

BMDFM FAQ

(A.k.a. “A Little Boy and His BMDFM”)

“And now, let's plunge into the dense fog!”

1. Why is the true multi-process model used in BMDFM but not multithreading?
2. Which build/revision of BMDFM is the latest?
3. How do you solve *termcap/terminfo* issues?
4. How do you change the default shared memory and semaphore limits under Linux?
5. How do you find out whether the OS is able to provide enough semaphores for BMDFM?
6. How do you start the BMDFM Server detached from a terminal and control it later?
7. How do you start many instances of BMDFM on the same machine?
8. How do you get a list of recognizable parameters for the BMDFM configuration profile?
9. Where can it be necessary to change the mounting address of the shared memory segment?
10. Is there any difference between a memory descriptor and a memory address?
11. How does BMDFM handle strings internally?
12. Why use *USER_IO*?
13. How do you evaluate the VM language expressions from C/C++ code?
14. How do you allocate/free shared memory from C/C++ code?
15. What is the optimal number of the BMDFM processes?
16. How do you rewrite application example from the BMDFM manual in pure VM language?
17. How serious is the performance degradation of pure unparallelled VM byte code?
18. How does the relaxed consistency model of shared memory influence BMDFM?
19. How does the BMDFM dataflow engine process an array?
20. Is there something in common between BMDFM and a multi-issue dynamic scheduling CPU?
21. How is the BMDFM Shared Memory Pool architected?

* * *

1. Why is the true multi-process model used in BMDFM but not 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 *MxI* runs all threads of the user process space through a single thread of kernel space relying on the kernel scheduler only. Threading model *IxI* 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. To make BMDFM portable across different SMP platforms and 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. No additional threading runtime library is required. Apparently, BMDFM is a scheduler itself.
- Performance is the most important point. Threading models are rather tailored to multi-tasking and programming convenience issues than to the true parallel processing. The multi-process model is more scalable and has better performance in practice than the threading model when running tasks on a big iron.

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 the *POSIX-threaded BMDFM*, which uses *POSIX semaphores* and *POSIX mutexes* as the synchronization primitives, exists and runs regression tests in the BMDFM labs. However, the *POSIX-threaded BMDFM* is not currently scheduled for inclusion into the public BMDFM distribution.

2. Which build/revision of BMDFM is the latest?

The *BMDFMfactory-5.9.9.tar.gz* standard distribution bundle contains *BM_DFM/Binaries/BuildStatus.txt* file listing the revision dates for all supported platforms:

BM_DFM/Binaries/BuildStatus.txt			
BMDFM 5.9.9 Build Status:			
Arch/Platform	Build		
Intel_Win32	13-Dec-2005	latest	release
Intel_Linux_32	25-Feb-2006	latest	release
Intel_FreeBSD_32	25-Feb-2006	latest	release
IA-64_Linux_64	25-Feb-2006	latest	release
AMDx86-64_Linux_64	25-Feb-2006	latest	release
Alpha_Tru64OSF1_64	25-Feb-2006	latest	release
Alpha_Linux_64	25-Feb-2006	latest	release
Alpha_FreeBSD_64	25-Feb-2006	latest	release
PA-RISC_HP-UX_32	25-Feb-2006	latest	release
PA-RISC_HP-UX_64	25-Feb-2006	latest	release
SPARC_SunOS_32	25-Feb-2006	latest	release
SPARC_SunOS_64	25-Feb-2006	latest	release
MIPS_IRIX_32	25-Feb-2006	latest	release
MIPS_IRIX_64	25-Feb-2006	latest	release
RS6000_AIX_32	25-Feb-2006	latest	release
RS6000_AIX_64	25-Feb-2006	latest	release
PowerPC_MacOS_32	25-Feb-2006	latest	release
Intel_Win32-UWIN	25-Feb-2006	latest	release
Intel_Win32-SFU	25-Feb-2006	latest	release
ARM_Linux_32	25-Feb-2006	latest	release
M68000_Linux_32	25-Feb-2006	latest	release
MIPSEL_Linux_32	25-Feb-2006	latest	release
MIPS_Linux_32	25-Feb-2006	latest	release
PA-RISC_Linux_32	25-Feb-2006	latest	release
PowerPC_Linux_32	25-Feb-2006	latest	release
PowerPC_Linux_64	25-Feb-2006	latest	release
S390_Linux_32	25-Feb-2006	latest	release
SPARC_Linux_32	25-Feb-2006	latest	release
SPARC_Linux_64	25-Feb-2006	latest	release
Intel_HP-UX_32	10-Mar-2006	latest	release
IA-64_HP-UX_64	10-Mar-2006	latest	release
Intel_SunOS_32	10-Mar-2006	latest	release
AMDx86-64_SunOS_64	10-Mar-2006	latest	release

