BMDFM FAQ

("A Little Boy and His BMDFM")

"And now, let's plunge into a dense fog!" :)

* * *

1. Why is the true multi-process model used in BMDFM additionally to multithreading?

There are a couple of reasons explaining why the true multi-process model is chosen for the BMDFM implementation:

- The threading models are different from OS kernel to OS kernel. Threading model ***Mx1*** runs all threads of the user process space through a single thread of kernel space relying on the kernel scheduler only. Threading model ***1x1*** runs each thread of the user process space through a separate dedicated thread of kernel space relying on the kernel scheduler only. Threading model ***MxN*** maps M threads of the user process space to N threads of kernel space relying on both the kernel scheduler and the user process multiplexing scheduler. In order to make BMDFM portable across different SMP platforms and the OS kernels, the true multi-process model is chosen. Such a solution compiles and runs under all OS kernels in the same way. No additional user process multiplexing scheduler is required.
- Performance is the most important point. **The multi-process model is more scalable and has better performance in practice than the multithreading model when running tasks on a big iron. Multithreading might work faster for multicores and many-cores.**

One serious reason that speaks for a threading approach is that it is a much cheaper way to create/dismiss a thread compared to the effort spent for a process fork. However, BMDFM does not fork processes at runtime; all processes are created at the initialization phase only. Note that BMDFM can be configured to run in the multithreaded mode as well as in the multi-process mode.

Note that ***POSIX-semaphores*** scale and perform better than ***SVR4-semaphores***. BMDFM can be configured using either ***POSIX-semaphores*** or ***SVR4-semaphores*** as synchronization primitives (on platforms where ***POSIX-semaphores*** are available for inter-process synchronization).

2. Where do you see version/build/revision of BMDFM?

The software revision can be seen in the command line prompt for each BMDFM utility as shown in the examples below:

Terminal (Intel x86-64; Linux)

```
$ fastlisp
fastlisp ==> stderr:
/* fastlisp.c - FastLisp Compiler with Runtime Environment.
                ------------------------------------------------------------
                Original 32-bit version for UNIX was founded && written by:
                                        Sancho Mining 05-05-1996 10:10:29.18pm */
  Usage0: fastlisp -h|--help
  Usage1: fastlisp -V|--versions
  Usage2: fastlisp [-q|--quiet] <FastLisp_file_name> [args...]
  Usage3: fastlisp [-c|--compile2disk] <FastLisp_file_name> [args...]
  Usage4: fastlisp [-q|--quiet] <Precompiled_FastLisp_file_name>

The following environment variable:
  FAST_LISP_CFGPROFILE_path="fastlisp.cfg"
specifies a configuration profile that can be used for the Global FastLisp
function definitions. The format of the configuration profile is:
  <(DEFUN ...)>{ <(DEFUN ...)>}  # <EOF>.

Compiled on: "Linux RedHatELS72VM 3.10.0-514.26.2.el7.x86_64 #1 SMP Fri Jun 30
05:26:04 UTC 2017 x86_64".
Compiled by: "gcc version 4.8.5 20150623 (Red Hat 4.8.5-11) (GCC) as [ELF 64-bi
t LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared lib
s), for GNU/Linux 2.6.32, stripped] at systime Fri Jul 13 13:01:55 CEST 2018".
RETURNED STATUS: ABNORMAL PROGRAM TERMINATION.
$
```

Terminal (Sun SPARC; SunOS)

```
$ BMDFMldr -h
BMDFMldr ==> stdout:
/* BMDFMldr.c - The External Task PROC Unit (The Loader and Listener Pair)
                for the "Broken Mind" Data-Flow Machine.
                ------------------------------------------------------------
                Original 32-bit version for UNIX was founded && written by:
                                        Sancho Mining 08-09-1996 4:59:39.14pm */
  Usage0: BMDFMldr -h|--help
  Usage1: BMDFMldr -V|--versions
  Usage2: BMDFMldr [-q|--quiet] <FastLisp_file_name> [args...]
  Usage3: BMDFMldr [-c|--compile2disk] <FastLisp_file_name> [args...]
  Usage4: BMDFMldr [-q|--quiet] <Precompiled_FastLisp_file_name>

Runtime environment variable dump:
  BM_DFM_CONNECTION_FILE_path="/tmp/.BMDFMsrv";
  BM_DFM_CONNECTION_NPIP_path="/tmp/.BMDFMsrv_npipe";

VERSION_BMDFM_SYS_: "Sancho M. BMDFMSys V5.9.9.".
Compiled on: "SunOS SunOS_Ultra45 5.10 Generic_147147-26 sun4u sparc".
Compiled by: "cc: Sun C 5.10 SunOS_sparc Patch 141861-09 2012/08/15 as [ELF 64-
bit MSB executable SPARCV9 Version 1, dynamically linked, stripped] at systime
Fri Jul 13 13:08:19 CEST 2018".
$
```

Terminal (SGI MIPS; IRIX)

```
$ fastlisp
fastlisp ==> stderr:
/* fastlisp.c - FastLisp Compiler with Runtime Environment.
                ------------------------------------------------------------
                Original 32-bit version for UNIX was founded && written by:
                                        Sancho Mining 05-05-1996 10:10:29.18pm */
  Usage0: fastlisp -h|--help
  Usage1: fastlisp -V|--versions
  Usage2: fastlisp [-q|--quiet] <FastLisp_file_name> [args...]
  Usage3: fastlisp [-c|--compile2disk] <FastLisp_file_name> [args...]
  Usage4: fastlisp [-q|--quiet] <Precompiled_FastLisp_file_name>

The following environment variable:
  FAST_LISP_CFGPROFILE_path="fastlisp.cfg"
specifies a configuration profile that can be used for the Global FastLisp
function definitions. The format of the configuration profile is:
  <(DEFUN ...)>{ <(DEFUN ...)>}  # <EOF>.

Compiled on: "IRIX64 SGImipsIRIX 6.5 07202013 IP35".
Compiled by: "MIPSpro Compilers: Version 7.4.4m as [ELF 64-bit MSB mips-4 dynam
ic executable MIPS - version 1] at systime Fri Jul 13 13:05:43 MET DST 2018".
RETURNED STATUS: ABNORMAL PROGRAM TERMINATION.
$
```

Terminal (HP PA-RISC; HP-UX)

```
$ BMDFMldr -h
BMDFMldr ==> stdout:
/* BMDFMldr.c - The External Task PROC Unit (The Loader and Listener Pair)
                for the "Broken Mind" Data-Flow Machine.
                ------------------------------------------------------------
                Original 32-bit version for UNIX was founded && written by:
                                        Sancho Mining 08-09-1996 4:59:39.14pm */
  Usage0: BMDFMldr -h|--help
  Usage1: BMDFMldr -V|--versions
  Usage2: BMDFMldr [-q|--quiet] <FastLisp_file_name> [args...]
  Usage3: BMDFMldr [-c|--compile2disk] <FastLisp_file_name> [args...]
  Usage4: BMDFMldr [-q|--quiet] <Precompiled_FastLisp_file_name>

Runtime environment variable dump:
  BM_DFM_CONNECTION_FILE_path="/tmp/.BMDFMsrv";
  BM_DFM_CONNECTION_NPIP_path="/tmp/.BMDFMsrv_npipe";

VERSION_BMDFM_SYS_: "Sancho M. BMDFMSys V5.9.9.".
Compiled on: "HP-UX c8k-HPUX B.11.23 U 9000/785 4042425048".
Compiled by: "HP ANSI C / C++ B3910B C.03.70 (HP92453-01 B.11.11.16 HP C Compil
er) as [ELF-64 executable object file - PA-RISC 2.0 (LP64) / HPPA64 (PA-RISC2.0
W)] at systime Fri Jul 13 13:10:52 METDST 2018".
$
```

Terminal (IBM POWER RS/6000; AIX)

```
$ BMDFMsrv -h
BMDFMsrv ==> stdout:
/* BMDFMsrv.c - The "Broken Mind" Data-Flow Machine SMP MIMD Server Unit.
                ------------------------------------------------------------
                Original 32-bit version for UNIX was founded && written by:
                                        Sancho Mining 20-07-1996 2:49:49.58pm */
  Usage0: BMDFMsrv [-d|--daemonize]
  Usage1: BMDFMsrv -h|--help
  Usage2: BMDFMsrv [-d|--daemonize] -n|--no-logs
  Usage3: BMDFMsrv [-d|--daemonize] -l|--logfile <log_file_name>

Runtime environment variable dump:
  BM_DFM_CFGPROFILE_path="./BMDFMsrv.cfg";
  BM_DFM_PROCstat_path="./PROCstat";
  BM_DFM_CPUPROC_path="./CPUPROC";
  BM_DFM_OQPROC_path="./OQPROC";
  BM_DFM_IORBPROC_path="./IORBPROC";
  BM_DFM_CONNECTION_FILE_path="/tmp/.BMDFMsrv";
  BM_DFM_CONNECTION_NPIP_path="/tmp/.BMDFMsrv_npipe";
  BM_DFM_EMERGENCY_IPC_FILE_path="./freeIPC.inf";
  BM_DFM_LOGFILE_KEEP_NxSIZE="10x10000000";
  BM_DFM_PROCLOGFILE_KEEP_NxSIZE="10x10000000";
  BM_DFM_PROCLOGFILE_path="./PROCs.log";

VERSION_BMDFM_SYS_: "Sancho M. BMDFMSys V5.9.9.".
Compiled on: "AIX IBMaix 1 7 00CE8BB34C00".
Compiled by: "IBM XL C/C++ for AIX, V13.1.3 (5725-C72, 5765-J07) Version: 13.01
.0003.0000 as [64-bit XCOFF executable or object module] at systime Fri Jul 13
13:09:43 DFT 2018".
$
```

Terminal (DEC Alpha RISC; Tru64 OSF1)

```
$ BMDFMtrc -h
BMDFMtrc ==> stdout:
/* BMDFMtrc.c - The Interactive Tracer Unit
                for the "Broken Mind" Data-Flow Machine.
                ------------------------------------------------------------
                Original 32-bit version for UNIX was founded && written by:
                                        Sancho Mining 18-09-1996 1:22:49.50am */
  Usage0: BMDFMtrc
  Usage1: BMDFMtrc -h|--help
  Usage2: BMDFMtrc -l|--log-screen [<log_file_name>]

Runtime environment variable dump:
  BM_DFM_CONNECTION_FILE_path="/tmp/.BMDFMsrv";
  BM_DFM_CONNECTION_NPIP_path="/tmp/.BMDFMsrv_npipe";

VERSION_BMDFM_SYS_: "Sancho M. BMDFMSys V5.9.9.".
Compiled on: "OSF1 DECtru64alpha V5.1 2650 alpha".
Compiled by: "Compaq C V6.5-303 (dtk) on HP Tru64 UNIX V5.1B (Rev. 2650) Compil
er Driver V6.5-302 (dtk) cc Driver as [COFF format alpha dynamically linked, de
mand paged executable or object module stripped - version 3.14-2] at systime Fr
i Jul 13 13:10:34 WEST 2018".
$
```

Terminal (Intel Xeon Phi MIC; Linux)

```
$ CPUPROC
/* CPUPROC.c - The CPU PROC
               a part of the "Broken Mind" Data-Flow Machine.
               ------------------------------------------------------------
               Original 32-bit version for UNIX was founded && written by:
                                       Sancho Mining 20-07-1996 2:49:49.58pm */
Error: wrong number of arguments.
Usage: CPUPROC should be used by the BM_DFM Server only.

VERSION_BMDFM_SYS_: "Sancho M. BMDFMSys V5.9.9.".
Compiled on: "Linux RedHatELS62VM 2.6.32-220.13.1.el6.x86_64 #1 SMP Thu Mar 29
11:46:40 EDT 2012 x86_64".
Compiled by: "icc Intel(R) C Intel(R) 64 Compiler XE for applications running o
n Intel(R) 64, Version 15.0.0.090 Build 20140723 (Copyright (C) 1985-2014 Intel
 Corporation. All rights reserved.) as [ELF 64-bit LSB executable, Intel K1OM,
version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.4.0, s
tripped] at systime Fri Jul 13 13:01:31 CEST 2018".
RETURNED STATUS: ABNORMAL PROGRAM TERMINATION.
$
```

Terminal (Intel x86-64; Apple MacOS X)

```
$ OQPROC
/* OQPROC.c - The OQ PROC
              a part of the "Broken Mind" Data-Flow Machine.
              ------------------------------------------------------------
              Original 32-bit version for UNIX was founded && written by:
                                      Sancho Mining 20-07-1996 2:49:49.58pm */
Error: wrong number of arguments.
Usage: OQPROC should be used by the BM_DFM Server only.

VERSION_BMDFM_SYS_: "Sancho M. BMDFMSys V5.9.9.".
Compiled on: "Darwin MacIntel 14.5.0 Darwin Kernel Version 14.5.0: Tue Apr 11 1
6:12:42 PDT 2017; root:xnu-2782.50.9.2.3~1/RELEASE_X86_64 x86_64".
Compiled by: "Apple LLVM version 7.0.2 (clang-700.1.81) as [Mach-O 64-bit execu
table x86_64] at systime Fri Jul 13 13:02:16 CEST 2018"
RETURNED STATUS: ABNORMAL PROGRAM TERMINATION.
$
```

Terminal (Intel Itanium IA-64 EPIC VLIW; HP-UX)

```
$ IORBPROC
/* IORBPROC.c - The IORBP PROC
                a part of the "Broken Mind" Data-Flow Machine.
                ------------------------------------------------------------
                Original 32-bit version for UNIX was founded && written by:
                                        Sancho Mining 20-07-1996 2:49:49.58pm */
Error: wrong number of arguments.
Usage: IORBPROC should be used by the BM_DFM Server only.

VERSION_BMDFM_SYS_: "Sancho M. BMDFMSys V5.9.9.".
Compiled on: "HP-UX IA64hpux B.11.31 U ia64 2897190201".
Compiled by: "cc: HP C/aC++ B3910B A.06.28 [Nov 21 2013] as [ELF-64 executable
object file - IA64] at systime Fri Jul 13 13:18:27 MESZ 2018".
RETURNED STATUS: ABNORMAL PROGRAM TERMINATION.
$
```

Terminal (MCST Elbrus VLIW; Linux)

```
$ PROCstat
/* PROCstat.c - The PROC stat
                a part of the "Broken Mind" Data-Flow Machine.
                ------------------------------------------------------------
                Original 32-bit version for UNIX was founded && written by:
                                        Sancho Mining 20-07-1996 2:49:49.58pm */
Error: wrong number of arguments.
Usage: PROCstat should be used by the BM_DFM Server only.

VERSION_BMDFM_SYS_: "Sancho M. BMDFMSys V5.9.9.".
Compiled on: "Linux debian4babayan-64 2.6.33-elbrus.033.3.42 #1 SMP Thu Apr 23
22:28:28 MSK 2015 e2k".
Compiled by: "lcc:1.19.11:Dec-13-2014:e2k-2c+-linux (gcc version 4.4.0 compatib
le) as [ELF 64-bit LSB executable, MCST Elbrus, version 1 (SYSV), dynamically l
inked (uses shared libs), for GNU/Linux 2.6.33, stripped] at systime Fri Jul 13
 14:15:11 MSK 2018".
RETURNED STATUS: ABNORMAL PROGRAM TERMINATION.
$
```

Terminals

3. How do you solve *termcap* issues?

BMDFM uses the standard ***termcap*** database for the terminal capabilities. Should BMDFM display incorrectly, please use the following troubleshooting procedures:

Check the ***$TERM*** environment variable whether it contains a correct terminal name, which can be found in the ***termcap*** database. If not, then set the one that is correct. If the standard ***termcap*** database is missing in the system then use the one provided with the BMDFM distribution:

Terminal (bash)

```
$ export TERM=vt100
$ export TERMCAP=/<full_path>/termcap
$
```

Terminal

If the running BMDFM instance is ***daemonized*** (detached from a terminal) then ***termcap*** is initialized with the following default ***termcap*** settings skipping all roundtrips to the ***termcap*** database:

***BMDFMsrv.log* (termcap record)**

```
[TermCap]: TERM=ansi.sys. Init TERMCAP for the BM_DFM console done.
[TermCap]: Current termcap settings:
[TermCap]: TERM_TYPE=`ansi.sys'; LINES_TERM=`25'; COLUMNS_TERM=`80';
[TermCap]: CLRSCR_TERM=`\e[m\e[7h\e[2J'; REVERSE_TERM=`\e[7m'; BLINK_TERM=``;
[TermCap]: BOLD_TERM=`\e[1m'; NORMAL_TERM=`\e[m'; HIDECURSOR_TERM=``;
[TermCap]: SHOWCURSOR_TERM=``; GOTOCURSOR_TERM=`\e[%d;%dH'.
[TermCap]: Remote terminal device driver installed.
```

