CVM: The Coherent Virtual Machine

Pete Keleher
University of Maryland
keleher@cs.umd.edu

CVM Version 0.9
August 1, 1998

1. What’s New in 0.3.................................................................2
2. Introduction.........................................................................2
3. Installation.........................................................................2
   3.1 Getting CVM.................................................................2
   3.2 Compiling CVM and System Requirements.....................2
       3.2.1 FreeBSD Kernel Mod?..............................................3
   3.3 Running CVM..................................................................3
       3.3.1 Configuration...........................................................3
       3.3.2 Command-line Options..........................................3
       3.3.3 Debugging CVM Applications.................................4
       3.3.4 Cleaning Up After CVM...........................................5
4. Programming with CVM.....................................................5
   4.1 Shared Memory Semantics and Synchronization..............5
   4.2 Example Program..........................................................5
   4.3 CVM’s API.....................................................................6
       4.3.1 Initialization...........................................................6
       4.3.23.3.2 Synchronization..............................................6
       4.3.33.3.3 Cleanup..........................................................7
   4.4 Fortran...........................................................................7
5. Extending CVM...................................................................7
   5.1 Communication............................................................7
   5.2 Protocol Extension........................................................8
6. Benchmarking CVM............................................................8
7. Future Plans.......................................................................9
8. References.........................................................................9
1. What’s New in 0.3

1) Section 5 on benchmarking CVM. 
2) Many bug fixes.
3) New protocols (two variants of home-based protocols).
4) New platforms (linux, freebsd, bip-mpi).
5) Fortran!
6) More details in this document.

2. Introduction

This report describes contains the primary description of the Coherent Virtual Machine (CVM) software distributed shared memory system. CVM is a user-level library that links with shared memory programs and enables them to exploit shared-memory semantics over message-passing hardware, i.e., networks of workstations or distributed-memory parallel machines such as the IBM SP-2.

CVM supports multiple protocols and consistency models. Like commercially available systems such as TreadMarks [1], CVM is written entirely as a user-level library and runs on most UNIX-like systems. Unlike TreadMarks, CVM was created specifically as a platform for protocol experimentation.

The system is written in C++, and opaque interfaces are strictly enforced between different functional units of the system whenever possible. The base system provides a set of classes that implement a generic protocol, lightweight threads, and network communication. The latter functionality consists of efficient, end-to-end protocols built on top of UDP.

New shared memory protocols are created by deriving classes from the base Page and Protocol classes. Only those methods that differ from the base classes’ methods need to be defined in the derived classes. The underlying system calls protocol hooks before and after page faults, synchronization, and I/O events take place. Since many of the methods are inlined, the resulting system is able to perform within a few percent of a severely optimized system, TreadMarks, running a similar protocol. However, CVM was designed to take advantage of generalized synchronization interfaces, as well as to use multi-threading for latency toleration. We therefore expect the performance of the fully functional system to improve over the existing base. NOTE: Several of these techniques are either not yet implemented or in an incomplete state.

Of course, the ultimate reference for CVM is the source itself.

3. Installation

3.1 Getting CVM

CVM’s home page is http://www.cs.umd.edu/projects/cvm.html. This document, together with complete source code, can be downloaded from the home page.

3.2 Compiling CVM and System Requirements

CVM currently supports the platforms shown in Table 1, with our primary environments being ‘rs6k’ and ‘bip_mpi’ (linux 2.0.29, myrinet, bip 0.41). We are currently compiling CVM using gcc version 2.7.2 and gmake 3.74 on all platforms.