BM_DFM/Binaries/BuildStatus.txt

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

Terminal (HP-UX)	Terminal (SunOS)
<pre> \$ fastlisp fastlisp ==> stderr: /* fastlisp.c - A FastLisp compiler with the run-time environment. ----- Original 32-bit version for UNIX was founded && written by: S.M. 05-05-1996 10:10:29.18pm */ Usage0: fastlisp -h --help Usage1: fastlisp -V --versions Usage2: fastlisp [-q --quiet] <lisp_file_name> [parms...] Usage3: fastlisp [-c --compile2disk] <lisp_file_name> [parms...] Usage4: fastlisp [-q --quiet] <precompiled_lisp_file_name> The environment variable defines a configuration profile that can be used for the Global FastLisp function definitions [Format of the cfg_profile is: <(defun ...) (defun ...) ... (defun ...)]: FAST_LISP_CFGPROFILE_path="fastlisp.cfg". Compiled on: "HP-UX obelix B.11.11 U 9000/785". Compiled by: "ccom: HP92453-01 B.11.11.06 HP C Compiler (ANSI) HPPA64 (hppa2.0w- hp-hpux11.11) as [ELF-64 executable object file - PA-RISC 2.0 (LP64)] at Sat Feb 25 14:59:00 MET 2006". RETURNED STATUS: ABNORMAL PROGRAM TERMINATION. \$ </pre>	<pre> \$ fastlisp fastlisp ==> stderr: /* fastlisp.c - A FastLisp compiler with the run-time environment. ----- Original 32-bit version for UNIX was founded && written by: S.M. 05-05-1996 10:10:29.18pm */ Usage0: fastlisp -h --help Usage1: fastlisp -V --versions Usage2: fastlisp [-q --quiet] <lisp_file_name> [parms...] Usage3: fastlisp [-c --compile2disk] <lisp_file_name> [parms...] Usage4: fastlisp [-q --quiet] <precompiled_lisp_file_name> The environment variable defines a configuration profile that can be used for the Global FastLisp function definitions [Format of the cfg_profile is: <(defun ...) (defun ...) ... (defun ...)]: FAST_LISP_CFGPROFILE_path="fastlisp.cfg". Compiled on: "SunOS wg-serv3 5.8 Generic_117350-27 sun4u sparc". Compiled by: "cc: Sun Workshop 6 update 2 C 5.3 Patch 111679-14 2004/02/20 as [E LF 64-bit MSB executable SPARCv9 Version 1, dynamically linked, stripped] at Sat Feb 25 14:04:01 MET 2006". RETURNED STATUS: ABNORMAL PROGRAM TERMINATION. \$ </pre>
<pre> \$ BMDFMSrv -h BMDFMSrv ==> stdout: /* BMDFMSrv.c - "Broken Mind" Data-Flow Machine SMP MIMD Server Unit. ----- Original 32-bit version for UNIX was founded && written by: S.M. 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_PROCCstat_path="/PROCCstat"; 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"; VERSION_BMDFM_SYS: "S.M. BMDFMSys V5.9.9.". Compiled on: "IRIX64 rapunzel 6.5 01062343 IP2T". Compiled by: "MIPSpro Compilers: Version 7.41 as [ELF 64-bit MSB mips-4 dynamic executable MIPS - version 1] at Sat Feb 25 16:24:12 MET 2006". \$ </pre>	<pre> \$ BMDFMSrv -h BMDFMSrv ==> stdout: /* BMDFMSrv.c - "Broken Mind" Data-Flow Machine SMP MIMD Server Unit. ----- Original 32-bit version for UNIX was founded && written by: S.M. 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_PROCCstat_path="/PROCCstat"; 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"; VERSION_BMDFM_SYS: "S.M. BMDFMSys V5.9.9.". Compiled on: "AIX ibmsmp 1 5 001058BA4C00". Compiled by: "VisualAge C++ Professional / C for AIX Compiler, Version 6 as [64- bit XCOFF executable (RISC System/6000)] at Sat Feb 25 16:53:00 MEZ 2006". \$ </pre>

