mpr − a memory allocation profiler for C/C++ programs

libmpr.a

To use *mpr* with a statically linked executable, link against `libmpr.a`. You will need to set the **MPRFI** environment variable − see the `mpr` and **MPRFI** sections below for further details.

libmpr.so

To use *mpr* with a dynamically linked executable, do one of the following:

1. dynamically link against `libmpr.so`
2. statically link against `libmpr.a`
3. use the dynamic linker/loader to load `libmpr.so` at execution time

Method (3) is obviously the most convenient, and can even be used with programs for which you don't have source − it is therefore the method of choice for most users. You will need to set the **MPRFI** and **LD_PRELOAD** environment variables − see the `mpr` and **MPRFI** sections below for further details.

mpr a.out [arg ...]

`mpr` is a simple wrapper that runs your program, after setting the environment variables **LD_PRELOAD** and **MPRFI** so that calls to `malloc()/free()` can be intercepted and logged.

```
% mpr perl −e 'print "goodbye, cruel world\n"'
% mpr python −c 'print "goodbye, cruel world"'
% mpr tcl −c 'puts "goodbye, cruel world"'
% mpr lua −e 'print "goodbye, cruel world"'
```

If your program has not been linked against libmpr.a or libmpr.so, `mpr` will add `libmpr.so` to the environment variable **LD_PRELOAD** to force `libmpr.so` to be loaded before your program runs.

If the **MPRFI** environment variable is unset, `mpr` will set it to `"gzip −1 >log.%p.gz"`. This will save the log messages to a file named `log.`*`pid`*`.gz` in the current directory, where *pid* is the process ID of the process that executes your program. If your program creates child processes, there may be multiple log files created. See the **MPRFI** section below for details on **MPRFI**, and the **MPRNOCHILD** section below for details on how to disable tracing child memory allocations.

NOTE: If `mpr` cannot find `libmpr.so`, it will print an error message to `stderr` and exit without running your program − you will need to manually set the environment variable **LD_LIBRARY_PATH** to include the directory containing `libmpr.so`.

NOTE: On NetBSD, you will need to manually set the environment variable **MPRLD_PRELOAD** to the complete path name for `libmpr.so`, since `ld.elf_so(1)` on NetBSD does not search **LD_LIBRARY_PATH** for **LD_PRELOAD** objects.

MPRFI

The **MPRFI** environment variable defines a shell pipeline that is used to filter the log messages emitted by *mpr*'s `malloc()/free()` hook functions. The pipeline is created by forking an **MPRSH** shell subprocess of the process running your program.

If you use the wrapper `mpr` to run your program and **MPRFI** is not already set, it will be set to `'gzip −1 >log.%p.gz'` automatically. The `'%p'` directive is replaced with the process ID of the process running your program.

If you only wish to catch memory leaks (i.e. if you are not interested in your program's entire allocation history), you can use `mprleak` in **MPRFI**:

```
% env MPRFI='mprleak >leaks.%p' mpr a.out ...
```

If you have statically linked `libmpr.a` into your program, you can disable the *mpr* hook functions by unsetting **MPRFI** or setting it to the empty string:

```
% env MPRFI= a.out ...
```

MPRNOCHILD

If the **MPRNOCHILD** environment variable is set, then *mpr* will not track memory allocations by child processes (i.e. it will not create multiple `log.`*`pid`* files). This can be useful if your program is a frontend that invokes other programs.

```
% env MPRNOCHILD=1 a.out ...
```

---

mprmap [−f|−F a.c,b.c,...] [−l] [−lx] [−i foo,bar,...] [−p] a.out [log]

`mprmap` maps program counters in an **MPRFI** log file to function names and, optionally, file names and line numbers.

```
% mprmap a.out <log
% mprmap a.out log
% mprmap a.out log.gz
```

To examine memory leaks, use `mprleak` as a pre−filter:

```
% mprleak log | mprmap a.out
```

Use option `−i` if you have "wrapper" functions around calls to `malloc()/free()` that are cluttering your call chains. Its argument is a list of comma−separated functions that you wish to ignore.

```
void *xmalloc(size_t sz)
{
        void *p = malloc(sz);
        if (!p && sz)
                abort();
        return p;
}

void *xrealloc(void *ptr, size_t sz)
{
        void *p = realloc(ptr, sz);
        if (!p && sz)
                abort();
        return p;
}
```