The main CVM directory has a source subdirectory (src), and applications subdirectory (apps), and a library build subdirectory for each supported architecture (alpha, rs6k, rs6k_mpi, etc.). Underneath the apps directory

<table>
<thead>
<tr>
<th>Processor</th>
<th>OS/Communications Library</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power-2 (SP-2)</td>
<td>AIX 4.?/UDP</td>
<td>rs6k</td>
</tr>
<tr>
<td>Power-2 (SP-2)</td>
<td>AIX 4.?/MPI</td>
<td>rs6k_mpi</td>
</tr>
<tr>
<td>Alpha</td>
<td>Digital Unix V4.0/UDP</td>
<td>alpha</td>
</tr>
<tr>
<td>Sparc</td>
<td>SunOS 5.5/UDP</td>
<td>solaris</td>
</tr>
<tr>
<td>Pentium</td>
<td>Linux 2.0.32/UDP</td>
<td>linux</td>
</tr>
<tr>
<td>Pentium</td>
<td>Linux 2.0.29/BIP-MPI [2]</td>
<td>linux_mpi</td>
</tr>
<tr>
<td>Pentium</td>
<td>FreeBSD/UDP</td>
<td>freebsd</td>
</tr>
</tbody>
</table>

Table 1: Current Platforms

is each of the applications supplied with the distribution. Application subdirectories contain a similar structure, allowing versions to be built for each architecture.
3.2.1 FreeBSD Kernel Mod?

In order to run pre-3.0 FreeBSD kernels (which don't exist yet), you must make a single change to the kernel source and recompile your kernel. The following is the change that needs to be made to the 2.2.7 kernel:

```
/usr/src/sys/i386/i386/trap.c

    if (i == 0)
        goto out;

    !                       ucode = T_PAGEFLT;
    break;

to

    if (i == 0)
        goto out;

    !                       ucode = (code << 5) + BUS_PAGE_FAULT;
    break;
```

3.3 Running CVM

3.3.1 Configuration

CVM needs two files to transparently spawn worker processes on other hosts:

- .cvmrc—CVM looks for a file called .cvmrc first in the same directory as the binary, and then in the user’s home directory. .cvmrc contains a list of hosts to run on, one per line. The host that CVM is started on must be the first host in this file. Sometimes pathnames are not uniform across machine boundaries. A pathname specifying the directory where the binary can be found can be specified after a machine name on any of the lines in .cvmrc. Alternatively, the -H flag can be used to specify hosts on command lines. There is currently no mechanism to specify pathnames via arguments. In situations where multiple network connections between two machines exist, the desired connection can usually be specified by using the correct variant of the name; e.g., ‘spanky01’ might name an Ethernet interface to spanky01, ‘spanky01h’ might name a connection to spanky01 over the SP-2’s high-performance switch, and ‘spanky01-a’ might name the ATM connection to spanky01.

- .netrc—CVM uses rsh to spawn worker processes on other hosts. In order to avoid being queried for a password for each worker process, you must set up a .netrc file on the master node (see man pages for detailed information), specifying userids and passwords for all nodes that might host worker processes. Fully-qualified names (e.g., herby.cs.umd.edu as opposed to herby) should be used in all cases.

3.3.2 Command-line Options

The CVM command line can specify arguments to the application, to the main CVM infrastructure, and directly to the protocol that has been chosen. Argument sets are separated by double dashes (“--”) on the command line. For example: ‘water -i sam8 -- -n2 -- -d4’ would pass ‘-i sam8’ to the application, ‘-n2’ to CVM proper, and ‘-d4’ to the chosen app (multi-writer LRC in this case).

Supported CVM options are the following:

- Use the runon() command on DEC Alpha multiprocessors when spawning worker threads.
- Turns on debugging output. See Section 3.3.3 for details. The CVM_DEBUG variable must be defined in global.h in order for this to do much. Undefine CVM_DEBUG to improve performance.
- Creates output files. Automatically invoked for -d and -T. Contains all output to stderr after initialization.
- Interprets ‘mach’ as the name of a machine on which to spawn a remote thread. Multiple -H commands can be used to specify all the machines for the run. The first -H must name the machine on which the application is initially run, subsequent arguments name other machines. For instance, to run water on machines spanky01 through spanky08, one can specify a command line of the form:

```
water -- -n8 -Hspanky01 -Hspanky02 ...
```

or, taking advantage of shell interpolation:

```
water -- -n8 -Hspanky0{1,2,3,4,5,6,7,8}
```
In both cases, the application must be started from `spanky01`.