Terminals

3. How do you solve *termcap/terminfo* issues?

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

To find out whether *termcap* or *terminfo* database is used in the build for your certain platform, open one of the BMDFM executables (*fastlisp*, *BMDFMsrv*, *BMDFMldr*, *BMDFMtrc* or *CPUPROC*) in a binary editor. Search for the specific string entry: `"/etc/termcap"` or `"/usr/share/lib/terminfo"`.

Check the `$TERM` environment variable whether it contains a correct terminal name, which can be found in the *termcap/terminfo* 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 TERMCAP=/<full_path>/termcap
$
```

Terminal

If the standard *terminfo* database is missing in the system then use the one provided with the BMDFM distribution:

```
Terminal (bash)
$ export TERMINFO=/<full_path>/<new_terminfo_directory>
$ mkdir $TERMINFO
$ tic terminfo
$
```

Terminal

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

```
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 (SAMPLE No# 2 for fastlisp or No# 3 for BMDFMldr)
...
Modifying the user's Lisp code (SAMPLE No# 3)...
(PROGN {(SETQ <termcap_var> <termcap_val> }<Lisp_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
(outf (prn_string_fmt) (clrscr_term))
(outf (prn_string_fmt) (gotocursor_term (>> (lines_term 1) (>> (columns_term 1))))

VM code fragment using predefined termcap variables (more correct)
(outf (prn_string_fmt) clrscr_term)
(outf (prn_string_fmt) (gotocursor1_term gotocursor_term (>> lines_term 1) (>> columns_term 1)))
```

VM code fragments

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

Under Linux, the default shared memory limit (both *shmmax* and *shmall*) is 32 MB; however, it can be changed in the proc file system (without a reboot). For example, to allow 64GB:

```
Terminal
$ su root
Password:
# echo 68719476736 >/proc/sys/kernel/shmall
# echo 68719476736 >/proc/sys/kernel/shmmax
# exit
$
```

Terminal

A user could put these commands into a script run 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 also be called explicitly from the command line:

```
Terminal
$ su root
Password:
# sysctl -w kernel.shmall=68719476736
# sysctl -w kernel.shmmax=68719476736
# exit
$
```

Terminal

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

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 off 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.
[SysMsg]: Organizing an abstract DFM STRUCTURE UNIT in the SHMEM_POOL:
[SysMsg]:   Initializing CPU PROCs state array...
[SysMsg]:   Organizing DFM IORBPs...
[SysMsg]:   Collecting the system semaphores for the OQ and DB...
[SysMsg]:   SemOQ=2000/2000, SemDBAreas=28000/28000.
[SysMsg]:   Organizing DFM OQ...
[SysMsg]:   Organizing DFM DB...
[MemPool]: Init the Shared Memory Pool done.
. . .
```

