SIGMA: A Simulator Infrastructure to Guide Memory Analysis

Luiz DeRose, K. Ekanadham, Jeffrey K. Hollingsworth, and Simone Sbaraglia

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
USA
{laderose,eknath}@us.ibm.com

Dept. of Computer Science
University of Maryland
College Park, MD 20740, USA
hollings@cs.umd.edu

University of Rome
"La Sapienza"
Rome, Italy
sbaragli@iac.rm.cnr.it

Abstract

In this paper we present SIGMA (Simulation Infrastructure to Guide Memory Analysis), a new data collection framework and family of cache analysis tools. The SIGMA environment provides detailed cache information by gathering memory reference data using software-based instrumentation. This infrastructure can facilitate quick probing into the factors that influence the performance of an application by highlighting bottleneck scenarios including: excessive cache/TLB misses and inefficient data layouts. The tool can also assist in perturbation analysis to determine performance variations caused by changes to architecture or program. Our validation tests using the SPEC Swim benchmark show that most of the performance metrics obtained with SIGMA are within 1% of the metrics obtained with hardware performance counters, with the advantage that SIGMA provides performance data on a data structure level, as specified by the programmer.

1. Introduction

Understanding and tuning memory system performance is a critical issue for most scientific programs. To help programmers tune their programs, a variety of tools have been created ranging from source code and binary analysis tools [1, 2, 3, 4, 5] to libraries and utilities to access hardware performance counters built into microprocessors [6, 7, 8, 9]. Depending on the type of problem being studied and stage of the tuning process (initial tuning of a new algorithm vs. fine-tuning for a specific platform), different tools are useful.

One area that has been lacking is a set of tools that allow programmers to understand the precise memory references in their program that are causing poor cache behavior. Fine-grained information such as this is useful for tuning loop kernels, understanding the cache behavior of new algorithms, and to investigate how different parts of a program compete for and interact within the memory subsystem.

In this paper we present a new data collection framework and family of cache analysis tools called SIGMA (Simulation Infrastructure to Guide Memory Analysis). The goal of the SIGMA environment is to provide detailed cache information by gathering memory reference data using software-based instrumentation, in order to provide feedback to programmers to help them to apply program transformations to improve cache performance. Typical tuning operations that programmers perform include padding of data structures to improve cache alignment, blocking (also known as tiling) of their code to provide cache re-use, and loop fusion, also to increase cache and register re-use. However, some of the challenges that users face when optimizing their applications are identifying which data structures are causing poor memory behavior and detecting which sections of the program would benefit from modifications such as blocking or fusion. The SIGMA framework provides an environment to help users to identify data structures and code segments that are causing poor program performance due to data layout without having to re-execute the program several times.

The SIGMA environment consists of a pre-execution tool that locates and instruments all instructions that refer to memory locations, a runtime data collection tool that performs a highly efficient lossless compression of the stream of memory addresses generated by the instrumentation, and a number of simulation and analysis tools that process the compressed memory reference trace to provide programmers with tuning information. We chose to use a post-compile instrumentation approach so that we would be able to gather data about the actual memory references generated by optimizing compilers rather than using source instrumentation which would gather data about the user specified array references. The simulation and analysis tools include a TLB simulator, a data cache simulator, a data prefetcher simulator, and a query mechanism that allows users to obtain performance metrics and memory usage statistics.

0-7695-1524-X/02 $17.00 (c) 2002 IEEE

2. SIGMA Design Overview

One of the main goals of SIGMA is to provide an infrastructure that facilitates quick probing into the factors that influence the performance of an application. Its aims are two-fold. Firstly, it can be used to highlight bottleneck scenarios, such as excessive cache/TLB misses and inefficient data layouts. Secondly and more importantly, it can assist in perturbation analysis, to determine performance variations caused by architectural and (program) structural changes. Varying cache parameters is an example of architectural change. Loop interchanges and padding data structures are examples of structural changes to a program. A chief characteristic of SIGMA is that it relates the analysis to the source programs by identifying the data structures, functions, and loop bodies as defined by the user.

