

pSigma: An Infrastructure for Parallel Application Performance Analysis using Symbolic Specifications

Simone Sbaraglia, Kattamuri Ekanadham, Saverio Crea
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA
{sbaragli,eknath,screa}@us.ibm.com

Seetharami Seelam
University of Texas at El Paso
El Paso, TX 79968, USA
seelam@cs.utep.edu

Abstract

In this paper we describe pSigma, which is an infrastructure for instrumenting parallel applications, enabling the users to probe into the execution of an application by intercepting its control-flow at selected points, the selection being made by lucid specifications using symbolic names of data structures and functions in the source program and performance-related information about the memory subsystem. This infrastructure assists application developers and system architects in examining performance events and relating them to the data and control components at the source program level. By providing details on how data is shared among parallel program components, it helps in exposing the performance effects induced by mutual interference or cooperation. Our event-based selection schema using symbolic names from the source domain results in a very user-friendly instrumentation mechanism.

1 Introduction

A variety of performance measurement, analysis, and visualization tools [3, 4, 5, 6] have been created to assist application programmers in tuning and optimizing scientific codes. However, there is still very limited support for fine-grained performance analysis of parallel programs running on real machines. Understanding and tuning memory system performance of scientific applications running on parallel machines is still a critical issue for programmers and architects.

Existing performance analysis and visualization tools tend to be *control-centric*, since they focus on the control structure of the program (e.g. loops and functions). However, users also need *data-centric* performance measurement infrastructures that can help them understand the precise memory references that are caus-

ing poor utilization of memory hierarchy, especially in a parallel environment.

In order to develop such performance tools, tool developers and researchers often rely on *instrumentation libraries* [2, 3, 11, 12] to intercept the execution of a program at selected events and insert *probe libraries* for measurement and analysis. Unfortunately, such instrumentation libraries are not “performance-oriented”, in that they do not provide the probe library with any information about the state of the machine, or the effect of the current operation on the system. For example, although they allow the probe library to be invoked upon the execution of a memory operation, they do not provide any information about the effect of the operation on the memory subsystem, such as whether it is a hit/miss in the cache, or whether it resulted in issuing any prefetches etc. Similarly, when the probe library is invoked as a result of a load operation, no symbolic information is provided as to which element and which data structure is being accessed.

Tool developers need flexible mechanisms to intercept the execution of a program at selected events based on information that relates performance to symbolic entities in the source program. This information must encompass execution events, such as accesses to array elements, as well as performance events such as cache misses exceeding a certain limit.

In this paper we present a new instrumentation infrastructure, which is an evolution of the **SIGMA** infrastructure for memory analysis, described in [1], to inject arbitrary user code into a parallel application binary. The user can describe the rules that should trigger the execution of the injected code using a specification that employs symbolic and performance related metrics. The infrastructure provides information at the granularity of individual load/store events for each thread of execution of an application and can be used for understanding sharing of data among parallel threads and the

resulting performance effects.

The remainder of this paper is organized as follows: In Section 2 we discuss the **pSigma** infrastructure for instrumentation of parallel applications. In Section 3 we describe the symbolic specification of events, in Section 4 we discuss the standardized event description interface to pass information to the event handler and in Section 5 we provide an illustrative example of use of **pSigma**. Finally, in Section 6, we describe one possible application of the infrastructure, to detect performance critical data structures in an OpenMP benchmark code. We present our conclusions in Section 7.

2 The pSigma Infrastructure

Figure 1 presents an overview of **pSigma** and the execution environment. On the top left corner of Figure 1, the three principal components of **pSigma** software are shown: a catalogue of *events*, a library of *event-handlers* and the module *psigmaInst*. An application user consults the event catalogue and composes a script describing a configuration for the memory subsystem, a list of events to be monitored, and the associated event-handlers, which can be picked from the **pSigma** library or supplied by the user. The *psigmaInst* module takes the script, the application binary and any user-supplied event-handlers and produces an instrumented binary that performs the desired probing.

