Tools for Characterizing Distributed Shared Memory Applications

Gheith A. Abandah
Computer Systems Laboratory
HP Laboratories
abandah@hpl.hp.com

November 20, 1996

Keywords: Distributed Shared Memory, Application Analysis, Trace Collection, Trace Analysis, Simulation.

Abstract

In order to support the design of future *distributed shared memory* (DSM) systems, we have developed a set of tools for characterizing DSM applications. These tools enable collecting and analyzing data, instruction, and I/O stream traces of DSM applications. They give characterization for several aspects of DSM applications including communication and sharing patterns. They also enable evaluating the performance of these applications on various DSM design alternatives. This report describes the implementation of these tools, their usage, and the format of their output files.

Contents

1 Introduction

This report describes a set of tools that were developed for characterizing *distributed shared memory* (DSM) applications. It describes the implementation of these tools, their usage, and the format of their output files. The development of these tools was initiated to support developing new DSM multiprocessor systems, Protic et al. survey DSM concepts and systems in [1]. The tools can also aid in application performance tuning.

There are three logical steps in conducting application characterization; problem definition, performance collection, and performance analysis. In *problem definition*, we need to specify the applications that we are interested in characterizing and the performance aspects that need to be characterized. We are interested in characterizing a wide range of applications including commercial, e.g. decision support and transaction processing; scientific, e.g. fluid computation and finite element analysis; and internet, e.g. WWW server and client applications. For all these applications, we are interested in characterizing the three traffic streams; data, instruction, and I/O.

Depending on the application, the characterization of one traffic stream might be more important than others. For example, data stream characterization has the highest priority in characterizing scientific applications because the data stream is responsible for most of the the system traffic, while characterizing the I/O stream might be as important as the data stream for some commercial applications.

There are several classes of techniques for *performance collection*, each with its own advantages and limitations. Hardware monitors, e.g. VAX microcode monitor [2], Convex Cxpa [3], and the IBM POWER performance monitor [4] provide low-level information using event counters, but require special hardware support. Code instrumentation, e.g. RYO [5] and Pixie [6], enables collecting various performance data and traces, but requires source code availability. Direct-execution simulation, e.g. Proteus [7] and SimOS [8], similar to code instrumentation, enables collecting various performance data and perturbs the execution as well, but does not require source code availability. SimOS collects traces for activities within the operating system in addition to the user-space activities. Because of the wide range of applications that we are interested in characterizing, we are collecting traces using code instrumentation and other methods. Trace collection using code instrumentation is described in this report.

Most of the available parallel *performance analysis* tools are developed for analyzing message-passing applications, e.g. Pablo [9], Medea [10], and Paradyn [11]. The tools presented in this report address the shortage of tools for analyzing shared-memory applications. They assist in the performance analysis of shared-memory applications and enable characterizing wide range of performance aspects.

We have developed two tools for analyzing shared-memory applications. The first tool is intended to generate abstractions that characterize the inherent application characteristics. The second tool is intended to predict the application performance on specific hardware

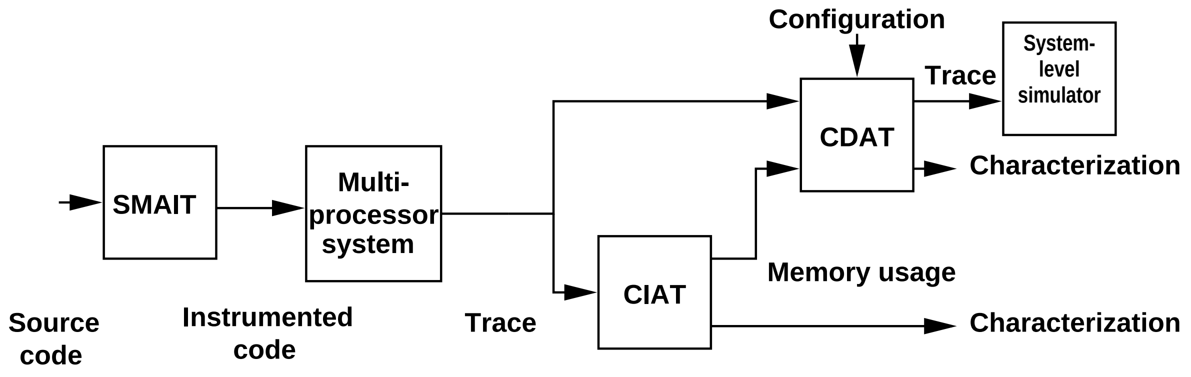

Figure 1: Application analysis methodology outline.

configurations. Given alternative hardware configurations, the second tool predicts the traffic flow volume and characteristics. It also generates traffic traces that are used to drive detailed system-level simulators.

Figure 1 shows an outline of our methodology in characterizing DSM applications. The figure shows that a shared-memory multiprocessor is used to collect traces by executing instrumented application codes. Nevertheless, other methods can be used for trace collection.

The *Shared-Memory Application Instrumentation Tool* (SMAIT) is used to instrument applications and is presented in more detail in Section 2. Instead of generating trace files, SMAIT can also pipe the traces to the analysis tools for *on-the-fly analysis.* On-the-fly analysis enables analyzing longer execution periods and solves the problem of having huge trace files.

The two analysis tools are the *Configuration Independent Analysis Tool* (CIAT) and the *Configuration Dependent Analysis Tool* (CDAT). CIAT generates characterization for data access instructions, branching, synchronization, communication patterns, and data sharing. Additionally, CIAT generates a *memory usage file* that specifies the usage statistics of all accessed memory pages. CDAT reads a *configuration file* that specifies a proposed system configuration and simulates the execution of the traces on this configuration. CDAT outputs characterization and trace files for the traffic that would occur when the application is run on the simulated configuration. CDAT can use the memory usage file to map memory pages to the simulated memory banks.