2.1 Abstract Representation of Program Execution

Since our aim is to assist the performance analysis by perturbing both the architectural parameters as well as program structure, we need to capture an abstract, but complete, information on the execution of a program, so that the behavior can be simulated for the altered architecture/program. Although we gather this information from the execution of the program on a specific architecture, we strip the architecture-specific information and make it as abstract as possible. Our abstract representation of an execution of a program has two components: *Structural Information* and *SIGMA Trace*, which are described below.

The machine code of the program is statically divided into basic blocks. The blocks are numbered sequentially, so that each instruction can be uniquely identified by the pair *(blockNumber, blockOffset)*. The *Structural Information* includes instruction addresses, opcodes and (register) dependence relations between instructions. The instruction semantics is the only architecture-dependent information, although most RISC architectures have similar instructions.

The *SIGMA Trace* of an execution captures the control flow as well as the memory addresses generated. It is a sequence of *block-instances* of the form: *block(blockNumber, blockOffset) {address-list}*. Each block instance specifies the sequence of instructions executed from the block with the given block number, starting from the block-offset till the end. The address-list gives the memory addresses generated by the memory-access instructions in that sequence. Thus, structural information and trace, together, enable us to reconstruct the entire execution of the program.

2.2 Trace Compression

Loops in programs provide an excellent opportunity for representing the trace information in a very compact form and we exploit this here. The body of a loop is a sequence of block instances, repeated many times, possibly with a different address-list each time a block instance is repeated. A reference to an element of an m-dimensional array, within a nest of n loops takes the general form, $AX+B$, where A is an m by n matrix, X is the n-dimensional iteration vector and B is an m-dimensional constant vector. If a is the base address of the array and each element of the array is e bytes long, then the address of the reference is given by $a+ e\ W^T\ (A\ X + B)$, where W is the m-dimensional vector of total sizes of each dimension of the array (*i.e.* successive products of the dimensions of the array). We can simplify the form to $c+ SX$, where the constant $c=a+eW^TB$ and the n-dimensional row-vector $S = e\ W^T\ A$. Thus, we can compactly represent the reference with the $n+1$-tuple $\langle c,S \rangle$, so that addresses can be generated for all values of the iteration vector X. A block instance within a nest of n loops will now be of the form *block(blockNumber, blockOffset) {<n+1_tuple>,<n+1_tuple>, ...}*. A loop having m iterations can be represented as *repeat(m){item-list}*, where each *item* is either a block instance at that level or a loop at the next level. Thus the format of the SIGMA Trace can be formally expressed as follows:

sigmaTrace ::= level_0_item, level_0_item,.....
level_n_item ::= ***block****(blockNum, blockOffset) { <n+1_tuple>, <n+1_tuple>, ...}* |
repeat*(iterCount) { level_n+1_item, level_n+1_item, ...}* |
repeat*(iterCount){level_n+1_item,level_n+1_item,...****exit****, level_n+1_item, level_n+1_item, ...}*

where the words ***block***, ***repeat*** and ***exit*** are keywords, and the number of tuples in a block instance must match the number of memory access instructions in it. The keyword ***exit*** indicates that the last iteration of the loop ends with it (rather than going all the way to the end). This feature was added since we found that a majority of the loops exhibit this behavior and it resulted in considerable saving in the length of a compressed trace (especially for nested loops).

2.3 Instrumentation and Trace Generation

The trace is generated by instrumenting selected instructions of the program. To instrument an instruction Z, which is at offset y in block b of the program, the instruction is replaced by a branch to a new code segment added at the

Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 **IEEE**

end of the program. The code segment saves the necessary state, computes the target address *a*, if Z is a memory access instruction, invokes a trace library supplying the triple (*b,y,a*), performs the instruction Z, restores the state, and branches back to the next instruction after the instrumented instruction. By instrumenting all memory access instructions and the first instruction executed in each branch instance, we have complete information as described in section 2.1.

