Optimizing Applications with `gcc` & `glibc`

Ulrich Drepper*
Cygnus Solutions
Sunnyvale, CA

August 9th, 1999

*drepper@cygnus.com

Contents

1 What is this about?

Most programmers think that the programs they write are fairly well written and perform as good as possible. In most cases this is not correct. Many people *think* they know how to do it right but they most probably miss something. Writing optimized programs is a learning process. One learns about a new technique every time one looks sharply at the own code and thinks about the interactions with the underlying libraries or the processor.

In this paper we will discuss several optimization techniques which the author learned in the past years of programming. It is certainly not a complete list nor is it a structured approach. It is merely a list of different techniques documented using illustrative examples. All examples are given in C but most of the techniques can be applied to C++ as well. Using templates as allowed in standard C++ is not at all covered.

This paper is *not* about optimizations performed in the optimizer of the compiler. This is a completely different field and a lot of literature exists describing the possibilities. What we will describe are changes to the source code and ways to find out where they are needed. The remainder of the paper is structured in five parts:

- Using optimizations performed at compile-time (starting at page 3).
- Helping the compiler to generate better code (starting at page 12).
- Knowing the libraries and understand the function implementation (starting at page 17).
- Writing better code in the first place (starting at page 30).
- Profiling (starting at page 34).

To read and understand the following text it is necessary that the reader knows how the preprocessor works, has advanced knowledge about the C programming language itself, and preferably knows the functions of the C language. The text also describes a few machine specific optimizations but it is OK to ignore these points if one is not familiar with the described architecture.

2 Using Optimizations Performed at Compile-Time

In this section we will describe optimizations which can always be performed without the fear of negative impacts. The optimizations are performed a compile time by the compiler. The worst case is that the code behaves as if no optimization at all is performed. Therefore these kind of optimizations should always be performed since they don't have a negative impact. But it should perhaps happen as the last step since it might disable some of the other optimizations.

2.1 Dead Code Elimination Works

Unexperienced users are often afraid of leaving dead code behind. Dead code is used to describe code which never gets executed. In most cases this fear is

unfounded. The compiler will in most cases recognize dead code and completely drop it from the program. When using gcc this is only true if optimization is enabled but this his hopefully always true. gcc generates truly horrible code when no optimization is enabled.
One not very serious example is this:

```
long int
add (long int a, void *ptr, int type)
{
   if (type == 0)
     return a + *(int *) ptr;
   else
     return a + *(long int *) ptr;
 }
```

Depending on the parameter `type` the object pointed to by the parameter `ptr` is either an `int` or a `long int`. On the first view the code makes perfect sense and seems to be optimal. But it is not since in some environments `int` objects and `long int` objects are actually the same. In this case the `if` and the `else` clause of the function execute exactly the same code. This can be avoided at no extra cost and the solution will work if `int` and `long int` are not the same.

```
long int
add (long int a, void *ptr, int type)
{
   if (sizeof (int) == sizeof (long int) || type == 0)
     return a + *(int *) ptr;
   else
     return a + *(long int *) ptr;
 }
```

The result of the `sizeof` operator is *always* known at runtime and therefore the added conditional expression always can be computed by the compiler. If the types have the same size the expression evaluates to true and therefore the `if` condition always is true. In this case the `else` clause is never used and the compiler can recognize this and completely elide it. If the types have different sizes the code is equivalent to the initial code of the example.

When writing low-level programs which are expected to run on a variety of platforms one often comes across situations like the above. It is important to remember tricks like the one just described. In some situations it is not possible or wanted to actually add something to the C code. To get the correct result one must the preprocessor to hide certain code complete. In this case one cannot use the `sizeof` operator. Instead one should use the macros from `limits.h`:

```
#include <limits.h>

long int
add (long int a, void *ptr, int type)
{
#if LONG_MAX != INT_MAX
   if (type == 0)
     return a + *(int *) ptr;
```

```
    else
#endif
      return a + *(long int *) ptr;
  }
```

Here the preprocessor does the work. In this example it is not necessary but it shows how it works. The preprocessor is required to be able to perform arithmetic operations and comparisons using the widest available representation (at least with 64 bits). Which version is the better cannot be answered clearly.[1] From the compiler perspective both ways are nearly identical since dead code elimination works.

2.2 Saving Function Calls

If the body of a function is very small the overhead associated with the function call can be really high compared to the time spend in the function. Typical examples of this in the standard library are many of the string and math functions. There are two solutions to this problem: to use macros or to use inline functions to implement the function in question.

The GNU CC manual says that inline functions are as fast as macros and much safer. This is true, in most cases. There are examples where the equivalent macro is faster and macros can, using other `gcc` features, be made as safe as inline functions. Generally the suggestion is to use inline functions as long as none of the optimizations depend on the use of a macro. Two reasons are:

- The use of `alloca` (see page 23).
- The use of `__builtin_constant_p`, see page 9.

In either case there are a few things which the programmer has to take care of. When using inline functions it is not necessary to treat parameters specially. The handling of parameters happens as in normal functions, i.e., each expression used for the parameters is evaluated exactly once. If the header containing the inline function is used outside the own project (e.g., if the project is a library) than it is important to make sure that used identifiers to no conflict with macros defined by the user and the system. The C library implementation therefore prepend `__` to all identifiers is situations like this. A user application cannot legally do this. One reasonably safe way is to append `__`.

A last problem with inline functions is that they are not always used if they are declared `extern`. The GNU C compiler never expands inline functions if no optimization is enabled. It also allows to disable the inline function use explicitly even if it performs all other optimizations. This means for the use of inline functions that one has to take this case into account and always provide real, non-inline implementations of these functions as well. If the compiler expands the inlines these functions might never be used. If one puts the code for these functions in an archive and then links against this archive instead of the object directly, the linker will add these objects to the final program only if the functions are really needed. Inline functions defined as `static` are always expanded but this overrides the judgment of the compiler whether it

[1]Some people are strongly opinionated. The guys at USL have not even implemented `#if` in the Plan 9 C compiler making it impossible to write it in the latter form.

is worthwhile to inline the function or not. Especially with the use of options like `-Os` (optimize for space) it is questionable whether `static` inline functions should be used.

Writing correct and safe macros is much harder. First of all, one has to protect the parameters. It is important to remember that the passed values for the parameters are passed verbatim, without evaluation to the places where the variable appears. This requires

- correctly adding parenthesis. One must always be prepared for the case where the parameter is not a simple value and variable.

  ```
  #define mult(a, b) (a * b)

  {
    int a = mult (1 + 2, 3 + 4)
  }
  ```

  The erroneous definition of `mult` above leads to the surprising result of 11 for `a`. The correct form is

  ```
  #define mult(a, b) ((a) * (b))
  ```

- that braces to introduce new blocks are not used directly. The following example shows a problem situation:

  ```
  #define scale(result, a, b, c) \
    {                                    \
      int c__ = (c);                     \
      *(result) = (a) * c__ + (b) * c__;   \
    }

  {
    int r;
    if (s != 0)
      scale (&r, e1, e2, s);
    else
      r = 0;
  }
  ```

  The surprising result of the expression above is that `result` always gets the value zero assigned. Correctly written the `mult` macro would not cause this problem:

  ```
  #define scale(result, a, b, c) \
    do {                                 \
      int c__ = (c);                     \
      *(result) = (a) * c__ + (b) * c__;   \
    } while (0)
  ```

- The above example already shows the next important point: don't introduce unwanted side effects by evaluating the expressions passed as parameters more than once. This is why the variable `c__` was introduced. Without the variable the expression would have been evaluated twice. There is also the problem that a parameter value is not evaluated at all. If the macro is meant as a replacement for a function this is also a problem. Therefore the `scale` macro is *not* written like this:

```
#define scale(result, a, b, c) \
  do {                                              \
    int c__ = (c);                                  \
    *(result) = c__ == 0 ? 0 : (a) * c__ + (b) * c__; \
  } while (0)
```

This could safe some time but would lead to strange results (compared to a real function) if the values passed for the second and third parameter have side effects.

- Also visible in the `scale` example is the problem macros have with non-trivial expressions. As soon as a variable is needed one cannot use simple expressions anymore. One has create a new block which of course means the macro cannot have a return value. One has to use awkward methods as in the `scale` example where the variable, the result has to be stored in, is given as a parameter. It would be much cleaner if the macro would return the value and one could assign it. This is not possible in standard C, but it is possible in GNU C.

```
#define scale2(a, b, c) \
  (__extension__
    ({                                \
      int c__ = (c);                  \
      (a) * c__ + (b) * c__;          \
    }))

{
  int r;
  if (s != 0)
    r = scale (e1, e2, s);
  else
    r = 0;
}
```

The GNU C feature used here is called “statement expression” and is described in the GNU CC manual. It basically is a normal block with the exception that the value of the last statement is passed up as the result of the expression. Please note that it is not necessary to use the `do ... while (0)` trick.