Fragment of ***BMDFMsrv.log*** file related to boot logs

The BMDFM runtime prefixes user's VM code with ***termcap*** variables. The variable names are the same as for the corresponding ***termcap*** functions and the assigned values are taken for the current terminal:

***fastlisp/BMDFMldr* log (PATTERN *No# 2* for *fastlisp* or *No# 3* for *BMDFMldr*)**

```
Modifying the FastLisp code (PATTERN No# 2)...
  (PROGN {(SETQ <termcap_var> <termcap_val>) }<FastLisp_prog>)
. . .
(PROGN
  (SETQ@S MAIN:TERM_TYPE@S "linux")
  (SETQ@I MAIN:LINES_TERM@I 25)
  (SETQ@I MAIN:COLUMNS_TERM@I 80)
  (SETQ@S MAIN:CLRSCR_TERM@S "\e[H\e[J")
  (SETQ@S MAIN:REVERSE_TERM@S "\e[7m")
  (SETQ@S MAIN:BLINK_TERM@S "\e[5m")
  (SETQ@S MAIN:BOLD_TERM@S "\e[1m")
  (SETQ@S MAIN:NORMAL_TERM@S "\e[m")
  (SETQ@S MAIN:HIDECURSOR_TERM@S "\e[?25l")
  (SETQ@S MAIN:SHOWCURSOR_TERM@S "\e[?25h")
  (SETQ@S MAIN:GOTOCURSOR_TERM@S "\e[%i%d;%dH")
```

Fragments of ***fastlisp/BMDFMldr*** log related to initialization phase

A user can choose to use ***termcap*** functions or variables. However, remember that the ***termcap*** functions are evaluated by the ***CPUPROC*** processes that could be started somewhere on a different terminal having different ***termcap*** settings:

VM code fragment using *termcap* functions

```
(if (|| (|| (!= term_type (term_type)) (!= lines_term (lines_term))) (!= columns_term (columns_term)))
  (while 1 (progn
    (outf "\nChoose terminal:\n" nil)
    (outf "  0 - TERM_TYPE=`%s';" term_type)     (outf " LINES_TERM=`%d';" lines_term)
    (outf " COLUMNS_TERM=`%d';\n" columns_term) (outf "  1 - TERM_TYPE=`%s';" (term_type))
    (outf " LINES_TERM=`%d';" (lines_term))     (outf " COLUMNS_TERM=`%d'.\n" (columns_term))
    (outf "Enter your choice (0 or 1) or press 'q' to quit:" nil)
    (setq ch (upper (scan_console 5000000)))
    (if (|| (== ch "Q") (== (asc ch) 3))
      (exit)
      (if (== ch "0")
        (break)
        (if (== ch "1")
          (progn
            (setq term_type (term_type))                  (setq lines_term (lines_term))
            (setq columns_term (columns_term))            (setq clrscr_term (clrscr_term))
            (setq reverse_term (reverse_term))            (setq blink_term (blink_term))
            (setq bold_term (bold_term))                  (setq normal_term (normal_term))
            (setq hidecursor_term (hidecursor_term))      (setq showcursor_term (showcursor_term))
            (setq gotocursor_term (gotocursor_term -1 -1))
            (break)
          )
          (if (== (asc ch) 0)
            nil
            (outf "\n\n*** Invalid selection ***\n" nil)
          )
        )
      )
    )
  ))
  nil
)
```

VM code fragment

4. How do you change the default shared memory and semaphore limits on Linux?

On Linux, the shared memory limits (both ***shmmax*** and ***shmall***) might be set to a low value by default. However, they can be changed on the ***/proc*** file system (no reboot needed). For example, to allow one 64GB:

```
Terminal
$ su
# echo 68719476736 >/proc/sys/kernel/shmall
# echo 68719476736 >/proc/sys/kernel/shmmax
#
```

Terminal

A user can add these commands into a script that is executed at boot time. Alternatively, a user can use the ***sysctl*** utility, if available, to control these parameters. The following lines can be added to a file called ***/etc/sysctl.conf***:

```
/etc/sysctl.conf
# . . .
kernel.shmall = 68719476736
kernel.shmmax = 68719476736
# . . .
```

/etc/sysctl.conf

This file is usually processed at boot time, but ***sysctl*** can be called from the command line as well.

```
Terminal
$ su
# sysctl -w kernel.shmall=68719476736
# sysctl -w kernel.shmmax=68719476736
#
```

Terminal

The same strategy can be applied to the default semaphore limits (***semmni***, ***semmsl*** and ***semmns***).

Consider configuring BMDFM with ***POSIX-semaphores***, which scale and perform better than ***SVR4-semaphores***. The number of ***POSIX-semaphores*** is not limited. ***POSIX-semaphores*** may have greater values than ***SVR4-semaphores*** (BMDFM resources are limited by the maximal semaphore value).

5. How do you find out whether the OS is able to provide enough semaphores for BMDFM?

If the OS kernel is configured with too few semaphore resources, BMDFM will not start at all, giving an error message indicating insufficient semaphore resources. Most critical consumers of the semaphore resources are ***OQ*** (***Operation Queue***) and ***DB*** (***Data Buffer***), depending on their sizes. The BMDFM boot logs show the number of semaphores in "<**obtained**>/<**required**>" format. Even if the log is scrolled out of the screen, all records can be found in the BMDFM server log file:

BMDFMsrv.log (successful sema4 record)

```
[OSInfo]: Current UNIX SVR4 IPC limits:
[OSInfo]:  sem: semaphore constants are not available.
[OSInfo]:  shm: shared memory constants are not available.
[OSInfo]: Current POSIX SEMA4 limits:
[OSInfo]:  sem: semaphore constants are not available.
[SysMsg]: Organizing an abstract DFM UNIT STRUCTURE in the SHMEM_POOL:
[SysMsg]:   Initializing CPU PROC state array...
[SysMsg]:   Organizing DFM IORBPs...
[SysMsg]:   Collecting system semaphores for the OQ and DB...
[SysMsg]:     SemOQ=2000/2000, SemDBAreas=28000/28000.
[SysMsg]:   Organizing DFM OQ...
[SysMsg]:   Organizing DFM DB...
```

Fragment of ***BMDFMsrv.log*** file related to boot logs

BMDFM can also function even with fewer semaphores than required. However, performance degradation can be observed in this case because all available semaphores are distributed along ***OQ*** and ***DB*** with certain interleaves. It is worth paying attention to the following possible warning message in the logs:

BMDFMsrv.log (not very successful sema4 record)

```
[OSInfo]: Current UNIX SVR4 IPC limits:
[OSInfo]:  sem: semaphore constants are not available.
[OSInfo]:  shm: shared memory constants are not available.
[OSInfo]: Current POSIX SEMA4 limits:
[OSInfo]:  sem: semaphore constants are not available.
[SysMsg]: Organizing an abstract DFM UNIT STRUCTURE in the SHMEM_POOL:
[SysMsg]:   Initializing CPU PROC state array...
[SysMsg]:   Organizing DFM IORBPs...
[SysMsg]:   Collecting system semaphores for the OQ and DB...
[SysMsg]:     WARNING!!! Poor resource of the system semaphores.
[SysMsg]:     SemOQ=412/3000, SemDBAreas=5507/40000.
[SysMsg]:   Organizing DFM OQ...
[SysMsg]:   Organizing DFM DB...
```

Fragment of ***BMDFMsrv.log*** file related to boot logs

It is also worth remembering the known fact that the semaphore resources (like all other IPC resources) can remain occupied in the OS kernel. Though BMDFM always cleans up its IPC resources correctly, it makes sense to check IPC resources after an unintentional crash situation. The standard ***ipcs*** and ***ipcrm*** utilities can be used for this purpose. Besides, BMDFM has its own utility called ***freeIPC***. This utility relies on the ***freeIPC.inf*** file with IPC resource descriptors used and recorded by the BMDFM Server.

Here is a hint on how to create a ***Purge_BMDFM.sh*** shell script able to purge the OS correctly from a single instance of BMDFM:

Purge_BMDFM.sh

```
#!/bin/sh

export BM_DFM_CONNECTION_FILE_path="/tmp/.BMDFMsrv";
export BM_DFM_CONNECTION_NPIP_path="/tmp/.BMDFMsrv_npipe";
export BM_DFM_EMERGENCY_IPC_FILE_path="./freeIPC.inf";

killall -9 BMDFMsrv BMDFMldr BMDFMtrc PROCstat CPUPROC OQPROC IORBPROC 2>/dev/null

rm -f $BM_DFM_CONNECTION_FILE_path $BM_DFM_CONNECTION_NPIP_path 2>/dev/null

freeIPC
```

Purge_BMDFM.sh shell script

Consider configuring BMDFM with ***POSIX-semaphores***, which scale and perform better than ***SVR4-semaphores***. The number of ***POSIX-semaphores*** is not limited. ***POSIX-semaphores*** may have greater values than ***SVR4-semaphores*** (BMDFM resources are limited by the maximal semaphore value).

6. How do you start the BMDFM Server detached from a terminal and control it later?

A user can start ***BMDFMsrv*** from the command line with ***--daemonize*** option. Later on, the started instance can be controlled through the BMDFM external named pipe. A second terminal can be used for dynamic logging. Such an open architectural approach even allows a user to write a kind of her/his own BMDFM Remote Console:

Terminal 0 (bash)

```
$ export BMDFM_LOG_FILENAME=BMDFMsrv.log
$ export BMDFM_ERR_FILENAME=BMDFMsrv.err
$ export BM_DFM_CONNECTION_NPIP_path=/tmp/.BMDFMsrv_npipe
$ BMDFMsrv --daemonize --logfile $BMDFM_LOG_FILENAME 2>$BMDFM_ERR_FILENAME &
$ echo command >$BM_DFM_CONNECTION_NPIP_path
$ echo command down down >$BM_DFM_CONNECTION_NPIP_path
$
```

Terminal 1 (csh)

```
$ setenv BMDFM_LOG_FILENAME BMDFMsrv.log
$ tail -f $BMDFM_LOG_FILENAME
[Vers]: Binary Modular Data-Flow Machine (BM_DFM) Release History
[Vers]: and Codenames:
[Vers]:    Years     | Versions    | BM_DFM Codename       | Release
[Vers]:   -----------+-------------+-----------------------+------------
[Vers]:    1996-1997 | 0.0.1-1.9.9 | "Bare Metal" DFM      | Official
[Vers]:    1998-1999 | 2.0.0-2.9.9 | "Big Monster" DFM     | Unofficial
[Vers]:    2000-2001 | 3.0.0-3.9.9 | "Beast Master" DFM    | Unofficial
[Vers]:    2002-2003 | 4.0.0-4.9.9 | "Behemoth Mighty" DFM | Official
[Vers]:    2004-2015 | 5.0.0-5.9.9 | "Broken Mind" DFM     | Official
[Vers]: VERSION_BMDFM_SYS_=`Sancho M. BMDFMSys V5.9.9.' # The BM_DFM Server.
[Vers]: VERSION_TERMCAP___=`Sancho M. TermCap v.1.2.0.' # Term capabilities.
[Vers]: VERSION_FSTLISP___=`Sancho M. FstLisp v.2.9.6.' # FastLisp RTEngine.
[Vers]: VERSION_CFLPUDF___=`Sancho M. CFLPUDF v.1.0.0.' # FastLSP UDFs in C.
[Vers]: VERSION_STRGLIB___=`Sancho M. StrgLib v.2.2.5.' # FstString library.
[Vers]: VERSION_MEMPOOL___=`Sancho M. MemPool v.2.8.8.' # ShMem Pool driver.
[TermCap]: ~~~~~~~~~~~~~~~~~~~~ Server is running on TERM=ansi.sys (80x25).
[SysMsg]: Overall machinery init for the virtual out-of-order general purpos
e processing was completed at systime Fri Nov 13 18:26:04 2015.
[SysMsg]: Going simultaneous jobs running all the threads in parallel...
[DFMSrv]: All resources were unhooked and invoked successfully!
[SysMsg]: The complete "Broken Mind" Data-Flow Machine Server has been fully
 started.
[SysMsg]: The entire DFM SMP MIMD architecture is ready for dynamic scheduli
ng now.
[Legacy_MainFrame_Initial_Greeting_Message]: GOOD EVENING.
[SysMsg]: A message routed out of the NPIPE at systime Fri Nov 13 18:26:13 2
015.
npipe[COMMAND]:  [MSG#0]

_____________
Console input:
[SysMsg]: ========== System time is Fri Nov 13 18:26:13 2015. ==========
[Err]: *** Boom! Invalid command!
[Msg]: Type `help' or `?' to see the list of possible commands!
[Msg]: The commands will also be accepted from the external named pipe:
[Msg]:   `/tmp/.BMDFMsrv_npipe' in "COMMAND <command>\n" format.
[SysMsg]: A message routed out of the NPIPE at systime Fri Nov 13 18:26:22 2
015.
npipe[COMMAND]: down down [MSG#1]

_____________
Console input: down down
[SysMsg]: ========== System time is Fri Nov 13 18:26:22 2015. ==========
[SysMsg]: Now, the BM_DFM Server is urgently going down...
[SysMsg]: Destroying the external connection file...
[SysMsg]: Destroying the ExtTask(Trace) nFIFO pipe...
[SysMsg]: Sending SIGINT to ExtTasks in TCZ...
[SysMsg]: Sending SIGTERM to ExtTasks in TCZ...
[SysMsg]: Sending SIGKILL to ExtTasks in TCZ...
[SysMsg]: Sending SIGKILL to ExtTraces in TPA...
[SysMsg]: Sending SIGKILL to the PROCstat...
[SysMsg]: Sending SIGKILL to the CPU PROCs...
[SysMsg]: Sending SIGKILL to the OQ PROCs...
[SysMsg]: Sending SIGKILL to the IORBP PROCs...
[SysMsg]: Invoking taskjob_end_callback()...
[SysMsg]: Deinitializing BM_DFM...
[DFMSrv]: Release semaphores done.
[DFMSrv]: Close msg PROC pipe done.
[MemPool]: The shared memory pool deinit done.
[SysMsg]: Destroying the freeIPC EMERGENCY CASE file...
[SysMsg]: SHUTDOWN completed at systime Fri Nov 13 18:26:23 2015.
[Legacy_MainFrame_Final_Message]: GOOD BYE.
[SysMsg]: Closing the logs `./BMDFMsrv.log'...
*** Logfile is closed at systime Fri Nov 13 18:26:23 2015 ***
^C
$
```

Terminal 0 and Terminal 1

Obviously, the best practice would be to source all BMDFM environment variables in a working shell and to create a script for the BMDFM Server console commands (one script for all commands or separate scripts for each command) as shown in the examples below:

BMDFMcmd.sh

```
#!/bin/sh

echo command $* >$BM_DFM_CONNECTION_NPIP_path;
tail -200 $BMDFM_LOG_FILENAME
```

downdown.sh

```
#!/bin/sh

echo command down down >$BM_DFM_CONNECTION_NPIP_path;
tail -100 $BMDFM_LOG_FILENAME
```

BMDFMcmd.sh and ***downdown.sh*** shell scripts

7. How do you start many instances of BMDFM on the same machine?

By default, it is not possible to start many instances of BMDFM on the same machine because the BMDFM Server checks for existence and creates both the ***/tmp/.BMDFMsrv*** connection file and the ***/tmp/.BMDFMsrv_npipe*** connection named pipe in the ***/tmp*** directory. However, those default names (as well as other default names) can be redefined via the corresponding environment variables. As an example, the following ***BMDFMrun0.sh*** shell script can start an additional unique local BMDFM instance:

BMDFMrun0.sh

```
#!/bin/sh

export BM_DFM_CFGPROFILE_path="./BMDFMsrv0.cfg";
export BMDFM_LOG_FILENAME="./BMDFMsrv0.log";
export BMDFM_ERR_FILENAME="./BMDFMsrv0.err";
export BM_DFM_CONNECTION_FILE_path="./.BMDFMsrv0";
export BM_DFM_CONNECTION_NPIP_path="./.BMDFMsrv0_npipe";
export BM_DFM_EMERGENCY_IPC_FILE_path="./freeIPC0.inf";

BMDFMsrv --logfile $BMDFM_LOG_FILENAME 2>$BMDFM_ERR_FILENAME
```

BMDFMrun0.sh startup shell script

It is also not a bad idea to source the mentioned variables in a user shell environment to be reused by ***BMDFMldr***, ***BMDFMtrc*** and ***freeIPC*** if necessary.

