User's Guide of IO-SIG Software Suite

Scalable Computing Software (SCS) Laboratory

Illinois Institute of Technology

May, 2010

Contents

0 Introduction

0.1 Who could use this document

This document is a user's guide for the IO-SIG software suite. The target readers are users that need to use the software suite to improve the I/O performance of their parallel computing systems to accelerate the executions of applications.

0.2 A introduction to IO-SIG software suite

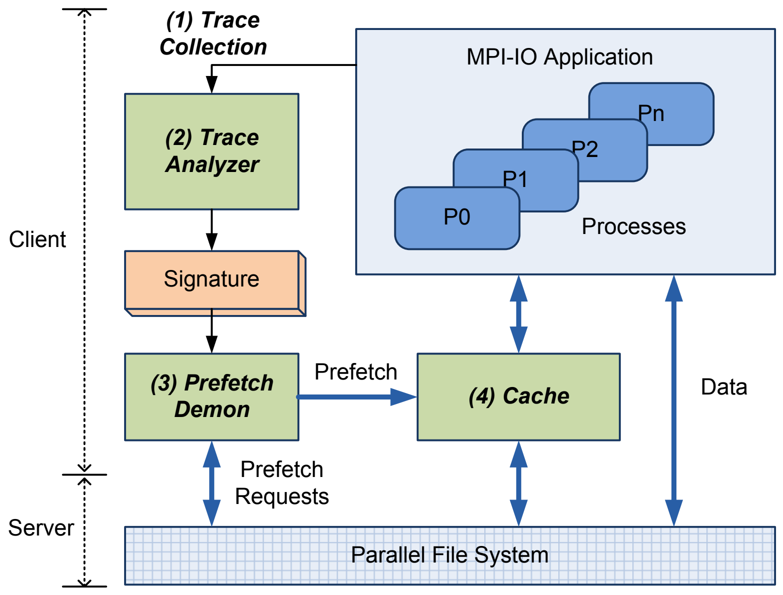

Figure 1: I/O Signature Sub-system overview.

IO-SIG is a suite of software that work together to enable data prefetching using MPI file caching and I/O Signatures in specific parallel I/O systems.

Figure 1 shows the prototype of a parallel computing system. In this system, multiple processes $P0 \sim Pn$ access the parallel file system for data, and in the IO-SIG subsystem the prefetch demon prefetches data from PFS to the client-side cache using the I/O Signature. With the help of IO-

SIG system, processes are able to access data from the local cache with lower cost instead from the parallel file system if the requested data is already prefetched.

The IO-SIG subsystem consists four components (the numbering is corresponding to Figure 1):

1. **Trace Collection**

   A tracing library is developed to collect information of an application's MPI-IO calls into trace files. The usage of this part is introduced in Section 1.

2. **Trace Analyzer/Signature Detection**

   Trace analyzer analyzes the trace files to generate a representation of an I/O access pattern, called *I/O Signature*. The usage of this part is introduced in Section 2.

3. **Prefetch Demon**

   A prefetch demon makes in-time data prefetching using both the I/O signature and the information of run-time I/O accesses. The demon runs alongside the main computing process during I/O operations to predict data requirements of the main thread and to bring the data into the client-side cache.

4. **Client-side cache**

   The client-side is used to store the prefetched data.

0.3 Dependencies

The following software packages are currently required by IO-SIG software suite:

MPICH2-1.1.1p1 The trace collection library currently only work with MPICH2-1.1.1p1. Later it will be updated to work with the latest MPICH2 release.

Python 2.6 [**or later**] The trace analyzer is currently written in Python.

protobuf 2.3 The trace analyzer uses protobuf to store the detected signatures.

1 Trace Collection

This section shows how to build the tracing library. All the following examples assume the user name is `me` and the home directory is `/home/me` where we do all the works.

1.1 Preparing MPICH2

Before compiling the tracing library, we should first compile MPICH2. Although the instructions on compiling MPICH2 can be found in its official installer's guide, we still repeat this compiling procedure here, not only for user's convenience, but also because it is a little different from the one in the official guide.