There remains one new feature introduced in the last example to be explained. The `__extension__` keyword added tells the compiler that the author knows s/he uses a GNU C extension. Therefore the compiler does not issue a warning even if it is asked to point out all ISO C violations. Therefore `__extension__` should be used in all headers which can be used outside the project.

2.3 Compiler Intrinsics

Most modern C compilers know intrinsic functions. These are special inline functions, which are provided by the compiler itself. Unlike inline functions they are always used, the compiler cannot opt for using an external implementation. Intrinsics known to `gcc` as of version 2.96 are:

- `__builtin_alloca`
  dynamically allocate memory on the stack
- `__builtin_ffs`
  find first bit set
- `__builtin_abs`, `__builtin_labs`
  absolute value of an integer
- `__builtin_fabs`, `__builtin_fabsf`, `__builtin_fabsl`
  absolute value of floating-point value
- `__builtin_memcpy`
  copy memory region
- `__builtin_memcmp`
  compare memory region
- `__builtin_memset`
  set memory region to given value
- `__builtin_strcmp`
  compare two strings
- `__builtin_strcpy`
  copy string
- `__builtin_strlen`
  compute string length
- `__builtin_sqrt`, `__builtin_sqrtf`, `__builtin_sqrtl`
  square root of floating-point value
- `__builtin_sin`, `__builtin_sinf`, `__builtin_sinl`
  sine of floating-point value
- `__builtin_cos`, `__builtin_cosf`, `__builtin_cosl`
  cosine of floating-point value
- `__builtin_div`, `__builtin_ldiv`
  integer division with rest
- `__builtin_fmod`, `__builtin_frem`
  module and remainder of floating-point division

There are a few more intrinsics but they are very useful. It is not guaranteed that all intrinsics are defined for all platforms. Therefore one must be prepared for the case that an intrinsic is not available and one has to use a real implementation.

One important and very useful feature of some intrinsic functions is that they can compute their results at compile-time if the parameters are constant at compile-time. E.g., it is possible that

```
strlen ("foo bar")
```

is directly replaced with the value seven. This is something which we will use in the remainder of this paper off and on.

2.4 `__builtin_constant_p`

Though the name `__builtin_constant_p` looks very much like the names of the intrinsic functions mentioned in the last section it is no intrinsic. It is instead an operator similar to `sizeof`. Since it follows the good old LISP tradition to use the ending `_p` one can see from the name that it is a predicate. It takes a single parameter and the return value is nonzero if the parameter value is constant at runtime.

This proofs to be a very useful thing to have. Many of the optimizations in the remainder of this text as well as many of the optimization implemented in the GNU C library headers depend on this feature. To show how it is used we continue the example from page 4. When third parameter is constant the type of the object pointed to by the second parameter can be deduced at compile time. Therefore we add in addition to the improved implementation above in the header with the prototype of this function the following macro:

```
#define add(a, ptr, type) \
  (__extension__                                     \
    (__builtin_constant_p (type)                     \
     ? ((a) + ((type) == 0                           \
               ? *(int *) (ptr) : *(long int *) (ptr)))  \
     : add (a, ptr, type)))
```

This macro changes the behavior of the `add` function only if the third parameter is constant. If it is not constant the real implementation is called.[2] Otherwise, the expression `(type) == 0` can be evaluated at compile time at the whole expression evaluates to either

```
(a) + *(int *) (ptr)
```

or

```
(a) + *(long int *) (ptr)
```

In this small example it might not be visible but the `__builtin_constant_p` operator allows to avoid code bloat in situation where a macro definition only leads to favorable code if due to compile-time computations, value propagation, and dead code elimination the code size is reduced drastically. The following real-world example shows this more clearly.

[2]It is hopefully clear why despite the call to `add` in the last line this is no recursive call to the macro.

```
#define strdup(s) \
  (__extension__                                             \
    (__builtin_constant_p (s) && __string2_1bptr_p (s)     \
     ? (((__const char *) (s))[0] == '\0'                  \
        ? (char *) calloc (1, 1)                           \
        : ({ size_t __len = strlen (s) + 1;                \
             char *__ret = (char *) malloc (__len);        \
             if (__ret != NULL)                            \
               __ret = (char *) memcpy (__ret, s, __len);  \
             __ret; }))                                    \
     : strdup (s)))
```

The use of `__builtin_constant_p` prevents the use of the whole bunch of lines of this macro if the parameter is not constant. For the sake of it, one should once look at the code to see how much code would have to be generated if one would always use the replacement. We simply could not write such a macro without `__builtin_constant_p` guarding the expansion (the `__string2_1bptr_p` is not interesting here; interested parties should look at the `<bits/string2.h>` header of a `glibc` 2.1 installation).

For a constant parameter `s` (which must be a string constant) most of the expression can be computed at runtime. Since the computer can see the first character of the string it knows whether the `calloc` call has to be made or whether the statement expression has to be executed. In the statement expression the result of the `strlen` call can be determined at compile-time. One can see that in either case the piece of code which remains a very small. Due to the optimizations performed by the compiler many of the operations of the code do not have be executed at runtime. In the above case, for a non-empty string, one would in the end have two function calls and an `if` expression but it would not be necessary to compute the string length which might be a big advantage.

2.5 Type-generic Macros

When writing a macro to help speeding up certain operations it is sometimes the case that one wants the same functionality for different type. For simple operations this is easy, one simply can let the compiler figure out how to use the arguments. It gets complicated as soon as one has to define variables inside the macro and if different functions depending on the used type have to be used.

Going back to the `scale` example, we might want to write a type-generic version of it. Instead of requiring the parameter `c` to be of type `int` we make it of whatever type the other parameters are. The compiler can help to figure this out. Since the result of operations on two numbers is of the larger type of the two operands the type we want to use for `c` is the same type as the one for `a + b + c`. Using a feature in `gcc` it is possible to define such a variable:

```
#define tgscale(result, a, b, c) \
  do {                                                      \
    __extension__ __typeof__ ((a) + (b) + (c)) c__ = (c);  \
    *(result) = (a) * c__ + (b) * c__;                     \
  } while (0)
```

This might be a bit confusing at first. But `__typeof__` (*o*) defines a type and it is the same type *o* has. Therefore the changed line as before defines a variable `c__` and assigns a value to it. What we gained through this change is that we now can call `tgscale` with arguments of type `int`, `long long int` and even `double` and we always get `__c` to be defined the best way with respect to the result.

There is another situation where `__typeof__` helps writing code even if not many variables are used and the common type has to be deduced. The ISO C9x standard introduces a new header `<tgmath.h>` allows the user to write code using the mathematical functions without taking care of the different variants for the different types. E.g., one can call `sin` and depending on the type of the parameter the correct function from the six possibilities is picked. A naïve approach to this problem would be:

```
#define sin(Val) \
  (sizeof (__real__ (Val)) > sizeof (double)      \
   ? (sizeof (__real__ (Val)) == sizeof (Val)     \
      ? sinl (Val) : csinl (Val))                 \
   : (sizeof (__real__ (Val)) == sizeof (double)  \
      ? (sizeof (__real__ (Val)) == sizeof (Val)  \
         ? sin (Val) : csin (Val))                \
      : (sizeof (__real__ (Val)) == sizeof (Val)  \
         ? sinf (Val) : csinf (Val))))
```

But this is not correct. Try to find out yourself before reading the footnote and looking at the correct code.[3] To correct the problem we have to introduce a variable and this is where `__typeof__` comes into play again.

```
#define sin(Val) \
  (__extension__                                          \
    ({ __typeof__ (Val) __tgmres;                         \
       if (sizeof (__real__ (Val)) > sizeof (double))     \
         {                                                \
           if (sizeof (__real__ (Val)) == sizeof (Val))   \
             __tgmres = sinl (Val);                       \
           else                                           \
             __tgmres = csinl (Val);                      \
         }                                                \
       else if (sizeof (__real__ (Val)) == sizeof (double))  \
         {                                                \
           if (sizeof (__real__ (Val)) == sizeof (Val))   \
             __tgmres = sin (Val);                        \
           else                                           \
             __tgmres = csin (Val);                       \
         }                                                \
       else                                               \
         {                                                \
           if (sizeof (__real__ (Val)) == sizeof (Val))   \
             __tgmres = sinf (Val);                       \
           else                                           \
             __tgmres = csinf (Val);                      \
```

[3]The above is a single expression and it must have exactly one statically determined type. This type must be the most general one to be able to represent values of all the other types without loss. Therefore the returned value is always of type `complex long double` which certainly is not what we want.

```
    }                                                          \
  __tgmres; }))
```

This example summarizes almost everything we discussed so far. If you would have been able to write this code yourself you have learned the lessons. Otherwise a few explanations. Because the code now is not a single expression the compiler does not find the most general type. In fact, the assignment to the variable `__tgmres` might force a conversion to a narrower type. But this never happens for any reached code: only one if the six assignments is really executed and here the assignment does not loose any precision. All the other cases are dead code and will be eliminated by the optimizer.