One more important thing to remember here is the number of used SVR4 semaphores. In other words, it is important to prevent a situation where one running BMDFM instance holds all available SVR4 semaphores in the system, blocking startup of other BMDFM instances. The ***OQ_DB_SEM_LIMIT*** configuration parameter of the BMDFM configuration profile serves exactly this purpose. The owner of a BMDFM instance is responsible to set this value correctly, based upon the number of all available SVR4 semaphores in the system and the number of BMDFM instances planned to be run simultaneously. All owners, for example, can have a kind of settlement agreement regarding the allowed SVR4 semaphore quota per instance.

Consider configuring BMDFM with ***POSIX-semaphores***, which scale and perform better than ***SVR4-semaphores***. The number of ***POSIX-semaphores*** is not limited. ***POSIX-semaphores*** may have greater values than ***SVR4-semaphores*** (BMDFM resources are limited by the maximal semaphore value).

8. How do you get a list of recognizable parameters for the BMDFM configuration profile?

The ***dfmkernel*** and ***dfmserver*** commands of the BMDFM Server console display all possible configuration parameters with their current values separated by "=" sign:

Output of *dfmkernel*

```
Console input: dfmkernel
[SysMsg]: ========== System time is Fri Nov 13 18:36:43 2015. ==========
[DFMKrnl]: Global parameters of the BM_DFM Kernel:
[DFMKrnl]: Operation Queue (OQ) size: Q_OQ=1000Entities.
[DFMKrnl]: Data Buffer (DB) size: Q_DB=500Entities.
[DFMKrnl]: I/O Ring Buffer Port (IORBP) size: Q_IORBP=100Entities.
[DFMKrnl]: Number of the IORBPs: N_IORBP=10.
[DFMKrnl]: Number of the main processes (CPU PROCs): N_CPUPROC=8.
[DFMKrnl]: Number of the OQ PROCs: N_OQPROC=8.
[DFMKrnl]: Number of the IORBP PROCs: N_IORBPPROC=8.
[DFMKrnl]: Block size used in OQ search algorithm is 62.
[DFMKrnl]: Size of caches in speculative prediction unit is 64000Bytes.
[DFMKrnl]: Associative hierarchy of speculative tagging max. 532000Bytes.
[DFMKrnl]: Display stall warnings: STALL_WARNINGS=NO.
[DFMKrnl]: Hard array synchronization: HARD_ARRAY_SYNCHRO=NO.
[DFMKrnl]: I/O synchronization of external task: EXT_IN_OUT_SYNCHRO=YES.
[DFMKrnl]: Compensate ShMem relaxed consistency: RELAXED_CNSTN_SM_MODEL=YES.
[DFMKrnl]: Use SVR4 or POSIX semaphores: POSIX_SEMA4_SYNC=RW+COUNT.
[DFMKrnl]:   SVR4 sema4 is replaced with POSIX sema4 where possible.
```

Output of *dfmserver*

```
Console input: dfmserver
[SysMsg]: ========== System time is Fri Nov 13 18:36:53 2015. ==========
[DFMSrv]: PIDs of the BM_DFM processes:
[DFMSrv]:  N#          | CPUPROCs    | OQPROCs     | IORBPPROCs | PROCstat
[DFMSrv]: -------------+-------------+-------------+------------+------------
[DFMSrv]:            0 |       14926 |       14934 |       14942 |       14925
[DFMSrv]:            1 |       14927 |       14935 |       14943 |
[DFMSrv]:            2 |       14928 |       14936 |       14944 |
[DFMSrv]:            3 |       14929 |       14937 |       14945 |
[DFMSrv]:            4 |       14930 |       14938 |       14946 |
[DFMSrv]:            5 |       14931 |       14939 |       14947 |
[DFMSrv]:            6 |       14932 |       14940 |       14948 |
[DFMSrv]:            7 |       14933 |       14941 |       14949 |
[DFMSrv]:   CPU PROC is multithreaded: CPUPROC_MTHREAD=NO.
[DFMSrv]:   OQ PROC is multithreaded: OQPROC_MTHREAD=NO.
[DFMSrv]:   IORBP PROC is multithreaded: IORBPPROC_MTHREAD=NO.
[DFMSrv]:   BMDFMldr is multithreaded: BMDFMLDR_MTHREAD=NO.
[DFMSrv]:   Thread-Local Storage (TLS) verification: MTHREAD_TLS_CHECK=NO.
[DFMSrv]: Allow CPU PROC Address Space Layout Randomization (ASLR): ALLOW_CPUPROC_ASLR=NO.
[DFMSrv]: Global parameters of the BM_DFM Server:
[DFMSrv]: AGGRESSIVE compilation: SPECULATIVE_RISC_ARCH = 1(yes).
[DFMSrv]: Own system SHM_SEMAPHORE: REENTERANT_SHMEM_POOL = 1(yes).
[DFMSrv]: PID of the BM_DFM Server is 14915.
[DFMSrv]: Number of SVR4 semaphores per group is 250.
[DFMSrv]: Maximal SVR4/POSIX semaphore value is 2147483647.
[DFMSrv]: ShMemPool mount address (0=auto): SHMEM_POOL_MNTADDR=999999999.
[DFMSrv]: ShMemPool size: SHMEM_POOL_SIZE=500000000Bytes.
[DFMSrv]: Number of ShMemPool banks: SHMEM_POOL_BANKS=10Banks.
[DFMSrv]: ShMemPool and sema4 permissions are: SHMEM_POOL_PERMS=432.
[DFMSrv]:   (0660=="rw-rw----").
[DFMSrv]: Array block size: ARRAYBLOCK_SIZE=64Entities.
[DFMSrv]: OQ function argument count: OQ_FUNC_ARG_COUNT=32Entities.
[DFMSrv]: Time to scan DFM for statistic: T_STATISTIC=1Second.
[DFMSrv]: Max number of OQ&&DB semaphores (0=unlim): OQ_DB_SEM_LIMIT=0.
[DFMSrv]: Number of the Trace Ports (TPs): N_TRACEPORT=5.
[DFMSrv]: Heartbeats for the CPU, OQ && IORBP PROCs: PROC_HEARTBEATS=YES.
[DFMSrv]: Detection of dataflow stall hazards: DFSTLHAZARD_DETECT=YES.
[DFMSrv]: Allow dropping nonproductive instructions: ALLOW_DROP_NONPROD=NO.
[DFMSrv]: Server console logs are enabled.
[DFMSrv]:   Logs are in `./BMDFMsrv.log'.
[DFMSrv]:   Keeping 10 old logfiles (10000000Bytes each).
[DFMSrv]: Registration logs for the CPU && IORBP PROCs: PROC_CPU_LOGS=NO.
[DFMSrv]: Runtime ErrCode for `ShMemPool space exhausted' is 252.
[DFMSrv]: Runtime ErrCode for `Dataflow stall hazard' is 253.
[DFMSrv]: Signal to reset/get used CPU time in child PROCs is 10 (irq).
[DFMSrv]: Signal to unhook child PROCs out of a semaphore is 12 (irq).
[DFMSrv]: Msg PROC unnamed pipe R/W IDs: rID=6, wID=7.
[DFMSrv]: External task named pipe `/tmp/.BMDFMsrv_npipe' R/W ID=8.
```

Output of the ***dfmkernel*** and ***dfmserver*** commands on the BMDFM Server console

9. Where can it be necessary to change the mounting address of the shared memory segment?

The shared memory segment is created, mounted and initialized by the BMDFM Server. Later on, all other BMDFM processes mount the shared memory segment to their own virtual address spaces. By default, the mounting address is chosen by the BMDFM Server and the OS automatically. This mounting address is the same (and it must be the same) for all other processes. The BMDFM Server is able to assign the mounting address automatically because the size of its code segment is practically the same as the code segment sizes of other processes and, additionally, a dynamic linker links practically against the same runtime libraries so that they do not overlap the virtual address space of the shared memory segment. The standard ***ldd*** utility is useful to get an idea of which runtime libraries are in use and which mounting address to choose manually if necessary:

```
Terminal
$ ldd BMDFMsrv
        linux-vdso.so.1 =>  (0x00007fff55dff000)
        libm.so.6 => /lib64/libm.so.6 (0x00000034a8600000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00000034a9200000)
        libc.so.6 => /lib64/libc.so.6 (0x00000034a8a00000)
        /lib64/ld-linux-x86-64.so.2 (0x00000034a8200000)
$ ldd BMDFMldr
        linux-vdso.so.1 =>  (0x00007fff0f3ff000)
        libm.so.6 => /lib64/libm.so.6 (0x00000034a8600000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00000034a9200000)
        libc.so.6 => /lib64/libc.so.6 (0x00000034a8a00000)
        /lib64/ld-linux-x86-64.so.2 (0x00000034a8200000)
$ ldd BMDFMtrc
        linux-vdso.so.1 =>  (0x00007fff4c078000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00000034a9200000)
        libc.so.6 => /lib64/libc.so.6 (0x00000034a8a00000)
        /lib64/ld-linux-x86-64.so.2 (0x00000034a8200000)
$ ldd PROCstat
        linux-vdso.so.1 =>  (0x00007fff11983000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00000034a9200000)
        libc.so.6 => /lib64/libc.so.6 (0x00000034a8a00000)
        /lib64/ld-linux-x86-64.so.2 (0x00000034a8200000)
$ ldd CPUPROC
        linux-vdso.so.1 =>  (0x00007fff947b3000)
        libm.so.6 => /lib64/libm.so.6 (0x00000034a8600000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00000034a9200000)
        libc.so.6 => /lib64/libc.so.6 (0x00000034a8a00000)
        /lib64/ld-linux-x86-64.so.2 (0x00000034a8200000)
$ ldd OQPROC
        linux-vdso.so.1 =>  (0x00007fffe51ff000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00000034a9200000)
        libc.so.6 => /lib64/libc.so.6 (0x00000034a8a00000)
        /lib64/ld-linux-x86-64.so.2 (0x00000034a8200000)
$ ldd IORBPROC
        linux-vdso.so.1 =>  (0x00007fff78bff000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00000034a9200000)
        libc.so.6 => /lib64/libc.so.6 (0x00000034a8a00000)
        /lib64/ld-linux-x86-64.so.2 (0x00000034a8200000)
$
```

Terminal

Even when a user extends the VM with his own implementations written in C/C++, those implementations are still linked against ***BMDFMsrv***, ***BMDFMldr*** and ***CPUPROC***, maintaining the equality of code segment sizes and the same set of runtime libraries.

However, the following exceptional cases exist where a manually chosen mounting address is required:

- A user prefers to link some of the BMDFM processes statically and some of them dynamically.
- A conditional compilation is applied that results in linking of different code sizes against the BMDFM processes (and possibly a different set of runtime libraries).

The ***SHMEM_POOL_MNTADDR*** configuration parameter of the BMDFM configuration profile lets you set the mounting address of the shared memory segment manually as needed.

10. Is there any difference between a memory descriptor and a memory address?

Memory descriptors are used only for backward compatibility with previous versions of BMDFM. The current implementation of BMDFM always returns memory address in case either of a memory descriptor or a memory address. This will also be supported in future versions of BMDFM. The two following VM code fragments are equivalent; the second one is recommended and preferable:

VM code fragment (obsolete)

```
(setq mem_descr (asyncheap_create size))
(setq mem_addr (asyncheap_getaddress mem_descr))
```

VM code fragment (recommended)

```
(setq mem_addr (asyncheap_create size))
```

VM code fragments

The returned memory addresses are always aligned to the size of a long integer (4 bytes in case of 32-bit BMDFM and 8 bytes in case of 64-bit BMDFM). All standard built-in ***asyncheap***-functions work correctly with such an alignment, they work even where float-alignment is required and on all RISC-processors (note: ***x86***, ***x86-64*** and ***IA-64*** are able to tolerate misaligned data in contrast to most RISC-processors). In most use cases, a user writes his own functions in C/C++ that are consumers of the returned memory addresses. Normally, it makes sense to align the addresses locally within every user-defined function, keeping the original addresses for ***asyncheap_delete*** function. The following is a recommended example for address alignment:

Pattern for the address alignment (pseudo-code)

```
addr -> block allocated with size alignment_size*(NumberOfEntities+1)
addr = addr + (alignment_size - abs(addr % alignment_size))
```

VM code

```
(defun float_size (len (dump_f2s 0.)))

(defun udf (progn
  (setq addr (+ 0 $1))

  # alignment
  (setq addr (+ addr (- (float_size) (iabs (% addr (float_size))))))

  # . . .

))

(setq addr (asyncheap_create (* (float_size) (++ NumberOfEntities))))
(udf addr)
(asyncheap_delete addr)
```

UDF written in C

```
#define ULO unsigned long int
#define SLO signed long int
#define DFL double

void udf(const ULO *dat_ptr, struct fastlisp_data *ret_dat){
  DFL *float_array;
  ret_ival(dat_ptr,(SLO*)&float_array);

  // alignment
  (ULO)float_array+=(sizeof(DFL)-(ULO)float_array%sizeof(DFL));

  // . . .

  return;
}
```

Address alignment in VM code or in C code

11. How does BMDFM handle strings internally?

BMDFM processes strings in the format similar to the ***Hollerith string representation*** using ***COW***-policy (***Copy-on-Write***). A string itself always stores its length followed by its contents terminated with number of zeros aligned to the size of ***long***. A pointer to the string always points to the string contents making it compatible with the standard null-terminated C-strings:

String format

```
CHR = char
ULO = unsigned long int

Addr Low                                                    Addr High

<ULO string_size><string><zero_char><zero_char_alignment_to_ULO_size>
                  ^
                  |
CHR *string_ptr -'
```

Sample C code using the BMDFM strings

```
#define CHR char

CHR *str0=NULL,*str1=NULL,*str2=NULL;
get_std_buff(&str0,"To be or not to be");
get_std_buff(&str1,"be");
get_std_buff(&str2,"compute");
upper(&str0,strtran(&str0,str0,str1,str2));
printf("`%s'\n",str0);
free_string(&str0);
free_string(&str1);
free_string(&str2);
```

BMDFM strings

The implemented set of the string processing functions is basically equal to the same set on the FastLisp level:

BMDFM string library

```
#define CHR char
#define UCH unsigned char
#define SCH signed char
#define USH unsigned short int
#define SSH signed short int
#define ULO unsigned long int
#define SLO signed long int
#define DFL double

CHR *mk_std_buff(CHR **buff, ULO size);
CHR *mk_std_buff_secure(CHR **buff, ULO size);
CHR *mk_fst_buff(CHR **buff, ULO size);
CHR *mk_fst_buff_secure(CHR **buff, ULO size);
CHR *get_std_buff(CHR **targ, const CHR *buff);
CHR *get_std_buff_secure(CHR **targ, const CHR *buff);
UCH notempty(const CHR *string);
ULO len(const CHR *string);
ULO at(const CHR *pattern, const CHR *among);
ULO rat(const CHR *pattern, const CHR *among);
UCH cmp(const CHR *string1, const CHR *string2);
SCH cmp_s(const CHR *string1, const CHR *string2);
CHR *equ(CHR **targ, const CHR *source);
CHR *equ_secure(CHR **targ, const CHR *source);
CHR *equ_num(CHR **targ, SLO num);
CHR *equ_fnum(CHR **targ, DFL fnum);
CHR *cat(CHR **targ, const CHR *source);
CHR *lcat(CHR **targ, const CHR *source);
CHR *space(CHR **targ, ULO pos);
CHR *replicate(CHR **targ, const CHR *source, ULO num);
CHR *left(CHR **targ, const CHR *source, ULO pos);
CHR *leftr(CHR **targ, const CHR *source, ULO posr);
CHR *right(CHR **targ, const CHR *source, ULO pos);
CHR *rightl(CHR **targ, const CHR *source, ULO posl);
CHR *substr(CHR **targ, const CHR *source, ULO from, ULO pos);
CHR *strtran(CHR **targ, const CHR *source, const CHR *pattern, const CHR *subst);
CHR *ltrim(CHR **targ, const CHR *source);
CHR *rtrim(CHR **targ, const CHR *source);
CHR *alltrim(CHR **targ, const CHR *source);
CHR *pack(CHR **targ, const CHR *source);
CHR *head(CHR **targ, const CHR *source);
CHR *tail(CHR **targ, const CHR *source);
CHR *lsp_head(CHR **targ, const CHR *source);
CHR *lsp_tail(CHR **targ, const CHR *source);
CHR *upper(CHR **targ, const CHR *source);
CHR *lower(CHR **targ, const CHR *source);
CHR *rev(CHR **targ, const CHR *source);
CHR *padl(CHR **targ, const CHR *source, ULO width);
CHR *padr(CHR **targ, const CHR *source, ULO width);
CHR *padc(CHR **targ, const CHR *source, ULO width);
CHR *strraw(CHR **targ, const CHR *source);
CHR *strunraw(CHR **targ, const CHR *source);
CHR *strdump(CHR **targ, const CHR *source);
CHR *string_time(CHR **targ);
CHR *strings_version(CHR **targ);
CHR *sch2str(CHR **targ, SCH num);
CHR *ssh2str(CHR **targ, SSH num);
CHR *slo2str(CHR **targ, SLO num);
CHR *ptr2str(CHR **targ, void *ptr);
CHR *dfl2str(CHR **targ, DFL num);
SCH str2sch(const CHR *string);
SSH str2ssh(const CHR *string);
SLO str2slo(const CHR *string);
void *str2ptr(const CHR *string);
DFL str2dfl(const CHR *string);
CHR *free_string(CHR **targ);
```

