

Project Report: Safe C using Fat Pointers

Miguel Silva

<http://andrew.cmu.edu/~miguel/safec>

May 7, 2009

1 Introduction

C and other unsafe languages allow programmers to easily access the wrong memory regions and to disregard the type system completely. This enables many security holes and bugs, the most well-known and historic relevant of those being the *array overflow* problem. Nonetheless, C remains one of the most used programming languages, with a large amount of software written in it, and several other languages that depend on its libraries.

In order to solve these issues, we have developed a method that uses *fat pointers* to ensure the code's memory safety. Our work is based on CCured [5–7] and applies some of the techniques used by that compiler to the LLVM IR. Specifically, we created an analysis based on a trimmed version of the CCured type system and implemented the following optimizations:

- **Partial Redundancy Elimination:** Usually, dynamic checks must be performed whenever a pointer is dereferenced. However, some of these checks may be redundant (if they are loop invariant, for instance), and can be eliminated using a special version of PRE for checks.
- **Forward Pointers:** Some pointers are only incremented and therefore we only need to keep track of the pointer's upper bound.
- **Loop Optimization:** We implemented an optimization similar to the one presented in [8], that moves some checks to the outside of loops, whenever it is possible to verify that the loop bounds never cause the pointer to overflow.
- **Loop Unroll:** We can also unroll a loop and check several pointers from different iterations at once

Our goal for this project was to create a method that does not require the source code to be rewritten, that works for any language, and that can be used to check if optimizations preserve safety. Additionally, if the code is legal we want it to run without raising exceptions and without major overhead.

The project's webpage is <http://andrew.cmu.edu/~miguel/safec>

2 Implementation

We implemented our method on top of LLVM Intermediate Representation as a whole-program analysis and transformation. This allows us to use our technique on other languages other than C and ensures that we can cover any unsafe pointer potentially introduced by other transformations.

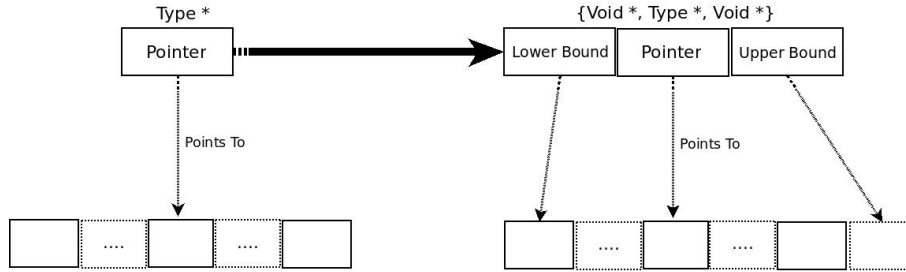


Figure 1: Structure of a fat pointer

2.1 Fat Pointers

Fat pointers are so called because they carry information about the pointer that is necessary to verify that a pointer is valid. In our case, fat pointers are structures containing three pointers: the original pointer and the pointers to the bounds of the memory region (figure 1).

We implemented a pass that detects any pointer p that must be turned into a fat pointer, based on following criteria:

- p is a fat pointer if p is subject to pointer arithmetic,
- or if p is the result of pointer arithmetic ($p = \text{getelempttr } \dots$),
- or if p is the result of an integer being cast to a pointer.

The first two rules exist to make sure we are able to raise an exception when a pointer goes beyond its memory regions' boundaries (in the presence of an array overflow, for instance), while the last one forbids pointers to arbitrary places in memory. This analysis uses unification to ensure that we preserve type safety whenever a type is changed from a normal pointer to a fat pointer.

After this analysis, each fat pointer $\text{type} * p$ must be transformed into the structure $\{\text{void} *, \text{type} *, \text{void} *\} p$. We also need to add new instructions that extract the original pointer whenever necessary, and we must initialize the bounds of the pointers:

- if $p = \text{alloca } \dots$, the bounds are p and $p + 1$
- if $p = \text{malloc } \text{type}, \text{size}$, the bounds are p and $p + \text{size}$
- if $p = \text{getelementptr } q, \dots$ the bounds are equal to the ones for q .
- if p is a null pointer or was cast from integer both bounds are 0

Checks are placed just before a *load* or *store* that dereferences p , and not when the pointer is created, since it is legal to have a invalid pointer that is never used.