3 Helping the Compiler

The GNU C compiler offers a few extensions which allow the compiler to describe the code more precisely and also features to influence the code generation. In this section we will describe some of the features in examples so that the reader can apply this later her/his own code.

3.1 Functions of No Return

Every bigger project contains at least one function which is used for fatal errors and which gracefully terminates the application. Such a function is often not treated in an optimal way since the compiler does not know that the function does not return. Take the following example function and the use in some code.

```
void fatal (...) __attribute__ ((__noreturn__));

void
fatal (...)
{
  ... /* Print error message. */ ...
  exit (1);
}

{
  ... /* read d */ ..
  if (d == 0)
    fatal (...);
  else
    a = b / d;
  ... /* and so on */ ...
}
```

The function `fatal` is guaranteed to never return since the function `exit` has the same guarantees. Therefore we annotated the prototype of the function with `__attribute__ ((__noreturn__))`. This `gcc` extension lets the author specify exactly what we just said: the function will never return.

Without this assurance the compiler would have to translate the `if` clause in the example to something which corresponds the following pseudo-code:

1 Compare `d` with zero
2 If not zero jump to 5
3 Call `fatal`
4 Jump to 6
5 Compute `b / d` and assign it to `a`
6 ...

This is not far from the optimum but there is unneeded code. The line 4 is never executed since the call to `fatal` does not return. If the compiler knows about this it will avoid this line and effectively transforms the source code to this:

```
{
  ... /* read d */ ..
  if (d == 0)
    fatal (...);
  a = b / d;
  ... /* and so on */ ...
}
```

Please note that the `else` is gone. This transformation would have been illegal without the knowledge about the behavior of `fatal`. Even if this is no big improvement and does not happen that frequently one should always think about marking function this way. The compiler will emit warnings about unreachable code if one forgets about the behavior of the not-returning function and adds some extra code after the function call.

3.2 Constant Value Functions

Some functions one writes only depend on the parameters passed into it and they have no side effects. Let us call them *pure functions.* This property is for the compiler not visible from the prototype and so it always has to assume the worst, namely, that the function has side effects. But this is something which prevents certain optimizations.

As an example take the `htons` function which either returns its argument (if the machine is big-endian) or swaps the byte order (if it is a little-endian machine). There are no side effects and only the parameter is used to determine the result. `htons` clearly is a pure function.

If we now would have the following code we would get a less than optimal result:

```
{
  short int serv = ... /* Somehow find this out */ ...;

  while (1)
    {
      struct sockaddr_in s_in;

      memset (&s_in, '\0', sizeof s_in);
      s_in.sin_port = htons (serv);

      ... /* lots of code where serv is not used */ ...
    }
}
```

This might be the outer loop of a network application which opens a socket connection to various hosts one after the other. The port to connect to is always the same since the variable `serv` does not change. Since we said that `htons` is a pure function this means that in every iteration of the loop the result of call to `htons` is the same. What we would like to see is an automatic transformation of the code to something like this:

```
{
  short int serv = ... /* Somehow find this out */ ...;
  serv = htons (serv); /* Once and for all compute the port */

  while (1)
    {
      struct sockaddr_in s_in;

      memset (&s_in, '\0', sizeof s_in);
      s_in.sin_port = serv;

      ... /* lots of code where serv is not used */ ...
    }
}
```

It is possible to achieve this by marking the `htons` function appropriately. `gcc` allows to give a function the attribute `__const__` which tells the compiler that the function is pure. I.e., if we would have added

```
extern uint16_t htons (uint16_t __x) __attribute__ ((__const__));
```

before the code of the initial example we would have given the compiler the opportunity to generate the code we want. Marking pure functions using `__const__` can mean quite an improvement and will and no case lead to worse code. Therefore one should always think about this optimization.

3.3 Different Calling Conventions

Each platform has specific calling conventions which makes it possible that programs/libraries written by different people with possibly different compilers can work together. These calling conventions were defined when the platform was young and maybe even the architecture/processor behaved differently.

Anyhow, there might be platforms and situations where one wants to use a different calling convention the compiler supports because it is more efficient. This is not possible in general but nobody can forbid using this internally in a project. If only functions which are never called outside the project are defined with a different calling convention there is no problem with this.

Especially on the Intel ia32 platform there are a few calling conventions supported by the compiler which are different from the standard Unix x86 calling conventions and which can occasionally sped up the program significantly. Other platforms might allow similar changes. The GNU C compiler manual explains the details. For this section we will restrict the descriptions to the x86 platform.

Changes to the calling conventions can be made in two ways: generally change the conventions by a command line option or individually change it via a function attribute. We will discuss only the latter since using a command line option is unsafe because exported functions might be effected or the command line option might be missing in another compiler run. One should always prefer function attributes.

3.3.1 `__stdcall__`

The first attribute changes the way the memory used to pass parameters is freed. Parameters are normally passed on the stack and at some point the stack pointer has to be adjusted to take this into account. The standard calling conventions on ia32 Unix is to let the caller correct the stack. This allows delaying the stack correction so that the effect of more than one function call can be corrected at once. On the other hand, if the `__stdcall__` attribute is defined for a function this signals that the function itself corrects the stack. This is not a bad idea on ia32 platforms since the architecture has a single instruction which allows returning from the function call and correcting the stack in one instruction. The effect can be seen in the following example.

```
int
__attribute__ ((__stdcall__))
add (int a, int b)
{
  return a + b;
}

int
foo (int a)
{
  return add (a, 42);
}

int
bar (void)
{
  return foo (100);
}
```

If this code gets translated the assembler output looks like this:

```
 8                    add:
 9 0000 8B442408              movl    8(%esp), %eax
10 0004 03442404              addl    4(%esp), %eax
11 0008 C20800                ret     $8
...
17                    foo:
18 0010 6A2A                  pushl   $42
19 0012 FF742408              pushl   8(%esp)
20 0016 E8E5FFFF              call    add
20      FF
21 001b C3                    ret
...
27                    bar:
28 0020 6A64                  pushl   $100
29 0022 E8E9FFFF              call    foo
29      FF
```

```
30 0027 83C404                 addl    $4, %esp
31 002a C3                     ret
```

Only the important lines are shown in the listing above. What has to be recognized is that the function `foo` does not have to do anything after the call of function `add`. The `ret` instruction in line 11 takes care of the memory need for the two parameters passed on the stack. The situation in function `bar` is different. Since `foo` is not marked with the `__stdcall__` attribute it does not free the memory and this has to be done in the caller. Therefore we see a stack pointer manipulation in line 30.

From this short example it seems that using `__stdcall__` only has advantages. Even the code generated is smaller (the size of the `ret` instruction increased by two bytes, but the `addl` instruction has three bytes). But this is not so. Since the compiler is clever and corrects the stack pointer for several function calls which are done is a row only once the gain in code size is not that big anymore. In addition to this future changes in the compiler will make handling the parameter allocation much faster and using `__stdcall__` will become counter-productive. Therefore this attribute should be used with care.

3.3.2 `__regparm__`

A more interesting function attribute is `__regparm__`. It is only available for the ia32 platform since most other platforms do not have a standard calling convention which would make this necessary.

Using the `__regparm__` attribute it can be specified how many integer and pointer parameters (up to three) are passed in registers instead of on the stack. This can make a significant difference especially if the work performed in the called function is not much and the parameters have to be used immediately. The following not very realistic example code shows this dramatically.

```
int
__attribute__ ((__regparm__ (3)))
add (int a, int b)
{
  return a + b;
}
```

Compiling this with optimizations leads to the following assembler output (only the important lines are shown).

```
 8                     add:
 9 0000 01D0                   addl    %edx, %eax
10 0002 C3                     ret
```

This is the optimal code which could be generated for the functions. Without the attribute the code would look as in the last section (see page 15). Using this way to pass parameters almost always has advantages. In the worst case the called function will store the variables itself somewhere on the stack if it needs the registers for some other computations. But in man cases the values can be used directly from the registers.

3.4 Sibling Calls

One optimization performed automatically by the compiler for many platforms is sibling call optimization. If a function call is the last thing happening in a function the code which usually gets generated looks like this:

	This is inside function F1
n	call function *F2*
$n+1$	execute code of *F2*
$n+2$	get return address from call in *F1*
$n+3$	jump back into function *F1*
$n+4$	optionally adjust stack pointer from call to *F2*
$n+5$	get return address from call to *F1*
$n+6$	jump back to caller of *F1*

This is not surprising but often not optimal. If no work has to be done in step $n+4$ or if this work can be moved before the call to *F2* the return from *F2* stops in *F1* only to jump again. It would be much better if the return from *F2* would immediately end up at the caller of *F1*.

To do this the subroutine call performed in *F1* must be changed into a jump which does not store a new return address. This way the return address normally used in *F1* would be used in *F2* and the result would be exactly as we want.

The compiler already performs this optimization occasionally and in future will do it more often. The consequence for the programmer is that s/he should try to arrange the code so that function calls happen as the last thing in a function.