Events can be of two types: Execution-specific events identify actions specified by the source program, such as the execution of a load, store or floating point instruction using a chosen data structure, or during the invocation of a certain function. Performance-specific events identify points during execution when certain criteria of performance metrics are satisfied, such as cache miss count exceeding a certain limit, an instruction or loop repeating a specified number of times.

The bottom of Figure 1 depicts the scenario when the instrumented binary is executed. The instrumented binary executes possibly multiple threads and whenever a designated event occurs, it generates an event-description packet. Memory related events are passed to the memory simulator, which simulates the cache structure for each processor. The packets carry the thread identification, so that corresponding caches are manipulated. After simulating the event in the cache structure, it generates event-descriptors that carry the results of cache accesses. All event-data packets are directed to the corresponding event-handlers that consume the data. All event-descriptors adhere to a standardized event-data format so that handlers can be developed using the uniform interface format. Event-data includes the identification of the data and control locations, in

terms of symbolic names used for the data structures and functions in the source program. For instance, it uniquely identifies the subject data element and array name in the program and the line number and function of the source program that is manipulating it during that event. Memory simulators can also optionally provide cache subsystem state, giving access results (hit/miss) at each level, information on evicted elements on cache fills and list of prefetched data elements.

An event handler can use the event information in arbitrary ways. Handlers can accumulate data and produce summaries at the end or enumerate the data and produce traces and profiles. The **pSigma** library provides a host of event handlers, that typically summarize event-data: they can provide compressed traces of execution, enumerations of selected event data, summaries of cache behavior, categorized by functions and data structures, etc.

In addition, **pSigma** system also provides a facility to specify memory subsystem parameters and data structure paddings that can be simulated. The handlers can also dynamically turn on/off the generation/handling of certain events based on the occurrence of other events.

3 Symbolic Specification of Events

To facilitate usability and for fine grained instrumentation of selected parts of a program **pSigma** provides an easy scripting language for expressing desired actions. The specification is a set of *directives*, each specifying an event and a corresponding action to be taken when that event occurs. The event can be one event or a logical combination of several events. An event is a predicate on the current operation (*eg.* operation is a load or it suffered an L1 miss), or a predicate on the accumulated state of the system (*eg.* accumulated L1 Miss count exceeds a constant). The library supplies a list of events and counters that are tracked in **pSigma**. An event can also be further qualified by a *context* which can specify that the event must be considered only when it relates to a named data structure and/or occurs within the execution of a specified function. An action can invoke a handler or print a counter.

```
directive ::= on compoundEvent do action
compoundEvent ::= event | compoundEvent and event
                | compoundEvent or event
event ::= currentEvent [ context ] | counterState relOp value
value ::= counterState | constant
currentEvent ::= load | store | L1Miss | L2Hit | ...
counterState ::= counterName [ context ]
counterName ::= cumL1Misses | cumL2Hits | ...
context ::= for structureName in functionName
action ::= call handlerName | print counterName
```