We have implemented the analysis and the code transformation as separate passes, and we execute the optimization passes in between them. The analysis pass creates a table of checks indexed by program points, and the optimizations delete or insert checks wherever necessary. This avoids having to insert code that may have to be deleted afterwards.

External Functions

Although our compiler is a whole program compiler, we do not recompile libraries. This leaves the question of how to handle functions that exist outside our compilation unit. Since we don't have access to the body of these functions, we cannot change the data layout of the memory used inside them. Therefore, given a pointer $\text{type} **p$, p itself may be a fat pointer (since we can always extract the original pointer from the fat pointer), but $*p$ must remain a normal pointer.

2.2 Errors Detected

The most important kind of error we want to be able to detect is pointer overflow. Although this is a common and important kind of error, a complete version of our analysis should also be able to determine if a pointer is cast to the wrong type. We tested the compiler with programs created with intentional bugs and successfully raised an exception whenever an invalid pointer was detected.

For example, many functions in C that deal with arrays also receive the size of that array as an argument:

```
int foo(int *a, int n){
    int i;

    for(i=0;i<n;i++){
        ...
        a[i]=...
        ...
    }
}
int main(){
    int a[10];

    foo(a,11);

    return 0;
}
```

The previous code would cause an error since we sent the wrong array size to function `foo`. Another way of doing this is to use a value to represent the end of the array and to stop the loop when we find that value. However, this not a good solution either: there may not be such a value (we may want to use any integer as a valid value) and the main problem - the fact that we have to trust the user to do the right thing - remains.

Since `foo`'s argument `a` is subject to pointer arithmetic, we must turn `a` into a fat pointer. This means that the function will receive not only `a` itself but it's actual bounds and we will be able to detect if anything goes wrong and abort the program.

Another example that raises an exception:

```
void foo(){
    int a;
    int *b=&a;

    b++;

    ...*b... //raises exception
}
```

Most of the programs from benchmark suites did not raise any exceptions; however, we did detect a pointer overflow (on `compress` from **SPEC95**) and an integer cast to pointer (on `mcf` from **versabench**). As far as we could assess, this last exception was not an actual problem since the integers had previously been cast from pointers.

3 Performance and Evaluation

In order to evaluate our method, we used several programs from different benchmark suites. `bmm`, `802.11`, `8b10b`, `ecbdes` and `beamform` come from the `versabench` benchmark suite; and `gzip`, `bzip2` and `crafty` are

from SPEC2000. `qsort`, `yacr2`, `anagram` and `KS` are part of a test suite used in a project similar to ours [1]. Finally, we also included a test (`SearchGame`) from the benchmark suite Fhourstone.

3.1 Fat Pointers

A large percentage of the pointers in our programs became fat pointers (figure 2(a)), although most those fat pointers are never checked (figure 2(b)). Nevertheless, the simple fact that a fat pointer exist means that we need to keep more information in memory.

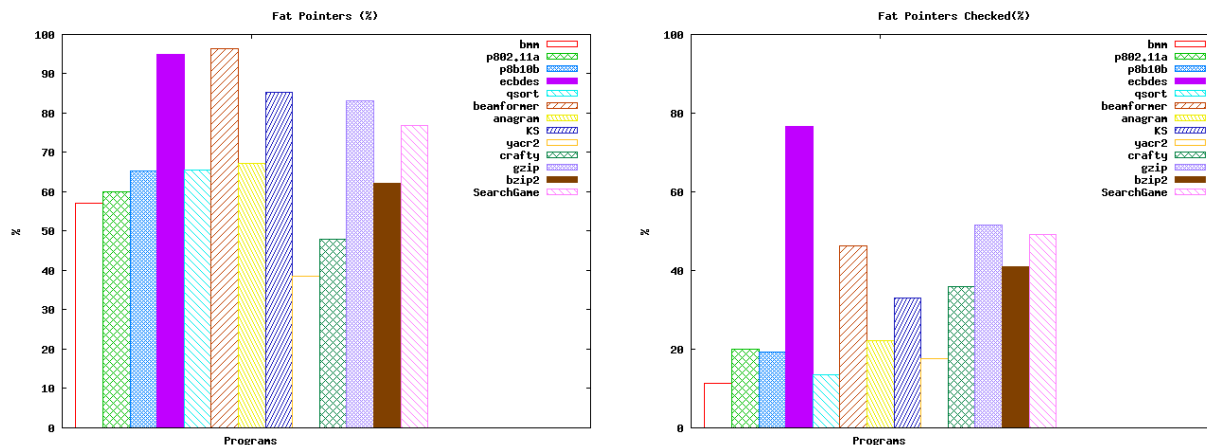


Figure 2: Percentage of fat pointers

Most of these fat pointers seem to be used inside loops, and they usually point to some element of an array. In fact, figure 3 plots the number of times the same memory region is checked in each program.

