# C-Mix: Making Easily Maintainable C-Programs run FAST

The C-Mix Group*, DIKU, University of Copenhagen

**Abstract**

C-Mix is a tool based on state-of-the-art technology that solves the dilemma of whether to write easy-to-understand but slow programs or efficient but incomprehensible programs. C-Mix allows you to get the best of both worlds: you write the easy-to-understand programs, and C-Mix turns them into equivalent, efficient ones. As C-Mix is *fully automatic*, this allows for faster and more reliable maintenance of software systems: system programmers need not spend hours on figuring out and altering the complicated, efficient code.

C-Mix is a *program specializer:* Given a program written in C for solving a general problem, C-Mix generates faster programs that solve more specific instances of the problem. Application areas include model simulators, hardware verification tools, scientific numerical calculations, ray tracers, interpreters for programming languages (Java bytecode interpreters, task-specific interpreters), pattern matchers and operating system routines.

C-Mix currently runs on Unix systems supporting the GNU C compiler, and treats programs strictly conforming to the ISO C standard. Future releases of C-Mix are intended to run on a variety of platforms.

## 1 Program specialization and partial evaluation

C-Mix performs program specialization by a technique called *partial evaluation* [2]. Given a source program and some of its input (the *specialization-time* data), it produces a so-called *residual* or *specialized* program. Running the residual program on the remaining input (the *residual-time* data) yields the same results as running the original program on all of its input; but potentially faster.

We use the word *spectime* to denote values and variables that are present at specialization time. The other values and variables in the program are *residual*.

C-Mix generates specialized versions of functions, unrolls loops, unfolds function calls and precomputes expressions and control constructs that does not depend on residual data. These transformations are similar to what optimizing compilers do, but since C-Mix takes some of the program's input into account, it can potentially do better. In addition, partial evaluation is based on inter-procedural analyses (including inter-procedural constant propagation), whereas most optimizing compilers only use intra-procedural analyses.

**Generality versus efficiency and modularity:** One often has a class of similar problems which all must be solved efficiently. One solution is to write many small and efficient programs, one for each. Two disadvantages are that much programming is needed, and maintenance is difficult: a change in outside specifications can require every program to be modified.

Alternatively, one may write a single highly parameterized program able to solve any problem in the class. This has a different disadvantage: *inefficiency*. A highly parameterized program can spend most of its time testing and interpreting parameters, and relatively little in carrying out the computations it is intended to do.

Similar problems arise with highly modular programming. While excellent for documentation, modification, and human usage, inordinately much computation time can be spent passing data back and forth and converting among various internal representations at module interfaces.

To get the best of both worlds: write only one highly parameterized and perhaps inefficient program; and *use a partial evaluator to specialize* it to each interesting setting of the parameters, automatically obtaining as many customized versions as desired. All are faithful to the general program, and the customized versions are often much more efficient. Similarly, partial evaluation can remove most or all the interface code from modularly written programs.

*Contact: Arne Glenstrup, Henning Makholm, or Jens Peter Secher: {panic,makholm,jpsecher}@diku.dk

C-Mix supports code that is *strictly conforming* to the ISO C standard. That is, code that depends on the details of data representation on the target platform (*e.g.*, mixes pointers and integers) will not be handled by C-Mix. However, in well-designed code these platform dependencies will usually be isolated in certain source files which can readily be excluded from the set of files that C-Mix specializes.

## 2 A simple example: specializing `printf` and `power`

Consider a source program containing a simplified implementation of the C library functions for computing the power function and formatting a string. Specializing the statements `v[0] = power(n,x);` `printf("Power = %d\n", v);` to fixed `n`=5 will yield the residual statements `v[0] = power_5(x);` `printf_res(v);`, where functions `power_5` and `printf_res` are defined in the residual program:

*Source Program*

```
void printf(char *fmt, int *values)
{ /* print formatted data */
  int i, j;
  /* Parse the format string */
  for (i = j = 0; fmt[i] != '\0'; i++) {
    if (fmt[i] != '%')
      putchar(fmt[i]);
    else {
      i++;
      switch (fmt[i]) {
      case 'd': /* %d: output int */
        sys_printf("%d", values[j]);
        j++; break;
      case '%':
        putchar('%'); break;
      default:
        putchar(fmt[i]); break;
  }}}
}

int power(int n, int x)
{ /* Return the nth power of x */
  int pow;

  pow = 1;
  while (n > 0) {
    pow = pow * x; n--;
  }

  return pow;
}
```