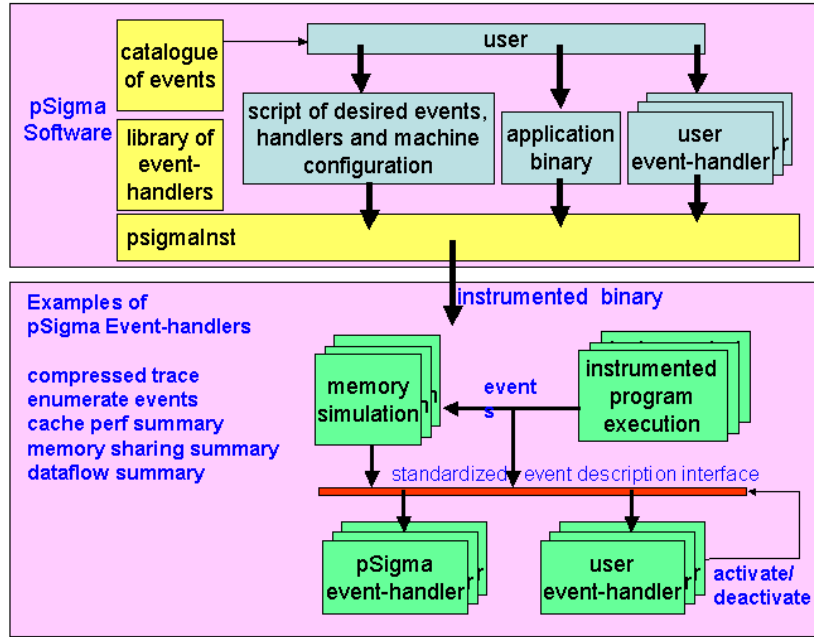


Figure 1. Structure of pSigma

Examples:

1. `on L1Miss for A in F do call myHandler`
2. `on L1Miss for A in F and`
`(cumL1Misses > 1000) for A in F`
`do call myHandler`
3. `pSigmaInst -p scriptFile myHandlers.a appbin`

In the first example, `myHandler` is invoked whenever an access to `A` in function `F` incurs a miss in L1. In the second example, the handler is called for each L1 miss on `A` in `F` after 1000 such misses have incurred. The third example illustrates how pSigma is invoked, supplying a script file, the library of user-supplied handlers and the application binary. The script file contains the event directives and the memory configuration of the system. The `pSigmaInst` command instruments the application binary as specified by the script and generates a new binary `app.inst` which includes all the necessary libraries. When this is executed, the handlers are invoked when the associated events occur.

Thus **pSigma** enables easy specification of very complex combination of conditions and provides sophisticated selection of events for which the user can obtain very detailed information on cache states and side-effects during execution. Using this infrastructure, users can quickly implement intelligent probes to get to the bottom of a performance problem.

4 Event Handler Interface

When the execution of an application is intercepted, **pSigma** provides the following information to the corresponding event handler. This interface is standardized so that all event handlers, including those to be supplied by the users, can be written for this interface. The event descriptor structure has three main components:

1. *InstructionData*: This includes the virtual address of the instruction that triggered the event, the opcode/type of the instruction (load, store, float etc.), the source line number and file to which this instruction belongs.
2. *MemoryData*: This indicates hit/miss at each level of cache and TLB that occurred when this instruction is executed, a list of evicted lines, prefetched lines that took place at each cache level during the execution of this instruction.
3. *CumulativeData*: This includes performance counters accumulated during program execution for each data structure and for each function.

5 An Illustrative Example

To illustrate the use of **pSigma**, we analyze here a simple Fortran sequential kernel, whose main loop is

```

parameter (nx=512, ny=512)
real a1(nx,ny), a2(nx,ny), a3(nx,ny)
do iter = 1, niters
  do id = 1, nd
    do jd = 1, nd
      do iy = 1, ny
        do ix = 1, nx
          ixy = ix + (iy-1)*nx
          a1(ix,iy) = a1(ix,iy)
&          + c11(ixy,id,jd)*b1(ixy,jd)
&          + c12(ixy,id,jd)*b2(ixy,jd)
&          + c13(ixy,id,jd)*b3(ixy,jd)
          a2(ix,iy) = a2(ix,iy)
&          + c21(ixy,id,jd)*b1(ixy,jd)
&          + c22(ixy,id,jd)*b2(ixy,jd)
&          + c23(ixy,id,jd)*b3(ixy,jd)
          a3(ix,iy) = a3(ix,iy)
&          + c31(ixy,id,jd)*b1(ixy,jd)
&          + c32(ixy,id,jd)*b2(ixy,jd)
&          + c33(ixy,id,jd)*b3(ixy,jd)
        end do
      end do
    end do
  end do
end do

```

Figure 2. Fortran Kernel

```

on load for a1 do call myHandler
on store for a1 do call myHandler
on load for a2 do call myHandler
on store for a2 do call myHandler
on load for a3 do call myHandler
on store for a3 do call myHandler
nCaches 2
TLBEntries 256
TLBAssoc 2-way
TLBRepl LRU
L1Size 64Kb
L1Linesize 128b
L1Assoc 128-way
L1Repl LRU
L2Size 4Mb
L2Linesize 128b
L2Assoc 2-way
L2Repl LRU

```

Figure 3. Event/Architecture Specification

shown in Figure 2. We instrumented the application to intercept all load and store operations for data structures *a1*, *a2* and *a3*. The memory system parameters used in this example are same as in the IBM Power3 system. The event description file is shown in Figure 3. On each load and store operation on the selected data structures *myHandler* is invoked. It accesses the *MemoryData* structure provided by the standardized event description interface and computes derived metrics such as hit and miss ratios of each cache level and for the TLB, as shown in Figure 4.