As we can see, most programs iterate over large memory regions, and the most likely cause of overhead is the fact that we are checking those pointers at each iteration.

Compared to CCured, we create a larger percentage of pointers. CCured uses symbolic analysis to statically detect if a pointer never gets outside its bounds, and in some cases is capable of backtracking and changing a fat pointer back into a normal pointer if that pointer is never used [5].

3.2 Overhead

Figure 4 shows the overhead of using our technique without any optimizations. The maximum slowdown is 4.5x and the average slowdown is 2.2x.

4 Optimizations

4.1 Forward pointers

The first optimization we implemented was the *Forward pointers optimization*. A forward pointer is a pointer that is only incremented, and therefore we only need to keep the information necessary to check if the pointer is always smaller than its upper bound (a Forward fat pointer is a structure $\{Type * p, void * u\}$, where p is the original pointer and u is the upper bound).

A forward pointer is detected by looking at the *getelementptr* instructions where the pointer is used (i.e the points where the pointer is subject to pointer arithmetic). If all the indices in all those instructions are positive (either a positive constant or an expression with only positive subexpressions and without

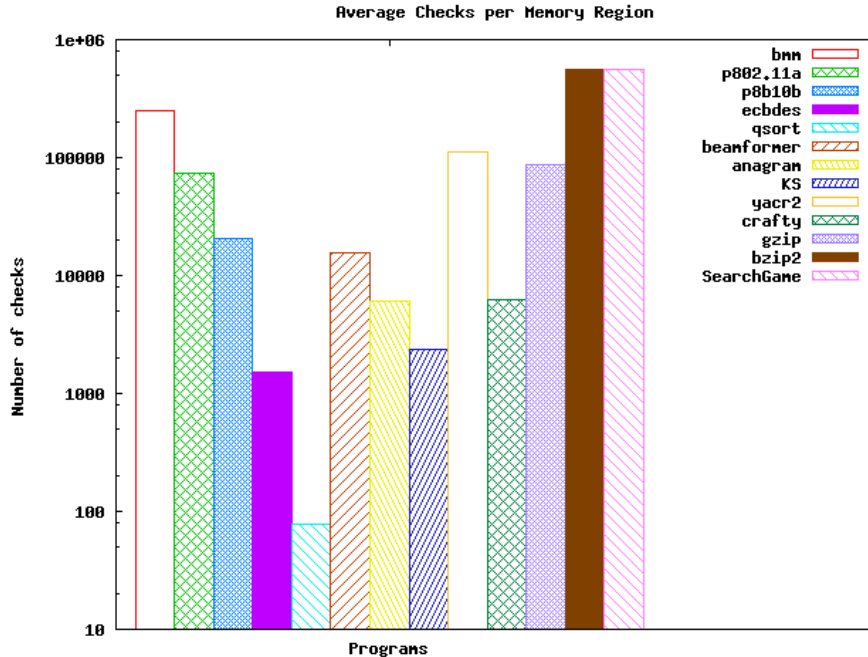


Figure 3: Average number of times a pointer from the same memory region is checked

subtractions), the pointer is a forward pointer. We found that, on average, 80% percentage of the pointers are Forward pointers.

Figure 5 shows the results for this optimizations, and we can see that most of the programs witness some improvement.

4.2 Partial Redudancy Elimination

In order to eliminate redudant checks we implemented a version of Partial Redudancy Elimination specifically designed for those dynamic checks (which we will simply call PRE). PRE uses the usual analysis [2], where $check(p)$ is locally transparent at the entry of block B if the block where p was defined strictly dominates B, and it is locally anticipatable if it is locally transparent and there is a check for p in B. We also preceded our optimization with LLVM’s *Loop Invariant Code Motion* and *Partial Redudancy Elimination* passes, since our pass only deals with redudancy on the checks themselves.

Unfortunately, there aren’t many redudant checks in our programs. As we have previously seen, most of the checks exist inside of loops but are not loop invariant. Therefore, the results (figure 6) of this optimization leave the overhead almost unchanged, with only two examples (802.11 and ecdbes) getting better performance.