*Residual Program*

```
void printf_res(int *values)
{

  putchar('P');
  putchar('o');
  putchar('w');
  putchar('e');
  putchar('r');
  putchar(' ');
  putchar('=');
  putchar(' ');
  sys_printf("%d", values[0]);
  putchar('\n');









}

int power_5(int x)
{
  int pow;

  pow = 1;
  pow = pow * x; pow = pow * x;
  pow = pow * x; pow = pow * x;
  pow = pow * x;

  return pow;
}
```

Note that the variable `n` has been specialized away: The calls to the specialized versions of `power` and `printf` now only take the residual variable as argument. The loop in the `power` function has disappeared and 5 multiplications remain. The loop in the `printf` function interpreting the format string has also been specialized away leaving only the code with side effects.

This residual program is much faster than the original general one, since the testing and updating of the loop variables has been eliminated.

## 3 A more subtle example: binary search

Binary search is an algorithm for detecting whether a sorted array contains a given element `x`. The problem is solved by keeping track of a range within the array in which `x` must lie, if it is there at all. Initially the range is the entire array, and in each iteration the range is halved. By comparing the middle element of the range to `x`, it is decided whether to continue looking in the upper or lower half. The process continues until `x` is found or the range becomes empty. The time to search an array of size $n$ is $O(log(n))$.

Instead of representing the range by its lower and upper values as in the "classical" version, the range is represented by its lower value `low` and an increment `mid`, thus *separating* the computations for the position of the range and its size. As usual, we ensure that the increment takes only powers of 2 as values.

```
int bsearch(int x, int *a)
{
    int mid = 512;
#pragma residual: bsearch::low
    int low = -1;

    if (a[511] < x)
        low = 1000 - 512;
    while (mid != 1) {
        mid = mid / 2;
        if (a[low + mid] < x)
            low += mid;
    }
    if (low + 1 >= 1000 || a[low+1] != x)
        return -1;
    return low + 1;
}
```

Specialization with respect to "no" spectime input may seem useless, but notice that the size of the array (1000) is "hard-coded" into the program. Thus, some data is present in the function already at specialization time.

The variable `low` can take 1000 different values, depending on the value of `x` and the array contents, so specialization with respect to this variable is likely to produce an enormous amount of residual code. To avoid this we can insert the C-Mix directive `residual` in the source program, which prevents C-Mix from trying to keep track of `low`'s values at specialization time. Note that this does not make `mid` residual, exactly because `mid` does not depend on `low` like in the "classical" implementation of the binary search algorithm.

Running this example through C-Mix we find that the residual program is small, and the division calculations involving `mid` have all been specialized away:

```
int bsearch(int x, int *a)
{
    int low;

    low = -1;
    if (a[511] < x)       low  = 488;
    if (a[low + 256] < x) low += 256;
    if (a[low + 128] < x) low += 128;
    if (a[low + 64] < x)  low += 64;
    if (a[low + 32] < x)  low += 32;
    if (a[low + 16] < x)  low += 16;
    if (a[low + 8] < x)   low += 8;
    if (a[low + 4] < x)   low += 4;
    if (a[low + 2] < x)   low += 2;
    if (a[low + 1] < x)   low += 1;
    if (low + 1 >= 1000 || a[low + 1] != x) return -1;
    else                                    return low + 1;
}
```

Had we specialized the program without the `residual` directive, the result would have been a rather large program (around 114 Kb), because the function is specialized with respect to many different values of the two spectime variables. The first few lines of the residual `bsearch` function look like this:

```
int bsearch(int v1, int *(v2))
{
  if (((v2)[511]) < (v1)) {
    if (((v2)[744]) < (v1)) {
      if (((v2)[872]) < (v1)) {
        if (((v2)[936]) < (v1)) {
          if (((v2)[968]) < (v1)) {
            if (((v2)[984]) < (v1)) {
              if (((v2)[992]) < (v1)) {
                if (((v2)[996]) < (v1)) {
                  if (((v2)[998]) < (v1)) {
                    if (((v2)[999]) < (v1)) {
                      return -1;
                    } else {
                      if ((0) || (((v2)[999]) != (v1))) {
                        return -1;
                      } else {
                        return 999;
                      }
                    }
                  } else {
                    if (((v2)[997]) < (v1)) {
                      if ((0) || (((v2)[998]) != (v1))) {
                        return -1;
                      } else {
                        return 998;
                      }
                    }
```