BMDFM string library

12. Why use *USER_IO*?

The direct purpose of BMDFM is fast parallel processing of data. If a specific input/output is required, it can be implemented as a standalone process providing data to BMDFM and taking processed data from BMDFM through files or pipes. However, it is not prohibited to implement such a specific input/output within BMDFM itself as user-defined functions extending the VM. In this case, it is not a big deal to write a couple of C-functions, something like ***device_open()***, ***device_read()***, ***device_write()*** and ***device_close()***. If access to such a device does not require having a process-associated descriptor with ***stateful*** data structures behind it, then there is no problem at all – ***stateless*** calls to the device will be synchronized on the BMDFM dataflow engine, and the ***CPUPROC*** processes will cooperatively execute the calls. The problem appears in a situation where the specific input/output requires a process-associated device descriptor having ***stateful*** data structures behind it, thus, the calls must be executed in the same process address space. Exactly for this purpose, the following VM functions are always executed by the ***BMDFMldr*** process but not ***CPUPROC*** processes:

VM functions

```
(accept <SVal_prompt_message_for_console_or_empty_for_stdin>)
(scan_console <IVal_wait_key_forever_if_1_or_useconds_if_positive>)
(file_create <SVal_file_name>)
(file_open <SVal_file_name>)
(file_write <IVal_file_descriptor> <SVal_string_to_be_written>)
(file_read <IVal_file_descriptor> <IVal_number_of_bytes_to_be_read>)
(file_seek_beg <IVal_file_descriptor> <IVal_offset_in_bytes_from_file_beginning>)
(file_seek_cur <IVal_file_descriptor> <IVal_offset_in_bytes_from_file_current_offset>)
(file_seek_end <IVal_file_descriptor> <IVal_offset_in_bytes_from_file_end>)
(file_close <IVal_file_descriptor>)
(file_remove <SVal_file_name>)
(user_io <IVal_user_defined_integer> <IVal_user_defined_string>)
```

List of the VM functions executed by ***BMDFMldr***

Hence, the following pattern is recommended to implement the specific input/output that requires a process-associated device descriptor:

VM code

```
(setq DEVICE_OPEN  (<< 1 20))
(setq DEVICE_READ  (<< 2 20))
(setq DEVICE_WRITE (<< 3 20))
(setq DEVICE_CLOSE (<< 4 20))

(setq XML_data (accept ""))  # input XML chunk
(setq descr (ival (user_io DEVICE_OPEN "Specific Device: XML")))
(user_io (| DEVICE_WRITE descr) XML_data)
(setq XML_data (user_io (| DEVICE_READ descr) ""))
(user_io (| DEVICE_CLOSE descr) "")
XML_data  # output XML chunk
```

USER_IO **callback written in C**

```
#define CHR char
#define SLO signed long int

#define DEVICE_OPEN  (SLO)(1<<20)
#define DEVICE_READ  (SLO)(2<<20)
#define DEVICE_WRITE (SLO)(3<<20)
#define DEVICE_CLOSE (SLO)(4<<20)

void user_io_callback(SLO usr_id, CHR **usr_buff){
  SLO operation=usr_id&(0xF<<20),descr=usr_id&0xFFFFF;
  switch(operation){
    case DEVICE_OPEN:
      equ_num(usr_buff,device_open(usr_buff)); break;
    case DEVICE_READ:
      get_std_buff(usr_buff,device_read(descr)); break;
    case DEVICE_WRITE:
      equ_num(usr_buff,device_write(descr,usr_buff)); break;
    case DEVICE_CLOSE:
      equ_num(usr_buff,device_close(descr));
  }
  return;
}
```

Specific input/output implemented via ***USER_IO***

13. How do you implement *termcap* via *USER_IO*?

The following usage model of ***termcap*** that is implemented via ***USER_IO*** works correctly for both ***fastlisp*** and ***BMDFMldr*** when running either VM code or precompiled VM code:

VM code

```
(user_io 0 "TERMCAP RESET")
(setq term_type          (user_io 0 "TERMCAP TERM_TYPE"))
(setq lines_term   (ival (user_io 0 "TERMCAP LINES_TERM")))
(setq columns_term (ival (user_io 0 "TERMCAP COLUMNS_TERM")))
(setq clrscr_term        (user_io 0 "TERMCAP CLRSCR_TERM"))
(setq reverse_term       (user_io 0 "TERMCAP REVERSE_TERM"))
(setq blink_term         (user_io 0 "TERMCAP BLINK_TERM"))
(setq bold_term          (user_io 0 "TERMCAP BOLD_TERM"))
(setq normal_term        (user_io 0 "TERMCAP NORMAL_TERM"))
(setq hidecursor_term    (user_io 0 "TERMCAP HIDECURSOR_TERM"))
(setq showcursor_term    (user_io 0 "TERMCAP SHOWCURSOR_TERM"))
(setq gotocursor_term    (user_io 0 "TERMCAP GOTOCURSOR_TERM"))
```

Using ***termcap*** via ***USER_IO***

Here is an implementation example:

USER_IO **callback written in C**

```
#include <fcntl.h>
#include <termcap.h>
#include <termio.h>
/* #include <termios.h> */

static struct tcap_{
  const CHR *TERM_TYPE;         /* TERM environment; */
  const ULO LINES_TERM;         /* number of the lines (li); */
  const ULO COLUMNS_TERM;       /* number of the columns (co); */
  const CHR *CLRSCR_TERM;       /* clr scr, cursor home (cl); */
  const CHR *REVERSE_TERM;      /* start reverse mode (mr); */
  const CHR *BLINK_TERM;        /* start blinking (mb); */
  const CHR *BOLD_TERM;         /* start bold mode (md); */
  const CHR *NORMAL_TERM;       /* end modes like mb,md,mr (me); */
  const CHR *HIDECURSOR_TERM;   /* cursor invisible (vi); */
  const CHR *SHOWCURSOR_TERM;   /* cursor visible (ve); */
  const CHR *GOTOCURSOR_TERM;   /* cursor move (cm). */
  CHR *term_type;
  ULO lines_term;
  ULO columns_term;
  CHR *clrscr_term;
  CHR *reverse_term;
  CHR *blink_term;
  CHR *bold_term;
  CHR *normal_term;
  CHR *hidecursor_term;
  CHR *showcursor_term;
  CHR *gotocursor_term;
  UCH tcap_initialized;
} tcap={
  (CHR*)"ansi.sys",(ULO)25,(ULO)80,(CHR*)"\033[m\033[7h\033[2J",
  (CHR*)"\033[7m",(CHR*)"",(CHR*)"\033[1m",(CHR*)"\033[m",(CHR*)"",
  (CHR*)"",(CHR*)"\033[%i%d;%dH",(UCH)0
};

void tcap_deinit(void){
  tcap.tcap_initialized=0;
  return;
}
void tcap_init(void){
  CHR *temp=NULL;
  char *term_data=NULL;
  int tty_term;
  struct winsize ws;
  get_std_buff(&tcap.term_type,tcap.TERM_TYPE);
  tcap.lines_term=tcap.LINES_TERM;
  tcap.columns_term=tcap.COLUMNS_TERM;
  get_std_buff(&tcap.clrscr_term,tcap.CLRSCR_TERM);
  get_std_buff(&tcap.reverse_term,tcap.REVERSE_TERM);
  get_std_buff(&tcap.blink_term,tcap.BLINK_TERM);
  get_std_buff(&tcap.bold_term,tcap.BOLD_TERM);
  get_std_buff(&tcap.normal_term,tcap.NORMAL_TERM);
  get_std_buff(&tcap.hidecursor_term,tcap.HIDECURSOR_TERM);
  get_std_buff(&tcap.showcursor_term,tcap.SHOWCURSOR_TERM);
  get_std_buff(&tcap.gotocursor_term,tcap.GOTOCURSOR_TERM);
  get_std_buff(&temp,getenv("TERM"));
  if(len(temp))
    equ(&tcap.term_type,temp);
  if(0<(signed)tgetent(NULL,tcap.term_type)){
    if((tty_term=open("/dev/tty",0))<0){
      tty_term=2;
      ioctl(tty_term,TIOCGWINSZ,&ws);
    }
    else{
      ioctl(tty_term,TIOCGWINSZ,&ws);
      close(tty_term);
    }
    if((tcap.lines_term=(ULO)ws.ws_row)<=0)
      tcap.lines_term=tcap.LINES_TERM;
    if((tcap.columns_term=(ULO)ws.ws_col)<=0)
      tcap.columns_term=tcap.COLUMNS_TERM;
    if(tgetstr((char*)"cl",&term_data)!=NULL)
      get_std_buff(&tcap.clrscr_term,term_data);
    if(tgetstr((char*)"mr",&term_data)!=NULL)
      get_std_buff(&tcap.reverse_term,term_data);
    if(tgetstr((char*)"mb",&term_data)!=NULL)
      get_std_buff(&tcap.blink_term,term_data);
    if(tgetstr((char*)"md",&term_data)!=NULL)
      get_std_buff(&tcap.bold_term,term_data);
    if(tgetstr((char*)"me",&term_data)!=NULL)
      get_std_buff(&tcap.normal_term,term_data);
    if(tgetstr((char*)"vi",&term_data)!=NULL)
      get_std_buff(&tcap.hidecursor_term,term_data);
    if(tgetstr((char*)"ve",&term_data)!=NULL)
      get_std_buff(&tcap.showcursor_term,term_data);
    if(tgetstr((char*)"cm",&term_data)!=NULL)
      get_std_buff(&tcap.gotocursor_term,term_data);
    free((void*)term_data);
  }
  free_string(&temp);
  tcap.tcap_initialized=1;
  return;
}

void user_io_callback(SLO usr_id, CHR **usr_buff){
  CHR *temp=NULL,*temp1=NULL,*temp2=NULL;
  equ(&temp,*usr_buff);
  if(cmp(head(&temp2,temp),get_std_buff(&temp1,"TERMCAP"))){
    tail(&temp1,temp);
    while(1){
      if(cmp(temp1,get_std_buff(&temp,"RESET"))){
        tcap_deinit();
        space(usr_buff,0);
        break;
      }
      if(!tcap.tcap_initialized)
        tcap_init();
      if(cmp(temp1,get_std_buff(&temp,"TERM_TYPE"))){
        equ(usr_buff,tcap.term_type);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"LINES_TERM"))){
        equ_num(usr_buff,(SLO)tcap.lines_term);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"COLUMNS_TERM"))){
        equ_num(usr_buff,(SLO)tcap.columns_term);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"CLRSCR_TERM"))){
        equ(usr_buff,tcap.clrscr_term);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"REVERSE_TERM"))){
        equ(usr_buff,tcap.reverse_term);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"BLINK_TERM"))){
        equ(usr_buff,tcap.blink_term);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"BOLD_TERM"))){
        equ(usr_buff,tcap.bold_term);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"NORMAL_TERM"))){
        equ(usr_buff,tcap.normal_term);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"HIDECURSOR_TERM"))){
        equ(usr_buff,tcap.hidecursor_term);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"SHOWCURSOR_TERM"))){
        equ(usr_buff,tcap.showcursor_term);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"GOTOCURSOR_TERM"))){
        equ(usr_buff,tcap.gotocursor_term);
        break;
      }
      break;
    }
  }
  free_string(&temp);
  free_string(&temp1);
  free_string(&temp2);
  return;
}
```

Implementation of ***termcap*** via ***USER_IO***

14. How do you evaluate the VM language expressions from C/C++ code?

The best way is to call ***mapcar*** function giving the artificially generated ***byte code preamble*** to its input. ***Mapcar*** accepts both VM language source and VM byte code. So, the idea of a byte code caching can be used to avoid redundant recompilation of the frequently evaluated expressions. The following approach will work correctly for both single-threaded and multithreaded BMDFM engines:

Evaluation of the VM language expressions from C/C++ code

```
#define CHR char
#define ULO unsigned long int
#define SLO signed long int

extern void func__mapcar(const ULO*, struct fastlisp_data*);
extern void func__dummy_s(const ULO*, struct fastlisp_data*);

#ifndef POSIXMTHREAD_NOT_SUPPORTED
__thread
#endif
struct{
  CHR *flp_expr;
  CHR *bytecode;
} flpeval_cache={NULL,NULL};
CHR flp_eval(CHR *flp_expr, struct fastlisp_data *ret_dat){
  CHR success=0,*flp_fnc=NULL,*temp=NULL;
  struct fastlisp_data res={1,1,0,0,{0},NULL,1,NULL,{NULL}};
  get_std_buff(&flp_fnc,flp_expr);
  if(notempty(flp_fnc)&&cmp(flp_fnc,flpeval_cache.flp_expr))
    equ(&flp_fnc,flpeval_cache.bytecode);
  lcat(&flp_fnc,slo2str(&temp,len(flp_fnc)));
  lcat(&flp_fnc,ptr2str(&temp,(void*)&func__dummy_s));
  lcat(&flp_fnc,temp);
  *((CHR**)flp_fnc)=flp_fnc+sizeof(ULO);
  func__mapcar((ULO*)flp_fnc,&res);
  if((res.array.mix+2)->value.ival||(res.array.mix+4)->value.ival)
    copy_flp_data(ret_dat,&res,0);
  else{
    copy_flp_data(ret_dat,res.array.mix+1,0);
    get_std_buff(&flpeval_cache.flp_expr,flp_expr);
    equ(&flpeval_cache.bytecode,(res.array.mix+7)->svalue);
    success=1;
  }
  free_flp_data(&res);
  free_string(&flp_fnc);
  free_string(&temp);
  return success;
}


/* Pattern example for a caller: */
SLO addr;
struct fastlisp_data res={1,1,0,0,{0},NULL,1,NULL,{NULL}};
if(flp_eval("(asyncheap_create 1024)",&res))
  addr=res.value.ival;
free_flp_data(&res);
```

Evaluation of the VM language expressions from C/C++ code

Here is an implementation of ***termcap*** via ***USER_IO*** that calls VM language from C code:

USER_IO **callback written in C that calls VM language**

```
/* The BMDFMldr module is capable of invoking/evaluating VM language
   expressions from C/C++ code (1-Capable;0-Unable).*/
UCH BMDFMldr_capable_call_VMcode_from_C=1;

void user_io_callback(SLO usr_id, CHR **usr_buff){
  CHR *temp=NULL,*temp1=NULL,*temp2=NULL;
  struct fastlisp_data res={1,1,0,0,{0},NULL,1,NULL,{NULL}};
  equ(&temp,*usr_buff);
  if(cmp(head(&temp2,temp),get_std_buff(&temp1,"TERMCAP"))){
    tail(&temp1,temp);
    while(1){
      if(cmp(temp1,get_std_buff(&temp,"TERM_TYPE"))
         ||cmp(temp1,get_std_buff(&temp,"CLRSCR_TERM"))
         ||cmp(temp1,get_std_buff(&temp,"REVERSE_TERM"))
         ||cmp(temp1,get_std_buff(&temp,"BLINK_TERM"))
         ||cmp(temp1,get_std_buff(&temp,"BOLD_TERM"))
         ||cmp(temp1,get_std_buff(&temp,"NORMAL_TERM"))
         ||cmp(temp1,get_std_buff(&temp,"HIDECURSOR_TERM"))
         ||cmp(temp1,get_std_buff(&temp,"SHOWCURSOR_TERM"))){
        lcat(&temp,sch2str(&temp2,'('));
        cat(&temp,sch2str(&temp2,')'));
        flp_eval(temp,&res);
        equ(usr_buff,res.svalue);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"LINES_TERM"))
         ||cmp(temp1,get_std_buff(&temp,"COLUMNS_TERM"))){
        lcat(&temp,sch2str(&temp2,'('));
        cat(&temp,sch2str(&temp2,')'));
        flp_eval(temp,&res);
        equ_num(usr_buff,res.value.ival);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"GOTOCURSOR_TERM"))){
        flp_eval("(gotocursor_term -1 -1)",&res);
        equ(usr_buff,res.svalue);
        break;
      }
      if(cmp(temp1,get_std_buff(&temp,"RESET"))){
        flp_eval("(reinit_terminal \"\")",&res);
        equ(usr_buff,res.svalue);
      }
      break;
    }
  }
  free_string(&temp);
  free_string(&temp1);
  free_string(&temp2);
  free_flp_data(&res);
  return;
}


# VM code
(user_io 0 "TERMCAP RESET")
(setq term_type          (user_io 0 "TERMCAP TERM_TYPE"))
(setq lines_term   (ival (user_io 0 "TERMCAP LINES_TERM")))
(setq columns_term (ival (user_io 0 "TERMCAP COLUMNS_TERM")))
(setq clrscr_term        (user_io 0 "TERMCAP CLRSCR_TERM"))
(setq reverse_term       (user_io 0 "TERMCAP REVERSE_TERM"))
(setq blink_term         (user_io 0 "TERMCAP BLINK_TERM"))
(setq bold_term          (user_io 0 "TERMCAP BOLD_TERM"))
(setq normal_term        (user_io 0 "TERMCAP NORMAL_TERM"))
(setq hidecursor_term    (user_io 0 "TERMCAP HIDECURSOR_TERM"))
(setq showcursor_term    (user_io 0 "TERMCAP SHOWCURSOR_TERM"))
(setq gotocursor_term    (user_io 0 "TERMCAP GOTOCURSOR_TERM"))
```