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.
[SysMsg]: Organizing an abstract DFM STRUCTURE UNIT in the SHMEM_POOL:
[SysMsg]:   Initializing CPU PROCs state array...
[SysMsg]:   Organizing DFM IORBPs...
[SysMsg]:   Collecting the 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...
[MemPool]: Init the Shared Memory Pool done.
. . .
```

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. 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

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 his/her 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
[TermCap]: ~~~~~Server is running on TERM=ansi.sys (25x80).
[SysMsg]: Total init for the VIRTUAL OUT-OF-ORDER PROCESSING was completed a
t Sat May 6 18:21:15 2006.
[SysMsg]: Going simultaneous jobs running all the threads in parallel...
[DFMSrv]: All resources were unhooked and invoked successfully!
[SysMsg]: "Broken Mind" Data-Flow Machine Server has been started.
[SysMsg]: The DFM SMP MIMD arch now is prepared for DYNAMIC SCHEDULING.
[Initial_MainFrame_Legacy_Greeting_Message]: GOOD EVENING.

Console input:
[SysMsg]: ===== System time is Sat May 6 18:22:36 2006. =====
[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.

Console input: down down
[SysMsg]: ===== System time is Sat May 6 18:23:32 2006. =====
[SysMsg]: The BM_DFM Server is urgently going down NOW...
[SysMsg]: Destroying the external connection file...
[SysMsg]: Destroying the ExtTask(Trace) nFIFO pipe...
[SysMsg]: Sending a SIGKILL to ExtTasks in ConnZone...
[SysMsg]: Sending a SIGKILL to ExtTraces in PlugArea...
[SysMsg]: Sending a SIGKILL to the PROCstat...
[SysMsg]: Sending a SIGKILL to the CPU PROCs...
[SysMsg]: Sending a SIGKILL to the OQ PROCs...
[SysMsg]: Sending a SIGKILL to the IORBP PROCs...
[SysMsg]: Deinitializing the BM_DFM...
[DFMSrv]: Release semaphores done.
[DFMSrv]: Close msg PROCs pipe done.
[MemPool]: Deinit shared memory pool done.
[SysMsg]: Destroying the freeIPC EMERGENCY CASE file...
[SysMsg]: SHUTDOWN complete at Sat May 6 18:23:32 2006.
[Final_MainFrame_Legacy_Message]: GOOD BYE.
[SysMsg]: Closing the logs './BMDFMsrv.log'...
*** Logfile is closed at Sat May 6 18:23:32 2006 ***
^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 -50 $BMDFM_LOG_FILENAME

downdown.sh
#!/bin/sh

echo command down down >$BM_DFM_CONNECTION_NPIP_path;
tail -30 $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 semaphores used. In other words, it is important to prevent a situation where one running BMDFM instance holds all available 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 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 semaphore quota per instance.

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

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

```
Output of dfmkernel
-----
Console input: dfmkernel
[SysMsg]: ===== System time is Sat May 6 17:53:43 2006. =====
[DFMKrnl]: Global parameters of the BM_DFM Kernel:
[DFMKrnl]: Operation Queue (OQ) size: Q_OQ=3000Entities.
[DFMKrnl]: Data Buffer (DB) size: Q_DB=1000Entities.
[DFMKrnl]: I/O Ring Buffer Port (IORBP) size: Q_IORBP=16Entities.
[DFMKrnl]: Number of the IORBPs: N_IORBP=5.
[DFMKrnl]: Number of the main processes (CPU PROCs): N_CPUPROC=4.
[DFMKrnl]: Number of the OQ PROCs: N_OQPROC=4.
[DFMKrnl]: Number of the IORBP PROCs: N_IORBPPROC=4.
[DFMKrnl]: Block size used in OQ search algorithm is 108.
[DFMKrnl]: Size of caches in speculative prediction unit is 48000Bytes.
[DFMKrnl]: Associative hierarchy of speculative tagging max. 3111000Bytes.
[DFMKrnl]: Display stall warnings: STALL_WARNINGS=NO.
[DFMKrnl]: Hard synchronization of the arrays: HARD_ARRAY_SYNCHRO=NO.
[DFMKrnl]: I/O synchronization of external task: EXT_IN_OUT_SYNCHRO=YES.
[DFMKrnl]: SM relaxed consistency compensation: RELAXED_CNSTN_SM_MODEL=YES.