Sections 3, 4, and 5 give more details about CIAT and CDAT. Section 6 describes another tool used for analyzing the variation of application behavior over time. Section 7 concludes the paper by stating the limitations and advantages of these tools and methodology. Appendix A specifies the formant of SMAIT trace files. Appendix B presents CIAT and CDAT command line options. Appendix C specifies CDAT trace format.

2 Trace Collection

SMAIT is based on RYO [5], a tool developed by Zucker and Karp for instrumenting PA–RISC [12] instruction sequences. RYO is a set of awk scripts that enable replacing individual machine instructions with calls to user written subroutines.

SMAIT is designed to enable collecting traces of multi-threaded shared-memory parallel applications. SMAIT has two parts, a perl script program for instrumenting PA–RISC assembly language files, and a run-time library that is linked with the instrumented program. The perl script program replaces some PA–RISC instructions with calls to the run-time library subroutines. During program execution, the run-time library generates one trace file per thread. SMAIT provides three levels of instrumentation:

- At *Level 1*, SMAIT instruments the procedure call instructions for some I/O, thread management, and synchronization subroutines. In this level, SMAIT enables collecting traces for the I/O stream and timing information. The run-time library generates one *call record* whenever an instrumented call instruction is executed. A call record contains the call type, time before and after the procedure call, and four argument fields. Appendix A describes the format of SMAIT traces.
- At *Level 2*, SMAIT additionally instruments all load and store instructions to collect traces for the data stream. The run-time library generates one *memory-access record* whenever an instrumented memory-access instruction is executed. A memory-access record contains the type of the instruction and the virtual address.
- At *Level 3*, SMAIT additionally instruments all branch instructions and the first instruction of every procedure in order to collect traces for the instruction stream. The run-time library generates one *branch record* for each taken branch. A branch record contains the virtual address of the branch instruction plus 4 and the virtual address of the branch target. Four bytes are added to the branch instruction address to account for the instruction in the delay slot after the branch instruction which, in PA–RISC architecture, is fetched and conditionally executed. The run-time library also generates one record whenever the first instruction of an instrumented procedure is executed. Such record contains the virtual address of the procedure start.

When Level 1 is used, the instrumented code is lightly perturbed and runs near the original uninstrumented speed. This level is mainly used for collecting timing information. When Level 2 or 3 is used, the instrumented code is heavily perturbed and runs about 70–100 times slower than the original uninstrumented code.

The following steps outline an example how to use SMAIT to instrument a simple matrix multiplication program and to collect traces on a Convex SPP–1600 multiprocessor [13].

1. Convert the high-level source files to assembly language files.

2. Instrument the assembly files using `smait.perl` and link with the two run-time library object files; `smait.o` and `smait_init.o`. Figure 2 shows an example of a make file that converts the Fortran program, `mat_mul.f`, into an assembly file, `mat_mul.s`, using the Convex Fortran Compiler `fc`, instruments it using `smait.perl`, compiles it into the object file `mat_mul.o`, and links the instrumented object file with the run-time library object files.

```
smait.o:  smait.c smait.h
          cc -O -c smait.c
smait_init.o:  smait_init.c smait.h
          cc -O -c smait_init.c
mat_mul.o:  mat_mul.f
          fc -O3 -S mat_mul.f
          sed 's,/dev/null,devnull,' < mat_mul.s > mat_mul.ss
          smait.perl -level=3 <mat_mul.ss >mat_mul.s
          fc -c mat_mul.s
mat_mul:  mat_mul.o smait.o smait_init.o
          fc -o mat_mul mat_mul.o smait.o smait_init.o
```

Figure 2: Example makefile showing SMAIT instrumentation.

3. Select the trace type by setting the environment variable `TRACE_FILE_TYPE`. SMAIT supports five options:
   - "normal", generates one uncompressed trace file per thread,
   - "compressed", generates compressed trace files,
   - "suppressed", does not generate any trace files,
   - "to_ciat", invokes CIAT and pipes the traces to it for on-the-fly analysis, and
   - "to_cdat", invokes CDAT and pipes the traces to it for on-the-fly simulation.
4. Select the trace file name by setting the environment variable `TRACE_FILE_NAME`.
5. Run the instrumented code to collect traces. Figure 3 shows how to set the environment variables and to run the instrumented code. The SPP–1600 command `mpa` is used to run `mat_mul` using 4 threads. The example shows that SMAIT run-time library reports the names of the 4 trace files, each file name specifies the number of threads and the thread ID.

Table 1 summarizes SMAIT overheads in collecting traces of a 256×256 matrix multiplication under the three instrumentation levels compared to the performance of the uninstrumented

```
% setenv TRACE_FILE_TYPE normal
% setenv TRACE_FILE_NAME trace_level_3
% mpa -max 4 mat_mul
Trace output stored in:
                    file trace_level_3.4.0 for thread 0
                    file trace_level_3.4.1 for thread 1
                    file trace_level_3.4.2 for thread 2
                    file trace_level_3.4.3 for thread 3
```

Figure 3: Example of collecting traces for 4 threads.

Table 1: SMAIT instrumentation overheads.

Instrumentation level	`mat_mul.o` size	Execution time	Trace size	Compressed size
No instrumentation	14 KB	1.2 sec	0	0
Level 1	16 KB	1.3 sec	2.2 KB	0.8 KB
Level 2	73 KB	83 sec	116 MB	19.8 MB
Level 3	87 KB	90 sec	127 MB	20 MB

program. Column 2 shows the size of the object file, `mat_mul.o`. Level 3 generates code that is about 6 times larger than the uninstrumented code. Column 3 shows the execution time. Level 3 results in a factor of 75 slowdown. Columns 4 and 5 shows the size of the four trace files for the normal and compressed types respectively. Compression reduces the trace size by a factor of 6.