Implementation of ***termcap*** via ***USER_IO*** that calls VM language from C code

15. How do you allocate/free shared memory from C/C++ code?

The standard calls to ***malloc()*** and ***free()*** will not target the Shared Memory Pool. One of the possible solutions is to evaluate the "***(asyncheap_create ...)***" and "***(asyncheap_delete ...)***" fastlisp expressions from C/C++ code. However, the direct calls to the ***asyncheap_create()*** and ***asyncheap_delete()*** implementations will run faster. The following approach will work correctly for both single-threaded and multithreaded BMDFM engines. The shared memory will be automatically freed after an external task is detached from the BMDFM server:

Shared memory operations from C/C++ code

```
/* Pure C */
#define CHR char
#define ULO unsigned long int
#define SLO signed long int

#ifdef _TO_BE_LINKED_AGAINST_CPUPROC_
  #define FLP_MALLOC par_func__asyncheap_create_j
  #define FLP_FREE   par_func__asyncheap_delete_j
#else
  #define FLP_MALLOC func__asyncheap_create_j
  #define FLP_FREE   func__asyncheap_delete_j
#endif  /* or do the module check at runtime
           using am_I_in_the_CPUPROC_module() */
extern void FLP_MALLOC(const ULO*, struct fastlisp_data*);
extern void FLP_FREE(const ULO*, struct fastlisp_data*);
extern void func__dummy_i(const ULO*, struct fastlisp_data*);

void *flp_malloc(SLO bytes){
  CHR *flp_fnc=NULL,*temp=NULL;
  SLO addr;
  struct fastlisp_data res={1,1,0,0,{0},NULL,1,NULL,{NULL}};
  slo2str(&flp_fnc,bytes);
  lcat(&flp_fnc,ptr2str(&temp,(void*)&func__dummy_i));
  lcat(&flp_fnc,temp);
  *((CHR**)flp_fnc)=flp_fnc+sizeof(ULO);
  FLP_MALLOC((ULO*)flp_fnc,&res);
  addr=res.value.ival;
  free_flp_data(&res);
  free_string(&flp_fnc);
  free_string(&temp);
  return (void*)addr;
}
void flp_free(SLO addr){
  CHR *flp_fnc=NULL,*temp=NULL;
  struct fastlisp_data res={1,1,0,0,{0},NULL,1,NULL,{NULL}};
  slo2str(&flp_fnc,addr);
  lcat(&flp_fnc,ptr2str(&temp,(void*)&func__dummy_i));
  lcat(&flp_fnc,temp);
  *((CHR**)flp_fnc)=flp_fnc+sizeof(ULO);
  FLP_FREE((ULO*)flp_fnc,&res);
  free_flp_data(&res);
  free_string(&flp_fnc);
  free_string(&temp);
  return;
}
```

```
// Pure C++
#define SLO signed long int

class foo{  // Use pattern for the class:
            //   foo *foo0=new foo,*foo1=new foo[2];
            //   delete foo0;
            //   delete[] foo1;
public:
  foo();
  ~foo();
  void *operator new(size_t size) throw (const char*);
  void *operator new[](size_t size) throw (const char*);
  void operator delete(void *p);
  void operator delete[](void *p);
};

foo::foo(){
}

foo::~foo(){
}

void *foo::operator new(size_t size) throw (const char*){
  void *ptr=flp_malloc((SLO)size);
  if(ptr==NULL)
    throw "Allocation failure.";
  return ptr;
}

void *foo::operator new[](size_t size) throw (const char*){
  void *ptr=flp_malloc((SLO)size);
  if(ptr==NULL)
    throw "Allocation failure.";
  return ptr;
}

void foo::operator delete(void *ptr){
  flp_free((SLO)ptr);
  return;
}

void foo::operator delete[](void *ptr){
  flp_free((SLO)ptr);
  return;
}
```

Shared memory access from C/C++ code

The similar strategy can be applied to other ***asyncheap*** functions. Use the standard ***nm*** utility to check the correct names of the functions you would like to link against:

Terminal

```
$ nm fastlisp.o
Symbols from fastlisp.o:
Name                              Value            Class Type   Size             Line  Section
. . .
func__asyncheap_create           |00000000000171d8|  T  | FUNC |00000000000000ec|      |.text
func__asyncheap_create_j         |00000000000172c4|  T  | FUNC |0000000000000100|      |.text
func__asyncheap_delete           |0000000000018bac|  T  | FUNC |00000000000000fc|      |.text
func__asyncheap_delete_j         |0000000000018ca8|  T  | FUNC |0000000000000110|      |.text
func__asyncheap_reallocate       |00000000000864c|  T  | FUNC |000000000000014c|      |.text
func__asyncheap_reallocate_j     |0000000000018798|  T  | FUNC |0000000000000180|      |.text
func__asyncheap_replicate        |0000000000018918|  T  | FUNC |0000000000000140|      |.text
func__asyncheap_replicate_j      |0000000000018a58|  T  | FUNC |0000000000000154|      |.text
. . .

$ nm CPUPROC.o
Symbols from CPUPROC.o:
Name                              Value            Class Type   Size             Line  Section
. . .
par_func__asyncheap_create       |000000000002bd8c|  T  | FUNC |000000000000020c|      |.text
par_func__asyncheap_create_j     |000000000002bf98|  T  | FUNC |000000000000021c|      |.text
par_func__asyncheap_delete       |000000000002e8fc|  T  | FUNC |00000000000001f8|      |.text
par_func__asyncheap_delete_j     |000000000002eaf4|  T  | FUNC |0000000000000208|      |.text
par_func__asyncheap_reallocate   |000000000002ddd4|  T  | FUNC |000000000000027c|      |.text
par_func__asyncheap_reallocate_j |000000000002e050|  T  | FUNC |00000000000002ac|      |.text
par_func__asyncheap_replicate    |000000000002e2fc|  T  | FUNC |00000000000002f8|      |.text
par_func__asyncheap_replicate_j  |000000000002e5f4|  T  | FUNC |0000000000000308|      |.text
. . .

$
```

Terminal

16. How do you attach to the BMDFM shared memory and allocate permanent data there?

An external application may attach and access the BMDFM shared memory using the following direct shared memory pool interface. The following example demonstrates this:

Direct shared memory pool interface

```
#define CHR char
#define UCH unsigned char
#define ULO unsigned long int

extern UCH attach_mempool(int sharedID, ULO mntaddr);
extern UCH detach_mempool(void);

extern void shmempool_on(void);
extern void shmempool_off(void);
extern UCH is_shmempool_on(void);

extern void *reallocpool(void *ptr, ULO size);
extern void freepool(void *ptr);
extern ULO getalignedsizepool(void *ptr);
extern void addmcastrefpool(void *ptr);

extern CHR *shmempoolLUT_add_key_value(CHR **value, const CHR *key);
extern CHR *shmempoolLUT_del_key_value(CHR **key);
extern CHR *shmempoolLUT_get_value(CHR **value, const CHR *key);
extern CHR *shmempoolLUT_get_entire_contents(CHR **contents);
extern void shmempoolLUT_purge_entire_contents(void);
```

Direct shared memory pool interface

C code: an external application allocates permanent data in the ShMemPool

```
int main(int argc, char *argv[]){
  CHR *BM_DFM_CONNECTION_FILE_path=NULL,*info=NULL,*temp=NULL,*temp1=NULL,
      *lut_key=NULL,*lut_value=NULL;
  ULO mntaddr;
  int f_descr,sharedID;
  struct _entry_point_struct{
    CHR *str0;
    CHR *str1;
    CHR *strN;
  } entry_point_struct={NULL,NULL,NULL},*entry_point_struct_ptr;

  if((BM_DFM_CONNECTION_FILE_path=getenv("BM_DFM_CONNECTION_FILE_path"))==NULL)
    BM_DFM_CONNECTION_FILE_path=(CHR*)"/tmp/.BMDFMsrv";

  if((f_descr=open(BM_DFM_CONNECTION_FILE_path,0))==-1){
    fprintf(stderr,"Fail opening file `%s'.\n",BM_DFM_CONNECTION_FILE_path);
    exit(1);
  }
  mk_fst_buff(&info,1024);
  read(f_descr,(void*)info,1024);
  close(f_descr);
  if(!cmp(left(&temp,info,9),get_std_buff(&temp1,"BMDFMsrv "))){
    fprintf(stderr,"`%s' is not the BM_DFM connection file.\n",
      BM_DFM_CONNECTION_FILE_path);
    exit(1);
  }
  tail(&info,info);
  head(&temp,info);
  tail(&info,info);
  sharedID=(int)atoi(temp);
  head(&temp,info);
  mntaddr=(ULO)atol(temp);

  shmempool_on();
  if(!attach_mempool(sharedID,mntaddr)){
    fprintf(stderr,"Cannot attach the shared memory pool.\n");
    exit(1);
  }

  if((entry_point_struct_ptr=(struct _entry_point_struct*)reallocpool(NULL,
     sizeof(struct _entry_point_struct)))==NULL){
    fprintf(stderr,"Memory allocation in the shared memory pool failed.\n");
    exit(1);
  }
  get_std_buff_secure(&entry_point_struct.str0,"String 0: I am in ShMemPool.");
  get_std_buff_secure(&entry_point_struct.str1,"String 1: I am in ShMemPool.");
  get_std_buff_secure(&entry_point_struct.strN,"String N: I am in ShMemPool.");
  *entry_point_struct_ptr=entry_point_struct;

  // Allocated entries are persistent.
  // Keep entry_point_struct_ptr somehow available for others, e.g.:
  //   shmempool_off();
  //   get_std_buff(&lut_key,"Key for our test allocations");
  //   equ_num(&lut_value,(SLO)entry_point_struct_ptr);
  //   shmempool_on();
  //   shmempoolLUT_add_key_value(&lut_value,lut_key);
  //   shmempool_off();
  // A consumer can initialize entry_point_struct_ptr like:
  //   shmempool_off();
  //   get_std_buff(&lut_key,"Key for our test allocations");
  //   shmempool_on();
  //   shmempoolLUT_get_value(&lut_value,lut_key);
  //   shmempool_off();
  //   entry_point_struct_ptr=(struct _entry_point_struct*)atol(lut_value);

  shmempool_on();
  if(!detach_mempool()){
    fprintf(stderr,"Cannot detach the shared memory pool.\n");
    exit(1);
  }
  shmempool_off();

  return 0;
}
```

An external application allocates permanent data in the ShMemPool

Add the code to your ***cflp_udf.c***. Link against one of ***BMDFMldr.o***, ***BMDFMsrv.o***, ***CPUPROC.o*** like e.g.:

- ***gcc -o MyProg cflp_udf.c CPUPROC.o -lpthread -lm***

17. What is the optimal number of the BMDFM processes?

Basically, the optimal number of the BMDFM processes (of each kind) is equal to the number of available system processors multiplied by 2. Recent server processors are very often the multi-core processors. Therefore, it is better to count the number of the BMDFM processes according to the number of cores or processing units.

However, it is important to know that ***CPUPROC*** processes mainly execute user code, ***IORBPROC*** processes run required dynamic scheduling routines and ***OQPROC*** processes perform speculative (somehow a little redundant) dynamic scheduling of dataflow instructions.

Suppose a user has one dedicated virtual partition on an IBM SMP mainframe based on the ***POWER*** architecture. This partition has 2 dedicated ***MCM*** (***Multi-Chip Modules***) having 4 processors per module and 16 cores per processor with ability to run 8 threads simultaneously on each core. Hence, the number of processing units is 2*4*16*8=1024, the following settings are recommended for such configuration:

BMDFMsrv.cfg

```
N_CPUPROC          =     2048 # Number of the CPU PROCs
N_IORBPPROC        =     2048 # Number of the IORBP PROCs
N_OQPROC           =     2048 # Number of the OQ PROCs
```

Settings for 1024 processing units

These mnemonic rules could be a good starting point for the initial settings. Later on, the number of the BMDFM processes can be experimentally tuned depending on application class and architecture of the SMP interconnections.

Note that the multithreaded mode can be configured as well (might be good for multicores and many-cores or for sharing objects in the process address space rather than in the shared memory pool):

BMDFMsrv.cfg

```
CPUPROC_MTHREAD    =      Yes # CPU PROC is multithreaded
OQPROC_MTHREAD     =      Yes # OQ PROC is multithreaded
IORBPPROC_MTHREAD  =      Yes # IORBP PROC is multithreaded
BMDFMLDR_MTHREAD   =      Yes # BMDFMldr is multithreaded
```

Settings for multithreaded mode

18. How do you implement a parallel recursive Fibonacci function?

Fibonacci numbers are the integer sequence produced by the following relationship:

Recursive Fibonacci algorithm (pseudo-code)

```
Fibonacci(0) = 0;
Fibonacci(1) = 1;
Fibonacci(N) = Fibonacci(N - 1) + Fibonacci(N - 2);
```

Recursive Fibonacci algorithm

Thus, the Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, . . .

Firstly, we implement a seamless single-threaded recursive Fibonacci function that is about to be added into the BMDFM configuration profile (or an alternative implementation in C):

Seamless single-threaded recursive Fibonacci implementation (VM code)

```
(defun FibonacciSeamless  # to be placed into .cfg
  (progn
    (setq n (+ 0 $1))
    (if (< n 2)
      n
      (+ (FibonacciSeamless (-- n))
         (FibonacciSeamless (- n 2))
      )
    )
  )
)
```

Seamless single-threaded recursive Fibonacci (alternative C-implementation)

```
#define ULO unsigned long int
#define SLO signed long int

SLO _dffib_FibonacciSeamless(SLO n){
  return noterror()&&n>1?_dffib_FibonacciSeamless(n-1)+_dffib_FibonacciSeamless(n-2):n;
}

void dffib_FibonacciSeamless(const ULO *dat_ptr, struct fastlisp_data *ret_dat){
  SLO n;
  ret_ival(dat_ptr,&n);
  if(noterror()){
    ret_dat->single=1;
    ret_dat->type='I';
    ret_dat->value.ival=_dffib_FibonacciSeamless(n);
  }
  return;
}

INSTRUCTION_STRU INSTRUCTION_SET[]={
  {"FIBONACCISEAMLESS",1,'I',(UCH*)"I",&dffib_FibonacciSeamless}
};
```

Seamless single-threaded recursive Fibonacci implementation

And then, we write a simple trivial implementation of our parallel multithreaded recursive Fibonacci function into the ***Fibonacci.flp*** file (note that we need neither special parallelization directives nor special reserved function names; we have "wrapped" the ***FibonacciSeamless*** function with the ***FibonacciCoordinator*** function in order to limit "unlimited parallelism"):

Parallel multithreaded recursive Fibonacci implementation (VM code)

```
(defun FibonacciCoordinator  # to be placed into .flp
  (progn
    (setq n (+ 0 $1))
    (setq spawn (+ 0 $2))
    (if (< n 2)
      n
      (if (> spawn 0)
        (+ (FibonacciCoordinator (-- n) (>> spawn 1))
           (FibonacciCoordinator (- n 2) (>> spawn 1))
        )
        (+ (FibonacciSeamless (-- n))
           (FibonacciSeamless (- n 2))
        )
      )
    )
  )
)

(defun Fibonacci
  (progn
    (setq n (+ 0 $1))
    (setq spawn (n_cpuproc))
    (FibonacciCoordinator n spawn)
  )
)

# main() begins here
(setq n (+ 0 $1))
(Fibonacci n)
```