It can be observed that the three data structures incur a large number of TLB misses. A close analysis of the code reveals that this is due to the data structure layout, that is such that *a1*, *a2* and *a3* map to the same TLB blocks. Since the TLB, in this experiment, is only 2-

way set associative and the three data structures are accessed in an interleaved manner, this layout causes a lot of misses. Most of these misses could be avoided by restructuring the data structure layout. Figure 5 illustrates a restructuring of the program, in which the matrices *a1*, *a2* and *a3* are padded so that they don't map to the same TLB block. This new layout incurs significantly less TLB misses, as shown in Figure 6.

	L1	L2	TLB
DATA: a1			
Load Hit Ratio	96.87%	99.99%	7.42%
Store Hit Ratio	99.22%	16.67%	-
DATA: a2			
Load Hit Ratio	96.88%	50.00%	7.44%
Store Hit Ratio	99.22%	16.67%	-
DATA: a3			
Load Hit Ratio	96.88%	59.99%	0.14%
Store Hit Ratio	99.22%	16.67%	-

Figure 4. Memory Profile for a1,a2,a3

```

parameter (nx=512, ny=512)
real a1(nx+1,ny), a2(nx+1,ny), a3(nx+1,ny)
do iter = 1, niters
  ...

```

Figure 5. Restructured Fortran Kernel

	L1	L2	TLB
DATA: a1			
Load Hit Ratio	96.87%	10.02%	99.90%
Store Hit Ratio	99.22%	15.53%	-
DATA: a2			
Load Hit Ratio	96.87%	34.22%	99.63%
Store Hit Ratio	99.22%	14.88%	-
DATA: a3			
Load Hit Ratio	96.87%	33.29%	96.51%
Store Hit Ratio	99.22%	14.42%	-

Figure 6. Memory Profile for New a1,a2,a3

6 OpenMP programs on pSigma

When threads concurrently executing on multiple processors make accesses to shared data, there are many opportunities for performance analysis and tuning. There are several degrees of freedom that can be exploited to improve performance in this area: careful scheduling of work items to threads, placement of threads on different processors and placement of data in the address space. All these decisions interact with the data sharing pattern dictated by the application and

Array	Function	Memory References	Stores	Cold Misses	c2c transfers
p	conj_grad@OL@B@OL@F	741241600	0	2624	1048576
a	conj_grad@OL@B@OL@F	741241600	0	16588	70057
z	conj_grad@OL@B@OL@13	29649664	0	2621	39427
z	con_grad@OL@B@OL@11	11200000	5600000	0	2395
colidx	conj_grad@OL@B@OL@F	741241600	0	0	403

Table 1. cache to cache transfers in NAS CG

```

1 !$omp do
2   do j=1,naa+1
3     ...
4     p(j) = r(j)
5   enddo
6   ...
7 !$omp do
8   do j=1,lastrow-firstrow+1
9     sum = 0.d0
10    do k=rowstr(j),rowstr(j+1)-1
11      sum = sum + a(k)*p(colidx(k))
12    enddo
13  enddo
14  q(j) = sum
15 enddo

```

Figure 7. Segment of the code from CG that had a large number of cache line transfers

influence the performance. For instance, when a producer and consumer of a data item are scheduled to run on different processors, the corresponding cache line will have to be transferred between the two processors, which could be avoided if the producer and consumer are assigned to the same thread or to the same processor. Another example is that of false sharing [7, 9, 8], where two threads access disjoint sets of data from the same cache line, but do not necessarily share any data from it. Because of coherence policies, this can result in the cache line ping-ponging between two processors, which can be avoided under some conditions. Analysis of such situations for potential performance improvements can be complex. But it must be aided by tools that provide detailed accounts of what is happening. Our goal is to provide an infrastructure for these studies where **pSigma** will be able to expose potential opportunities and also provide simulated results when data is repartitioned, work is repartitioned among threads and schedules are rearranged.