1.1.1 Download and Untar the Packages

Download MPICH2 from the official website, currently we have to use the 1.1.1p1 version because the tracing library can only be built based on this version. Later it will catch up with the latest MPICH2 release.

```
$ cd /home/me/iosig
$ wget http://www.mcs.anl.gov/research/projects/mpich2/downloads/\
tarballs/1.1.1p1/mpich2-1.1.1p1.tar.gz
$ tar zxvf mpich2-1.1.1p1.tar.gz
$ ls
mpich2-1.1.1p1  mpich2-1.1.1p1.tar.gz
```

1.1.2 Configure MPICH2

Choose an installation directory (the default is `/usr/local/bin`):

```
mkdir /home/me/mpich2-install
```

Configure the source code:

```
$ cd /home/me/iosig/mpich2-1.1.1p1
$ ./configure --prefix=/home/me/mpich2-install
```

The official installer's guide suggests to use a separated building directory to keep the source code clean, and then do both the configuration and compiling in that separated directory. Here we must do the "`./configure`" in the original source path. Otherwise, the tracing library won't be built successfully.

After configuring the source code, tracing library can be compiled successfully following the instruction in Section 1.2. However to use the library, we still need the MPICH2 environment, so build and install it using:

```
$ make
$ make install
```

1.2 Building the Tracing Library

Now it's ready to compile the tracing library. Download the tarball and untar it:

```
$ cd /home/me/iosig
$ wget http://www.cs.iit.edu/~scs/iosig/trace_collect.tar.gz
$ tar zxvf trace_collect.tar.gz
$ ls
mpich2-1.1.1p1  mpich2-1.1.1p1.tar.gz  trace_collect  trace_collect.tar.gz
$ cd trace_collect
$ ls Makefile
Makefile
```

Edit the first two lines in the file `Makefile` that can be found alongside with the source code, declare the variable `MPIPATH` as the path of the built source of MPICH2 (in this guide it's `/home/me/iosig/mpich2-1.1.1p1`) and declare the variable `INSTALL_DIR` as the `lib` directory under the MPICH2 installation path (in this guide it's `/home/me/mpich2-install/lib/`).

Then it's ready to build the tracing library using:

```
$ make
```

After it's done, we can find the library file "`libpushio.a`" in the directory "`trace_collect`". To install this library into the standard MPICH2 library path, just do:

```
$ make install
```

1.3 Using the Tracing Library

To use the tracing library "`libpushio.a`" we build in Section 1.2, only one thing to do is compile the MPI program and link the library to get the executable file. For example, if the usual command to compile "`test_mpiio.c`" is:

```
$ mpicc -o test_mpiio test_mpiio.c
```

Then the new command:

```
$ mpicc -o test_mpiio test_mpiio.c -lpushio
```

will get the executable file with MPIIO tracing enabled. While running this executable file using the usual `mpirun` or `mpiexec` command, traces of all MPIIO operations will be collected and saved into trace files placed in the same directory with the executable file. For each single MPI process, one trace file will be generated.

2 IO Signature Detection

After trace file are generated, they can be analyzed to detect any I/O patterns among them. The detected patterns are stored as an I/O signatures. An I/O signature contains information regarding the strides between successive I/O accesses (spatial pattern), how many times the pattern is repeated, its temporal pattern, the size of requested data, etc.

2.1 I/O Patterns

The current released software is able to detect the following I/O signatures:

- Contiguous
- Non-contiguous: Fixed Strided, 2D Strided, Negative Strided
- Combination of above patterns

2.2 I/O Signatures Notation

The I/O signature can be given in two forms; the first describing the sequence of I/O accesses in a pattern and the second identifying I/O patterns. We call the description of a sequence of I/O accesses in a pattern a *Trace Signature*, and the abstraction of a pattern a *Pattern Signature.* Using the five dimensions mentioned above, *Trace Signature* takes the form as follows:

{I/O operation, initial position, dimension, ([{offset pattern}, {request size pattern}, {pattern of number of repetitions}], [...]), # of repetitions}

Take the fixed strided *Trace Signature* in Section 2.3.1 as an example:

```
{MPI_READ, 0, 1, ([{0, 1, (1048576, 1)}, 1048576, 1]), 250}
```

From this signature, we can know, the application does a sequence of 250 MPI_READ's. The first MPI_READ start at the offset 0, and the subsequence offset increase by 1048576 each time, and the size is consistantly 1048576. For more information on signature notations, please see the publication [1] listed in Section 5.1.