3 Trace Analysis

This section is an introduction to CIAT and CDAT, it describes some trace analysis techniques that are common in both tools.

Both tools accept two sets of trace files. Each set is made up of p trace files coming from p threads of execution. The first optional set is called *call traces* and contains traces of the I/O and synchronization calls. The second set is required and called *detailed traces* and contains the call traces, data stream traces, and instruction stream traces—if available. Although all the records in the call traces are also in the detailed traces, the call traces are used because they usually come from a less perturbed execution and their time stamps are closer to the uninstrumented execution. The call traces are collected using SMAIT instrumentation level 1, and the detailed traces are collected using SMAIT instrumentation level 2 or 3.

Both tools assume that the traces come from an application with one or more *execution* *phases* where each phase has its own properties. Currently, the supported phases are *serial* and *parallel* phases and *user-defined* phases. In a serial phase, thread 0 is the only active thread and other threads, for a multi-thread execution, are idle. In a parallel phase, multiple threads are active. The tools recognize the two phases from the trace records of the thread spawn and thread join calls that activate and deactivate threads between serial and parallel phases. The user-defined phases are recognized when the tools encounter special *marker* records (see Appendix A). The marker records can be generated by instrumenting the high-level source code.

Both tools perform analysis per phase and report characterization statistics at the end of each phase. Both tools also report the aggregate characterization statistics of all phases at the end. Each line in the report file has a distinctive *tag*. The tag format is "RrTtLll:", where r is the phase number, t is the thread number, and ll is a distinctive data type number. The aggregate statistics lines have r='x'. When t='x', the line refers to data that is applicable to all the active threads.

Both tools manage one *pseudo clock* per thread to interleave processing multiple traces. The clocks are initialized to zero at the start of the first phase. A thread clock is incremented by one whenever an instruction is processed for that thread. The clocks are synchronized at the end of each phase and after emerging from a synchronization barrier to the value of the largest clock.

Within a serial phase, both tools process the trace records of thread 0 until reaching a thread spawn call. Within a parallel phase, both tools process trace records from all the available threads until reaching a thread join call. A trace record is processed for the thread with the smallest clock. In case all threads have same clock value, trace records are processed in a round robin method.

The tools also support traces that do not contain information about thread management and synchronization. They support analyzing T trace files of T different processes by interleaving these traces on p processors. This is accomplished by making use of the time stamps of the call records that imply process synchronization like `semop`. The tools logically break the traces into *slices*. A slice is a sequence of memory-access and branch records that are surrounded by two call records. The end time of the head call record is used as the start time of the slice, and the start time of the tail call record is used as the end time of the slice. Call records that do not imply process synchronization like disk read are part of the slice body. The tools employ a *Scheduler* that sorts the slices into a list according to their start time and schedule them to the available processors.

The Scheduler tries to keep the processors busy. It schedule the slice at the list head whenever there is free processor, the slice is from a trace file that does not have any other scheduled slice, and its start time is larger than the end time of the scheduled slices. This greedy scheduling sometime results in process migration among processors. Process migration can be prohibited by selecting the *affinity* option, see Appendix B. The Scheduler supports time-slicing to ensure that long slices do not hog processors more than a user-defined slice length.

In order to improve processor utilization, the Scheduler also swaps out a process when it is blocked waiting for disk read.

The tools reconstruct an approximation of the instruction stream using the branch records. This is accomplished by maintaining one program counter per thread. Whenever this program counter is incremented by one instruction word, the instruction at the old address is fetched.

The program counter is initialized whenever a procedure start record is encountered and is incremented by one instruction word for every memory-access record. For branch records, the program counter is first compared against the source address of the branch record. The program counter is usually less than the source address because instructions other than memory-access instructions are not represented in the trace. Hence, the program counter is iteratively incremented by one instruction word at a time until it matches the source address. Secondly, the target address is copied to the program counter.

The above algorithm generates an approximation of the instruction stream because the memory-access instructions in a block between a procedure start or a target address and the next taken branch are crammed to the block start. Better instruction stream reconstruction can be achieved by referring to the executable file to find the relative addresses of the memory-access instructions.

4 Configuration Independent Analysis

CIAT is intended to capture inherent application characteristics that do not change from one configuration to another. Configuration here refers to the configuration of the multiprocessor that runs the application. The configuration of a multiprocessor includes the way processors are clustered in a hierarchy, the interconnection topology, the coherence protocols, and the cache configurations.

CIAT uses many counters for counting various events and uses a memory structure to keep track of data and code accesses. The phase statistics are reported in a report file at the end of each phase and the aggregate statistics are reported in the report file at the end of the last phase. Additionally, at the end of the last phase, CIAT scans the memory structure and reports memory usage statistics in the report file and generates a memory usage file that summarizes the memory usage of each touched page. Subsection 4.5 presents more information about the memory usage file.

The following example shows how to use CIAT to analyze the call traces `trace_level_1.4` and the detailed traces `trace_level_3.4` of a 4-thread run.

```
% ciat trace_level_1.4 trace_level_3.4
```

For this example, the output report file name is `trace_level_3.4.ciat_report` and the output memory usage file name is `trace_level_3.4.memory`.