Fibonacci.flp containing parallel multithreaded recursive Fibonacci implementation

19. How do you rewrite application example from the BMDFM manual in pure VM language?

The application example from the BMDFM manual can be rewritten using e.g. asynchronous heaps:

fastlisp.cfg/BMDFMsrv.cfg

```
(defun dhtpipe0_generate  # $1=array, $2=n, $3=m.
  (progn
    (setq array (+ 0 $1))
    (setq n (+ 0 $2))
    (setq m (+ 0 $3))
    (setq m_1 (-- m))
    (setq n_1 (-- n))
    (for i 0 1 n_1
      (for j 0 1 m_1
        (asyncheap_putfloat array (*+ m i j) (frnd 1.))
      )
    )
    array
  )
)

(defun dhtpipe0_dht  # $1=target_array, $2=n, $3=m,
                     # $4=source_array.
  (progn
    (setq target_array (+ 0 $1))
    (setq n (+ 0 $2))
    (setq m (+ 0 $3))
    (setq source_array (+ 0 $4))
    (setq c1 (/. (*. 2. (pi)) n))
    (setq s1 (/. (*. 2. (pi)) m))
    (setq m_1 (-- m))
    (setq n_1 (-- n))
    (for p 0 1 n_1
      (for q 0 1 m_1 (progn
        (setq s 0.)
        (for i 0 1 n_1
          (for j 0 1 m_1
            (setq s (*+. (asyncheap_getfloat source_array
              (*+ m i j)) (cas (*+. c1 (*. p i) (*. s1 (*.
              q j)))) s))
          )
        )
        (asyncheap_putfloat target_array (*+ m p q) s)
      ))
    )
    target_array
  )
)
```

```
(defun dhtpipe0_idht  # $1=target_array, $2=n, $3=m, $4=source_array.
  (progn
    (setq target_array (+ 0 $1))
    (setq n (+ 0 $2))
    (setq m (+ 0 $3))
    (setq source_array (+ 0 $4))
    (setq c1 (/. (*. 2. (pi)) n))
    (setq s1 (/. (*. 2. (pi)) m))
    (setq m_1 (-- m))
    (setq n_1 (-- n))
    (for p 0 1 n_1
      (for q 0 1 m_1 (progn
        (setq s 0.)
        (for i 0 1 n_1
          (for j 0 1 m_1
            (setq s (*+. (asyncheap_getfloat source_array (*+ m i j))
              (cas (*+. c1 (*. p i) (*. s1 (*. q j)))) s))
          )
        )
        (asyncheap_putfloat target_array (*+ m p q) (/. (/. s n) m))
      ))
    )
    target_array
  )
)

(defun dhtpipe0_compare  # $1=array0, $2=array1, $3=n, $4=m.
  (progn
    (setq array0 (+ 0 $1))
    (setq array1 (+ 0 $2))
    (setq n (+ 0 $3))
    (setq m (+ 0 $4))
    (|| (> (fabs (-. (asyncheap_getfloat array0 0) (asyncheap_getfloat
      array1 0))) 1e-10) (> (fabs (-. (asyncheap_getfloat array0 (*+ m
      (-- n) (-- m))) (asyncheap_getfloat array1 (*+ m (-- n) (-- m)))))
      1e-10))
  )
)
```

dhtpipe0.flp

```
(progn
  (outf
    "Pipeline calculation of the 2D nonseparative Hartley transform.\n\n" 0)
  (setq m (ival (accept "M-value of M*N-matrix: ")))
  (setq n (ival (accept "N-value of M*N-matrix: ")))
  (setq numb (ival (accept "How many input data packs: ")))
  (setq arrays_size (* (* m n) (len (dump_f2s 0.))))
  (for i 1 1 numb (progn
    (outf "Sequence %ld:" i)
    # 1.
    (setq inp_array_sync (& 0
      (setq inp_array_addr (asyncheap_create arrays_size))
    ))
    # 2.
    (setq inp_array_sync (& 0
      (setq inp_array_addr
        (dhtpipe0_generate (| inp_array_sync inp_array_addr) n m))
    ))
    # 3.
    (setq dht_array_sync (& 0
      (setq dht_array_addr (asyncheap_create arrays_size))
    ))
    # 4.
    (setq dht_array_sync (setq inp_array_sync (& 0
      (setq dht_array_addr
        (dhtpipe0_dht (| dht_array_sync dht_array_addr) n m inp_array_addr))
    )))
    # 5.
    (setq idht_array_sync (& 0
      (setq idht_array_addr (asyncheap_create arrays_size))
    ))
    # 6.
    (setq idht_array_sync (setq dht_array_sync (& 0
      (setq idht_array_addr
        (dhtpipe0_idht (| idht_array_sync idht_array_addr) n m dht_array_addr))
    )))
    # 7.
    (setq inp_array_sync (setq idht_array_sync (& 0
      (setq cmp_res (dhtpipe0_compare inp_array_addr idht_array_addr n m))
    )))
    (outf " %s.\n" (if cmp_res "Fail" "Ok"))
    # 8.
    (asyncheap_delete (| inp_array_sync inp_array_addr))
    (asyncheap_delete (| dht_array_sync dht_array_addr))
    (asyncheap_delete (| idht_array_sync idht_array_addr))
  ))
  ""
)
```

fastlisp.cfg/BMDFMsrv.cfg and ***dhtpipe0.flp***

20. How serious is the performance degradation of pure unparalleled VM byte code?

For the performance test, a test program was rewritten in pure ANSI C, in Java and in the native VM language. This testbench comes from the area of discrete trigonometric transformations, namely the “2D non-separate Hartley transform”. Full source code can be found in the BMDFM distribution package. Below, only fragments of the code are given for comparison:

Pure ANSI C code fragment (*testbench.c*)

```
void dht(DFL *target_array, SLO n, SLO m, DFL *source_array){
  SLO i,j,p,q;
  DFL pi,c1,s1,sum,tmp;
  pi=3.1415926535897932;
  c1=2*pi/n;
  s1=2*pi/m;
  for(p=0;p<n;p++)
    for(q=0;q<m;q++){
      sum=0;
      for(i=0;i<n;i++)
        for(j=0;j<m;j++)
          sum+=(*(source_array+i*m+j)*(cos(tmp=c1*p*i+s1*q*j)+sin(tmp)));
      *(target_array+p*m+q)=sum;
    }
  return;
}
```

Java VM code fragment (*testbench.java*)

```
public static void dht(double target_array[], int n, int m,
   double source_array[]){
  int i,j,p,q;
  double pi,c1,s1,sum,tmp;
  pi=3.1415926535897932;
  c1=2*pi/n;
  s1=2*pi/m;
  for(p=0;p<n;p++){
    for(q=0;q<m;q++){
      sum=0;
      for(i=0;i<n;i++){
        for(j=0;j<m;j++){
          sum+=(source_array[i*m+j]*(Math.cos(tmp=c1*p*i+s1*q*j)+Math.sin(tmp)));
        }
      }
      target_array[p*m+q]=sum;
    }
  }
  return;
}
```

Native VM code fragment (*testbench.flp*)

```
(defun dht (progn
  (setq target_array (+ 0 $1))
  (setq n (+ 0 $2))
  (setq m (+ 0 $3))
  (setq source_array (+ 0 $4))
  (setq c1 (/. (*. 2. (pi)) n))
  (setq s1 (/. (*. 2. (pi)) m))
  (setq m_1 (-- m))
  (setq n_1 (-- n))
  (for p 0 1 n_1
    (for q 0 1 m_1 (progn
      (setq s 0.)
      (for i 0 1 n_1
        (for j 0 1 m_1
          (setq s (*+. (asyncheap_getfloat source_array (*+ m i j))
            (cas (*+. c1 (*. p i) (*. s1 (*. q j)))) s))
        )
      )
      (asyncheap_putfloat target_array (*+ m p q) s)
    ))
  )
))
```

Benchmarked fragments of code

The testbench was benchmarked on various processors (***Opteron***, ***Itanium***, ***POWER***) demonstrating nearly the same average performance degradation ratio on these processors:

Benchmarks

```
Pure ANSI C compiled machine code: 100sec. (1.0 - baseline)
   Java VM running Java byte code: 300sec. (3.0 - times slower)
Native VM running BMDFM byte code: 550sec. (5.5 - times slower)
```

Test results

Thus, the performance degradation of pure unparalleled VM byte code is 5.5 times compared to ANSI C compiled machine code. As a conclusion, it is worth highlighting two general ideas:

- BMDFM that runs application byte code (preferably structured in coarse-grain functions) on an 8-way SMP machine can outperform unparalleled ANSI C compiled machine code.
- Use of VM becomes much more efficient when the VM is extended with C-implementations of frequently used coarse-grain functions.

21. How does the relaxed consistency model of shared memory influence BMDFM?

Although the question of how consistent shared memory is seems simple, it is remarkably complicated, as is shown with a simple example:

```
Process 0 shares A and B (pseudo-code)
a=1;
// . . .
a=0;
if(b){
  // . . .
}
Process 1 shares A and B (pseudo-code)
b=1;
// . . .
b=0;
if(a){
  // . . .
}
```

Concurrent processes running on different processors

Assume that the processes are running on different processors, and that locations of A and B are originally cached by both processors with the initial value of 1. If writes always take immediate effect and are immediately seen by other processors, it will be impossible for both if-statements to evaluate their conditions as true, since reaching the if-statement means that either A or B must have been assigned the value 0. But suppose the write invalidate is delayed, and the processor is allowed to continue during this delay – then it is possible that both processes have not seen the invalidation for B and A, respectively, before they attempt to read the values. In other words, processed data can be invisible for the other processor because data has not even left the boundaries of the processor where it was processed.

The most straightforward model for memory consistency is called ***sequential consistency***. Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved. Sequential consistency eliminates the possibility of some non-obvious execution in the previous example, because the assignments must be completed before the if-statements are initiated. The sequential consistency model has a performance disadvantage.

To provide better performance, researchers and architects have designed ***relaxed consistency*** of shared memory, which yields a variety of models including ***weak ordering***, the ***Alpha consistency model***, the ***PowerPC consistency model***, and ***release consistency*** depending on the details of the ordering restrictions and how synchronization operations enforce ordering. The main idea from the programmer's point of view is that data becomes consistent when a synchronization primitive is called.

And now let's go back to BMDFM. Speculative parallel ***OQPROC*** scheduling processes, for the sake of performance, call only a reduced number of necessary synchronization primitives. Normally, such a strategy is acceptable when running BMDFM, for example, on the ***Intel*** architecture, which tends to be more ***sequentially consistent***. A problem can appear when running BMDFM, for example, on the IBM ***POWER*** architecture exploiting ***relaxed consistency*** – a dead stall can be observed. Experimentally, such a stall can happen one time per month in average when running BMDFM in an intensive batch mode on an 8-way ***POWER*** machine.

BMDFM has built-in facilities to compensate the influence of the relaxed consistency model of shared memory. These compensation mechanisms are activated by the ***RELAXED_CNSTN_SM_MODEL*** configuration parameter of the BMDFM configuration profile, and they are activated by default. It is strongly recommended to keep them activated if the consistency model of SMP machine is not clear enough.

22. How does the BMDFM dataflow engine process an array?

The addressed issue is very interesting and very sensitive in all known implementations of dataflow machines. For example, the famous ***Monsoon*** dataflow machine project (Motorola Cambridge Research Center) provides a classical solution of this problem based on ***i-structures***, that is fairly efficient, however, still not efficient enough. BMDFM uses the advanced approach described below.

Arrays are not contexted data – this would be too expensive. By default, BMDFM accesses array's members in parallel, detecting overwritten values. An overwritten value is detected as a violation of the ***single assignment paradigm***. For most typical cases like the following, this approach works well, causing no violation:

Pseudo-code

```
for(i=0;i<=N;i++)
  a[i]=...;
for(i=0;i<=N;i++)
  b[i]=...a[i]...;
```

Fragment without violation of single assignment

If a violation of single assignment is detected, then BMDFM recommends using the ***HARD_ARRAY_SYNCHRO*** configuration parameter of the BMDFM configuration profile. In the case of hard array synchronization, BMDFM tracks all array accesses and does assignments sequentially. Thus, no contentions appear, and, besides, such a sequential fine-grain access works faster anyway than the fine-grain access round trips through the dataflow machinery.

Let's describe the use cases of array processing in BMDFM.

USECASE 0: There are multiple fine-grain assignments of the array's members running in parallel without using ***HARD_ARRAY_SYNCHRO*** serialization. Having the input code fragment described below, the generated input to the BMDFM dataflow engine works correctly in parallel because arrays are local for ***func0*** and ***func1*** and, thus, in the different contexts:

Initial sequence (pseudo-code)

```
for(i=0;i<=N;i++)
  a[i]=...;
for(i=0;i<=N;i++)
  a[i]=...;
```

Generated input to the BMDFM dataflow engine (pseudo-code)

```
func0(array){
  for(i=0;i<=N;i++)
    array[i]=...;
  return array;
}
func1(array){
  for(i=0;i<=N;i++)
    array[i]=...;
  return array;
}
a=func0(a);
a=func1(a);
```

Fragments for USECASE 0

USECASE 1: Assigned values are heavyweight computations. Then serialization of ***HARD_ARRAY_SYNCHRO*** ensures correctness and at the same time does not bring any performance degradations:

Initial sequence (pseudo-code)

```
for(i=0;i<=N;i++)
  a[i]=func();
```

Generated input to the BMDFM dataflow engine (pseudo-code)

```
for(i=0;i<=N;i++){
  temp=func();  // contexted, heavy-weight computations are parallel.
  a[i]=temp;    // sequential, no performance degradations.
}
```

Fragments for USECASE 1

USECASE 2: Array processing is done in a coarse-grain fashion. In this case, the above mentioned ***func0*** and ***func1*** are seamless for the dataflow scheduler, thus, the dynamic scheduler is not aware of the arrays at all:

Pseudo-code

```
func0(array){  // defined as a seamless function
  for(i=0;i<=N;i++)
    array[i]=...;
  return array;
}
```

```
func1(array){  // defined as a seamless function
  for(i=0;i<=N;i++)
    array[i]=...;
  return array;
}
```

Fragments for USECASE 2

USECASE 3: Finally, the arrays can be processed as normal arrays programmed in C via pointers. In this case, the parallel array processing is reduced to the known case of "Synchronization of Asynchronous Coarse-Grain and Fine-Grain Functions".

23. How do you enable late binding for a precompiled program?

Suppose, we have a ***problem.flp*** calling (***mapcar***) with UDF's that are defined in .cfg, e.g.:

```
Terminal
$ cat problem.flp
# . . .
(fastlisp "(flog1 -2 -8)")  # (fastlisp) and (flog1) are defined in .cfg

$ fastlisp -c problem.flp
. . .
3.0000000000000000E+00

$ fastlisp -q problem.flx
[Syntax error 3]: In function `(Main)': undefined function name `FLOG1': (flog1 -2 -8)

$
```

Terminal

When running a precompiled program, .cfg is not loaded in sake of performance.

Note that ***`BMDFMldr -c problem.flp; BMDFMldr -q problem.flz`*** works correctly since .cfg is loaded by the BMDFM Server.