3.5 Using `goto`

Despite what Dijkstra says, using the `goto` command sometimes has advantages. One should, though, not use it without knowing the effect. Using `goto`s definitely disturbs the compiler to some extend since the various flow analysis mechanisms don't work so well anymore.

Before using `goto` one should look at the generated assembler code. It will also be necessary to understand the branch prediction mechanism of the processor. Equipped with this knowledge one can insert in strategic places `goto`s where normally the translation of a loop or conditional statement would lead to different code. `goto`s also might enable writing loops differently. With `goto` it is possible to leave a loop at an arbitrary point.

It is not possible to give a general advice when to use `goto`. But if one decides to use it one should make sure to take measurements of the runtime. Normally the compilers do a quite good job and adding `goto`s might even hurt. For a concrete example how to use `goto` see page 32.

4 Knowing the Libraries

Programs are normally not written stand-alone. Instead they take advantage of libraries which already exist on the system. The most important library is the C library whose minimal content is defined in the ISO C standard. Other standards and system-dependent extension broaden the range of available functions.

A good programmer knows at least most of the available functionality in the system libraries. But to truly master the programming it is also necessary to at least have a grasp of how the functions are implemented. This allows the programmer to avoid using those functions which are slower in favor of those which do a comparable job but perform faster.

In the remainder of this section we will see several examples of functions with similar functionality but different runtime characteristics. The reader should be able to apply the knowledge easily to own programs.

4.1 `strcpy` vs. `memcpy`

Handling strings is in C programs a frequent task. There is no string data type and so the operations have to performed by hand. This has the advantage that the programmer can take into account the overall use of the string and is not limited to implement the immediate need only. E.g., If three strings have to be concatenated it is not necessary to first allocate memory for the concatenating of the first two strings, copy over the two strings and then perform the concatenation with the third string. Instead it is possible to allocate immediately enough memory for the total result and then copy the strings.

This degree of freedom leads to very different approaches people take and most of them are all but optimal. The least people can do is to take the correct functions for the task.

At this point we only want to point out a few general points. Later sections will give concrete examples. Here we will discuss the differences between the `mem*` and the `str*` functions so that this knowledge can be used later.

For using the `mem*` functions one has to know the size of the region to copy. This is the case most of the time since a correctly written program does not simply copy a string of unknown length (this could lead to crashes and can open security holes). The main difference between the two function families is that the `str*` functions normally don't need a separate length counter but on the other hand the `mem*` functions know about the length of the region and therefore can perform the work word-wise. We will describe here the characteristics of the most important functions.

`strcpy`	`memcpy`
two values needed: source and destination pointer	three values needed: source and destination pointer as well as length count
works bytewise	can work word-wise

`strncpy`	`memcpy`
three values needed: source and destination pointer as well as length count	Likewise
two abort criteria: length reached and NUL byte found	one abort criteria: length reached
works byte-wise	can work word-wise
no `gcc` intrinsic	`gcc` intrinsic
fills rest of target buffer with NUL	just stops copying

What is here exemplified with memcpy and the string functions strcpy and strncpy is also true for several other function combinations (e.g., memchr and strchr). It is therefore important to study the available mem* and str* functions are try to find out where their functionality overlaps.

The number of values needed is not an issue when the actual library function is called since then the compiler can use the registers marked as call-clobbered. But it is a problem if the functions are inlined. In this case more used values means increased register pressure in the current function. For machines with small register sets it might mean spilling. But also from the algorithm's point of view more values *can* mean more complicated code. But this is not necessarily the case. In general is true that fewer used values are better then more.

Of importance as soon as the functions are used on strings which are not only a few characters long is whether the processing can happen word- or byte-wise. The str* functions all work byte-wise because the lengths of the strings are not known.[4] On the other hand the lengths of the memory regions handed over to the mem* functions are always known since they are given as an argument to the function call. A last important differentiator is the number of abort criteria. The more criteria there are, the more complex the loops are, the slower the code is. The table above mentions this for the
memcpy and strncpy functions: memcpy stops when all bytes are copied. The strncpy function has to check for the NUL byte *and* check the length parameter to not exceed the maximal number of characters to copy. The latter is clearly slower.

The recommendation for the use of the three functions above is therefore:

- Never use strncpy unless it fits exactly in the current situation. Normally the string size is known.
- If the strings to copy are known to be short use strcpy.
- If it is possible that the handled strings might be longer use memcpy. The use of memcpy will not hurt for short strings either since the performance difference for short strings is not big.

We will see in the remainder of this paper a few more examples of the use of mem* and str* functions. In many cases one can possibly gain a lot by careful analysis of the situation. But the rule of thumb, to use the mem* functions when possible, should lead to overall good results.

4.2 strcat and strncat

A golden rule of optimal string handling code is:

> Never ever use strcat and strncat!

It is always wrong to use these two functions. Before concatenating a string to another, one has to know whether there is enough room. For this it is necessary to know the current end of the existing string and the length of the string to be copied. But then it is completely unnecessary to use strcat. The following is similar to often seen code:

[4] This must be clarified a bit. "Working byte-wise" here means that every single byte must be examined. It is possible that str* functions read memory word-wise (which is possible with aligned accesses). But still every single byte must be tested for a NUL byte.

```
{
  char *buf = ...;
  size_t bufmax = ...;

  /* Add ‘s’ to the string in buffer ‘buf’.  */
  if (strlen (buf) + strlen (s) + 1 > bufmax)
    buf = (char *) realloc (buf, (bufmax *= 2));

  strcat (buf, s);
}
```

This looks quite nice from the programmers point of view but the `strcat` functions is expensive. What the function has to do first is to search for the end of the existing string. This is equivalent to the `strlen` call, it therefore duplicates work already done. Next the string `s` must be copied. But though it is known from the `strlen` call how long the string is the `strcat` function has to perform a normal string copy operation (see the last section for why this is not good). It is much better to write the code like this:

```
{
  char *buf = ...;
  size_t bufmax = ...;
  size_t slen;
  size_t buflen;

  /* Add ‘s’ to the string in buffer ‘buf’.  */
  slen = strlen (s) + 1;
  buflen = strlen (buf);

  if (buflen + slen > bufmax)
    buf = (char *) realloc (buf, (bufmax *= 2));

  memcpy (buf + buflen, s, slen);
}
```

This version counters both disadvantages of `strcat`: when copying we are not looking through the existing string again since we know how long it is and simply can add at the end. Also the copying happens using `memcpy` because we know how long the string `s` is. We will give two more examples on how to implement string concatenation by implementing two often needed functions, this time with error checking:

```
char *
concat2 (const char *s1, const char *s2)
{
  size_t s1len = strlen (s1);
  size_t s2len = strlen (s2) + 1;
  char *buf = (char *) malloc (s1len + s1len);

  if (buf != NULL)
    (void) memcpy (mempcpy (buf, s1, s1len), s2, s2len);

  return buf;
}
```

This code uses a function which is not defined by the ISO C or Unix standard and didn't appear so far. mempcpy, available in the GNU libc, works like memcpy. It copies the given number of bytes from the source to the target buffer. But instead of returning a pointer to the beginning of the buffer it returns a pointer just after the last copied byte.

In the code above it can be seen how this different behavior can be used. The returned value can be used immediately in the next function call since it is exactly the position where the next copy operation must be started. In addition, by returning the pointer which just was used for copying the mempcpy function can be implemented a bit faster than the memcpy function. The latter must return the beginning of the buffer which might not be in a register anymore.

What why is then the function memcpy used to perform the second copying in the example above if mempcpy is possibly faster? The first observation is that we don't need the return value and it therefore does not matter from the correctness standpoint which function is used. The answer to the question is: gcc knows about memcpy and has an intrinsic function but it does not (in the moment) know about mempcpy. Therefore the use of memcpy in some situations is faster.

Now for a bit more complicated example which does not allow such an easy argumentation as the function above. We implement a function which concatenates arbitrary many strings. The problem here is that we cannot easily save the lengths of the participating strings in a pair of variable. Or can we?

```
char *
concat (const char *s, ...)
{
  size_t nlens = 127;   /* Minimal maximal number of parameters.  */
  size_t *lens = (size_t *) alloca (nlens * sizeof (size_t));
  size_t cnt = 0;
  va_list ap;
  va_list ap_save;
  const char *cp;
  size_t total;
  char *retval;

  if (s == NULL)
    return (char *) calloc (1, 1);

  total = lens[cnt++] = strlen (s);
  va_start (ap, s);
  __va_copy (ap_save, ap);

  while ((cp = (const char *) va_arg (ap, const char *)) != NULL)
    {
      if (cnt == nlens)
        {
          size_t *newp = (size_t *) alloca ((nlens *= 2)
                                            * sizeof (size_t));
          lens = (size_t *) memcpy (newp, lens, cnt * sizeof (size_t));
        }
      total += lens[cnt++] = strlen (cp);
    }
```

```
  retval = (char *) malloc (total + 1);
  if (retval != NULL)
    {
      char *endp = (char *) mempcpy (retval, s, lens[0]);
      cnt = 1;

      while ((cp = (const char *) va_arg (ap_save, const char *))
             != NULL)
        endp = (char *) mempcpy (endp, cp, lens[cnt++]);

      *endp = '\0';
    }

  return retval;
}
```

This code might need a little bit of explanation. First, the dots in the parameter list this time really mean a variable length parameter list and not, as in earlier examples, that something is left out. We allow the function to take arbitrary many parameters (within the limits of the compiler, of course). All parameters must be strings except for the last one which must be a `NULL` pointer. Even only a `NULL` pointer is allowed.