The table below shows the run-times of the various versions of binary search we have seen. The function was called 1000 times. The runtime is shown in CPU user seconds, and the code size is the size of the object file measured by `size`. The speedup is the ratio between the running time for the original and the specialized program; similar for the code size blowup. The programs were compiled with the Gnu C-compiler `gcc` with option `-O2`, and the programs were executed on an HP9000/735.

| Program | Runtime (sec) | | | Code size (bytes) | | |
|---|---|---|---|---|---|---|
| | Orig | Spec | Speedup | Orig | Spec | Blowup |
| bsearch2 (residual `low`) | 7.7 | 5.2 | 1.5 | 96 | 200 | 2 |
| bsearch1 (spectime `low`) | 7.7 | 4.6 | 1.7 | 96 | 24520 | 255 |

The cost of forcing `low` to be residual is a increase in running time by 0.6 seconds (or 13 percent), which is acceptable since the specialized program is 100 times smaller! Careful inspection of the programs and other experiments show that the size of the residual program in the first example is proportional to the size of the array, whereas it is proportional to the logarithm of the size in the second example.

This experiment shows both a strength and a weakness of partial evaluation. It is pleasing that we can achieve a good speedup despite the modest code blowup, but it requires some insight to discover that the variable `low` should be residualized. However, once the `residual` directive has been added, the program can be *automatically* specialized whenever the initial value of `mid` and the array size changes.

Finally it is worth mentioning that it is also possible to specialize binary search with respect to a known array. In that case even higher speedups can be expected. Experiments show speedups of 2.8 and 3.7 and code blow-ups of 385 and 363 [1]. Code blowup of this size is quite acceptable, if it results in such good speedups, and the table is not too big.

## 4 Tool structure

C-Mix works together with existing C and C++ compilers when producing the residual program. The C-Mix core system produces from the source program a C++ program text called the *generating extension*. When the generating extension is run (after having been compiled with a C++ compiler) it reads in the specialization-time inputs and emits the final C source for the residual program.

There are several advantages of this approach. The first is that since the spectime actions are performed by the generating extension, C-Mix does not itself have do know how to execute them—that is left for the C++ compiler to decide. (Many earlier partial evaluators were "monolithic" and needed in effect to contain an interpreter for the source language to be able to perform specialization-time actions).

Second, the user of C-Mix can link the generating extension together with object code of his own. That means that the user can provide functions that can be called at specialization time, without C-Mix having to analyse and understand their source code. This is particularly convenient if the spectime input has a high-level format that have to be parsed before it can be used. C-Mix performs rather badly when faced with the output of parser generators such as `yacc`—but since none of the parser code is supposed to be present in the residual program anyway, the parser can be compiled as-is and linked into the generating extension without confusing C-Mix.

After having run the generating extension the finished residual program has to be compiled with a normal C compiler. Our experience is that the transformations done by C-Mix works well together with optimizing C compilers. Rather than performing optimizations which the compiler would have done in any case, partial evaluation seems to open up new possibilities for compiler optimization. This is mainly because the control structure of the residual program is often simpler than that of the source program, and because, generally, the residual program contains less possibilities of aliasing.

## 5 Conclusion

C-Mix is a partial evaluator for specializing C programs. It is automatic and handles all C programs that are strictly conforming to the ISO C standard. Using C-Mix one can obtain *efficient* programs from larger but *easy-to-read* programs, and in this way reduce the slow and error-prone job of maintaining highly efficient but highly unreadable software.

C-Mix is available free of charge from DIKU. The project's home page is http://www.diku.dk/research-groups/topps/activities/cmix.html

## References


[1] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[2] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.