Alternatively, CIAT can be invoked by SMAIT for on-the-fly analysis as follows:

```
% setenv TRACE_FILE_TYPE to_ciat
% setenv TRACE_FILE_NAME trace_level_3
% setenv CALL_FILE_NAME trace_level_1
% mpa -max 4 mat_mul
```

The environment variable `TRACE_FILE_NAME` is used for naming the report and memory usage files. The environment variable `CALL_FILE_NAME` is optional and is used to specify the names of the call trace files. On-the-fly analysis is slower than generating traces. When the example outlined in Section 2 is run with on-the-fly analysis, the execution time rises from 90 sec to 280 sec (a factor of 230 total slowdown).

CIAT provides the option of generating a trace file of the key communication events. This trace file facilitates conducting time distribution analysis of the communication patterns in an application using the tool described in Section 6. The format of this file is outlined in Subsection 4.4.

The following subsections present the application characteristics that are reported by CIAT. The characteristics are classified into five groups; memory-access instructions, instruction stream, procedure calls, communication patterns, and memory usage.

4.1 Memory-access Instructions

These statistics are found by counting occurrences of the different types of load and store instructions. CIAT reports the following in number and percentage:

1. Memory-access instructions for both types, load and store.
2. Load instructions classified by the size of the accessed data and the register type involved; byte, halfword, word, float, and double-float.
3. Store instructions classified by the size of the accessed data and the register type involved; zero-byte, byte, halfword, three-byte, word, float, and double-float.
4. Frequency of load and store strings according to the string length. A load string of length l is a sequence of l consecutive loads from one thread not interrupted by stores. Similarly, A store string of length l is a sequence of l consecutive stores from one thread not interrupted by loads.

4.2 Instruction Stream

CIAT reports the following information about the instruction stream:

1. Number of fetched instructions.
2. Number of taken branches.
3. Total number of procedure calls.
4. Number of calls to instrumented procedures.

4.3 Procedure Calls

CIAT reports summaries about the procedure calls that have call records in the trace file. For each call type, CIAT reports number of occurrences, time spent in the call, and an accumulation of the amount field. For the I/O calls, this summary specifies:

1. Number of calls of each type.
2. Time spent in each call type.
3. Amount of data in bytes transfered by each call type.

For the synchronization and thread management subroutine calls, the summary specifies:

1. Number of times the subroutine was called. This number gives an indication of the parallelism grain size.
2. Time spent in each subroutine. This time is useful to estimate the synchronization overhead and the load balance conditions.

4.4 Communication Patterns

In a shared-memory application, processors communicate by accessing shared memory. CIAT reports the amount of communication and the communication patterns for every execution phase. Specifically, CIAT reports the following:

1. Number of *read-after-write* accesses (RAW): A RAW access occurs when one or more processors load a memory location that was stored by a processor. This pattern does not include the case when only one processor loads a memory location that was stored by this same processor. This is a common communication pattern, it occurs in a producer-consumer situation where one processor produces data and one or more processors consumes it.

2. *Sharing degree* for RAW. This is a vector **S**, where **S**$[k]$ is the number of times that a memory location was read by k processors after being written.

3. Number of *write-after-read* accesses (WAR): A WAR access occurs when a processor stores a memory location that was loaded by one or more processors. This pattern does not include the case where a processor stores to a memory location that was only loaded by itself. This is also a common pattern, it occurs when a processor writes data that was read by other processors.

4. *Invalidation degree* for WAR. This is a vector **I**, where **I**$[k]$ is the number of times that a memory location was written after being previously read by k processors.

5. Number of *write-after-write* accesses (WAW): A WAW access occurs when a processor stores to a memory location that was stored by another processor. This is a less common pattern, it occurs when multiple processors write without reading, or when processors take turns on a memory location where in each turn a processor writes and reads.

6. Number of *read-after-read accesses* (RAR): A RAR access occurs when a processor loads a memory location that was loaded by another processor and the first visible access to this location is a load. This is an uncommon pattern, it occurs in bad programs that read uninitialized data. Nevertheless, CIAT often encounters this pattern when the data is initialized in untraced routines.

When instructed by the command line option, CIAT dumps a trace for the above communication events. The trace file has a header that specifies its format and contains a record for each of the four communication events. Each record contains fields for the clock, event type, processor, and degree. The degree is 1 for WAW, equals the invalidation degree for WAR, and undefied for RAW and RAR.

4.5 Memory Usage

CIAT reports the following summary of the memory usage for the instruction and data streams at the end of the report file:

1. Total number of touched pages.

2. Number of touched pages that are shared. A *shared page* is a page touched by more than one thread.

3. Number of touched pages that are range shared. A *range-shared page* is a page touched by more than one thread, or is in an address range reserved for shared pages, see Appendix B.

4. Total number of touched memory locations in bytes.

5. Number of touched data memory locations in bytes.
6. Number of touched code memory locations in bytes.
7. Number of shared memory locations in bytes. A shared memory location is a location accessed by more than one thread.
8. Number of touched memory locations in bytes that are page shared. A page-shared memory location is a location in a shared page.
9. Number of touched memory locations in bytes that are range shared. A range-shared memory location is a location in a range-shared page.
10. Total number of memory accesses.
11. Number of data accesses.
12. Number of data accesses to shared memory locations.
13. Number of data accesses to shared pages.
14. Number of data accesses to range-shared pages.
15. Number of code accesses.
16. Number of code accesses to shared memory locations.
17. Number of code accesses to shared pages.
18. Number of code accesses to range-shared pages.
19. Data Locality Index, which is calculated as the number of accessed data bytes divided by the number of touched data memory locations.
20. Code Locality Index, which is calculated as the number of fetched instructions divided by the number of touched instructions.

In addition to the above aggregate statistics, CIAT reports one record for each touched page in the memory usage file. The record format is specified in the file header. Each record contains the page number, number of touched bytes, number of touched code bytes, number of shared bytes, number of data accesses, number of code accesses, number of shared data accesses, number of shared code accesses, and the page owner.