In the code fragment above, `xmalloc()/xrealloc()` are wrappers around `malloc()/realloc()` that check for allocation failures. Since they are not directly responsible for the allocations, you can ignore them with:

```
% mprmap −i xmalloc,xrealloc ...
```

As indicated above, "," (comma) is the default separator for the list of functions to be ignored with option `−i`. If you are using `mprmap` with a C++ program, you can use option `−I` to specify a different separator since "," may appear in a C++ function's signature:

```
% mprmap −I'#' −i 'objalloc(void *, size_t)#objalloc(size_t)' a.out log
```

Option `−f` displays file names in addition to function names. This only works for those parts of your program that were compiled with the `−g` option of the C compiler. Parts of your program that were not compiled with `−g`, or whose symbol tables have been stripped, will not display file names. Option `−l` displays line numbers in addition to file names and function names − it should be used in conjunction with option `−f`.

```
% mprmap −f −l −i xmalloc,xrealloc a.out log
```

Option `−F` is similar to option `−f`, but it restricts the set of source files for which file names are displayed. For example, to display file names and line numbers only for files `a.c` and `b.c`:

```
% mprmap −F a.c,b.c −l a.out log
```

To display file names and line numbers only for source files in directory `src/dir1`:

```
% mprmap −F "`cd src/dir1; echo *.c`" −l a.out log
```

Option `−p` is useful if you are using `mprmap` with a C++ program. By default, `mprmap` will print the full signature of each C++ function − this can result in unwieldy looking output. Option `−p` omits the parameter list portion of the signature.

---

mprchain [−c N] [−C N] [−w N] [−n] [−m] [log]

`mprchain` groups memory allocations by call chains. Its stdin should be connected to the stdout of `mprmap`.

```
% mprmap a.out log | mprchain
```

To see memory leaks grouped by call chains, use `mprleak` as a pre−filter to `mprmap`:

```
% mprleak log | mprmap a.out | mprchain
```

`mprchain` displays the call chain in column 1, the number of allocations/leaks by the call chain in column 2, the total amount of memory allocated/leaked by the call chain in column 3, and the percentage of memory allocated/leaked by the call chain in column 4. `mprchain` also prints a separator line "−−" between each call chain, which helps to distinguish between call chains that span multiple lines.

Option `−c` sets the length of the call chains (default=999999).

```
% mprmap −f −l a.out log | mprchain −c9
```

Option `−C` sets the number of call chain entries to display per line. By default, `mprchain` tries to squeeze as many entries as it can onto one line.

```
% mprmap −f −l a.out log | mprchain −C1
```

Option `−w` sets the width of the display (default=80).

```
% mprmap −f −l a.out log | mprchain −w132
```

Option `−n` omits the percentage column.

Option `−m` displays, in column 3, the *maximum* amount of memory that was allocated at any time by the call chain (instead of the *total* amount of memory allocated by the call chain).

```
main()
{
        int i;
        char *p1, *p2;
        for (i=0; i<10; i++) {
                p1 = malloc(200);
```

```
                free(p1);
        }
        p1 = malloc(150);
        p2 = malloc(100);
        free(p1);
        free(p2);
}
```

In the code fragment above, without option `−m`, `mprchain` will show `main()` as having allocated a *total* of 2250 bytes; with option `−m`, it will show `main()` as having allocated a *maximum* of 250 bytes.

---

mprsize [−n] [log]

`mprsize` groups memory allocations by size. Its input should be the the **MPRFI** log file.

```
% mprsize <log
% mprsize log
% mprsize log.gz
```

To see memory leaks grouped by size, use `mprleak` as a pre−filter.

```
% mprleak log | mprsize
```

`mprsize` displays the size in column 1, the number of allocations/leaks of that size in column 2, the amount of memory allocated/leaked by that size in column 3, and the percentage of memory allocated/leaked by that size in column 4.

Option `−n` omits the percentage column.