Second, we are using the same method as in the `concat2` function: we first determine how much memory is needed, allocate it and then copy the strings. An alternative approach would be to enlarge the destination buffer on demand, possibly several times during the function run. While this is possible we don't implement the function this way since the resizing with the associated copying of existing content is very costly. Please note in the code above the user of the return value of `memcpy`

The problem the chosen implementation faces is that with the arbitrary number of parameters it is not so easy to remember all the string lengths which must be determined first to determine how much memory is needed. In addition we don't want to add any limitations. Therefore the function allocates memory for the string lengths on the fly. Since this information is not needed outside the function the memory can be allocated using `alloca` (for a detailed discussion of `alloca` see the next section). Even though this also involves copying it is not as bad as copying the string since a) all objects are of fixed width (while strings can be arbitrarily long) and b) because this probably never has to happen since concatenations of more than 127 strings at once are not often needed.[5]

Interesting, since not often used, is also the use of `__va_copy`. This allows portably to walk over a parameter list twice. It is not generally possible to simply assign two objects of type `va_list` to one another.

The rest of the function is easy. The special case of only a `NULL` pointer argument is handled early. The `calloc` call allocates a memory region of 1 byte length and initializes it to zero, which makes it a zero-length string. The code to copy the strings is without surprising. We are now using `mempcpy` for all the copy operations to always get a pointer to the following byte. Since we never copy the NUL byte terminating the string we must in the end explicitly terminate the string in the buffer which will be returned.

[5]Those who noticed that the handling of the array `lens` is not perfect since it wastes stack space are correct. But this is an example only.

Just for comparison, here is how the copying loop of an implementation which uses strcat could look like:

```
char *
concat (const char *s, ...)
{
  size_t cnt = 0;
  va_list ap;
  va_list ap_save;
  const char *cp;
  size_t total;
  char *retval;

  if (s == NULL)
    return (char *) calloc (1, 1);

  total = strlen (s);
  va_start (ap, s);
  __va_copy (ap_save, ap);

  while ((cp = (const char *) va_arg (ap, const char *)) != NULL)
    total += strlen (cp);

  retval = (char *) malloc (total + 1);
  if (retval != NULL)
    {
      strcat (retval, s);

      while ((cp = (const char *) va_arg (ap_save, const char *))
             != NULL)
        strcat (retval, cp);
    }

  return retval;
}
```

This looks much simpler. But this implementation is horrible. The complexity is $O(n \times m)$, where n is the average lengths of the strings and m the number of strings. The problem is that strcat has to scan over the existing text over and over again. The optimized implementation above has the expected complexity of $O(n)$. Even if the non-linear nature of the strcat based implementation does not kick in for few and small strings it is nevertheless noticeable even then. Hopefully this is enough evidence to proof the statement from the beginning of this section.

4.3 Optimized memory allocation

The example in the previous section already used two different kinds of memory allocation: conventional allocation (malloc, calloc) and stack-based allocation with alloca. The use of alloca isn't necessary and can easily be replaced by a malloc call. The question now is why should optimized programs use alloca wherever possible.

To answer the question it is necessary to understand how the two groups of functions work. The `malloc` group requests the memory it needs from the kernel for permanent use (until it is freed). Traditionally this allocation was on the so called head, an area designated by a break pointer which can be modified by the `sbrk` system call. Modern `malloc` implementations use on some systems for large memory areas a different method. They allocate the memory using `mmap`. This has on some systems the advantage that resizing is very cheap. In any case the new pieces of memory must be somehow noted in the internal data structures which the `malloc` implementation keeps to handle frequent freeing and re-allocating efficiently. At the minimum the size of the block must be remembered somewhere. A call to `malloc` is therefore not cheap. On a modern system one would have to allow at least 100 cycles. If the memory actually has to be retrieved from the kernel the number instantly rises to several thousand (one or two orders of magnitude more).

On the other hand the implementation of the `alloca` function is trivial. At least if the compiler directly supports it as an inline. This is what we assume throughout the whole paper. In this case the `alloca` call is a simple manipulation of the stack pointer. The stack pointer is corrected to leave the specified number of bytes, given in the arguments, between the last used object on the stack and the current stack pointer. The starting address of the block is the result of the function call. I.e., we are talking about a single assembler instruction. The `alloca` implementation therefore is two orders of magnitude faster than the optimal case of calling the `malloc` function.

If this is not reason enough, there is another big advantage. While the program must call `free` on the returned pointer, the memory allocated with `alloca` gets automatically recycled as soon as the function is left. The `free` call must not be underestimated. It is often more expensive than the `malloc` call since it has to enqueue the new block in the internal data structures and it has to see whether it has to return memory to the system. This can make the call very expensive.

This brings up the question why is there a `malloc` call if `alloca` has all the advantages. The question was already partly answered in the last paragraph. `alloca` can only be used if the memory block is only used in the current function of in functions called by it. The memory block is invalid as soon as the function returns from the current function. Therefore `malloc` must be used if the live range of the object must extend over the use of the current functions. Another limitation of `alloca` is that most systems install a not too generous limit of the stack size. This is done for safety reasons to catch unlimited recursion early. For `alloca` this means that large memory allocations must happen using `malloc` since the heap has much less restrictions. In addition the `malloc` implementation and the kernel can handle large allocations must better this way (at least on systems using `mmap`).
Now it's time for an example. `alloca` is extremely useful for making temporary copied. This is how it should not be done:

```
int
tempcopy (const int *a, int n)
{
  int *temp = (int *) malloc (n * sizeof (int));
  int_fast32_t cnt;
```

```
  int result;

  if (temp == NULL)
    return -1;

  for (cnt = 0; cnt < n; ++cnt)
    temp[cnt] = a[cnt] ^ 0xffffffff;

  result = foo (temp, n);

  free (temp);

  return result;
}
```

As discussed above the `malloc` call is much more expensive and we also need a `free` call. The following is better:

```
int
tempcopy (const int *a, int n)
{
  int *temp = (int *) alloca (n * sizeof (int));
  int_fast32_t cnt;

  for (cnt = 0; cnt < n; ++cnt)
    temp[cnt] = a[cnt] ^ 0xffffffff;

  return foo (temp, n);
}
```

This function is not only faster, it is also smaller due to the two dropped function calls. And it could allows more optimization due to the sibling function call at the end. And there is one more point: it is not necessary to test for the success of the `alloca` call. It always succeeds since it is only a simple pointer manipulation. If the maximal stack size is reached the problem will not become visible in the `alloca` call but instead in the first access of this memory. This is quite dangerous but if the stack is reasonably sized and one does not put too big objects on the stack it should never give any problems. And if there are problems whey do not result in silent errors but instead cause the application to crash which then can be analyzed easily.

One kind of object which frequently has to be duplicated temporarily are strings. This is why the GNU libc provides two special features to ease this: `strdupa` and `strndupa`. The behavior is comparable to the functions `strdup` and `strndup` with the one difference that the returned strings are allocated using `alloca` instead of `malloc`. But this automatically means that `strdupa` and `strndupa` must be macros and no functions!
One could think `strdupa` could be implemented like this:

```
/* Please note this is WRONG!!!  */
#define strdupa(s)                                           \
  (__extension__                                             \
    ({                                                       \
      __const char *__old = (s);                             \
```

```
    size_t __len = strlen (__old) + 1;                    \
    (char *) memcpy (__builtin_alloca (__len), __old, __len);  \
  }))
```

But the `memcpy` is *very* wrong! Everybody who uses `alloca` must be aware of this problem. We already explained that `alloca` works by manipulating the stack pointer. But on some systems parameters for function calls are also put on the stack. If this happens for the above `memcpy` call we could get the following sequence of operations:

1. push `__len` on the stack, change stack pointer
2. push `__old` in the stack, change stack pointer
3. modify stack pointer for newly allocated object
4. push current stack pointer on stack, change stack pointer
5. call `memcpy`
6. `...`

We can now see why this is wrong. The memory allocated for the `alloca` call is in the middle of the parameter list. This can never work. Therefore everybody using `alloca` must remember *never* to call `alloca` in the parameter list of a function call. This includes of course hidden `alloca` calls as in `strdupa`.

4.4 Some more Memory Issues

Beside the existence and possibilities of `alloca` there are some more issues one should know about memory allocation to write optimal code.

