++) {
        ary[i][j] = 0;
    }
}
```

since *len* is invariant in both loops.

## 7.2 Invariant conditionals

Currently, if the check is within a conditional,

```
for (i = 0 ; i < len ; i++) {
    if (cond) {
        CHECK_BOUND(ary_length, i);
        ary[i] = 0;
    }
}
```

then the optimizer fails. However, if it can be show that the conditional is invariant in the loop, then we can hoist the check as well:

```
if (invariant_cond)
    CHECK_BOUND(ary_length, len-1);
for (i = 0 ; i < len ; i++) {
    if (invariant_cond) {
        ary[i] = 0;
    }
}
```

## 7.3 Derived induction variables

If a variable $i$ is an induction variable, and $j$ is defined as $c \cdot i + d$ for some invariants $c$ and $d$, then we can say that $j$ is a derived induction variable in the family of $i$: it can be computed directly from $i$. Therefore we should be able to hoist code like:

```
for (i = 0 ; i < len ; i++) {
    j = 2*i + 1;
    CHECK_BOUND(ary_length, j);
    ary[j] = 0;
}
```

to

```
CHECK_BOUND(ary_length, 2*(len-1)+1);
for (i = 0 ; i < len ; i++) {
    j = 2*i + 1;
    ary[j] = 0;
}
```

There are many other such possible additions of varying complexity.

# 8 References

[Spodik00] Bodik, R., R. Gupta, and V. Sarkar. *ABCD: Eliminating Array-Bounds Checks on*

*Demand*. SIGPLAN 2000.
[Appel98] Andrew Appel. *Modern Compiler Implementation in C*. Cambrige Press, Cambridge, UK, 1998.
[Aho86] A. Aho, R. Sethi, J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

Information on CCured and CIL can be found at `http://manju.cs.berkeley.edu/ccured`
`http://manju.cs.berkeley.edu/cil`

Information on queens and sieve can be found at
`http://mathworld.wolfram.com/QueensProblem.html`
`http://mathworld.wolfram.com/SieveofEratosthenes.html`