---

mprleak [log]

`mprleak` identifies those allocations that lead to memory leaks. Its input should be the **MPRFI** log file. It is most often used as a pre−filter for `mprmap` or `mprsize`.

```
% mprleak <log | mprmap a.out
% mprleak log | mprmap a.out
% mprleak log.gz | mprmap a.out
```

To examine memory leaks grouped by call chains:

```
% mprleak log | mprmap a.out | mprchain
```

To examine memory leaks grouped by size:

```
% mprleak log | mprsize
```

If you only wish to catch memory leaks in your program (i.e. if you are not interested in your program's entire allocation history), you can use `mprleak` in **MPRFI**.

```
% env MPRFI='mprleak >leaks.%p' mpr a.out ...
```

---

mprhisto [−c N] [−w N] [−b N] [log]

`mprhisto` displays a memory allocation histogram. Its input should be the **MPRFI** log file.

```
% mprhisto <log
% mprhisto log
% mprhisto log.gz
```

`mprhisto` uses "*" (asterisk) to represent a 1KB block of allocated memory. For example, to find the maximum amount of memory (within 1KB) that was allocated at any point during the life of your program:

```
% mprhisto log | awk '{if (length>m) m=length} END {print m}'
```

Use option `−b` to change the default block size. For example, to see a histogram with a block size of 2KB:

```
% mprhisto −b2048 log
```

Option `−c` sets the length of the call chains to display alongside the histogram (by default, no call chains are displayed). If you use option `−c`, you should pre−filter the **MPRFI** log file through `mprmap` so that call chain program counters are mapped to function names. For example, to see a histogram with a block size of 2KB and call chains of length 5:

```
% mprmap a.out log | mprhisto −b2048 −c5
```

Note that without option `−c`, `mprhisto` will not show adjacent histogram entries of the same length (i.e. `mprhisto` only shows changes in the amount of allocated memory in multiples of the block size). If you use option `−c`, `mprhisto` is more verbose and prints a line for *every* `malloc()/free()` (i.e. every line in the **MPRFI** log file).

Option `−w` sets the width of the display (default=80), and is useful when you use option `−c` to show call chains alongside the histogram. With option `−c`, `mprhisto` tries to right justify the call chains to line up in column 80 (the histogram is easier to visualize by separating it from the call chains). However, if the output scrolls off the end of the screen or if the call chains are too close to the histogram, you can

1. use option `−b` to increase the block size (so that the histograms are shorter)
2. use option `−c` to decrease the length of the call chains
3. use option `−w` to increase the width of the display

For example, to see a histogram with a block size of 2KB and call chains of length 5 on a 132−column wide display:

```
% mprmap a.out log | mprhisto −w132 −b2048 −c5
```

MPRAWK

If the **MPRAWK** environment variable is set, it names the AWK interpreter used by the *mpr* scripts. The default interpreter used is `awk`.

The quality of your AWK interpreter can make a big difference to the runtime of `mprmap`. I suggest that you install Michael Brennan's excellent `mawk` on your system. Even GNU `gawk`, which is the standard AWK interpreter on most Linux systems, loses − I have seen improvements of over 500% using `mawk` vs `gawk`. For example, to force all the *mpr* scripts to use `mawk`:

```
% export MPRAWK='mawk −W sprintf=4096'
```

MPRTMP

If the **MPRTMP** environment variable is set, it names the directory in which the *mpr* scripts will create their temporary output files. The default directory is `/tmp`. For example, to force all temporary files to be created in the `tmp` subdirectory of your home directory:

```
% export MPRTMP=$HOME/tmp
```

MPRSYNC

If the **MPRSYNC** environment variable is set, it will force all output to the **MPRFI** pipe to be line buffered, instead of 8KB buffered. This may be useful if you wish to visualize the **MPRFI** pipe while your program is running.

```
% env MPRSYNC=1 MPRFI=mprhisto mpr a.out
```

You may need to force your `awk` interpreter to unbuffer or line−buffer stdout in order for this to work. For example, if you're using `mawk`:

```
% env MPRAWK='mawk −W interactive' MPRSYNC=1 MPRFI=mprhisto mpr a.out
```

MPRSH

If the **MPRSH** environment variable is set, it names the shell program used to run the **MPRFI** pipeline. The default shell used is `/bin/sh`. It is required on systems like Solaris8, where `/bin/sh` sometimes fails (for unknown reasons).

```
% export MPRSH=/bin/ksh
```

MPRLD_PRELOAD

As mentioned in the **mpr** section above, on NetBSD you must manually set the environment variable **MPRLD_PRELOAD** to the complete path name for `libmpr.so`, since `ld.elf_so(1)` on NetBSD does not search **LD_LIBRARY_PATH** for **LD_PRELOAD** objects.

```
% export MPRLD_PRELOAD=/usr/local/lib/libmpr.so
```