The nonzero costs of a call to any of the memory allocation functions was already mentioned in the last section. Especially the `realloc` function is possibly slow since in the worst case it has to do the work of an `malloc`, `memcpy`, and `free` call all at once.

The `malloc` implementation will try to keep the amount of used memory as low as possible. I.e., memory which is freed could be reused in a later `malloc` call. To do this the implementation uses sophisticated data structures to make this possible. Things can work pretty smoothly if memory needs would never be able to grow. But occasionally a program calls `realloc` to resize the buffer. Shrinking is not a problem, but growing the buffer is. Since memory usage has to be kept minimal allocated buffer lie back to back in memory. But this means that there is often no room to grow a buffer.

Therefore a `realloc` call might have to allocate a completely new buffer. In this case the content of the old buffer must be copied over and the internal data structures must be updates to contain the old buffer as free.

Programs using dynamic memory allocation therefore should try to determine the amount of needed memory first. As long as the amount is not underestimated and not heavily overestimated, the numbers need not be 100% accurate. Shrinking the buffer by a few bytes is not a problem. Getting at least a usable guess for the amount of memory needed often is not easy, especially if it involves a lot of computation which would have to be repeated when copying in the new buffer. That it is possible to handle even these situations efficiently shows the implementation of `concat` above. To preserve the results of costly computations in the first phase one uses memory allocated on the stack.

A final point about memory allocation is to mention `calloc`. Most people only know `malloc`, `really`, and `free`. But ISO C defines another function.

calloc, as malloc, returns a newly allocated block of memory. The size is not given as a single integer value but instead as two values, which must be multiplied to get the total size. The big difference to malloc is that before the allocated memory is returned to the caller it is initialized with NUL bytes.

```
/* Allocate NMEMB elements of SIZE bytes each, all initialized to 0.  */
void *calloc (size_t __nmemb, size_t __size);
```

Now the reader can of course ask why this is important. It is easily possible to call memset after an successful malloc call and initialize the memory. This is true, but in some situations calloc does not have to call memset because the memory is already zeroed. This happens if the malloc implementation gets the memory from the kernel via a mmap call. The memory returned by mmap is guaranteed to be filled with NUL bytes (unlike the memory made available by a sbrk call). A good malloc implementation, e.g. the one in the GNU libc, keeps track of this fact and avoids the memset call whenever possible.

For this reason it is always, especially for large allocation requests, better to call calloc instead of calling malloc and clearing the memory afterwards with a memset/bzero call.

4.5 Using the Best Types

The following is not really part of the library but instead part of the library headers. We are talking about available types and choosing the best one.

The ISO C9x standard will feature an important new header: <stdint.h>. This header will contain definitions for now types which will be portably usable over all ISO C9x compliant platforms.

The first problem these new types are solving is a longstanding one. To reliably exchange data from one system to another one can either encode the data in textual form (though this leaves to problem of different character sets open) or one uses the same binary encoding. The latter requires the very same interpretation of each byte. This is not a problem for single-byte object (like strings). But if a object consists of more than one byte we get problems with endianess. The main problem, though, is that types like long int have no fixed representation over different platforms. There is no way to write a ISO C compliant program which does not have this problem. ISO C9x will solve this problem with types like int32_t which has a fixed length but we won't discuss this here.

We want to discuss two other categories of types introduced in ISO C9x. The first group consists of int_least8_t, uint_least8_t, int_least16_t, etc, for various sizes up to at least 64. The difference between int8_t and int_least8_t and all the other pairs is that the former has a guaranteed size for the objects while the latter only guarantees that values which use up to the given number of bits, can be stored without loss. An object of type int_least*_t type is at least as big as one of the fixed with type but might possibly be larger.

The question now is where can this be useful. The answer is: wherever values are stored with a minimal known range and the exact representation is not important and some waste of memory is acceptable. This happens quite frequently. E.g., a not too large array containing 16 bit values which are frequently used can be stored, e.g., in an array of int or an array of int_least16_t. The difference might be dramatic. The int_least16_t type can be adapted for the processor architecture in use and might be much bigger, e.g., 64 bits. This would allow accessing the array values much faster if the architecture does not allow directly to access sub-word-size memory values. This is the case for many modern architectures. Whenever one creates data structures where the access might be performance critical, one should think about using the types above to allow the processor to work best.

While the int_least*_t types are mainly used for data objects another group of types is mainly used for the use in program code. Very often programs code must have variables which are used as counters.

```
{
  short int n;
  ...
  for (n = 0; n < 500; ++n)
    c[n] = a[n] + b[n];
  ...
}
```

While this seems logical (a short int on interesting machines has at least 16 bits) since it makes the variable n smaller than as if we would use int, the code is not really good. The variable n is kept in a register and therefore the size of the variable does matter as long as it does not exceed the size of the register. Sometimes compilers recognize situations like the above and simply perform the operations which are fastest even if they are not correct for the given type (short int in the above case). But it is nevertheless better to help the compiler doing this. ISO C9x introduces appropriate types for this. The changed example looks like this:

```
{
  int_fast16_t n;
  ...
  for (n = 0; n < 500; ++n)
    c[n] = a[n] + b[n];
  ...
}
```

In this version the author expresses everything the compiler has to know. The counter variable must have at least 16 bits to hold the values from 0 to 500. How big the variable is actually is uninteresting, the program must only run fast. This allows the compiler/library to pick the best size for this definition which in most cases is a type with the same size as the registers.

4.6 Non-Standard String Functions

The designers of the standard library added several useful functions which together cover most of the needed functionality. This does not mean that the provided set of functions allow optimal programs. In this section some functions from the repertoire of the GNU libc will be introduced which add additional functionality which allows writing more optional programs.

A repeating task in programs which handle strings is to find the end of the string for further processing. This is often implemented like this:

```
{
  char *s = ... /* whatever needed */...;

 s += strlen (s);

  ... /* add something at the end of the string */ ...
}
```

This is not terribly efficient. The strlen function already had a pointer to the terminating NUL byte of the string. The addition simply recomputes this result. It is more appropriate to write something like the following:

```
{
  char *s = ... /* whatever needed */...;

  s = strchr (s, '\0');

  ... /* add something at the end of the string */ ...
}
```

Here we get immediately the result from the function call since the result of the strchr call is a pointer to the byte containing the searched value. Since we are searching for the NUL byte this is the end of the string. But this is worse than the original version. The problem is that the strchr function has two termination criteria: the given character matches of the end of the string is reached. That both test are the same in the above case is not seen (at least it is not guaranteed). The GNU libc contains a function which can be used in this situation and which does not have this problem.

```
{
  char *s = ... /* whatever needed */...;

  s = rawmemchr (s, '\0');

  ... /* add something at the end of the string */ ...
}
```

The rawmemchr function is much like the memchr function but it does not take a length parameter and therefore performs only one termination test. It terminates only if the given character is found. This makes `rawmemchr (s, '\0')` the exact equivalent to `s + strlen (s)`. The implementation of the rawmemchr function is very simple and fast. It is especially fast on the Intel x86 architecture where it can effectively implemented with a single instruction.

To see the function in action we take a look at a piece of code which can be found in this form or another in many programs. It handles values given in the PATH-like style where a string contains individual values separated by a specific character, a colon in many cases. Code to iterate over all the individual values and produce NUL terminated strings from them could be done like this:

```
{
  const char *s = ... /* whatever needed */...;

  while (*s != '\0')
```

```
    {
      char *copy;
      const char *endp = strchr (s, ':');
      if (endp == NULL)
        endp = rawmemchr (s, '\0');

      copy = strndupa (s, endp - s);
      ... /* use copy */ ...

      if (*s != '\0')
        ++s;
    }
}
```

We are using the rawmemchr function to find the end of the string if there is no colon anymore. The copy on which the rest of the function is working on is created using strndupa. This introduces no arbitrary limits (as a static buffer) and is fast (unlike a malloc call). The above construct of finding a specific character and, failing that, returning the end of the string appears so often, that the GNU libc contains a specific function for this. This function is a slightly modified version of strchr. The original code would scan the last part of the input string twice although already the strchr call almost had the result. This deficiency is fixed by the new function.

```
{
  const char *s = ... /* whatever needed */...;

  while (*s != '\0')
    {
      const char *endp = strchrnul (s, ':');
      char *copy = strndupa (s, endp - s);
      ... /* use copy */ ...

      if (*s != '\0')
        ++s;
    }
}
```

This is the ultimate solution for this problem. The strchrnul function always returns the value we are interested in and it does not cost anything extra; the strchrnul function is even a big faster than strchr since no special return value has to be prepared for the case that a NUL byte is found.

The lesson from this section should be: library functions are useful and often highly optimized for the specific purpose. But there is no guarantee that they are the best solution in every situation they are used in. There might be better and generally interesting functions and maybe the GNU libc already provides them.

5 Writing Better Code

Using the correct functions and types and helping the compiler to generate better code can only help that much if the general algorithm and use of the functions isn't good. In this section we will describe in various examples for what to look for and how to improve algorithms.