- \(-i \ <num>\) Resets statistics after the \(num\)th iteration. This is only relevant if `cvm_timestep()` is called by your application each iteration. By default, this value is ‘1’. You can set it to zero if you desire. **YOU SHOULD USE THIS FOR TIMING, SEE SECTION 6.**

- \(-m \ <num>\) Specifies the initial ownership assignments of pages. The default (0) causes all pages to be assigned to processor 0. \(-m1\) causes the assignment to be round-robin, whereas \(-m2\) causes the shared space of an \(n\)-processor execution to be divided into \(n\) contiguous chunks.

- \(-n \ <num>\) Specifies number of nodes to be used by the system. A total of ‘\(n\)’ machines must be specified either by the `.cvmrc` file or through repeated \(-H\) arguments.

- \(-p \ <num>\) Pre-allocates \(num\) pages. Since pages are dynamically allocated, this is only useful in special circumstances, for example, in those applications that allocate large amounts of data after initialization is complete.

- \(-P \ <num>\) Sets the default page size, which must be a multiple of two of the system page size. The default page size is 8k bytes

- \(-s\) Turns on statistics gathering.

- \(-S\) Turns on timing statistics.

- \(-t \ <num>\) Uses per-node multi-threading to hide communication latency. Generates \(num\) threads on each node (processor). See the file `dcs97.ps.gz` in the docs directory for details. Default is one thread per processor.

- \(-T\) Uses the `/tmp` directory for debugging output.

- \(-X \ <num>\) Chooses a protocol. Supplying either 0, 1, or 2 as the argument to \(-X\) chooses one of the three protocols provided with CVM. ‘0’ is the default. The protocols are the following:

  0) **Lazy multi-writer (LMW)**—Lazy Release Consistency (LRC), allowing multiple writers to simultaneously access the same page without communication. “Diffing” is used to summarize modifications and to resolve multiple concurrent modifications to the same page. Advantages include low numbers of messages, low bandwidth, good toleration of false sharing, and good overall performance. Disadvantages are complexity, “diffs” (inelegant), LRC programming model (synchronization must be correct).

  1) **Lazy single-writer (LSW)**—Uses LRC, but differs in that only a single writer is allowed to modify any given page at a time. Since there is no problem with merging multiple concurrent diffs, they are not used. Advantages over LMW are simplicity, and a slight performance edge in cases of no false sharing. No diffs. Disadvantages are poor performance in the face of false sharing (the ping-pong effect), and higher bandwidth requirements (since small modifications can no longer be summarized in a diff rather than sending the entire page). The ping-pong effect is mitigated to some extent by allowing each processor to have access to any new page for a *delta* of time before it can be invalidated or stolen away.

  2) **Sequentially consistent single-writer (SEQ)**—Uses sequential consistency rather than LRC, and only a single writer. All the disadvantages of LSW apply and are usually more pronounced. Additionally, there are extra messages for invalidations. Advantage is the programming model, no dependence on synchronization. SEQ also uses a *delta* interval in order to somewhat minimize the effects of false sharing.

  3) **Home-based multi-writer (home)**—Uses a home-based protocol multi-writer LRC protocol modeled after that described in [3].

Command-line options after a second instance of “```--```” are passed to the chosen protocol (`lmw` by default). The following are options for the protocols supplied with CVM:

- \(-d \ <num>\) (`lmw`) multiplies the space allocated for diffs by \(num\). The effect is to decrease the frequency of garbage collections.

- \(-i \ <num>\) (`lmw`) multiplies the space allocated for interval structures by \(num\). As above.

- \(-m \ <num>\) (`home`) migrate page ownership after barrier \(num\). Can only be done once.

There are other options, but most refer to untried, untested, experimental code and you are advised not to use them.