The page owner is undefined and equals -1 for a shared page. For a private page, one that was touched by only one thread, the page owner equals the thread number.

5 Configuration Dependent Analysis

CDAT is intended to predict the traffic that would be generated by an application on a proposed DSM multiprocessor configuration. CDAT is a simple trace-driven simulator that has cache, memory, bus, and internode interconnection models. These models can be arranged in a hierarchical configuration as specified by the configuration file, see Subsection 5.1. CDAT generates abstractions and flow parameters that can be used to parameterize workload generators. CDAT uses many counters for counting various events. The phase statistics are reported in a report file at the end of every phase and the aggregate statistics are reported at the end of the last phase.

Additionally, CDAT generates two types of traces; *general* and *detailed.* The general trace contains the requests and returns generated by normal and I/O processors as seen at the interconnection networks. Processor requests are generated on cache misses and DMA activities and processor returns are generated on cache replacements. The detailed trace contains records for all transactions that are interchanged to satisfy processor requests and returns. Appendix C specifies these transactions and gives the format of these trace files.

CDAT modeling of DSM systems is flexible to enable experimenting with various DSM design options. For example, it enables evaluating various coherence protocol alternatives and implementation techniques. Supporting various coherence protocols is possible because the coherence protocol handling subroutines are modular and the user can specify the wanted protocol as one of the configuration file options. Supporting different hardware implementations is accomplished by using the general CDAT trace to feed system-level simulator. The general trace gives the system-level simulator the freedom of using various DSM hardware implementations.

Although CDAT does not keep track of time explicitly, it approximates time by tagging the generated transactions by the pseudo clock. The clock equals the instruction number that caused the transaction. CDAT assumes that each instruction takes one clock cycle.

One of the problems that faces CDAT is mapping virtual addresses to the distributed physical memory banks. CDAT has several policies for mapping memory pages. Some of these policies rely on the information gathered by CIAT about memory usage. CDAT reads the memory usage file and maps the used virtual pages into physical memory banks according to the policy specified in the configuration file. CDAT has the ability to allocate private memory in the local node, interleave shared memory across nodes, and replicate some shared memory in multiple nodes, e.g. for shared code pages.

The following example shows how to use CDAT to analyze the call traces `trace_level_1.4` and the detailed traces `trace_level_3.4` for a DSM multiprocessor specified by the configuration file `dsm.cfg`.

```
% cdat trace_level_1.4 trace_level_3.4 dsm.cfg
```

For this example, the output report file is `trace_level_3.4.cdat_report`. CDAT can also be invoked by SMAIT for on-the-fly simulation in a similar way as CIAT. Setting the environment variable `CONFIGURATION_FILE_NAME` specifies the configuration file name. When the example outlined in Section 2 is run with on-the-fly simulation, the execution time rises to 175 sec (a factor of 146 total slowdown).

CDAT reports number and percentage of the following flow parameters for the data and instruction streams of normal processors. Additionally, similar parameters are reported for the I/O stream of the I/O processors.

1. Cache hit.
2. Cache miss.
3. Cache misses that are satisfied locally.
4. Cache misses that are satisfied from a remote node.
5. Cache misses that are satisfied from a local memory bank.
6. Cache misses that are satisfied from a local cache.
7. Cache misses that are satisfied from a remote memory bank.
8. Cache misses that are satisfied from a remote cache.

CDAT reports statistics about the traffic transactions. Some of the transaction types are listed in Appendix C. CDAT reports three aggregate numbers for each phase:

1. Total number of transactions.
2. Number of processor requests.
3. Average latency, which is estimated as the number of critical-path transactions divided by the number of processor requests.

CDAT reports the number of occurrences of each transaction type. CDAT also reports a state table for the processor requests and returns. Each processor request or return type has one column. The various rows specify the number of occurrences of the different states of the requested or returned lines. The states are classified according to the line home node (local or remote), presence (local, remote, home), and cache status (idle, shared, exclusive, or dirty). The table also shows the average sharing degree of remotely shared lines.

CDAT also reports the number of active cycles, number of idle cycles, and utilization of each processor.

5.1 Configuration File

The configuration file specifies the following aspects:

- Number of nodes, valid values are 1 through 32 nodes.
- Number of processors per node, valid values are 1 through 128 processors and the total number of processors should not exceed 128 processors.
- Number of memory banks per node, valid values are 1 through 32 banks.
- Line size in bytes.
- Thread mapping to processors, specifies the node and processor each thread is mapped to.
- Cache coherence protocol. Currently, there are 8 variants of a directory based cache coherence protocol. These variants differ in the cache status of remote loaded lines, the processor cache action on replacing exclusive lines, and the support of direct processor to processor transactions. Table 2 summarizes these variants.
- Memory allocation policy. The currently supported policies are:
    1. Round Robin 1 (RR1), memory pages are interleaved in a round robin scheme across all the available nodes. The memory usage file is not used. When the command line option -HPUX is selected, pages that are in a private memory range are allocated locally.
    2. Round Robin 2 (RR2), this policy is similar to RR1 but code pages are replicated in all active nodes. The memory usage file is used to find the code pages.
    3. Oracle policy. This policy relies on the memory usage file so that code pages are replicated, private pages are allocated locally, and shared pages are interleaved in a round robin scheme.

Table 2: Cache coherence protocol options.

Option	Remote lines	Exclusive replacement	Direct transactions
1	Maybe loaded exclusive	Announced	Supported
2	Loaded shared	Announced	Not supported
3	Loaded shared	Not announced	Not supported
4	Maybe loaded exclusive	Not announced	Not supported
5	Maybe loaded exclusive	Not announced	Supported
6	Maybe loaded exclusive	Announced	Not supported
7	Loaded shared	Not announced	Supported
8	Loaded shared	Announced	Supported

4. First Touch (1Touch). In this policy, a page is allocated in the same node where it is referenced for the first time. The memory usage file is not used.