5.1 Writing and Using Library Functions Correctly

This paper showed in the earlier section that choosing the correct functions is important as is writing sometimes new functions which fulfill the job better. But writing new functions and using other ones to do this also contains a lot of situations where one can introduce problems. By a simple example we show some of the things one has to take care of.
The ISO C library does not contain any function to duplicate a string. We ignore for a moment that the GNU libc already contains an implementation of the `strdup` function and assume we want to write it now. A first attempt could look like this:

```
char *
duplicate (const char *s)
{
  char *res = xmalloc (strlen (s) + 1);
  strcpy (res, s);
  return res;
}
```

We use the `xmalloc` function which is often used in GNU packages to provide a failsafe `malloc` implementation. After reading the previous sections of this paper it be clear that we can do better by not using `strcpy` and reusing the result of the `strlen` call. Second try:

```
char *
duplicate (const char *s)
{
  size_t len = strlen (s) + 1;
  char *res = xmalloc (len);
  memcpy (res, s, len);
  return res;
}
```

This is better but we missed one very often missed optimization: most functions are functions in the mathematical sense and have a return value. One cannot be reminded often enough on that. After fixing this we end up with the following form:

```
char *
duplicate (const char *s)
{
  size_t len = strlen (s) + 1;
  return (char *) memcpy (xmalloc (len), s, len);
}
```

That's much nicer and even looks shorter than the original implementation. To stress it once more: the return values of functions can be used directly! This is not UCSD Pascal. Especially the `memcpy` function has a return value which many people simply forget. In this situation the code change can safe a load from the memory where the buffer pointer is kept since the return value of the memcpy call can be used directly. Additionally the compiler now could perform a sibling call optimization.

But there is one more optimization which could be performed at compile time. If the argument to duplicate is a constant string we could compute the length of the string at compile-time. But with a simple function call this is not possible. Therefore we add a wrapper macro which recognizes this case. The following code only works with gcc.

```
#define duplicate(s) \
  (__builtin_constant_p (s)          \
   ? duplicate_c (s, strlen (s) + 1)  \
   : duplicate (s))
```

We introduced the __builtin_constant_p operator already on page 9. It should therefore be clear what the macro does. The missing duplicate_c function is easily written:

```
char *
duplicate_c (const char *s, size_t len)
{
  return (char *) memcpy (xmalloc (len), s, len);
}
```

Finally we ended up with a highly optimized version which takes advantage of all compile time optimization, which enables the compiler to generate optimal code and which uses the existing functions in an optimal way. Ideally every function one writes should be optimized that carefully. It is not hard if one only takes care of these three steps:

1. Are the correct functions used or are there better ones available?
2. Do I use the functions I use in the optimal way? Are the return values used?
3. Are all computations which can be carried out at compile time done and used?

5.2 Computed gotos

Sometimes functions cannot be broken up in smaller pieces for design or performance reasons. Then one could end up with a large function with many conditionals which slow down the execution. A solution would be a kind of state machine. The traditional and simple way to implement a state machine is to have one big switch statement with a single state variable controlling which case is used.

This general form is very often not necessary since in most cases it is not necessary to be able to go over from each state into another jump. What is actually a better implementation is a jump table which can be adopted for each situation. In standard C it is not possible to write jump tables but it is with gcc's computed gotos. As an example we use the following code.

```
{
  ...
  switch (*cp)
  {
  case 'l':
    islong = 1;
    ++cp;
    break;
  case 'h':
    isshort = 1;
    ++cp;
    break;
  default:
  }

  switch (*cp)
  {
  case 'd':
    ... /* handle this */ ...
    break;
  case 'g':
    ... /* and code for this */ ...
    break;
  }
}
```

This is with lots of code left out from a piece of code in the GNU libc where now jump tables are used: the `printf` implementation. The problem is the processing of the format string. Many optional character can precede the actual format. So we have to test for them (e.g., the modifiers `'l'` and `'h'`) even though we might find out in the first `switch` statement that we already found a format character, e.g., `'d'`. What we rather would like to do is to jump directly to the format handling instead of the `default` case where we start performing the test again. Using jump tables this is possible.

```
{
  static const void *jumps1[] =
  {
    ['l'] = &&do_l,
    ['h'] = &&do_h,
    ['d'] = &&do_d,
    ['g'] = &&do_g
  };
  static const void *jumps2[] =
  {
    ['d'] = &&do_d,
    ['g'] = &&do_g
  };

  goto *jumps1[*cp];

do_l:
  islong = 1;
  ++cp;
```

```
    goto *jumps2[*cp];

  do_h:
    isshort = 1;
    ++cp;
    goto *jumps2[*cp];

  do_d:
    ... /* handle this */ ...
    goto out;

  do_g:
    ... /* and code for this */ ...
    goto out;

  out:
  }
```

This might look frightening and complex but it is not. The jump table syntax has to be learned but it is noting but an array of pointers. The elements of these arrays can then be used by a `goto` instruction. By finding the array elements using the current format string character we are emulating the `switch` statements above. But it should be noted that if the first character is directly a format character, we jump directly to the code performing the handling of the formatted output. Only if we actually see a modifier character we add some extra steps. Since (in this simplified situation) it is not valid to have repeated modifiers we have for the jumps out of the modifier handling code a different jump table. It is possible to have arbitrary many of them.

The code above is not complete. E.g., the handling of invalid characters is not correct as the `goto`s would use NULL pointers in the uninitialized array spots or even access memory outside the array boundaries. Also, accessing the array using the character as an index wastes a lot of array space. One should come up with a tighter packing method.

To see how this can be done and for a real world, complex example take a look at the `vfprintf.c` file in the GNU libc sources. The file is far too big to be printed here.

6 Profiling

When one has performed all the obvious optimizations there remains the mean of profiling to find out where the time in the program is spend and work on those functions. Profiling is supported on most systems, more or less accurately. In general there are two kinds of profiling:

- Timer-based. This allows to find out where the most time is spend.
- Call-arc based. This allows to find out what functions are called how often and from where.

The peak values for in both counts must not always fall together. Many simple functions are called very often and still do not contribute prominently to the overall runtime. Nevertheless this peaks in the call count chart indicate a possible place where inlining might help.

On most Unix system one can compile programs using the `gprof` method. Systems using GNU libc and Solaris can perform another kind of profiling which is implemented using the dynamic linker.

6.1 gprof Profiling

Traditional profiling is implemented by compiling all sources which should participate in the profiling with a special option. This causes the compiler to generate some extra code which records the execution at runtime. The compiler would have to be called like this:

```
gcc -c foo.c -o foo.o -pg
```

The `-pg` option instruct the compiler to add the extra code. When linking the program another decision can be made. If the user also wants to know about the time spend in function and the calls made to functions in the C library s/he can link against a special version of the C library be adding the `-profile` option:

```
gcc -o foo foo.o -profile
```

Otherwise the normal library is used only only the function of the program are instrumented. To get results the program must be executed. Once the program terminated the user can find a file named `gmon.out` in the initial working directory. This file, together with the executable, serves as the input for a program named `gprof`. We will show the various outputs of this program in a small example. The following, horrible code is used.

```
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char *argv[])
{
  char *buf = NULL;
  size_t buflen = 0;
  size_t bufmax = 0;
  char *line = NULL;
  size_t linelen = 0;
  size_t cnt;

  while (!feof (stdin))
    {
      size_t len;
      if (getline (&line, &linelen, stdin) == EOF)
        break;
      len = strlen (line);
      if (len == 0)
        break;
      if (buflen + len + 1 > bufmax)
        {
  buf = realloc (buf, bufmax = (2 * bufmax + len + 1));
          buf[buflen] = '\0';
        }
```

```
      strcat (buf, line);
      buflen += len;
    }

  for (cnt = 0; cnt < buflen - strlen (argv[1]); ++cnt)
    {
      size_t inner;

      for (inner = 0; inner < strlen (argv[1]); ++inner)
        if (argv[1][inner] != buf[cnt + inner])
          break;
      if (inner == strlen (argv[1]))
        printf ("Found at offset %lu\n", (unsigned long int) cnt);
    }

  return 0;
}
```

This problem probably violates all of the rules defined in the previous sections. And it indeed runs very slowly. Using the `gprof` output we can see why. To do this we run the program and we get in the end a file `gmon.out`. Now we start the `gprof` program to analyze it. Without options the programs the output consists of two parts. We will explain them here.