Output of dfmserver
-----
Console input: dfmserver
[SysMsg]: ===== System time is Sat May 6 17:54:10 2006. =====
[DFMSrv]: The BM_DFM System processes' PIDs:
[DFMSrv]:  N#          | CPUPROCs   | OQPROCs    | IORBPPROCs | PROCstat
[DFMSrv]:  -----+-----+-----+-----+-----
[DFMSrv]:          0 |         141 |         145 |         149 |         140
[DFMSrv]:          1 |         142 |         146 |         150 |
[DFMSrv]:          2 |         143 |         147 |         151 |
[DFMSrv]:          3 |         144 |         148 |         152 |
[DFMSrv]: Global parameters of the BM_DFM Server:
[DFMSrv]: AGGRESSIVE compilation: SPECULATIVE_RISC_ARCH = 1(yes).
[DFMSrv]: Own system SHM_SEMAPHORE: REENTERABLE_SHMEM_POOL = 1(yes).
[DFMSrv]: The BM_DFM Server PID=139.
[DFMSrv]: Number of semaphores per group is 250.
[DFMSrv]: Maximal semaphore value is 32767.
[DFMSrv]: ShMemPool mount address (0=auto): SHMEM_POOL_MNTADDR=0.
[DFMSrv]: Array block size: ARRAYBLOCK_SIZE=20Entities.
[DFMSrv]: OQ function arguments count: OQ_FUNC_ARG_COUNT=80Entities.
[DFMSrv]: Time to scan DFM for statistic: T_STATISTIC=1Second(s).
[DFMSrv]: Max number of OQ&&DB semaphores (0=ulim): OQ_DB_SEM_LIMIT=0.
[DFMSrv]: Number of the Trace Ports (TPs): N_TRACEPORT=3.
[DFMSrv]: Logs registration for the CPU && IORBP PROCs: PROC_CPU_LOGS=NO.
[DFMSrv]: Signal to reset/get used CPU time in CHILDS is 10 (irq).
[DFMSrv]: Signal to unhook CHILDS out from a semaphore is 12 (irq).
[DFMSrv]: Msg PROCs unnamed pipe R/W IDs: rID=5, wID=6.
[DFMSrv]: External task named pipe '/tmp/.BMDFMsrv_npipes' R/W ID=7.
[DFMSrv]: Semaphore permissions are 0x01B4.
```

Output of *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 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
    libm.so.6 => /lib64/tls/libm.so.6 (0x0000002a9566d000)
    libc.so.6 => /lib64/tls/libc.so.6 (0x0000002a957c5000)
    /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)
$ ldd BMDFMldr
    libm.so.6 => /lib64/tls/libm.so.6 (0x0000002a9566d000)
    libc.so.6 => /lib64/tls/libc.so.6 (0x0000002a957c5000)
    /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)
$ ldd BMDFMtrc
    libc.so.6 => /lib64/tls/libc.so.6 (0x0000002a9566d000)
    /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)
$ ldd PROCstat
    libc.so.6 => /lib64/tls/libc.so.6 (0x0000002a9566d000)
    /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)
$ ldd CPUPROC
    libm.so.6 => /lib64/tls/libm.so.6 (0x0000002a9566d000)
    libc.so.6 => /lib64/tls/libc.so.6 (0x0000002a957c5000)
    /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)
$ ldd OQPROC
    libc.so.6 => /lib64/tls/libc.so.6 (0x0000002a9566d000)
    /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)
$ ldd IOBPBPROC
    libc.so.6 => /lib64/tls/libc.so.6 (0x0000002a9566d000)
    /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)
$
```

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: *IA-32*, *IA-64* and *AMDx86-64* 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
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 (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

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 zeros up 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	
UCH = unsigned char ULO = unsigned long int	
Addr Low	Addr High
<ULO StrByteLength><String><ZeroCharTerminator><ZeroCharJustificationToSizeOfULO>	
	 UCH *StrPtr -'
Sample code using the BMDFM strings	
<pre>#define UCH unsigned char UCH *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); /* ==> 'TO COMPUTE OR NOT TO COMPUTE' */ free_string(&str0); free_string(&str1); free_string(&str2);</pre>	

BMDFM strings

The BMDFM internal string processing functions repeat those defined in VM language:

BMDFM string library	
<pre>#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 UCH *mk_std_buff(UCH **buff, ULO size); UCH *mk_fst_buff(UCH **buff, ULO size); UCH *get_std_buff(UCH **targ, UCH *buff); UCH notempty(UCH *string); ULO len(UCH *string); ULO at(UCH *pattern, UCH *among); ULO rat(UCH *pattern, UCH *among); UCH cmp(UCH *string1, UCH *string2); SCH cmp_s(UCH *string1, UCH *string2); UCH *equ(UCH **targ, UCH *source); UCH *equ_num(UCH **targ, SLO num); UCH *equ_fnum(UCH **targ, DFL fnum); UCH *cat(UCH **targ, UCH *source); UCH *catl(UCH **targ, UCH *source); UCH *space(UCH **targ, ULO pos); UCH *replicate(UCH **targ, UCH *source, ULO num); UCH *left(UCH **targ, UCH *source, ULO pos); UCH *leftr(UCH **targ, UCH *source, ULO posr); UCH *right(UCH **targ, UCH *source, ULO pos); UCH *rightl(UCH **targ, UCH *source, ULO posl); UCH *substr(UCH **targ, UCH *source, ULO from, ULO pos); UCH *strtran(UCH **targ, UCH *source, UCH *pattern, UCH *subst);</pre>	<pre>UCH *ltrim(UCH **targ, UCH *source); UCH *rtrim(UCH **targ, UCH *source); UCH *alltrim(UCH **targ, UCH *source); UCH *pack(UCH **targ, UCH *source); UCH *head(UCH **targ, UCH *source); UCH *tail(UCH **targ, UCH *source); UCH *flp_head(UCH **targ, UCH *source); UCH *flp_tail(UCH **targ, UCH *source); UCH *upper(UCH **targ, UCH *source); UCH *lower(UCH **targ, UCH *source); UCH *rev(UCH **targ, UCH *source); UCH *padl(UCH **targ, UCH *source, ULO width); UCH *padr(UCH **targ, UCH *source, ULO width); UCH *padc(UCH **targ, UCH *source, ULO width); UCH *straw(UCH **targ, UCH *source); UCH *strunraw(UCH **targ, UCH *source); UCH *strdump(UCH **targ, UCH *source); UCH *string_time(UCH **targ); UCH *strings_version(UCH **targ); UCH *sch2str(UCH **targ, SCH num); UCH *ssh2str(UCH **targ, SSH num); UCH *slo2str(UCH **targ, SLO num); UCH *ptr2str(UCH **targ, void *ptr); UCH *dfl2str(UCH **targ, DFL num); SCH str2sch(UCH *string); SSH str2ssh(UCH *string); SLO str2slo(UCH *string); void *str2ptr(UCH *string); DFL str2dfl(UCH *string); UCH *free_string(UCH **targ);</pre>

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 <func_prompt_message_for_console_or_empty_for_stdin>)
(scan_console <wait_keypress_forever_if_1_or_useconds_if_ positive>)
(file_create <file_name>)
(file_open <file_name>)
(file_write <file_descriptor> <string_to_be_written>)
(file_read <file_descriptor> <number_of_bytes_to_be_read>)
(file_close <file_descriptor>)
(file_remove <file_name>)
(user_io <int_usr_id> <str_usr_buff>)
```

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 UCH unsigned 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, UCH **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 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 UCH unsigned char
#define ULO unsigned long int
#define SLO signed long int

extern void func_mapcar(ULO*, struct fastlisp_data*);
extern void func_dummy_s(ULO*, struct fastlisp_data*);

struct{
    UCH *flp_expr;
    UCH *bytecode;
} flpeval_cache={NULL,NULL};

UCH flp_eval(UCH *flp_expr, struct fastlisp_data *ret_dat){
    UCH 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);
    cat1(&flp_fnc,slo2str(&temp,len(flp_fnc)));
    cat1(&flp_fnc,ptr2str(&temp,(void*)&func_dummy_s));
    cat1(&flp_fnc,temp);
    *((UCH**)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(flpeval("(asyncheap_create 1024)",&res))
    addr=res.value.ival;
free_flp_data(&res);
```