**3.3.3 Debugging CVM Applications**

There are two basic avenues to debugging application on CVM or CVM itself:

**Output files**—Turning on either the \(-d\), \(-E\), or \(-T\) options causes each thread to redirect stderr to an output file in the same directory as the application. With the \(-d\) option (and `CVM_DEBUG` defined in `global.h`), a large
(easily 100 megs!) amount of debugging output is spewed to per-thread files. The files are named `out.x.y`, where `x` is the process id of the thread associated with the file and `y` is the total number of threads (or processors, if no additional threads are requested). Most of this output is likely to be unintelligible without study of CVM’s source code. However, in the event of a crash with `-d` turned on, even a very cursory examination of the output files should give you a good idea of where the processes were when the crash occurred. One special case is if one of the files ends with a ‘`segv nonsense`’ comment. This usually refers to a bad pointer de-reference, often caused by incorrect programs.

**Debuggers**—Both `gdb` and `dbx` can be used with CVM. The usual approach is to start the application, log in to the machine, on which the thread of interest is running, and then stop the thread by using the ‘`attach`’ command from with the debugger. Judicious insertions of `sleep()` calls into CVM’s code can ensure that the offending process does not die before you have a chance to attach it. Core dumps do not work with CVM.

### 3.3.4 Cleaning Up After CVM

Worker processes created by CVM on other machines will all usually die if any one process dies. However, if a glitch occurs during startup (i.e. before the ‘`Initialization Complete`’ message appears), processes must be manually killed. The distribution includes a `kill_orphans` perl script (located in `bin` subdirectory) that attempts to automate this process. Invoking the script from an `apps/<app>/<arch>` directory allows the script to extract the application name and pass it to invocations of `lookfor` perl script on the other machines specified in the `.cvmrc` file. You may have to fiddle with these scripts to get them to work in your environment. However, the main `INSTALL` script (will update the perl path interactively) will update the perl path in the file.

### 4. Programming with CVM

#### 4.1 Shared Memory Semantics and Synchronization

CVM supports the abstraction of a shared space visible to multiple threads, each of which runs on a different machine. These threads communicate via `shared-memory semantics`; e.g., memory allocated through CVM’s `cvm_alloc()` call is visible to all threads in the system, regardless of location.

CVM supports several different protocols, each of which implements a slightly different memory model. The default protocol supports a multi-writer version of Lazy Release Consistency (LRC) [4]. The memory models supported by LRC protocols differ slightly from the “standard” shared-memory semantics in that two processes are only guaranteed to see the same view or version of a given shared object if they have synchronized with respect to each other, even if only indirectly. Such `must` be accomplished through synchronization provided by the CVM system, i.e., `cvm_lock()` or `cvm_barrier()`.

#### 4.2 Example Program

The following is a complete, runnable CVM program that fills an array in parallel, with the number of parallel threads (each on a different machine) controlled by a command-line argument.

```c
#include <cvm.h>

#define DIM 4096
int *arr;
int sz = DIM;

void worker()
{
    int i, start = (DIM/cvm_num_procs) * PID;
    int end = start + DIM/cvm_num_procs);

    if (end >= DIM) end = DIM-1;
    for (i = start; i < end; i++) {
        arr[I] = PID;
    }
}

main(int argc, char **argv)
{
    int c;
```
while ((c = getopt(argc, argv, "s:")) != -1) {
    switch (c) {
    case 's':
        sz = atoi(optarg);
        break;
    }
}
cvm_startup(argc, argv);
arr = (int *)cvm_alloc(sz);
cvm_create_procs(worker);
worker();
cvm_finish();
}

The program could be run with an array size of 1k on four processors via the following command line:
a.out -s 1024 -- -n4

4.3CVM’s API

The following summarizes CVM’s application-programmer interface:

- **cvm_startup()**—reads all arguments after the double dash (which must be present), and initializes CVM.
- **cvm_alloc()**—allocates a chunk of shared memory and stores the address of the memory into the variable arr. At this point, the program is still single-threaded, so all threads will eventually see this value.
- **cvm_create_procs(worker)**—creates n-1 (in this case three) worker processes on machines specified by the .cvmrc file, and starts a single thread on each machine in the worker() function.
  1. Each thread at this points inherits an exact copy of the master thread’s global, heap, and stack data. This implies that each thread, for example, will have it’s own copy of the arr variable, and all will point to the same address in shared space.
  2. After this point, local program globals (such as arr), heap data, and stack data will diverge from machine to machine.
  3. cvm_alloc() calls are not allowed after the cvm_create_procs() call completes. Think of this as the end of your initialization phase. Indeed, CVM’s automatic timing starts at this point and continues until cvm_finish() is called.
- The worker() routine is called directly by the master process in order to have the master perform a portion of the work.
- When worker() completes at the worker threads, cvm_finish() is implicitly called.

4.3.1 Initialization

- **cvm_startup(int argc, char **argv)**—is called after your application processes its own arguments. The intention is to support programs that are invoked as follows:
  
  \[
  \text{program <your options> -- <CVM options> -- <protocol options>}
  \]
  
  Hence, your application should always include a getopt loop prior to calling cvm_startup().
- **cvm_alloc(int sz)**—allocates shared memory. This is the sole means of allocating shared memory, hence all shared data in CVM programs is necessarily dynamically allocated (this is not always true, look at suif_if.c and fort_if.c for examples of sharing statically-allocated data). All calls to cvm_alloc() must complete prior to the cvm_create_procs() call.
- **cvm_create_procs(func_ptr worker)**—creates threads on remote machines. If n threads are specified on the command line, n-1 remote threads are created at this stage. The machines are determined either through the .cvmrc startup file, or on the command line via the -H option. All remote threads are started in the function specified by the argument. Timing starts at the end of this routine.

4.3.2 Synchronization

- **cvm_lock(int id)**—acquires a global lock on the lock specified by ‘id’. The current maximum number of locks is 5000. This can be changed in $CVM/include/cvm.h.
- **cvm_unlock(int id)**—releases a global lock.
• **cvm_barrier(int id)**—performs a global barrier. Barriers prevent any process from proceeding until all processes have arrived. The ‘id’ parameter is currently ignored.

### 4.3.33.3 Cleanup

• **cvm_finish()**—is only called by the “master” process. Causes the master to wait until all worker process has completed. It then prints elapsed time and shuts CVM down.

• **cvm_exit(char *, ...)**—is called in case of error for a quick exit.

### 4.4 Fortran

CVM has crude support for fortran programs on AIX, OSF on Alphas, and Solaris on Sparc processors. The primary problem with supporting Fortran is that Fortran variables are allocated statically. Memory protection operations do not work on this part of the data segment for several architectures. Hence, the standard approach to supporting Fortran is to remap Fortran shared variables to a different address, on which protection operations can work. However, this is definitely a dicey operation. Nonetheless, CVM comes with a Fortran version of gauss that compiles and runs on AIX and Solaris (Sparc).

**Interface**

Shared Fortran variables must all be declared in a single common block, called ‘cvm_shared_common’. A 16k buffer variable should be added to the end of the common block, and the first variable and this buffer should be passed to ‘cvm_startup()’. Fortran versions of ‘cvm_barrier()’, ‘cvm_lock()’, ‘cvm_unlock()’, ‘cvm_finish()’, and ‘cvm_exit()’ are supplied. Additionally, the Fortran variables ‘cvm_num_procs’ and ‘cvm_proc_id’ are also defined.

*Per-thread multi-threading does not currently work w/ Fortran.*

### 5. Extending CVM

CVM was written with an eye to extension. Essentially, adding a new protocol to CVM consists of:

1) creating the new protocol by deriving new objects from Protocol and Page,
2) modifying the CVM library makefile (Make.common) to include the new sources, and
3) modifying startup.c to create the new protocol and add it to protObjs (search for ‘lsw’ and duplicate for your protocol).
4) modifying msg.h to include any new message and data types.
5) modifying comm.c to include a string identifying each new message and data type.

In the future, each page will potentially run a different protocol. However, currently all pages must use the same protocol. The following is a sketch of the way protocol and communication objects interact. The true documentation of CVM, of course, is the code itself.