4.3 Loop Bounds Checks (LBC)

In section 3.1 we stated that many of our fat pointers are used inside loops. Assuming we can calculate the bounds of those loops then it should be possible to replace each check inside a loop’s body with another one that ensures, before the execution of the loop, that given those loop bounds the pointer will always point to the correct memory region.

Let p be a pointer whose checks we are trying to hoist out of a loop. In order to do this we first need to ensure:

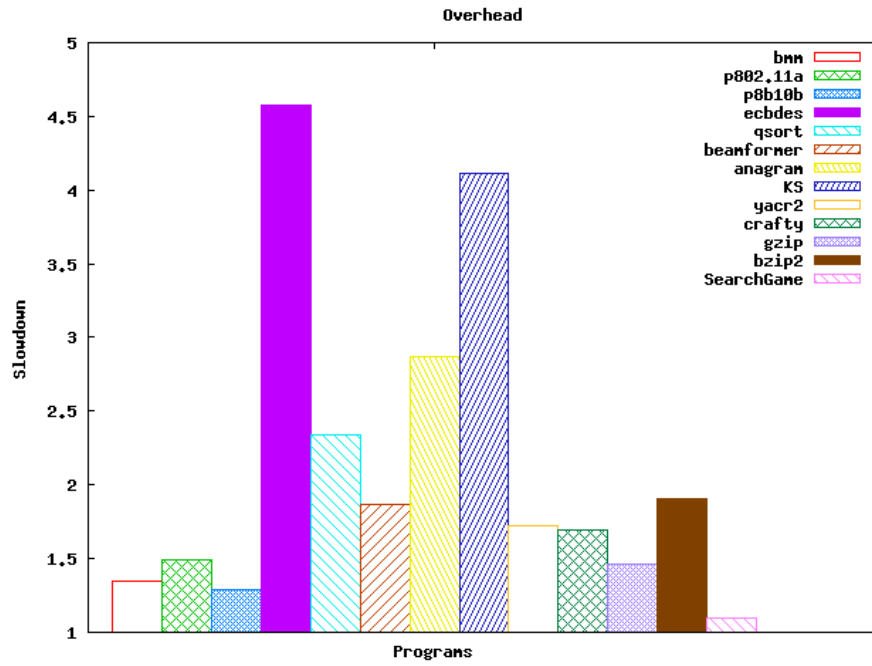


Figure 4: Slowdown with no optimizations

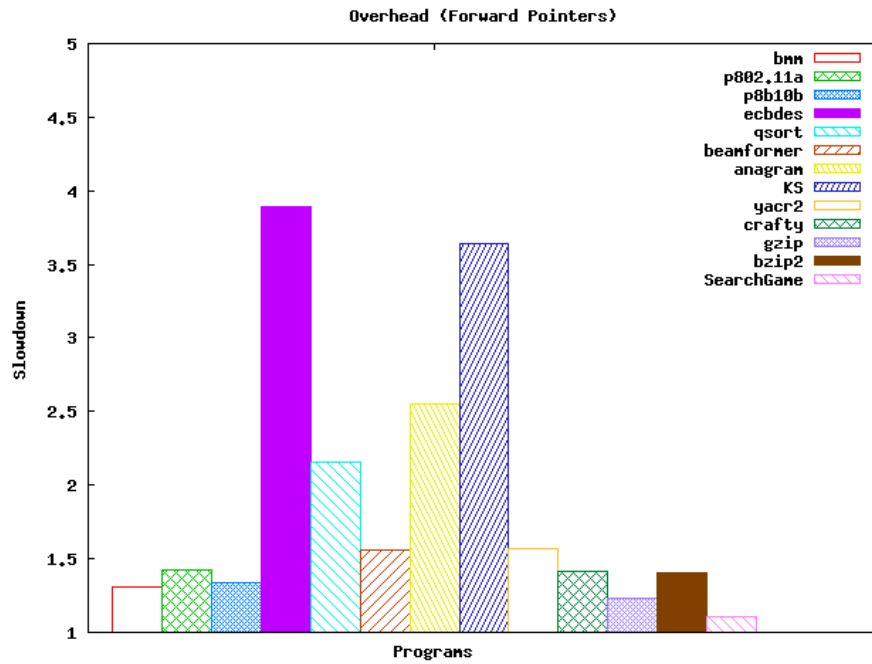


Figure 5: Slowdown with Forward Pointers optimization

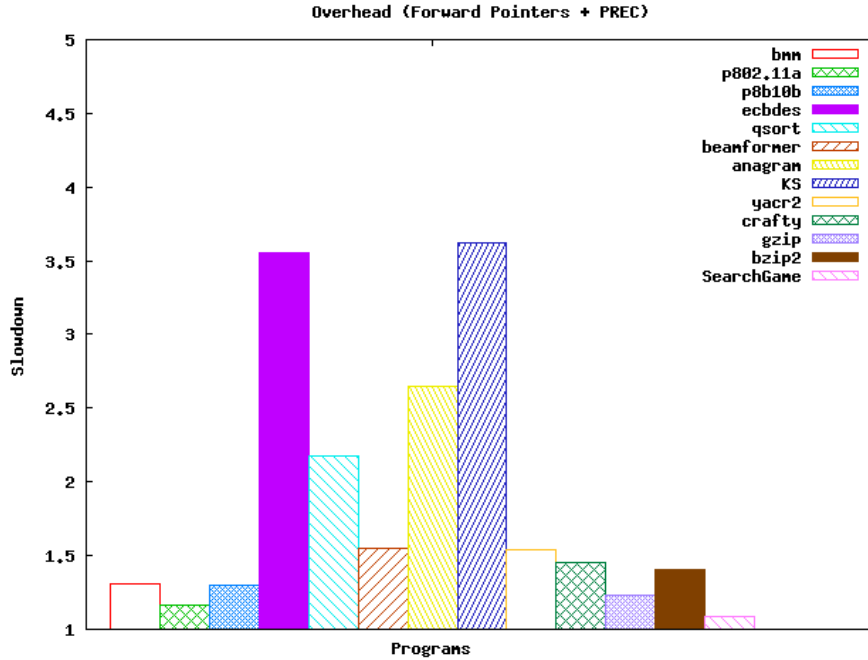


Figure 6: Slowdown with Forward Pointers and PRE

- The loop has a canonical induction variable
- The termination condition is of the form $a * i + b \leq N$ or $a * i + b \geq N$, where i is the loop's canonical induction variable, and a , b and N are loop invariant
- The increment is also of the form $i = a * i + b$, where a , b are loop invariant and a is positive.
- $p = \text{getelementptr } p', d_1, d_2, \dots, d_n$ (i.e. $p = p'[d_1][d_2] \dots [d_n]$), where every d_j is a linear combination of the canonical induction variables and p' is loop invariant
- p is not inside an *if*.

Alternatively, we can also apply this optimization if the following conditions are true:

- The loop has a canonical induction variable