- General trace file generation; 0: not generated, 1: generated.
- Detailed trace file generation; 0: not generated, 1: generated.
- Processor data cache configuration, e.g. size, line size, and associativity.
- Processor instruction cache configuration.
- Interconnect cache configuration.

6 Time Distribution Analysis

Time distribution analysis is handled by a separate tool called Time Distribution Analysis Tool (TDAT). TDAT is used to analyze event traces generated by CIAT or CDAT. The event trace is an ASCII file that has one record per event. Each record starts with a clock field and may have other optional fields.

TDAT puts the events into bins according to the clock field. The bin width can be specified by the user. TDAT finds the total number of events, number of bins, average event rate, minimum and maximum rates, standard deviation, density function, and distribution function.

The following example shows how to use TDAT to analyze `event_trace` using 1000-cycle bins for 4 processors.

```
% tdat event_trace event.time 1000 4 > event.freq
```

The output file `event.time` contains the number of events in each bin. The example shows that the output data is directed to the file `event.freq`.

7 Conclusions

In this report, we have presented some of our tools for characterizing DSM applications and supporting the design of future DSM systems. These tools are continuously updated and improved to answer an increasingly expanding list of needed application characterization and design options. Future updates will be reflected in the release notes of these tools.

Some of the limitations of the methodology outlined in this report are that trace collection using SMAIT requires the availability of application sources. Additionally, the trace collection does not include the activities inside the operating system or shared libraries. While the collected traces are of the PA–RISC instruction set architecture (ISA), future DSM systems use different ISA. Thus a method is required to translate the performance of these traces to future systems.

On the other hand, some of the advantages of this methodology are that we collect parallel traces for applications that use the shared-memory programming paradigm. Such traces are of higher importance than other traces, e.g. serial traces, for developing DSM systems. Nevertheless, CIAT and CDAT can be used to analyze traces collected by other tools. Analyzing the performance of an application for runs of varying number of processors enables application characterization as a function of the number of processors. Moreover, time distribution analysis is also supported. CIAT and CDAT use simple models, thus they can be used to do fast analysis of long traces. They can also be used in on-the-fly mode to save the burden of generating huge trace files. Through CDAT, application analysis directly supports evaluating various design options for future systems.

Acknowledgment

I would like to thank Tom Rokicki for his assistance in implementing these tools, Rajiv Gupta and Josep Ferrandiz for their guidance, Lucy Cherkasova for her constructive discussions and comments, and Milon Mackey for providing valuable traces that enabled extending the tools.

8 References

[1] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: Concepts and systems," *IEEE Parallel and Distributed Technology*, pp. 63–79, Summer 1996.

[2] D. Clark and H. Levy, "Measurement and analysis of instruction use in the VAX–11/780," in *Proc. The 9th Annual Symposium on Computer Architecture*, pp. 9–17, 1982.

[3] CONVEX Computer Corporation, 3000 Waterview Parkway, P.O. Box 833851, Richardson, TX 75083–3851, *CXpa Reference Manual*, second ed., March 1993. Order No. DSW–253.

[4] E. H. Welbon, C. C. Cha-Nui, D. J. Shippy, and D. A. Hicks, "The POWER2 performance monitor," *IBM J. Research and Development*, vol. 38, pp. 545–554, Sept. 1994.

[5] D. F. Zucker and A. H. Karp, "RYO: a versatile instruction instrumentation tool for PA–RISC," Technical Report CSL–TR–95–658, Stanford University, Jan. 1995.

[6] M. D. Smith, "Tracing with Pixie." ftp document, Center for Integrated Systems, Stanford University, Apr. 1991.

[7] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl, "PROTEUS: a high-performance parallel-architecture simulator," Tech. Rep. MIT/LCS/TR-516, Massachusetts Institute of Technology, Sept. 1991.

[8] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer simulation: The SimOS approach," *IEEE Parallel and Distributed Technology*, Fall 1995.

[9] D. Reed *et al.*, "Scalable performance analysis: The Pablo performance analysis environment," in *Proc. IEEE Scalable Parallel Libraries Conf.*, 1993.

[10] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and T. Daniele, "Medea: A tool for workload characterization of parallel systems," *IEEE Parallel and Distributed Technology*, vol. 3, pp. 72–80, Winter 1995.

[11] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn parallel performance measurement tool," *Computer*, vol. 28, pp. 37–46, Nov. 1995.

[12] Hewlett-Packard, *PA-RISC 1.1 Architecture and Instruction Set*, third ed., Feb. 1994.

[13] T. Brewer, "A highly scalable system utilizing up to 128 PA-RISC processors," in *Digest of papers, COMPCON'95*, pp. 133–140, Mar. 1995.

A SMAIT Trace Format

SMAIT generates one binary trace file for each thread. A trace file is made up of a collection of *simple records*. Each simple record has 2 fields and each field is 4 Bytes long and contains an unsigned integer number.

There are three types of records that are supported by SMAIT; call, memory-access, and branch. The following is a description of the three record types.

A.1 Call Record

Call records are used for tracing I/O, synchronization, and thread management calls. Each call record is made up of 6 simple records with a total of 12 fields. The first field in each simple record is a concatenation of the number of the simple record within the call record, the call type, and a constant. The first hex digit (least significant) specifies the simple record number. The next two digits specify the call type according to Table 4. The most significant five hex digits are the constant digits 00001.

The start time field contains the time stamp in microseconds before calling the subroutine. The end time field contains the time stamp in microseconds after calling the subroutine.