Here is a simple solution on how we can run our precompiled ***problem.flx*** in a way that (***mapcar***) still works for .cfg-defined UDF's. Conventional wisdom tells us to write a trivial just-one-line-of-code ***flxrunner.flp*** helper:

```
Terminal
$ cat flxrunner.flp
(fastlisp (rightl (get_file $1) (<< (len (dump_i2s 0)) 1)))  # (fastlisp) and (get_file) are in .cfg

$ fastlisp -q flxrunner.flp problem.flx
3.0000000000000000E+00

$
```

Terminal

24. How do you fix unresolved dependencies introduced by vendor's proprietary compiler?

In order to achieve better optimization level on different target platforms BMDFM is built using vendor's proprietary compilers for the target platforms where possible. Users might want to write their own C-interface extensions and rebuild BMDFM using e.g. publicly available ***gcc*** compiler. This works pretty well in general, however, sometimes leading to harmless side-effects and minor inconvenience issues:

```
Terminal
$ source /opt/intel/composer_xe_2015.0.090/bin/compilervars.sh intel64
$ icc -mmic -o BMDFMsrv BMDFMsrv.o cflp_udf.o -lm -lpthread

$ /usr/linux-k1om-4.7/bin/x86_64-k1om-linux-gcc -o BMDFMsrv BMDFMsrv.o cflp_udf.o -lm -lpthread
. . .
BMDFMsrv.o: In function `copy_flp_data_':
(.text+0xe946): undefined reference to `_intel_fast_memset'
BMDFMsrv.o: In function `copy_flp_data_':
(.text+0xe98b): undefined reference to `_intel_fast_memcpy'
BMDFMsrv.o: In function `_reallocpool':
(.text+0x27136): undefined reference to `_intel_fast_memmove'
BMDFMsrv.o: In function `rat':
(.text+0x29491): undefined reference to `_intel_fast_memcmp'
. . .
$
```

Cross-compiling for native Intel Xeon Phi MIC on Linux

```
Terminal
$ cc -q64 -o CPUPROC CPUPROC.o cflp_udf.o -lm -lpthread

$ gcc -maix64 -o CPUPROC CPUPROC.o cflp_udf.o -lm -lpthread
. . .
ld: 0711-317 ERROR: Undefined symbol: .__xl_log
ld: 0711-317 ERROR: Undefined symbol: .__xl_exp
ld: 0711-317 ERROR: Undefined symbol: .__xl_cos
ld: 0711-317 ERROR: Undefined symbol: .__xl_sin
ld: 0711-317 ERROR: Undefined symbol: .__xl_atan
ld: 0711-317 ERROR: Undefined symbol: .__xl_tanh
. . .
$
```

Compiling for RS/6000 on POWER AIX

The solution is to use another linker or link explicitly against an appropriate vendor's library, e.g.:

```
Terminal
$ /usr/linux-k1om-4.7/bin/x86_64-k1om-linux-gcc -o BMDFMsrv BMDFMsrv.o cflp_udf.o -lm -lpthread
. . .
BMDFMsrv.o: In function `copy_flp_data_':
(.text+0xe946): undefined reference to `_intel_fast_memset'
BMDFMsrv.o: In function `copy_flp_data_':
(.text+0xe98b): undefined reference to `_intel_fast_memcpy'
BMDFMsrv.o: In function `_reallocpool':
(.text+0x27136): undefined reference to `_intel_fast_memmove'
BMDFMsrv.o: In function `rat':
(.text+0x29491): undefined reference to `_intel_fast_memcmp'
. . .
$ /usr/linux-k1om-4.7/bin/x86_64-k1om-linux-gcc -o BMDFMsrv BMDFMsrv.o cflp_udf.o -lm -lpthread
-L/opt/intel/composer_xe_2015.0.090/compiler/lib/mic/ -lirc
$
```

Cross-compiling for native Intel Xeon Phi MIC on Linux

If the required vendor's library is not available then (in case of urgency) an own trivial stub implementation can be added to the C-interface extension, e.g.:

```
C code
void *_intel_fast_memset(void *s, int c, size_t n){
  return memset(s,c,n);
}
void *_intel_fast_memcpy(void *dest, const void *src, size_t n){
  return memcpy(dest,src,n);
}
void *_intel_fast_memmove(void *dest, const void *src, size_t n){
  return memmove(dest,src,n);
}
int _intel_fast_memcmp(const void *s1, const void *s2, size_t n){
  return memcmp(s1,s2,n);
}

double __xl_log(double x){
  return log(x);
}
double __xl_exp(double x){
  return exp(x);
}
double __xl_cos(double x){
  return cos(x);
}
double __xl_sin(double x){
  return sin(x);
}
double __xl_atan(double x){
  return atan(x);
}
double __xl_tanh(double x){
  return tanh(x);
}
```

Own trivial stub implementation of the missing functions

25. How do you build *fastlisp.exe* with MS VS linking against *cygwin1.dll*?

Download and install latest version of Cygwin. You will need latest ***cygwin1.dll*** and ***crt0.c***. You will also need to build your own ***my_crt0.c*** into a DLL in a Cygwin prompt.

Download ***impdef.exe*** for Windows. Use the ***impdef.exe*** program to generate a ***cygwin1.def*** file for the ***cygwin1.dll*** in a Windows prompt:

```
Windows prompt
> impdef cygwin1.dll >cygwin1.def
>
```

Generation of definition file

Use the MS VS linker (***lib.exe***) to generate an import library in a Windows prompt:

```
Windows prompt
> "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat"
> lib /def:cygwin1.def /out:cygwin1.lib
>
```

Generation of import library file

Create a file ***my_crt0.c*** with the following contents:

```
C code
#include <sys/cygwin.h>
#include <stdlib.h>

typedef int (*MainFunc)(int argc, char *argv[], char **env);

void my_crt0(MainFunc f){
  cygwin_crt0(f);  /* cygwin1.dll needs to be initialized */
}
```

Auxiliary C file

Use ***gcc*** in a Cygwin prompt to build ***my_crt0.c*** into a DLL (e.g. ***my_crt0.dll***):

```
Cygwin prompt
$ gcc my_crt0.c -shared -o my_crt0.dll
$
```

Generation of DLL

Generate ***my_crt0.def*** and ***my_crt0.lib*** files for the ***my_crt0.dll*** in a Windows prompt:

```
Windows prompt
> impdef my_crt0.dll >my_crt0.def
> lib /def:my_crt0.def /out:my_crt0.lib
>
```

Generation of definition file and import library file

Copy ***crt0.c*** from your Cygwin installation and include it into your sources for MS VS. Modify it to call ***my_crt0()*** instead of ***cygwin_crt0()***. Build your object files using the MS VS compiler ***cl.exe***, e.g.:

```
Windows prompt
> cl cflp_udf.c crt0.c /o "fastlisp.exe" /D "_NOT_UNIX_" /link /NODEFAULTLIB fastlisp.o my_crt0.lib
cygwin1.lib
>
```

Generation of executable file

Note that if you are using any other Cygwin based libraries then you will probably need to build them as DLL's using ***gcc*** and then generate import libraries for the MS VS linker.

26. How do you start BMDFM on Windows with Cygwin?

Please, read the following user story:

BMDFM and Cygwin

```
- I have an old version of Cygwin and I cannot start BMDFM. Probably my Cygwin version does not support POSIX for 100%.

- No problem. We can try to start BMDFM with your current Cygwin version. Which error message did you get?

- Single-threaded version works fine, but when I start BMDFMsrv I get the following error:

  [Msg]: Determining the system semaphore parameters...
  Cannot determine sems_per_group for SVR4 sema4.
  ProcName=BMDFMsrv, PID=4104, tID=4104, Module=mem_pool, Function=sem4svr4_determ(), Location=2.
  semget(key_t key = 0(IPC_PRIVATE), int nsems = 1, int shmflg = 1968(512(IPC_CREAT)|1024(IPC_EXCL)|432(permissions)))

- What did you get when your try ipcs?

- $ ipcs
  ipcs: msgctl: Function not implemented

- Ok. You need to start Cygwin server:

  $ /usr/sbin/cygserver.exe &
  [1] 3280
  cygserver: Initialization complete.  Waiting for requests.

- BMDFM still does not start and hangs while starting PROCstat:

  [SysMsg]: Forking up and handshaking the PROCstat daemon...
  PROCstat daemon does not respond.
  ProcName=BMDFMsrv, PID=2292, tID=2292, Module=BMDFMsrv, Function=main(), Location=125.

- Try to attach strace to the PROCstat process. What do you see?

- $ strace -p 2412
  . . .
  136  432630 [main] PROCstat 2412 __set_errno: static int semaphore::wait(semaphore**):3925 setting errno 22
   59  432689 [main] PROCstat 2412 __set_errno: static int semaphore::wait(semaphore**):3925 setting errno 22
   50  432739 [main] PROCstat 2412 __set_errno: static int semaphore::wait(semaphore**):3925 setting errno 22
  . . .

- Ok. Your Cygwin does not fully support POSIX semaphores. Seems they do not work for multi processes.
  Let us switch BMDFM to SVR4 semaphores.
  Please, comment the following line in your BMDFMsrv.cfg file:

  #POSIX_SEMA4_SYNC   = RW+Count # Replace None/RW/RW+Count SVR4 with POSIX sema4

- BMDFM still does not start:

  [SysMsg]: Setting up the Task Connection Zone (TCZ)...
  Cannot create semaphore.
  ProcName=BMDFMsrv, PID=4720, tID=4720, Module=BMDFMsrv, Function=main(), Location=50, SysCall=semget(), errno=28
   : No space left on device

  *** EMERGENCY EXIT from the BM_DFM Server session. ***
  BM_DFM KERNEL PANIC, RETURNED STATUS: ABNORMAL PROGRAM TERMINATION.

- Ok. There is insufficient number of semaphores in the default configuration of the Cygwin server.
  Change the following settings in your /etc/cygserver.conf file and restart the Cygwin server:

  # kern.ipc.semmni: Maximum no. of semaphore identifiers hold concurrently.
  # Default: 10, Min: 1, Max: 1024
  kern.ipc.semmni 1024

  # kern.ipc.semmns: Maximum no. of semaphores hold concurrently.
  # Default: 60, Min: 1, Max: 1024
  kern.ipc.semmns 1024

  # kern.ipc.semmsl: Maximum no. of semaphores per semaphore id.
  # Default: 60, Min: 1, Max: 1024
  kern.ipc.semmsl 1024

- BMDFMsrv starts now. But BMDFMldr fails to start:

  $ BMDFMldr hello.flp
  Current termcap settings:
    TERM_TYPE=`xterm'; LINES_TERM=`35'; COLUMNS_TERM=`124';
    CLRSCR_TERM=`\e[H\e[2J'; REVERSE_TERM=`\e[7m'; BLINK_TERM=`\e[5m';
    BOLD_TERM=`\e[1m'; NORMAL_TERM=`\e[0m'; HIDECURSOR_TERM=`\e[?25l';
    SHOWCURSOR_TERM=`\e[?12l\e[?25h'; GOTOCURSOR_TERM=`\e[%i%d;%dH'.
  Reading the `/tmp/.BMDFMsrv' BM_DFM connection file...
  Opening the `/tmp/.BMDFMsrv_npipe' BM_DFM named FIFO pipe...
  Cannot open the named fifo pipe for R/W `/tmp/.BMDFMsrv_npipe'.
  ProcName=BMDFMldr, PID=5048, tID=5048, Module=BMDFMldr, Function=main(), Location=11, SysCall=open(), errno=16
   : Device or resource busy

  $ ls -la /tmp/.BMDFMsrv_npipe
  prw-rw-rw- 1 user None 0 Apr 13 23:42 /tmp/.BMDFMsrv_npipe

- What do you see when you start BMDFMldr with strace?

- $ strace BMDFMldr hello.flp
  . . .
  27  205505 [main] BMDFMldr 1120 open: -1 = open(/tmp/.BMDFMsrv_npipe, 0x8002), errno 16
  . . .

- Ok. Your Cygwin does not fully support named pipes. Let us do the following trick in the shell where you run your BMDFMldr:
  $ echo -n >npip
  $ export BM_DFM_CONNECTION_NPIP_path=npip
  $ tail -f npip >/tmp/.BMDFMsrv_npipe &
  $ BMDFMldr hello.flp

- Thank you! Everything works fine now.
```

BMDFM and Cygwin

27. Why cannot *sem_maxval* be determined for POSIX sema4?

The BMDFM Server may fail to start with an error message regarding POSIX sema4, e.g. when starting on Windows SFU/SUA:

```
C:\HOME\BMDFM_Win32-SFU-SUA - cmd.exe
[SysMsg]: Squeezing nested PROGN statements in the Global FastLisp function set...
[SysMsg]:   Redundant nested PROGN statements removed: 0.
[SysMsg]: Resolving data types in the Global FastLisp function set...
[SysMsg]: Compiling the Global FastLisp function source code (Pass One)...
[DFMKrnl]: Compiled Global function bytecode size is 7964Bytes.
[SysMsg]: Linking compiled Global function bytecode (Pass Two)...
[Msg]: Determining the system semaphore parameters...
                    * Stay alert! Information you are about to view is being logged as `BOOT DUMP'.


Cannot determine sem_maxval for POSIX sema4.
ProcName=BMDFMsrv, PID=1793, tID=1793, Module=mem_pool, Function=sem4posix_determ(), Location=4.

C:\HOME\BMDFM_Win32-SFU-SUA>
```

Starting the BMDFM Server in a Windows prompt

The reason might be that the POSIX sema4 functionality is not fully supported by the OS. This can be easily tested as shown below:

```
C:\HOME\BMDFM_Win32-SFU-SUA - cmd.exe
C:\HOME\BMDFM_Win32-SFU-SUA> C:\Windows\SUA\bin\ed
a
/* posixsem4_test.c */

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

int main(void){
  sem_t sem;              /* posix sema4 */
  int pshared=1;          /* share sema4 among processes */
  unsigned int value=0;   /* initial sema4 value */

  printf("sem_init(shared) has returned %d\n",sem_init(&sem,pshared,value));
  perror(NULL);
  return 0;
}
.
w posixsem4_test.c
364
q

C:\HOME\BMDFM_Win32-SFU-SUA> C:\Windows\SUA\opt\gcc.4.2\bin\gcc -L /dev/fs/C/Windows/SUA/usr/lib/x86 -
o posixsem4_test posixsem4_test.c

C:\HOME\BMDFM_Win32-SFU-SUA> posixsem4_test
sem_init(shared) has returned -1
Not supported

C:\HOME\BMDFM_Win32-SFU-SUA>
```

Testing whether POSIX sema4 can be shared among processes

In case where the POSIX sema4 cannot be shared among processes, BMDFM has to be configured for using SVR4 sema4 functionality:

```
BMDFMsrv.cfg
# . . .
POSIX_SEMA4_SYNC    =    None # Replace None/RW/RW+Count SVR4 with POSIX sema4
# . . .
```

SVR4 sema4 settings

28. How do you run BMDFM on Linux with *glibc* that is older than required by BMDFM?

This is the error you might get when you run BMDFM against an old ***glibc***, e.g.:

```
Terminal
$ BMDFMsrv
BMDFMsrv: /lib64/ld-linux.so.2: version `GLIBC_2.12' not found
BMDFMsrv: /lib64/libc.so.6: version `GLIBC_2.12' not found
BMDFMsrv: /lib64/libpthread.so.0: version `GLIBC_2.12' not found
$
```

Unresolved externals due to older library version

In order to stay non-intrusive to the system, it is possible to have multiple versions of ***glibc*** on the same system. Download and install required version of ***glibc*** into your private (e.g. ***/home/myglibc64***) directory to be used and linked against BMDFM.

Dynamically linked ELF-executables always specify a dynamic linker or interpreter, which is a program that actually loads the executable along with all its dynamically linked libraries. The absolute path to the interpreter (e.g. ***/lib64/ld-linux.so.2*** on 64-bit Linux) is hard-coded into the executable at link time. This absolute path can also be changed after the link is done, e.g. by a binary editor modifying the interpreter section of executable. However, this is not quite trivial because the path of the new interpreter may be longer than the old one. Download and install ***patchelf*** utility that takes care of increasing the executable size with sufficient space at the beginning to contain the new interpreter field. Note that the resulting executables may be one page (usually 4KB) larger.