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

14. 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	
<pre> /* Pure C */ #define UCH unsigned 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 extern void FLP_MALLOC(ULO*, struct fastlisp_data*); extern void FLP_FREE(ULO*, struct fastlisp_data*); extern void func__dummy_i(ULO*, struct fastlisp_data*); void *flp_malloc(SLO bytes){ UCH *flp_fnc=NULL,*temp=NULL; SLO addr; struct fastlisp_data res={1,1,0,0,{0},NULL,1,NULL,{NULL}}; slo2str(&flp_fnc,bytes); cat1(&flp_fnc,ptr2str(&temp,(void*)&func__dummy_i)); cat1(&flp_fnc,temp); *((UCH**)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){ UCH *flp_fnc=NULL,*temp=NULL; struct fastlisp_data res={1,1,0,0,{0},NULL,1,NULL,{NULL}}; slo2str(&flp_fnc,addr); cat1(&flp_fnc,ptr2str(&temp,(void*)&func__dummy_i)); cat1(&flp_fnc,temp); *((UCH**)flp_fnc)=flp_fnc+sizeof(ULO); FLP_FREE((ULO*)flp_fnc,&res); free_flp_data(&res); free_string(&flp_fnc); free_string(&temp); return; } </pre>	<pre> // Pure C++ #define SLO signed long int class foo{ // Use pattern for the class: // foo *foo0=new foo,*fool=new foo[2]; // delete foo0; // delete[] fool; 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; } </pre>

Shared memory operations 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         |000000000001864c| 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     |000000000002d4d4| 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

15. 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. 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. Therefore, it is recommended to set the number of **OQPROC** processes approximately equal to half of the available processing units.

Suppose a user has one dedicated virtual partition on an IBM SMP mainframe based on the **POWER** architecture. This partition has two dedicated **MCM (Multi-Chip Modules)** typically having four processors per module and two cores per processor. Hence, the number of processing units is $2*4*2=16$, and the following settings are recommended for such a configuration:

<i>BMDFMSrv.cfg</i>			
# . . .			
N_CPUPROC	=	16	# Number of the CPU PROCs
N_IORBPROC	=	16	# Number of the IORB PROCs
N_OQPROC	=	8	# Number of the OQ PROCs
# . . .			

Settings for 16 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 according to the nature of the running application tasks and the architecture of the SMP interconnections.

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

For those who run statically linked BMDFM with disabled C-interface (*BMDFM599free_80386Linux32static.tar.gz* bundle, *BMDFM599free_pariscHPUX32static.tar.gz* bundle, *BMDFM599free_sparcSunOS32static.tar.gz* bundle and *BMDFM599free_rs6000_AIX32static.tar.gz* bundle), it is recommended to rewrite application example from the BMDFM manual using 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
  )
)
)
)

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))
  ))
)
)
)
)