In the spawn record, argument 1 is a positive integer that specifies an identifying number for the parallel region at the thread spawn and argument 2 specifies the number of spawned threads. For the barrier record, argument 1 is the address of the synchronization variable and argument 2 specifies the number of threads waiting on the barrier.

Table 3: Call record format.

Simple record number	Field 1	Field 2
1	00001xx0	Start time
2	00001xx1	End time
3	00001xx2	Argument 1
4	00001xx3	Argument 2
5	00001xx4	Argument 3
6	00001xx5	Argument 4

Table 4: Call types.

xx	Subroutine
01	Thread spawn, cps_spawn
02	Thread join, cps_join
03	Barrier synchronization
04	Begin ordered section, cps_begin_order
05	End ordered section, cps_end_order
06	Begin critical section, cps_begin_critical
07	End critical section, cps_end_critical
10	Marker, used in the high-level source code
11	File read
12	File write
13	Load and clear work, ldcw
14	Semaphore operation, semop
15	Semaphore control operation, semctl
16	Generic system call
17	Start trace record
18	End trace record

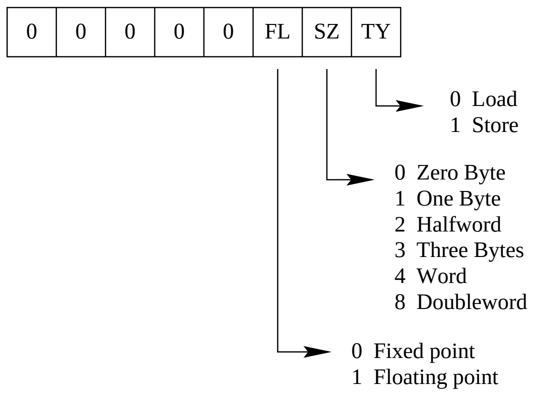

Figure 4: Type field format of memory-access records.

The marker record is generated by a special call inserted in the source code. The call template is `smait_marker(int *cmd, int *num)`, where `cmd` is a command that controls the generation of detailed traces and `num` is a user-specified distinguishing number. Command 0 has no effect, command 1 stops the thread detailed tracing, command 2 resumes thread detailed tracing, command 3 stops detailed tracing of all threads, and command 4 resumes detailed tracing of all threads.

For the file read and write records, argument 1 is the virtual address of the memory buffer, argument 2 is the number of bytes moved, and argument 3 is the file handle that specifies the accessed file. For the semop record, the four arguments specify the semaphore identifier, number, operation, and flags respectively. For the semctl record, the four arguments specify the semaphore identifier, number, command, and value respectively. Argument 1 in the generic record specifies the system call identifier.

Argument fields not defined above are not used.

A.2 Memory-access Record

Memory-access records are used for tracing memory-access instructions. Each memory-access record is made up of 1 simple record with a total of 2 fields. The first field specifies the memory-access type and the second field contains the virtual address. As shown in Figure 4, the first hex digit in the type field specifies the access type, the second hex digit specifies the size of data accessed, the third hex digit specifies the register type involved in the memory access, and the most significant five hex digits are the constant digits 00000.

For example, the type field 00000040 refers to an instruction that loads one word into a fixed-point register, and 00000181 refers to an instruction that stores a double-word from a floating-point register to memory.

A.3 Branch Record

Branch records are used for tracing procedure starts and taken branch instructions. Each branch record is made up of 1 simple record with a total of 2 fields. The most-significant hex digit of the first field specifies one of three record types.

- *Procedure start* record when the most-significant hex digit is 4. For example, the simple record 40000000 00004C00 is generated when starting to execute a procedure at address 4C00.
- *Taken branch* record when the most-significant hex digit is 8. The rest of the first field specifies the address of the instruction after the taken branch, and the second field specifies the target address. For example, the simple record 80004C18 00004D20 is generated for a taken branch instruction at address 4C14 to the target address 4D20.
- *Procedure call* record when the most-significant hex digit is C. The rest of the first field specifies the address of the instruction after the procedure call instruction, and the second field specifies the target procedure address. For example, the simple record C000542C 00005630 is generated for a procedure call instruction at address 5428 to the target procedure at address 5630.

B Command Line Options

This appendix specifies the command line options acceptable by CIAT and CDAT. Unless otherwise stated the following options apply to both tools. The simplest way to use CIAT and CDAT on trace files collected using SMAIT is

`ciat [calls_file.p] trace_file.p, and`

`cdat [calls_file.p] trace_file.p cfg_file,`

where `trace_file.p` is the start of the names of `p` detailed trace files, `calls_file.p` is the start of the names of `p` optional call trace files, and `cfg_file` is CDAT configuration file. In this case, CIAT generates a report file named `trace_file.p.ciat_report` and memory usage file named `trace_file.p.memory`. CDAT generates a report file named `trace_file.p.cdat_report`.

Alternatively, for traces collected by other tools, the following format can be used

`ciat [options] [-c call_list] -t trace_list, and`

`cdat [options] [-c call_list] -t trace_list cfg_file,` where `trace_list` is a file containing a list of detailed trace file names, `call_list` is a file containing a list of call trace file names, and *options* can be zero or more of the options shown below. In this case, `trace_list` is used in naming the output files.