- The termination condition is of the form $a * i + b \leq N$ or $a * i + b \geq N$, where i is the loop's canonical induction variable, and a , b and N are loop invariant
- The increment is of the form $i = i + b$, where b is loop invariant.
- $p = \text{getelementptr } p, d$ (i.e. $p = p + d$), where p is a phi instruction and d is constant
- p is not inside an *if*.

Figure 7 shows examples of these two cases.

Let L and U be the bounds of the memory region visited by the loop. These requirements allow us to easily calculate the initial and last values of the canonical induction variables, and use them to find the values of L and U . For instance, in the first example in figure 7, $L = a$ and $U = a + (N - 1)$ (we replaced i by its last value), and in the second example $L = p$ and $U = p + d * (N - 1)$ (we multiply d by the trip count).

```

int a[N];

for(int i=0; i< N; i++){
    p= a+i;
    ...
}

int *p;

for(int i=0; i< N; i++){
    p= p+d;
    ...
}

```

Figure 7: Two diferent loops that can be optimized using LBC

If all the necessary conditions are true, then we calculate L and U at the loop's pre-header and check if they are inside the appropriate bounds. This way, we can discard the checks for p that exist within the loop's body.

We were able to optimize several of our examples using LBC. In particular, cases like `bmm` (which implements matrix multiplication) ¹ are made mostly of loops with bounds that are easy to calculate, allowing us to hoist out all the checks and greatly improve the performance.