The first part is the flat profile. Here every function is listed with the number of times the function is called and the time spent executing it. The beginning of the output is this:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 95.27      8.66     8.66    10445     0.83     0.83  strcat
  2.86      8.92     0.26                             __mcount_internal
  0.55      8.97     0.05  1327730     0.00     0.00  strlen
  0.44      9.01     0.04      108     0.37     0.37  read
  0.44      9.05     0.04                             mcount
  0.22      9.07     0.02        1    20.00  8790.00  main
  0.11      9.08     0.01    10551     0.00     0.00  memcpy
  0.11      9.09     0.01       58     0.17     0.17  write
  0.00      9.09     0.00    20950     0.00     0.00  flockfile
  0.00      9.09     0.00    20950     0.00     0.00  funlockfile
  0.00      9.09     0.00    10548     0.00     0.00  memchr
  0.00      9.09     0.00    10446     0.00     0.00  feof
  0.00      9.09     0.00    10446     0.00     0.00  getdelim
  0.00      9.09     0.00    10446     0.00     0.00  getline
```

What we can see is that of the total runtime of 8.79 seconds the program spent 95% in the `strcat` function. This again shows how evil `strcat` is. The chart also shows how calls to `strlen` are made. The function is executable very quickly so we have no hit by the profiling interrupt, but 1.3 million calls to process 10445 lines of input is too much.

More detailed information about the contexts in which the functions are called can be found in the second part, the call graph. Here every function is listed with the places from which it is called and the function which are called from it. This is an excerpt from the same run as the flat profile output:

```
index % time    self  children    called     name
                0.02    8.77       1/1           __libc_start_main [2]
[1]    100.0    0.02    8.77       1         main [1]
                8.66    0.00   10445/10445       strcat [3]
                0.05    0.00 1327730/1327730     strlen [4]
                0.00    0.05   10446/10446       getline [6]
                0.00    0.01      58/58          printf [13]
                0.00    0.00      15/15          realloc [23]
                0.00    0.00   10446/10446       feof [28]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    8.79                 __libc_start_main [2]
                0.02    8.77       1/1           main [1]
                0.00    0.00       1/1           exit [39]
-----------------------------------------------
                8.66    0.00   10445/10445       main [1]
[3]     98.5    8.66    0.00   10445         strcat [3]
-----------------------------------------------
                0.05    0.00 1327730/1327730     main [1]
[4]      0.6    0.05    0.00 1327730         strlen [4]
-----------------------------------------------
```

This output shows all the functions called from `main`. For this simple program there are no surprises and we could have predicated the output easily. But if the program is more complicated a function might be called from different places and then it is useful to know from which places how many calls are made. The content of the column titled “called” consists of two parts (except for the line with the function this is all about). The left part is the number of calls made to this function from this place. The right column specifies the total number of calls. For all functions all calls come from `main`.
Now we try to improve the program a bit and use the following modified version:

```
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char *argv[])
{
  char *buf = NULL;
  size_t buflen = 0;
  size_t bufmax = 0;
  char *line = NULL;
  size_t linelen = 0;
  size_t cnt;
  size_t argv1_len = strlen (argv[1]);

  while (!feof_unlocked (stdin))
    {
      size_t len;
      if (getline (&line, &linelen, stdin) == EOF)
        break;
```

```
      len = strlen (line);
      if (len == 0)
        break;
      if (buflen + len + 1 > bufmax)
        {
  buf = realloc (buf, bufmax = (2 * bufmax + len + 1));
          buf[buflen] = '\0';
        }
      memcpy (buf + buflen, line, len);
      buflen += len;
    }

  for (cnt = 0; cnt < buflen - argv1_len; ++cnt)
    {
      size_t inner;

      for (inner = 0; inner < argv1_len; ++inner)
        if (argv[1][inner] != buf[cnt + inner])
          break;
      if (inner == argv1_len)
        printf ("Found at offset %lu\n", (unsigned long int) cnt);
    }

  return 0;
}
```

All we changed is to use `memcpy` instead of `strcat`, to use `feof_unlocked` instead of `feof` and to precompute `strlen (argv[1])` and reuse the value. The results are dramatic:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 40.00      0.02     0.02        1 20000.00 40000.00  main
 20.00      0.03     0.01    20996     0.48     0.48  memcpy
 20.00      0.04     0.01    10548     0.95     0.95  memchr
 20.00      0.05     0.01                             __mcount_internal
  0.00      0.05     0.00    10504     0.00     0.00  flockfile
  0.00      0.05     0.00    10504     0.00     0.00  funlockfile
  0.00      0.05     0.00    10446     0.00     0.00  feof_unlocked
  0.00      0.05     0.00    10446     0.00     1.44  getdelim
  0.00      0.05     0.00    10446     0.00     1.44  getline
  0.00      0.05     0.00    10446     0.00     0.00  strlen
```

The total program runtime went down to 40 milliseconds. Most of the time now is spend in the application itself. The calls to `memcpy`, which replaced the `strcat` calls, do not play any significant role. Also, the number of calls to `strlen` went down dramatically.

This exampled showed how the profiling possibilities can be used to pinpoint the most time consuming part of the program. With the different output modes it is then easy to locate the places where calls are made and possibly rewrite the code. The results, as can be seen above, can be dramatic.

6.2 `sprof` Profiling

In the last section we have mentioned that profiling is possible with and without taking the library function into account. For the latter case one has to provide the `-profile` option and gets the result of the last section where timing and call counts are given for all the library functions. What wasn't said is that the resulting binary is statically linked. The special library version necessary to support `-profile` is only available as an archive.

The reason for this is the way profiling is implemented. The algorithms need a single text section for the whole program. This is not the case if shared objects are used and therefore they cannot be used. At least not before the profiling code is completely rewritten.

But this means that programs are not really profiled in the same form they would later be used. Normally every application is linked dynamically. Therefore realistic profiling should allow profiling shared objects.

In GNU libc 2.1 (and also on Solaris) this possibility is implemented. It allows to profile single shared objects, for one executable or systemwide. I.e., it does not allow profiling all shared objects of an application but exactly one. And it also does not allow profiling the application code itself with an shared object.

These all are significant restrictions but you can solve these problems partly using the static profiling using `gprof`. The profiling of a single shared object as implemented provides something which is not available from static profiling: it is possible to profile the use of a shared object by several applications at the same time and contributing to the same output file. Profiling a single application will allow optimizing only the *use* of a library. But to optimize the library itself it is necessary to see the data from uses of different programs. And having all the data (optionally) combined in one single file is even better.

And there is one more good thing about the `sprof` approach: there is no need to recompile any code. The normal code which is used for everyday operations is the one which gets debugged. This means we need absolutely no preparation to start profiling.

```
LD_PROFILE=libc.so.6 LD_PROFILE_OUTPUT=. /bin/ls -alF ~
```

Executing this command on a Linux/x86 system (where the SONAME of the C library is `libc.so.6`) normally executes the program. But during the execution the file `libc.so.6.profile` in the current working directory (specified by the `LD_PROFILE_OUTPUT` environment variable) is filled with profiling information. We can execute the program or a completely different program arbitrary many times, even in parallel, and they all can contribute to the profiling data. Once enough data is collected one can look at the content using the `sprof` program (on Solaris systems the `gprof` program must be used).

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 50.00      0.02     0.02      341    58.65           __lxstat64
 25.00      0.03     0.01     1362     7.34           _IO_str_init_static
 25.00      0.04     0.01       72   138.89           strcasecmp
```

```
  0.00      0.04     0.00     4836     0.00            strcmp
  0.00      0.04     0.00     2872     0.00            mempcpy
  0.00      0.04     0.00     2568     0.00            flockfile
  0.00      0.04     0.00     2568     0.00            funlockfile
  0.00      0.04     0.00     2420     0.00            memcpy
```

The output looks like the output of gprof and this is of course intended. We see the functions which were used most, see how often they were called and the time the contribute to the total runtime.

```
[85]     0.0    0.00    0.00        0         _nl_make_l10nflist [85]
                0.00    0.00        3/3           argz_count [658]
                0.00    0.00        1/19          cfree [543]
                0.00    0.00        4/12          stpcpy [636]
                0.00    0.00        2/2           argz_stringify [665]
                0.00    0.00        2/2420        memcpy [643]
                0.00    0.00        3/425         malloc [541]
-----------------------------------------------
                0.00    0.00        1/1           _nl_expand_alias [79]
[175]    0.0    0.00    0.00        1         bsearch [175]
-----------------------------------------------
[177]    0.0    0.00    0.00        0         msort_with_tmp [177]
                0.00    0.00      506/2420        memcpy [643]
                0.00    0.00     2466/2872        mempcpy [631]
-----------------------------------------------
                0.00    0.00        1/3           read_alias_file [80]
                0.00    0.00        2/3           <UNKNOWN>
[178]    0.0    0.00    0.00        3         qsort [178]
                0.00    0.00        1/19          cfree [543]
                0.00    0.00        1/425         malloc [541]
                0.00    0.00        2/1391        __errno_location [12]
```

The output also contains the call graph. We can exactly analyze from where each function was called how often. If the name is <UNKNOWN> it is a call from the main program or another library.

Profiling shared objects is a very powerful mean to optimized them. It is not meant to optimize applications but to optimize the system-wide use of the library. It is very well possible that libraries should be optimized differently on different systems. In future there will be tools which interpret the sprof output appropriately. For now one can use sprof for the library as if it is the program one wants to optimize.