#### 5.1 Communication

All protocol communication is accomplished through Msg objects. The following is a condensed definition of Msg:

```c
class Msg {
    void send(int type, int to);
    Msg *wait();
    void forward(int to);
    void reply(int reliable = FALSE);
    int add(int type, char *buf, int buf_sz, int copy=TRUE);
    char *retrieve_type(int type);
};
```

A request is sent to another processor by:

1) Creating a new Msg object.
2) Adding data via add.
3) Calling send.
4) Waiting for a reply via wait, which returns another Msg object containing the reply message.
5) Data is retrieved from the Msg via retrieve_type, and then ...
6) … both the original message and the reply must be deleted (manual garbage collection).
The reply method is used to reply to a `Msg`. If the `reliable` parameter is true, the `Msg` must not be deleted, the system retains the reply in order to automatically handle duplicated message. If `reliable` is `FALSE`, the protocol code must delete the `Msg` itself.

5.2 Protocol Extension

The following is a condensed version of the `Protocol` class:

```c
class Protocol {
    int msg_handler(Msg *msg); // each protocol has a chance at each Msg
    void add_to_lock_grant(Msg *msg, int from);
    void read_from_lock_grant(Msg *msg, int from);
    void add_to_lock_request(Msg *msg, int from);
    void read_from_lock_request(Msg *msg, int from);
};
```

Deriving a new protocol from the above class, overloading those methods that are applicable creates new protocols. Most important are the `msg_handler()` and synchronization methods. An incoming message is presented to each protocol in turn until one claims it by returning `TRUE`. The synchronization methods are called at the appropriate place by the synchronization classes, allowing each protocol to append information to outgoing messages and to retrieve information from incoming messages.

The synchronization methods are needed for protocols based on synchronization, such as release consistency or lazy release consistency, and clearly are not needed for sequential consistency. For the latter, only the `msg_handler()` method is necessary, together with an appropriate `Page` derivation:

```c
class Page {
    char *addr;
    int is_readable();
    int is_writable();
    void make_readable();
    void make_writable();
    void make_unreadable();
    void fault(int writing);
};
```

Only the last method needs to be overridden. The `fault()` method will usually interact with protocol routines to support the consistency model.

6. Benchmarking CVM

CVM was designed and is being distributed to the research community in order to facilitate protocol experimentation. It is not necessarily designed for ease of use or fast startups. More to the point, we only care about CVM’s steady-state performance, so we have eschewed distributed initialization in favor of a simpler, but slower, approach. Therefore:

**PLEASE DO NOT REPORT CVM TIMES THAT INCLUDE STARTUP!**

Ignoring this plea will not add anything to the sum of information known about DSM, it will just obscure the real tradeoffs between the protocols that you are looking at. The way to avoid this is the ‘-i’ option and the `cvm_timestep()` function (actually, it’s a macro from ‘cvm.h’). Most shared-memory applications are iterative, and have an explicit timestep loop. Put a call to `cvm_timestep()` at the beginning of the loop, and then run the app w/ a ‘-i2’ command-line option in order to have CVM zero it’s timer at the beginning of iteration 2.

Second, if at all possible:

**IMPLEMENT YOUR PROTOCOL ON TOP OF CVM!**

This serves two purposes. First, it levels the playing field. By having both your protocol and CVM’s protocols built on top of the same communication structure protocol-specific effects can more easily be isolated. Second, you can then send your protocol to me and I can redistribute it with future CVM releases. This will allow us to make
progress towards the holy grail, or a freely available DSM that can be used to do meaningful and repeatable experiments.

7. Future Plans

Future versions of CVM will include the following:

1) Adaptive load-balancing. Lightweight threads will allow thread migration across nodes. CVM will have an open architecture that allows new adaptation algorithms to be experimented with.
2) Multiple protocols in the same run. Protocols will be able to be specified on a per-page or object basis.
3) Adaptive protocol switching.

8. References