However, some cases, such as `anagram` remain unchanged. `anagram` is a particularly difficult example to optimize because most of its loops are of the form:

```

while(io_f(...)=...){
...
}

```

(where `io.f` is an input/output function) or:

```

while(*p=='\0'){
...
}

```

and it was impossible for our optimization to determine their bounds. `ks` also experienced limited performance increase due to the fact that some of its loops allocate and use new memory regions at each iteration. Figure 8 shows the results of using this optimization.

4.4 Loop Unroll and Pointer Chain Checks elimination (PCE)

We define a *Pointer chain* as a sequence of pointers of the form:

$$\begin{aligned}
 p_1 &= a + b_1 \\
 p_2 &= a + b_2 \\
 &\dots
 \end{aligned}$$

¹It may seem strange why this particular example has only a 1.5 slowdown to begin with. The fact that `bmm` outputs its results to a file, which would dominate the execution time of the program, is probably the reason why this happens

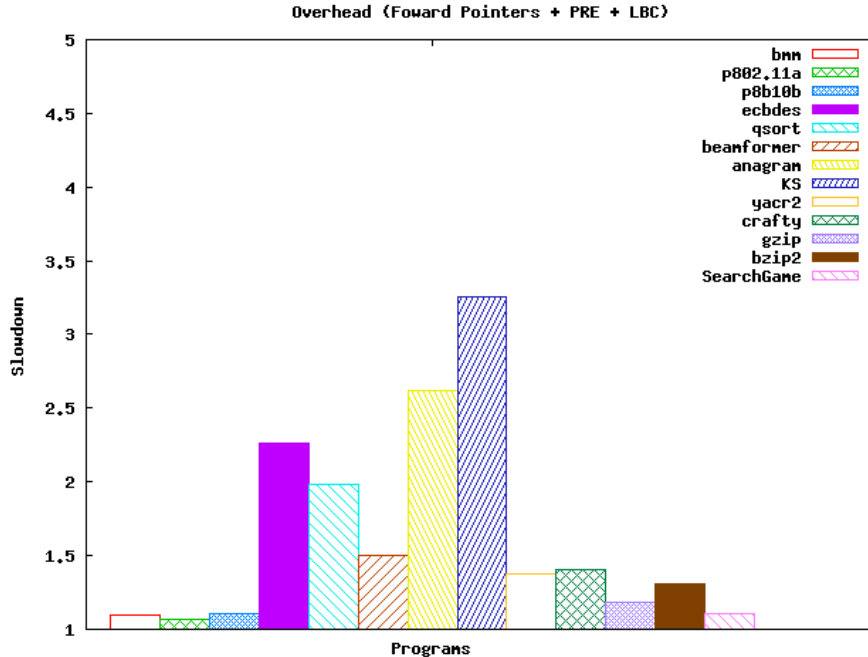


Figure 8: Slowdown with Forward Pointers, PRE and LBC

$$p_n = a + b_n$$

or

$$\begin{aligned}
 p_1 &= a + b_1 \\
 p_2 &= p_1 + b_2 \\
 &\dots \\
 p_n &= p_{n-1} + b_n
 \end{aligned}$$

where all p_j are in the same block, $b_1 > b_2 > \dots > b_n$ (or $b_1 < b_2 < \dots < b_n$) and p_n is used somewhere in the same block. Since we know that we will have to calculate p_n and dereference it, we can eliminate the checks for all other p_j and just check p_n instead (**note:** the check must be placed before the first use of p_1).

This technique should be particularly useful if we do loop unrolling. We used LLVM's *Loop Unroll pass*, but the pass did not unroll most of the interesting loops. We were still able to optimize some examples (`beamformer`, `ks` and `yacr2` 9), although several of the chains we found already existed in the original code, prior to unrolling.

With all optimizations in place, we get a maximum slowdown of 3x and average of 1.5x.

4.5 Related work

As previously stated, our approach is similar to the one designed for the CCured [5–7] compiler. The major difference is the fact that CCured requires some code rewriting and is a source-to-source compiler. Also our optimizations focus on trying to find checks that are somehow redundant in order to eliminate them.