- `-ciat_events`: this option is only valid with CIAT and it enables the generation of CIAT event trace.
- `-HPUX`: this option can be used when the trace is collected on a system running the HPUX operating system. In this case, the tools assume that the virtual addresses 0x40000000 through 0x7fffffff are private addresses.
- `-p`: this option implies that the thread spawn and join calls are invisible and the tools should start analysis with a parallel phase.
- `-i` *p*: this option instructs the tools to interleave the trace files listed in `trace_list` on *p* processors.
- `-a`: this option instructs the scheduler to maintain affinity of the processes, i.e. a process is always scheduled at the same processor. In the default mode the scheduler can move a process among the available processors to maximize utilization.
- `-rr_interleaving`: this option can be used with `-i` to use a simple round robin trace interleaving on the *p* processors. If this option is not selected, the tools do slice interleaving as explained in Section 3.
- `-k`: when this option is used, the tools keep long slices scheduled until they finish. The default is to swap out long slices.
- `-s` *v*: this option specifies the length *v* in instructions after which a long slice is scheduled off its processor. The default value is 20,000 instructions.
- `-handle_io`: this option instructs the tools to simulate the traffic implied by disk read and write calls. The default is not to handle I/O.
- `-buffered_io`: instructs the tools to simulate buffered disk I/O. The default is raw disk I/O.
- `-asynch_io`: to use asynchronous disk I/O, a process does not wait for disk I/O completion. The default is synchronous disk I/O where a process is blocked waiting for I/O completion.
- `-distributed_io`: when this option is selected, each processor performs its disk I/O through the local I/O processor. The default mode is centered disk I/O where all disk I/O is done through the I/O processor in node 0.
- `-dma_bw` *v*: this option specifies the disk DMA bandwidth in MB/sec. The default value is 20 MB/sec.

- `-disk_latency` v: this option specifies the average disk latency in msec. The default value is 10 msec.
- `-ipc` v: this option is used to specify the processor IPC number (instructions per cycle). The default value is 2.
- `-h`: instructs the tools to print the command line options.

C CDAT Trace Format

This appendix is intended to describe the format of the two types of CDAT traces; *general* and *detailed*. The general trace is composed of one trace file for each processor and each I/O processor. Normal node processors are numbered 0, 1, . . . , and I/O node processors are numbered -1, -2, ... Each record in the general trace is generated to report one cache miss or replacement or to report the change in the processor status between active and idle. The record format is

`clock addr code proc node bank hnode status sharing_list -1,`

where

- `clock`: is the pseudo clock.
- `addr`: is the cache line virtual address.
- `code`: is the record type code, request and return records have the same code numbers as defined below. The start active status record code is 91 and the start idle status record code is 92.
- `proc`: is the number of the requesting processor in its node.
- `node`: is the node number of the requesting processor.
- `bank`: is the number of the memory bank in the home node.
- `hnode`: is the home node number of the line.
- `status`: is the status of the line; 1: idle (not cached), 2: shared, 3: exclusive, and 4: dirty.
- `sharing_list`: is a vector that specifies for each processor whether it has a copy of this line; 0: does not have a copy, and 1: has a copy.
- "`-1`": indicates sharing list end.

For example, the record

```
11351 277248 2 0 3 1 2 4 0 0 1 0 0 0 0 0 -1
```

says that at clock 11351 processor 0 of node 3 has a load miss for the line 277248 which has home at bank 1 of node 2. This line is dirty in proc 2 of node 0 (assuming 4 processors per node).

CDAT detailed trace file contains all the sequences of transactions of all processors. Each transaction can be classified into one of the following six classes (the supported transactions and their codes are listed for each class, also shown is the format of each class):

- *Processor request*, processor to memory, response is required:
  - read_shar (01), on instruction fetch miss.
  - read_shar_or_priv (02), on load miss.
  - read_priv_own (03), on store miss.
  - req_inv (04), on store hit on a shared line.
  - read_current (05), used by the I/O processor to read the current copy of a line without joining its sharing list.
  - write_purge (06), used by the I/O processor to update a memory line, invalidating cached copies, and ignoring write backs.

  Record format: `clock addr code proc node bank hnode`

- *Memory response*, memory to processor:
  - data_shar (11), shared copy of a cache line.
  - data_priv (12), private copy of a cache line.
  - alloc_done (13), private ownership granted.
  - write_purge_done (14), write_purge completed.

  Record format: `clock addr code bank hnode proc node`

- *Processor return*, processor to memory, no response is needed:
  - write_back (21), for dirty lines.
  - update_data (22), as a response to snoop hit on a dirty line. This transaction is not reported in the general trace.
  - update_tag (23), relinquishing a clean private line (exclusive).

  Record format: `clock addr code proc node bank hnode`

- *Memory recall*, memory to processor, response is required:
  - recall_shar (31), recall transaction for read_priv.
  - recall_shar_or_priv (32), recall transaction for read_shar_or_priv.
  - recall_priv_own (33), recall transaction for read_priv_own.
  - inv (34), invalidate remote copies for req_inv.

  Record format: `clock addr code bank hnode proc node`

- *Cache_to_cache recall response*, processor to processor:
  - c2c_data_shar (41), shared copy of the line, need to forward data to home.
  - c2c_data_priv (42), private copy of the line, need to forward data to home.
  - c2c_data_priv_own (43), private copy of the line, no need to forward data.

  Record format: `clock addr code proc node proc node`

- *Recall response*, processor to memory:
  - recall_c2c_data (51), after c2c_data_shar or c2c_data_priv.
  - recall_c2c_done (52), after c2c_data_priv_own.
  - inv_done (53), response to inv.
  - recall_nack_shar (54), negative acknowledgment due to not dirty line.
  - recall_nack_inv (55), negative acknowledgment due to not available line.
  - recall_data (56), returning dirty line to memory.

  Record format: `clock addr code proc node bank hnode`

CDAT generates a sequence of transactions in response to a cache miss or replacement. Each sequence of transactions starts with either a processor request or a processor return transaction. A sequence that starts with a processor return transaction has a singular transactions. Transaction 22, update_data, can end the first three processor request sequences.