CVM: The Coherent Virtual Machine

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

CVM Version 0.1
November 21, 1996

1. Introduction

This tech 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 distribute-memory parallel machines such as the IBM SP-2.

CVM supports multiple protocols and consistency models. Like commercially available systems such as TreadMarks [keleherdwarkadas94], 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 class's methods need to be defined in the derived class. 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.

The rest of this report proceeds as follows. Section 2 describes how to install CVM, Section 3 describes how to run and debug applications on CVM. Section 4 describes how to extend CVM by defining new protocols. Finally, Section 6 describes future extensions that we plan to add to the core system.

2. Installation

2.1 Getting CVM

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

2.2 Compiling CVM

CVM currently supports four distinct platforms, Sparcs running SunOS 4.1, Sparcs, running SunOS 5.5 (Solaris 4.0), the IBM SP-2 running AIX 4.1, and DEC Alpha workstations running Digital Unix 3.2D-1. We are currently compiling CVM using gcc version 2.7.2 and gmake 3.74 on all platforms.

The first release of CVM uses UDP/IP communication only. Future versions will support MPI as well.

Figure X shows CVM's general heirarchy. The main CVM directory has a source subdirectory, and apps subdirectory, and library build subdirectories for each supported architecture. Underneath the apps directory are each of the application supplied with the distribution. Each application subdirectory contains a similar structure, allowing versions to be build for each architecture.

In general, to build a CVM application, you first need to build the appropriate library, and then the application. To build 'Water' on Digital Unix, for example:

- cd cvm/alpha
- gmake
- cd ../apps/water/alpha
- gmake

3. Running Applications on CVM

3.1 Configuration

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

- **.cvmrc -** CVM looks for a file in your home directory called `.cvmrc`. `.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. Alternatively, the -H flag can be used to specify hosts on command lines. In situations where multiple network connections between two machines exists, 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 `rexec` 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, 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.

3.2 Command-Line Options

The *cvm_startup()* routine parses all options after the first double dash (--) on the command line. The following are supported options:

-a — Uses the *runon()* command on DEC Alpha multiprocessors when spawning worker threads.

-A <num> — Multiplies the default page size by 'num'

-d — Turn on debugging output. See Section 3.3 for details. The CVM_DEBUG variable must be defined in global.h in order for this to do much. Undef CVM_DEBUG to improve performance.

-E — Create output files. Automatically invoked for -d, -F, -T.

-F — Collect "history" information. This is basically a history of events, such as how long each thread waited at barriers for the last 100 barriers. This is printed to the out.0.x file.

-H <mach> — Interpret '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`, I 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`.

-n <nodes> — Specify number of threads in system. A total of 'n' machines must be specified either by the .cvmrc file or through repeated -H arguments.

-s — Turn on statistics gathering.

-S — Turn on timing statistics. These can adversely affect performance, but usually not by much.

-T — Use the /tmp/ directory for debugging output.

-X <ind> — Choose 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:

1) *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).
2) *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 concur-

rent 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 be allowing each processor to have access to any new page for a *delta* of time before it can be invalidated or stolen away.

3) *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.

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

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, -F, or -T options causes each thread to create 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 meg) 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.

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.

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. This can usually be semi-automated via awk and shell scripts.

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) [Keleher, 1992 #315]. 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.

```
#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) {
                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

4.3 CVM'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 points, 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 initilization 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.

Each thread is assigned a process ID, accessed through the *PID* macro. The total number of parallel threads, specified via the '-n' cvm option, is accessed via the *cvm_num_procs* integer variable.

Threads in this application do not share data, and hence do not need to synchronize. However, you applications presumably will.

The complete source to three applications, *water*, *barnes*, and *sor* are included in this distribution.

4.3.1 Initialization

- *cvm_startup(int argc, char **argv)* - This routine is called after your app process its own arguments. The intention is to support programs that are invoked as follows:

```
program <your options> -- <CVM options>.
```
Hence, your application should always include a *getopts* loop prior to calling *cvm_startup()*.

- *cvm_alloc(int sz)* - allocate 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.

4.3.3 Cleanup

- *cvm_finish()* - called only by the master process. Causes the master to wait until all worker process have completed. It then prints elapsed time and shuts CVM down.
- *cvm_exit(char *, …)* - called in case of error for a quick exit.

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 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, right now 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:

```
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
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 Msg and the reply must be deleted.

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:

```
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);
};
```

New protocols are created by deriving a new protocol from the above class, overloading those methods that are applicable. Most important are the ‘msg_handler’ and synchronization routines. An incoming Msg 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:

```
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. Future Plans

Future versions of CVM will include the following:

1) Per-node multi-threading support. This support can be used for latency toleration as well as providing a programming model that is independent of the underlying architecture.
2) 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.
3) Multiple protocols in the same run. Protocols will be able to be specified on a per-page or object basis.
4) Adaptive protocol switching.
5) High-level object support of consistency protocols. See the *Sparcs* paper at http://www.cs.umd.edu/~keleher.