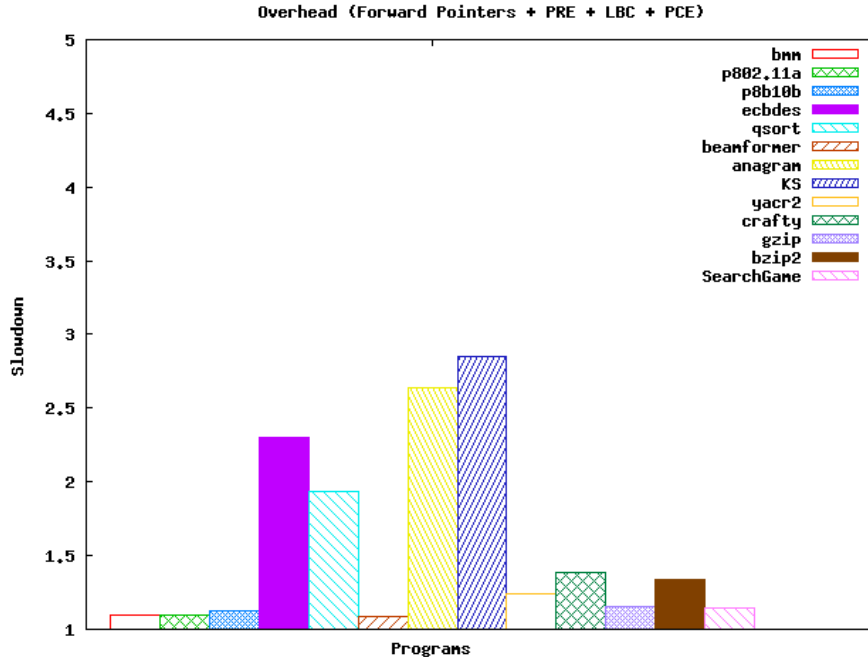


Figure 9: Slowdown with Forward Pointers, PRE, LBC and PCE

CCured, on the other hand, uses PRE and Forward Pointers but also includes other additional analysis to discard some unnecessary fat pointers.

In [9], the author describes a compiler that accepts any unaltered C program and ensures the result is safe. However, this compiler depends heavily on runtime checks and analysis, and even with optimizations there are still cases that suffer a slowdown of 5x.

It is also worth mentioning the existence of safe C dialects, such as Popcorn [4] and Cyclone [3].

4.6 Future Work and Conclusion

During the course of this project we were able to implement an extension for the LLVM compiler that detects possible pointer overflows and raises an exception if necessary. Our technique is not complete since we don't raise exceptions for cases where a pointer is cast to the wrong type, a situation that can also prevent a program from being safe.

In the future, we would also like to include a symbolic solver in our compiler to help us detect, during compilation time, that a pointer with statically known bounds will never overflow, thus eliminating all the overhead associated with turning that pointer into a fat pointer. For example, some arrays have exactly the same size as the loops that use them, and if we could detect this statically, those pointers could remain as normal pointers.

We implemented our analysis on top of LLVM IR, making it possible for other people to create C-like languages (such as Popcorn and Cyclone) without having to worry about safety.

In terms of performance, our optimizations were able to eliminate a significant percentage of overhead, although the final slowdown observed in some examples is still too high. We believe that the previously mentioned symbolic solver could be used to find more cases where our LBC optimization can be applied.

Also, a better *loop unrolling* pass is necessary since we feel that most of the potential of our current PCE optimization is left unexplored.

Finally, there are some cases, such as **anagram**, that we were unable to optimize. **Anagram** is particularly interesting since the bounds cannot be calculate statically nor can they be calculated dynamically before the execution of the loop, and therefore we really need to check the pointers at each iteration. For cases of the form:

```
while(*p=='\0'){  
...  
}
```

a possible solution could be to check before the loop if '\0' appears inside the memory region. However, this solution would still cause (potentially more) overhead and assumes we know which value is used to represent the end of the array.

References

- [1] Pointer-intensive benchmark suite
<http://pages.cs.wisc.edu/~austin/ptr-dist.html>.
- [2] Seth Goldstein. Lecture 20: **PRE** and loop invariant code motion
<http://www.cs.cmu.edu/afs/cs/academic/class/15745-s09/www/lectures/lect20-pre.pdf>.
- [3] Trevor Jim, Greg Morrisett, Dan Grossman, and Michael Hicks. Abstract cyclone: A safe dialect of c.
- [4] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *In Second Workshop on Compiler Support for System Software*, pages 25–35, 1999.
- [5] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured documentation.
- [6] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Taming c pointers.
- [7] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.
- [8] AJ Shankar. Loop optimization for ccured.
- [9] Oiwa Yutaka. *Implementation of a Fail-Safe ANSI C Compiler*. PhD thesis.