Since a *Trace Signature* contains more information than a *Pattern Signature*, in the current release, the program only give the *Trace Signature*.

2.3 Using Trace Analyzer

2.3.1 Download Trace Analyzer

Download the source code from the website and unpack it:

```
$ cd /home/me/iosig/
$ wget http://www.cs.iit.edu/~scs/iosig/trace_analyzer.tar.gz
$ tar zxvf trace_analyzer.tar.gz
$ cd trace_analyzer
```

2.3.2 Run the Trace Analyzer

The trace analyzer scan the spcified trace file and detect the signature from it. After that, the signature list is stored into a protobuf file.

To run the trace analyzer, be sure that protobuf is already installed in the system. Here is the instruction on how to install the latest protobuf.

```
$ wget http://protobuf.googlecode.com/files/protobuf-2.3.0.tar.gz
$ tar zxvf protobuf-2.3.0.tar.gz
$ cd protobuf-2.3.0
$ ./configure
$ make
$ make check
$ sudo make install
```

The above instruction uses the default installation options, and in this case `protobuf` is installed into `/usr/local/lib`, which is not in the environmental varible “`$LD_LIBRARY_PATH`”. So before using protobuf, you need to do the following:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib/
```

After that just simply compile the protobuf.

```
$ cd trace_analyzer
$ make
protoc --cpp_out=. --java_out=. signatureList.proto
Writing shortcut script detect_iosig...
```

Then the program can be directly run using the generated script. The following command prints the help page of the program:

```
$ ./detect_iosig
I/O Signature Detection Program
Detect the I/O Signatures in a I/O trace file
Usage: python sig.py -f filename [options][source]
   or: ./detect_iosig [options][source]
Options:
  -h, --help                show this help
  -d                        show debugging information while parsing
  -f ..., --filename=...    the filename of the trace file
  -b ..., --blksz=...       size of block for block based detection
  -r ..., --range=...       # of entries will be processed
  -m ..., --formatFile=...  the properties file for the trace format information
```

Under the source code directory "`trace_analyzer`" there is a folder named as "`trace_samples`" which contains several simple trace files for user to test and get familiar with the program. For example, to run a test on a trace file with 1-D fixed stride pattern, just use the command:

```
$ ./detect_iosig -m ./standard.properties -f ./trace_samples/trace_26502.out -r 250
The following signatures are detected:
{MPI_READ, 0, 1, ([{0, 1, (1048576, 1)}, 1048576, 1]), 250}
```

Section 2.3.3 explains the meanings of each options in the Python command line.

2.3.3 Understanding the Command line Options

-h, –help This option will cause all the other options invalid and the program will print the help information and then terminate. The same effect can be achieved by running the program with none options.

-d With this option, the program will print out all the debug information. Usually debug information are only useful for developers on this program.

-f ..., –filename=... This option cannot be eliminated for a meaningful execution, it specify the input trace file through its file name.

-b ..., –blksz=... While giving this option, the program use block-based trace analysis during detecting signatures, and the size of the data block is specified by this option's parameter. Section 2.3.5 explains what the block-based analysis is.

-r ..., –range=... A trace file usually contains many I/O access entries, this options give a max range on the number of entries that are analyzed in one single run. While the range parameter is larger than the total number of entries in the trace file, it means the whole trace file is analyzed.

-m ..., –formatFile=... If a trace file is generated by our tracing library, then its format is familiar to the trace analyzer program. To handle all the other traces in different formats all over the world, we need to use this option along with the filename of a format properties file. This file tells the trace analyzer how the information is organized in the trace files. Section 2.3.4 describes how to use this properties file.

2.3.4 Using Format Properties File

`.properties` is a file extension for pure text files mainly used to store the configurable parameters of an application. Each parameter is stored as a pair of strings, one storing the name of the parameter (called the key), and the other storing the value. Each line in a .properties file normally stores a single property. Several formats are possible for each line, including $key = value$, $key : value$, and $keyvalue$.

Let's take a look at the "`standard.properties`":