Adjust BMDFM to use ***glibc*** from your private ***/home/myglibc64*** directory:

```
Terminal
$ for i in fastlisp BMDFMsrv PROCstat IORBPROC OQPROC CPUPROC BMDFMldr BMDFMtrc freeIPC; do patchelf -
-print-interpreter $i; done
/lib64/ld-linux.so.2
/lib64/ld-linux.so.2
/lib64/ld-linux.so.2
/lib64/ld-linux.so.2
/lib64/ld-linux.so.2
/lib64/ld-linux.so.2
/lib64/ld-linux.so.2
/lib64/ld-linux.so.2
/lib64/ld-linux.so.2

$ gcc -m64 -Wl,--rpath=/home/myglibc64 -Wl,--dynamic-linker=/home/myglibc64/ld-linux.so.2 -o fastlisp
fastlisp.o cflp_udf.o -lm -lpthread
$ gcc -m64 -Wl,--rpath=/home/myglibc64 -Wl,--dynamic-linker=/home/myglibc64/ld-linux.so.2 -o BMDFMldr
BMDFMldr.o cflp_udf.o -lm -lpthread
$ gcc -m64 -Wl,--rpath=/home/myglibc64 -Wl,--dynamic-linker=/home/myglibc64/ld-linux.so.2 -o BMDFMsrv
BMDFMsrv.o cflp_udf.o -lm -lpthread
$ gcc -m64 -Wl,--rpath=/home/myglibc64 -Wl,--dynamic-linker=/home/myglibc64/ld-linux.so.2 -o CPUPROC
CPUPROC.o cflp_udf.o -lm -lpthread

$ patchelf --set-rpath /home/myglibc64 --set-interpreter /home/myglibc64/ld-linux.so.2 PROCstat
$ patchelf --set-rpath /home/myglibc64 --set-interpreter /home/myglibc64/ld-linux.so.2 IORBPROC
$ patchelf --set-rpath /home/myglibc64 --set-interpreter /home/myglibc64/ld-linux.so.2 OQPROC
$ patchelf --set-rpath /home/myglibc64 --set-interpreter /home/myglibc64/ld-linux.so.2 BMDFMtrc
$ patchelf --set-rpath /home/myglibc64 --set-interpreter /home/myglibc64/ld-linux.so.2 freeIPC

$ for i in fastlisp BMDFMsrv PROCstat IORBPROC OQPROC CPUPROC BMDFMldr BMDFMtrc freeIPC; do patchelf -
-print-interpreter $i; done
/home/myglibc64/ld-linux.so.2
/home/myglibc64/ld-linux.so.2
/home/myglibc64/ld-linux.so.2
/home/myglibc64/ld-linux.so.2
/home/myglibc64/ld-linux.so.2
/home/myglibc64/ld-linux.so.2
/home/myglibc64/ld-linux.so.2
/home/myglibc64/ld-linux.so.2
/home/myglibc64/ld-linux.so.2
$
```

Switching BMDFM to pull libraries from another location

29. Why does it seem like BMDFM keyboard input is delayed on Linux Alpha-based machines?

Linux for Alpha-processor-based machines preserves some compatibility with OSF/1. Keyboard input is buffered to hold keystrokes in a buffer before they are processed. The buffer size can be seen by attaching ***strace*** to ***BMDFMsrv*** when the BMDFM server waits for the keyboard input from the BMDFM server console:

Terminal on Alpha server

```
$ strace -v -s 1000 -p <PID_of_BMDFMsrv>
. . .

ioctl(3, TCGETA, {c_iflags=0x4300, c_oflags=0x3, c_cflags=0xb0f, c_lflags=0x5cf,
 c_line=0, c_cc="\x03\x1c\x7f\x15\x04\x00\x00\x00"}) = 0
ioctl(3, TCSETAW, {c_iflags=0x4300, c_oflags=0xc03, c_cflags=0xb0f, c_lflags=0x4
c1, c_line=0, c_cc[_VMIN]=4, c_cc[_VTIME]=0, c_cc="\x03\x1c\x7f\x15\x04\x00\x00\
x00"}) = 0
select(8, [3 5 7], NULL, NULL, NULL)    = 1 (in [3])
read(3, "a", 1)                         = 1
ioctl(3, TCSETAW, {c_iflags=0x4300, c_oflags=0x3, c_cflags=0xb0f, c_lflags=0x5cf
, c_line=0, c_cc="\x03\x1c\x7f\x15\x04\x00\x00\x00"}) = 0

. . .

ioctl(3, TCGETA, {c_iflags=0x4300, c_oflags=0x3, c_cflags=0xb0f, c_lflags=0x5cf,
 c_line=0, c_cc="\x03\x1c\x7f\x15\x04\x00\x00\x00"}) = 0
ioctl(3, TCSETAW, {c_iflags=0x4300, c_oflags=0xc03, c_cflags=0xb0f, c_lflags=0x4
c1, c_line=0, c_cc[_VMIN]=4, c_cc[_VTIME]=0, c_cc="\x03\x1c\x7f\x15\x04\x00\x00\
x00"}) = 0
select(8, [3 5 7], NULL, NULL, NULL

. . .
$
```

Terminal

Similar information can be obtained by ***stty*** running in terminal where the BMDFM server is executed:

Terminal on Alpha server

```
$ stty -a -F /dev/tty
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
werase = ^W; lnext = ^V; discard = ^U; min = 4; time = 0;
-parenb -parodd -cmspar cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany -imaxbel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke -flusho
$
```

Terminal

Changing the keyboard buffer size to 1 fixes the problem (note that the keyboard buffer size has to be set to 1 separately for each terminal where BMDFM is used):

Terminal on Alpha server

```
$ stty -icanon min 1 -F /dev/tty

$ stty -a -F /dev/tty
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
werase = ^W; lnext = ^V; discard = ^U; min = 1; time = 0;
-parenb -parodd -cmspar cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany -imaxbel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke -flusho
$
```

Terminal

30. How do you add “NUMA-awareness” to BMDFM?

Non-Uniform Memory Access (NUMA) design divides memory into multiple memory nodes, which are local to one or more CPUs. The local memory node can be accessed faster than the other memory nodes. From this perspective, a NUMA system can be viewed as a set of SMP systems: each NUMA node acts as an SMP system. NUMA nodes are connected via some sort of system interconnect. A crossbar or point-to-point link are the most common types of such interconnects. Modern servers with multiple CPU sockets usually have NUMA architecture. NUMA configuration can be displayed by ***numactl***:

Terminal (Sun X4600M2 8x[Opteron/4cores]; Linux)

```
$ numactl --hardware
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3
node 0 size: 32254 MB
node 0 free: 32184 MB
node 1 cpus: 4 5 6 7
node 1 size: 32244 MB
node 1 free: 32174 MB
node 2 cpus: 8 9 10 11
node 2 size: 32255 MB
node 2 free: 32171 MB
node 3 cpus: 12 13 14 15
node 3 size: 32255 MB
node 3 free: 32144 MB
node 4 cpus: 16 17 18 19
node 4 size: 32255 MB
node 4 free: 32117 MB
node 5 cpus: 20 21 22 23
node 5 size: 32255 MB
node 5 free: 32130 MB
node 6 cpus: 24 25 26 27
node 6 size: 32255 MB
node 6 free: 32129 MB
node 7 cpus: 28 29 30 31
node 7 size: 32255 MB
node 7 free: 32184 MB
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  12  12  14  14  14  14  16
  1:  12  10  14  12  14  14  12  14
  2:  12  14  10  14  12  12  14  14
  3:  14  12  14  10  12  12  14  14
  4:  14  14  12  12  10  14  12  14
  5:  14  14  12  12  14  10  14  12
  6:  14  12  14  14  12  14  10  12
  7:  16  14  14  14  14  12  12  10
$
```

Terminal (IBM S822LC 2x[POWER8/10cores/SMT8]; Linux)

```
$ numactl --hardware
available: 2 nodes (0,1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
node 0 size: 262144 MB
node 0 free: 253490 MB
node 1 cpus: 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97
98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114
115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131
132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148
149 150 151 152 153 154 155 156 157 158 159
node 1 size: 262144 MB
node 1 free: 259213 MB
node distances:
node   0   1
  0:  10  40
  1:  40  10
$
```

Terminals

Node distance matrices of these machines show that the neighbor memory node can be accessed up to 1.2 - 1.6 times slower than the local memory node for Sun X4600M2 and up to 4 times slower than the local memory node for IBM S822LC.

The standard ***libnuma*** library provides NUMA interface that allows one to build required NUMA policy depending on application business logic. The simplest NUMA policy for BMDFM that makes sense would be to manage ***process affinity*** in such a way that each ***CPUPROC*** process or thread runs on a dedicated NUMA node performing memory allocation locally on this node for the fastest local memory access on this node:

C code for “NUMA-awareness” added to *cflp_udf.c*

```
/* cflp_udf.c */
// . . .
#include <numa.h>  /* include NUMA interface (link CPUPROC against libnuma library: -lnuma) */
// . . .
int NUMA_Nodes=1;  /* number of configured NUMA nodes */
// . . .

void startup_callback(void){
  // . . .
  if(am_I_in_the_CPUPROC_module()){
    if(numa_available()==-1)  /* check whether NUMA functionality is available */
      fprintf(stderr, "startup_callback(): WARN: NUMA functionality is not available!\n");
    else{
      NUMA_Nodes=numa_num_configured_nodes();  /* number of configured NUMA nodes */
      if(NUMA_Nodes<1){
        fprintf(stderr, "startup_callback(): WARN: no configured NUMA nodes!\n");
        NUMA_Nodes=1;
      }
      fprintf(stderr, "startup_callback(): INFO: number of configured NUMA nodes: %d.\n", NUMA_Nodes);
      if(numa_run_on_node(get_id_cpuproc()%NUMA_Nodes)==-1)  /* run process or thread on NUMA node */
        fprintf(stderr, "startup_callback(): WARN: numa_run_on_node() failed! %d\n", errno);
      else{
        numa_set_localalloc();  /* allocate memory on local node */
        fprintf(stderr, "startup_callback(): INFO: CPUPROC proc/thread %ld runs on NUMA node %ld.\n",
          get_id_cpuproc(), get_id_cpuproc()%NUMA_Nodes);
      }
    }
    // . . .
  }
  // . . .
  return;
}
```

The simplest NUMA policy for BMDFM: run each ***CPUPROC*** process or thread on a dedicated NUMA node with local memory allocation

31. Is there something in common between BMDFM and a multi-issue dynamic scheduling CPU?

Both are dataflow machines and have a lot of common internal architectural solutions. Understanding of the following similarities helps us to use BMDFM more efficiently:

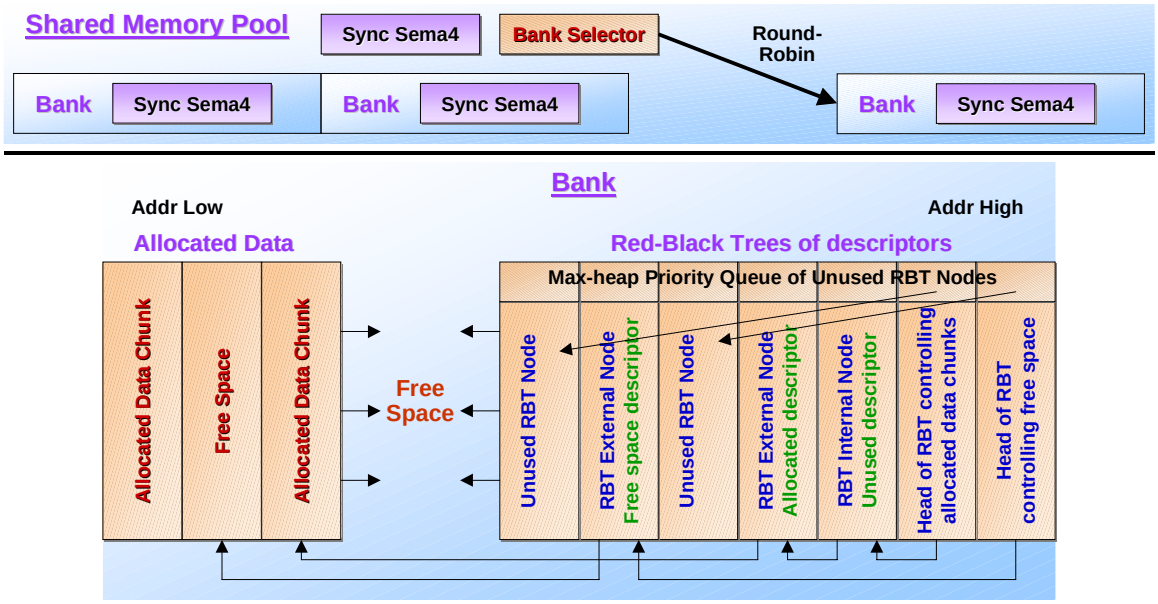
Multi-issue dynamic scheduling processor	BMDFM
Hardware dataflow machine that uses ***Tomasulo's algorithm*** to exploit ***instruction-level parallelism***.	Software dataflow machine that uses the ***Tagged-token principle*** to exploit ***thread-level parallelism*** of virtual machine instructions.
Dataflow is local within one processor chip. Execution units are the processing elements of the ***ALU*** and ***FPU***.	Dataflow is global within multi-core SMP machine. Execution units are the processors and cores themselves.
Tomasulo's approach defines the ***reservation station*** as a unit that is used for ***register renaming***.	BMDFM defines ***contexted data structure*** for each variable using nearly the same principle.
To feed its own dataflow avoiding stalls, the processor requires multiple flows of the ***instruction fetch***.	To feed the BMDFM dataflow engine avoiding bottlenecks, the ***BMDFMldr*** External Task Loader and Scheduler sustains multiple flows of ***marshaled clusters***.
To fill dataflow resources more efficiently, a concept of ***simultaneous multithreading*** is used that naturally matches the register renaming principles.	To fill dataflow resources more efficiently, many ***BMDFMldr*** processes can connect to the ***Task Connection Zone*** of BMDFM simultaneously, which naturally matches the contexted data principles.
To avoid stalls in the internal ***RISC-pipelines***, ***instruction prefetch*** and ***branch prediction*** units are used.	Same ideas are used to avoid stalls: ready VM-instructions are prefetched into the ***CPUPROC*** pipelines; recurrence is predicted to reduce scheduling effort for tagging ready VM-instructions.
Predicated instructions are used to shift conditional branches from the pipeline into the logic of the instruction itself.	***User-defined coarse-grain VM-instructions*** are defined as seamless blocks to move scheduling-expensive pieces of code into the logic of such a VM-instruction itself.

Comparison table

32. How is the BMDFM Shared Memory Pool architected?

The BMDFM Shared Memory Pool is divided into banks. Each bank is protected by a semaphore. Data chunks are allocated starting from lower addresses of the bank. A bank's control structures are in the higher addresses. These control structures are two ***Red-Black-Trees*** of descriptors (to be more precise, two ***Red-Black-Trees*** of ***RBT-nodes*** where external nodes store the descriptors): one RBT with descriptors pointing to the allocated chunks of data, and the other RBT with descriptors pointing to the holes of free space. Each RBT-node has a reserved field. Because all RBT-nodes are allocated linearly, it makes it possible that all reserved fields comprise a ***Max-heap Priority Queue***, which is used as storage for pointers to unused RBT-nodes.

Thus, allocation and freeing of memory blocks basically invoke a sequence of insert/delete operations in two Red-Black-Trees (O(log n)). Each RBT, in its own turn, uses the Max-heap Priority Queue to allocate its internal and external nodes (again O(log n)). Max-heap guarantees that the root of the Priority Queue always points to an unused RBT-node with the highest address. This address and such a node will be used first, ensuring a compact node allocation and making life of RBT node lazy garbage collector much easier.



*** Banks are thread-safe for parallel allocation**
*** Alloc() and free() invoke RBT insert/delete operation and Max-heap queuing for RBT nodes**
*** When free space exhausted, the RBT node lazy garbage collector is triggered or next bank is chosen**

The architecture of the Shared Memory Pool

The ***shmempool*** command of the BMDFM Server console displays current state of the Shared Memory Pool:

Output of *shmempool*

```
Console input: shmempool
[SysMsg]: ========== System time is Fri Nov 13 18:38:58 2015. ==========
[MemPool]: * STATUS OF THE SHARED MEMORY DRIVEN BY THE RE-ENTRANT CODE *
[MemPool]: Shared memory segment ID=229379.
[MemPool]: SHMEM_POOL_SIZE: 500000000Bytes (10 BANKS of 49999896 each).
[MemPool]: Shared memory segment has been attached at 0x000000003B9AC000.
[MemPool]: Shared memory segment permissions are: 0660=="rw-rw----".
[MemPool]: Using POSIX sema4 sync instead of SVR4 sema4 sync.
[MemPool]: Red-Black Tree (RBT) node size: 72Bytes.
[MemPool]: Number of reserved RBT-nodes: 13.
[MemPool]:<BANK#: Entities, FirstEntSpaceAfter, Free(Max), Fragmentation.>
[MemPool]: B#0: Ent=161, FA=160, Free=49601248(49601248), Frag=0.00%.
[MemPool]: B#1: Ent=161, FA=160, Free=49684504(49684504), Frag=0.00%.
[MemPool]: B#2: Ent=161, FA=160, Free=49720272(49720272), Frag=0.00%.
[MemPool]: B#3: Ent=161, FA=160, Free=49648192(49648192), Frag=0.00%.
[MemPool]: B#4: Ent=161, FA=160, Free=49635600(49635600), Frag=0.00%.
[MemPool]: B#5: Ent=161, FA=160, Free=49706240(49706240), Frag=0.00%.
[MemPool]: B#6: Ent=161, FA=160, Free=48657528(48657528), Frag=0.00%.
[MemPool]: B#7: Ent=160, FA=159, Free=49712464(49712464), Frag=0.00%.
[MemPool]: B#8: Ent=160, FA=159, Free=49720872(49720872), Frag=0.00%.
[MemPool]: B#9: Ent=160, FA=159, Free=49718248(49718248), Frag=0.00%.
[MemPool]: Memory Pool TOTAL:
[MemPool]:   Number of allocated entities: 1607.
[MemPool]:   Number of all/(LazyGarbageCollected) RBT-nodes: 3224/(3224).
[MemPool]:   Allocated size: 3960944Bytes.
[MemPool]:   Free space/(LargestFreeBlock): 495805168/(49720872)Bytes.
[MemPool]:   Fragmentation of holes: 0.00%.
[MemPool]:   Number of extra multicast references: 0.
```

Output of the ***shmempool*** command on the BMDFM Server console

 # <EOF>