(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))
    )
  )
)
)
)
)

```

fastlisp.cfg/BMDFMsrv.cfg and dhtpipe0.flp

17. How serious is the performance degradation of pure unparallelled 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 packages, here only fragments of the code are given for comparison:

<pre> 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,cl,s1,sum,tmp; pi=3.1415926535897932; cl=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=cl*p*i+s1*q*j)+sin(tmp)); } *(target_array+p*m+q)=sum; } } } return; } </pre>
<pre> 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,cl,s1,sum,tmp; pi=3.1415926535897932; cl=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=cl*p*i+s1*q*j)+Math.sin(tmp)); } } target_array[p*m+q]=sum; } } return; } </pre>
<pre> 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 cl (/ (* 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 (+ cl (* p i) (* s1 (* q j))) s)))) (asyncheap_putfloat target_array (+ m p q) s)))))) </pre>

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 unparallelled 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 unparallelled 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.

18. 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
a=1;
// . . .
a=0;
if(b){
  // . . .
}
Process 1 shares A and B
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 *POWER5* 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.

19. 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
<pre>for(i=0;i<=N;i++) a[i]=...; for(i=0;i<=N;i++) b[i]=...a[i]...;</pre>

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.

USE CASE 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)
<pre>for(i=0;i<=N;i++) a[i]=...; for(i=0;i<=N;i++) a[i]=...;</pre>
Generated input to the BMDFM dataflow engine (pseudo-code)
<pre>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);</pre>

Fragments for USE CASE 0

USE CASE 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)
<pre>for(i=0;i<=N;i++) a[i]=func();</pre>
Generated input to the BMDFM dataflow engine (pseudo-code)
<pre>for(i=0;i<=N;i++){ temp=func(); // contexted, heavy-weight computations are parallel. a[i]=temp; // sequential, no performance degradations. }</pre>

Fragments for USE CASE 1

USE CASE 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
<pre>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; }</pre>

Fragments for USE CASE 2

USE CASE 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".

20. 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:

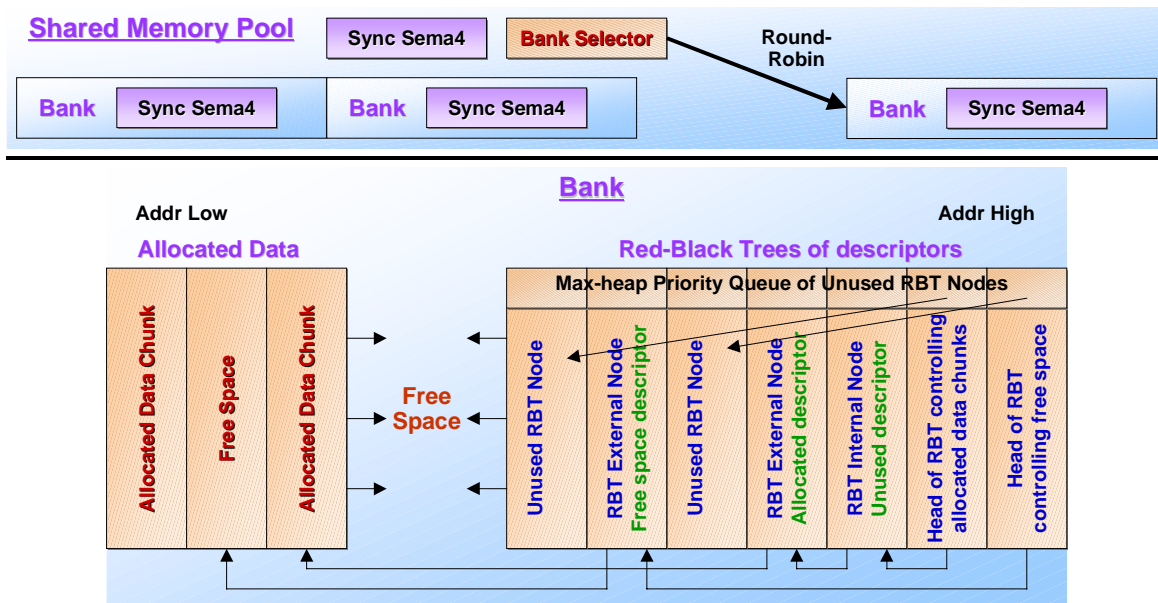
Multi-issue dynamic scheduling processor	BMDFM
Hardware dataflow machine that uses Tomasulo's algorithm to exploit instruction-level parallelism .	Software dataflow machine that uses the Tagged-token principle to exploit thread-level parallelism of virtual machine instructions.
Dataflow is local within 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

21. 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 the life of the 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 Sat May 6 18:05:13 2006. =====
[MemPool]: ** STATUS OF THE SHARED MEMORY DRIVEN BY REentrant CODE **
[MemPool]: Shared memory segment ID=622593.
[MemPool]: SHMEM_POOL_SIZE: 68719476736Bytes (4 BANK(S) of 17179869104 each).
[MemPool]: Shared memory segment has been attached at 0x2000000000400000.
[MemPool]: Shared memory segment permissions are 0x01B4.
[MemPool]: Red-Black Tree node size: 72Bytes.
[MemPool]:<BANK#: Entities, FirstEntSpaceAfter, Free(Max), Fragmentation.>
[MemPool]: B#0: Ent=6059, FA=2580, Free=14169231272(12023410128), Frag=71.30%.
[MemPool]: B#1: Ent=6062, FA=708, Free=11669047872(10036833488), Frag=29.62%.
[MemPool]: B#2: Ent=6060, FA=1739, Free=11807538304(10650111824), Frag=21.55%.
[MemPool]: B#3: Ent=6059, FA=939, Free=12261846632(12258315984), Frag=0.07%.
[MemPool]: Memory Pool TOTAL:
[MemPool]: Number of allocated entities: 24240.
[MemPool]: Number of all/(LazyGarbageCollected) RBT-nodes: 49680/(48566).
[MemPool]: Allocated size: 18808315296Bytes.
[MemPool]: Free space/(LargestFreeBlock): 49907664080/(12258315984)Bytes.
[MemPool]: Fragmentation of holes: 26.26%.
[MemPool]: Number of extra multicast references: 13.
    
```

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



<EOF>