```
$ cat ./standard.properties
skip_lines: 5
mpi_rank: 1
file_id: 2
size: 4
pos: 3
op: 6
```

We can see there are six properties in this file. The means of these properties are:

skip_lines Usually, a trace file might include several instructive non-trace information in the first several lines. The value of this property can tell the program how many lines should be skipped before analyzing the real trace entries.

mpi_rank, file_id, size, pos, op While analyze each line of the trace file, the program first replaces all the non-regular (not in "_", A-Z, a-z and 0-9) characters with white space, after that, each line becomes one list of words. Each of these five properties specifies the location (word index) of the field (key) in each trace entries. For example, with `mpi_rank:2`, the program knows the word with index 2 in each line is the "rank of MPI process" for this single MPIIO operations.

2.3.5 Original v.s. Block-based Trace Analysis

Block-based Trace Analysis is based on the following two facts:

1. While the I/O pattern of applications are too complex, it's hard to detect the patterns.

2. The cache manages data in a block-based way, also the disk reads data in a block-based way.

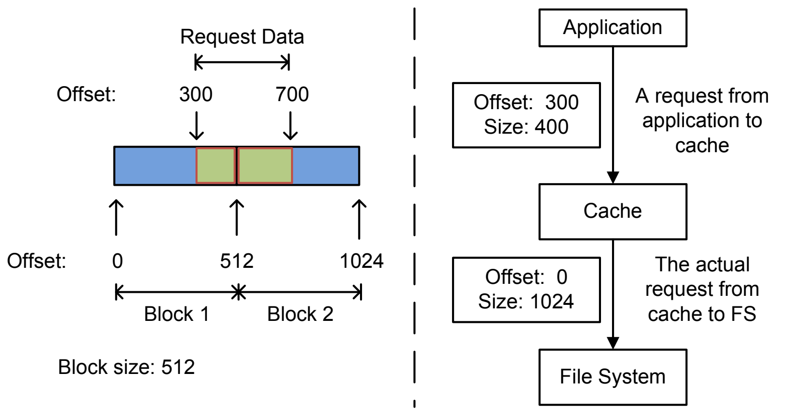

Figure 2: Block-based data organization in cache and disk I/O.

Figure 2 shows the block-based data organization in cache and disk I/O. While application issues a request with offset 300 and size 400, this request actually contains the last half portion of block 0 and the first half portion of block 1, in this situation, the disk will read both two blocks (block 0 and 1) and the cache will store both of them too.

Based on this particular fact, the program provide one feature that can analyze the traces in a block-based way. In other words, to do this, the program first transfer the original offset and size values to the block-based ones and then conduct the usual trace analysis. Experimental results shows this approach make it easier to detect access patterns for applications with complex accesses.

3 Prefetching Daemon

A prefetch demon does the following tasks.

1. It runs alongside the main computing process during I/O operations to predict data requirements of the main thread and to bring the data into the client-side cache.

2. It monitors the cache status to see whether the action of main thread is in accordance with the I/O signature. If not, the prefetching daemon automatically adjust the signature according the run-time traces.

This part of software is still under preparation for release and will be ready soon.

4 Client Side Cache

This part of software is still under preparation for release and will be ready soon.

5 Resources

5.1 Related Publication

1. S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O Prefetching Using MPI File Caching and I/O Signatures," in Proceedings of the 2008 ACM/IEEE conference on Supercomputing. Piscataway, NJ, USA: IEEE Press, 2008, pp. 112.

5.2 Online Resources

The homepage of the project of I/O Signature System Software is http://www.cs.iit.edu/ scs/iosig/. This document and the source code of these software are available on this homepage.

To view the source code online, please use FishEye the online source code explorer also available at the project homepage.

5.3 Reporting Errors

If you ever found any errors or bugs with the IO-SIG software suite, please report them to Dr. Xian-He Sun <sun@iit.edu> or Yanlong Yin <yyin2@iit.edu>. We appreciate it very much. The TRAC (available at http://trac.edgewall.org/) site of this project is under construction.