In this section we provide an example of a simple observation of how many times cache lines are being

transferred between processors. Some of these may be avoided and some may not be. Here we just provide a count for all such transfers. Our tool, known as *c2c*, provides a count of cache-to-cache transfers under the following conditions. It assumes an infinite cache system. The first time a cache accesses a line, that is counted as a cold access. All other accesses, ideally should not cause a fault, under the infinite cache assumption. However, in a multi-processor situation, a writer invalidates the line from all other caches and hence the caches may have to obtain the line again for further accesses. Since all caches are infinite, all these will be cache to cache transfers. This count gives an idea about the flow of information and can be used to guide the schedules and data mapping.

We instrumented the CG OpenMP benchmark from NPB ([10]) suite. A segment of this code, which is repeated a large number of times, is shown in Figure 7. Two loops manipulate the array *p*. The first loop initializes it and the second loop accesses it through in-direction. Both loops are parallelized (statically scheduled) by OpenMP directives. Since the producer and consumer for the same data may run on two different processors, this will cause some cache to cache transfers. Table 1 shows the transfer counts categorized by function and data structure. Our tool indicated that the data structure *p* has suffered more than 80% of the total invalidations in the program. This information is useful for tuning the application for improved performance.

7 Conclusion

In this paper we presented the **pSigma** infrastructure for instrumentation of parallel applications. This infrastructure allows the user to probe into the execution of the application at selected points, which can be specified in a natural language using symbolic names of data structures and functions and performance-related information. Furthermore, the infrastructure transparently provides performance-related metrics which can assist in developing data-centric performance monitoring and analysis tools.

We demonstrated one possible application of the

infrastructure by implementing a tool which detects cache-to-cache transfers in OpenMP applications.

References

- [1] L. Derose, K. Ekanadham, J.K. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of Supercomputing '02*, November 2002.
- [2] Luiz DeRose, Ted Hoover Jr., and Jeffrey K. Hollingsworth. The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.
- [3] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, November 1995.
- [4] Luiz DeRose and Daniel Reed. Svpablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proceedings of the International Conference on Parallel Processing*, pages 311–318, August 1999.
- [5] Bernd Mohr, Allen Malony, and Janice Cuny. TAU Tuning and Analysis Utilities for Portable Parallel Programming. In G. Wilson, editor, *Parallel Programming using C++*. M.I.T. Press, 1996.
- [6] Jerry Yan, Sekhar Sarukkai, and Pankaj Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software Practice & Experience*, 25(4):429-461, April 1995.
- [7] Tor E. Jeremiassen , Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *ACM SIGPLAN Notices*, 30(8):179-188, August 1995.
- [8] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, Per Stenström, The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th annual international symposium on Computer architecture*, p.88-97, May 16-19, 1993, San Diego, California, United States.
- [9] Joseph Torrellas , Monica S. Lam , John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches, *IEEE Transactions on Computers*, 43(6):651-663, June 1994.
- [10] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. *NAS Technical Report NAS-99-011*, NASA Ames Research Center, Moffett Field, CA, 1999.
- [11] S.J. Eggers, D.R. Keppel, E.J. Koldinger, and H.M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the 1990 ACM SIGMETRICS conference on Tue, 28 Sep 2004*, Simone Sbaraglia wrote: *Measurement and Modeling of Computer Systems*, p.37-47, April 1990.
- [12] Jordi Caubet, Judit Gimenez Jesus Labarta, Luiz DeRose, and Jeffrey Vetter. A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications. In *Proceedings of the Workshop on OpenMP Applications and Tools- WOMPAT 2001*, pages 53 – 67, July 2001.