The trace library (linked into the application by our instrumentation tool) maintains a buffer. Each time the library is invoked, the corresponding triple is appended to the buffer, and the buffer is searched backwards to find any matching triple that corresponds to the previous execution of the same block instance. As an example, suppose the suffix of the current buffer is the sequence of triples: (b_1,y_1,a_1), (b_2,y_2,a_2),... (b_n,y_n,a_n) as depicted in Figure 1(a). If this is followed by the sequence of triples (b_1,y_1,d_1), (b_2,y_2,d_2),…(b_n,y_n,d_n), as shown in Figure 1(b), then a loop is detected with the strides given by $s_i = d_i - a_i$. At this point, as illustrated in Figure 1(c), an iteration counter $x=2$ is started, indicating that the body can be repeated for x=0,1. As long as the subsequent triples match the pattern (b_1,y_1,a_1+x*s_1), (b_2,y_2,a_2+x*s_2),…(b_n,y_n,a_n+x*s_n), the iteration counter x is incremented. Whenever the match fails, as illustrated in Figure 1(d), the matched part of the loop is recorded in the encoded form, using the ***repeat/exit*** clause described in Section 2.2, as depicted in Figure 1(e). This indicates that the loop can be repeated for x=0..k-1, except that the last iteration is terminated at the exit clause. To detect nested loops, the compression algorithm is applied again to the output of the first level of compression.

A chief advantage of this method of compressing the trace is that the trace can be compressed *online*, as the application program executes. This avoids the need to store large traces first and post-process them later for compression. Furthermore, it is easy to see that one can reverse the above algorithm to generate the sequence of triples from a compressed trace in an incremental manner. Thus all the tools that process the trace can operate upon the compressed trace.

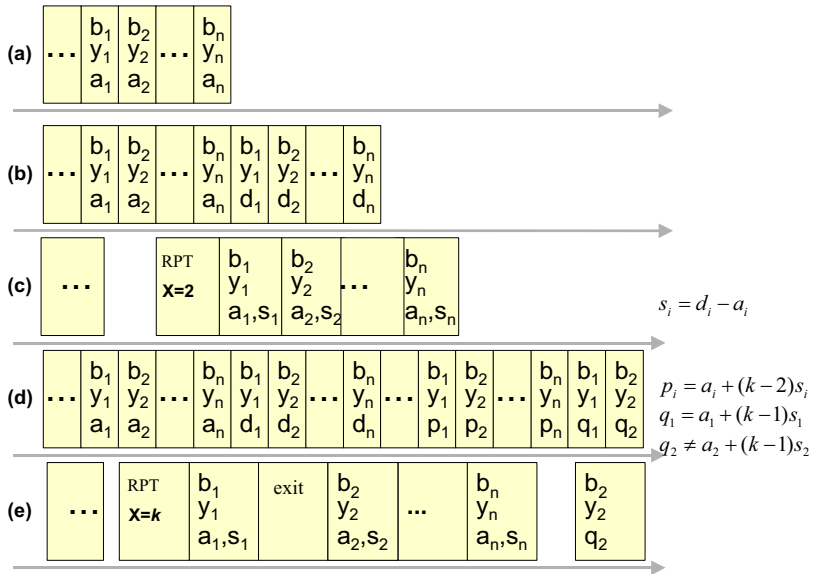


Figure 1: Schematic view of operations on the buffer during compression

This method has some limitations. Clearly the size of the buffer limits the extent to which repeated patterns can be recognized. While small buffers miss some repeated patterns, large buffers increase the runtime overhead by requiring larger number of comparisons. The above method of matching nested loops requires that the iteration counts for inner loops must be constants and cannot be functions of outer iteration variables (as is the case for some computations involving wavefronts or recursive function calls). It is possible to extend the notation of the ***repeat*** clause, to handle such cases. However, we have not pursued this approach in this study. Finally, loops do contain references that do not generate address sequences in the form specified in section 2.2. Even if they miss the pattern occasionally because of conditionals, the matching algorithm fails and a longer trace will be generated. Again, while further modifications can be made to the format to accommodate such cases, we have not pursued them here.

Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 **IEEE**

In spite of these limitations, we have found substantial reductions in trace sizes, as evidenced by the statistics presented in Table 1. We observe two extremes in the compression ratio when running the matrix multiplication kernels (blocked and recursive). In both cases we multiplied two matrices of 128 by 128 elements. With the blocked matrix multiplication kernel, which is characterized by its six levels of loop nest and very regular memory access pattern, the compression algorithm was able to compress up to the sixth level, achieving a compression rate of more than 20,000. On the other hand, with the recursive matrix multiplication kernel, we observe the limitations of our approach for random and irregular accesses. In this case, the compression rate was only 1%, which was due to the loop used to initialize the arrays. With the applications from the SPEC benchmark, we observed an average of compression rate of three orders of magnitude, but again with extremes in both sides. Swim spends most of its execution time on loops with 2 nesting levels accessing data with regular strides, while wupwise has several loop nests, but several subroutine calls inside of the inner loop, and irregular memory access.

Program	*Memory References (M)*	*Uncompressed trace (Mbytes)*	*Compressed trace (Mbytes)*	*Nesting level*	*Compression Ratio*
Blocked matrix mult.	6.5	29.7	0.0014	6	21,214.29
Bitonic sort	16.3	100.0	66.5	1	1.50
Recursive matrix mult.	18.9	102.1	101.1	2	1.01
Red black SOR	137.1	574.3	14.6	2	39.34
SPEC applu	59.5	243.6	0.17	4	1,432.94
SPEC hydro2d	130.3	643.8	0.83	2	775.66
SPEC mgrid	100.2	415.0	0.40	3	1,037.50
SPEC swim	147.9	685.6	0.011	3	62,327.27
SPEC wupwise	375.9	2,073.3	477.6	4	4.34

Table 1: Trace compression statistics

2.4 Source Code and Data Structure Mapping

Performance Analysis is best understood when the observations are expressed in terms of functions and data structures defined in the source code. For instance, it is more useful to know that the accesses to array "A" from the function "F" are causing several cache misses, rather than indicating excessive cache misses for this application. In order to provide such relation to the source programs, two key mappings are required: for each memory reference made, we gather the source line in the program that made the reference and the element of the data structure that corresponds to the reference. Debugger information stored in an executable created with the debug option[1], contains mappings to go from instruction to source line and a mapping to go from data structure to its address and type information. Using this, we can connect every reference triple (*b,y,a*) in the SIGMA Trace to the data structure and the place of reference in the source code. Section 3 describes how we use these mapings in the performance analysis.

3. Using SIGMA for Memory Analysis

As shown in Figure 2, the process flow of memory analysis with SIGMA is divided into three main steps: *instrumentation/execution*, *simulation*, and *querying*. The source code is initially instrumented and executed as described in Section 2.3, in a completely transparent manner, as it does not require any directives or source code modifications. This yields the (program/data) structural information file and the compressed trace file.

The simulation phase takes these two files and architectural and data layout specifications of interest to the user. The architectural specifications include cache hierarchy and cache parameters. Data layout specifications contain internal and external padding for desired data structures. A feeder module dynamically uncompresses the trace and suitably modifies the addresses to reflect the given layout specifications. This sequence is fed to a module that simulates the specified architecture. The result is a condensed *repository* of *statistics* on memory accesses. The repository is organized in such a manner that statistics can be tailored to provide a host of sectional views based on functions and data structures. This is implemented by the *query* module, which provides an interactive interface through which the user can view different views of the statistics. One can repeatedly view various statistics from a given repository. One can generate different repositories for different architectural/lay out specifications. Finally, one can make source modifications and rerun the whole thing. This provides a flexible 3-tier framework to do "what-if" analysis with progressively faster turn-around times.

[1] Notice that with most compilers, the use of the debug option (-g) does not inhibit optimizations.

Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 **IEEE**

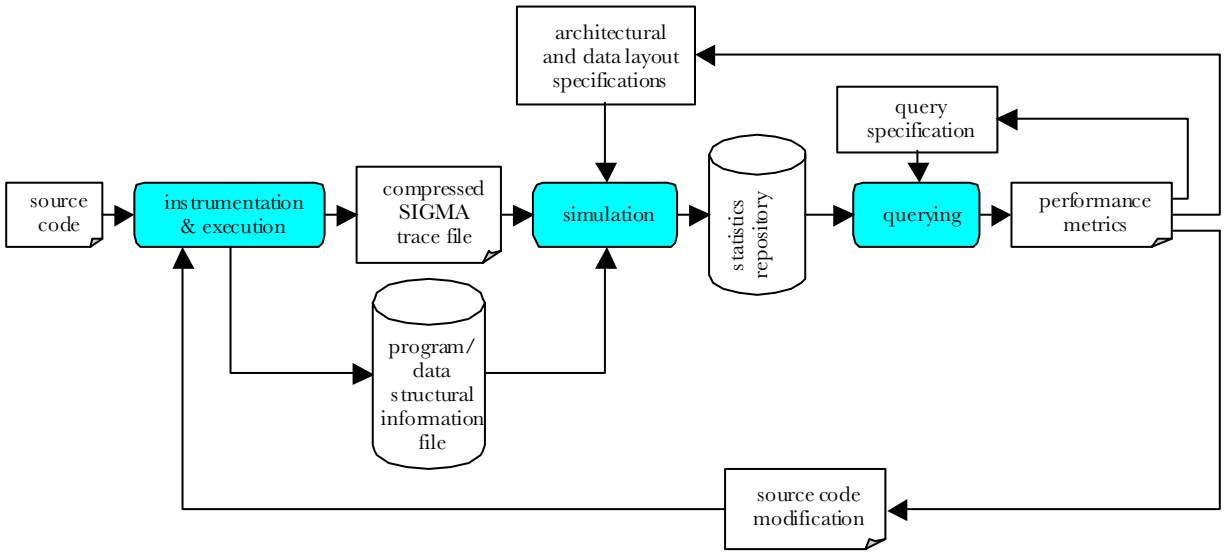


Figure 2: Memory analysis flow with SIGMA

3.1 Simulator

The purpose of the simulator is to project the performance of the same memory reference trace on a memory subsystem of choice. By repeating the simulation with an altered architecture, one can do perturbation analysis and answer questions like, "should we increase the size of the cache or increase its associativity?". The behaviors emulated include, data prefetcher, TLB and multiple levels of caches, with parameterized size, associativity, line size, write-back/writethru policy, replacement policy etc. The simulation collects statistics like the number of cache hits/misses and number of loads/stores. Two novelties of the simulator are worth mentioning. First the simulator operates on the compressed trace, by dynamically uncompressing the trace and feeding one address at a time to the memory hierarchy. Secondly, the addresses are passed through *filters* on their way to the memory. A filter suitably modifies an address to reflect the given layout specifications (see Section 3.2 for more details). Work is underway to extend this for multiple processors and to collect statistics on false sharing. We are also investigating to extend the filters to do more sophisticated transformations, such as blocking of well-known operations.

3.2 Data Structure Padding

A technique that attempts to improve program performance by reducing misses due to lack of cache associativity in the memory subsystem is to include buffers between array definitions, in order to space the variables with respect to each other, or to increase one of the dimensions of a multidimensional data structure with the purpose of spacing the columns/rows of the data with respect to each other. These techniques are known as external and internal padding respectively. Padding may improve program performance when data is laid out in such a way that several frequently used data structures map into the same cache set or TLB line. With SIGMA one can analyze the effect of modifications in the data structure layout, without having to modify the source code and re-execute the program. This is achieved by re-mapping the data structure to different relative locations in memory, which correspond to external or internal padding.

External padding works as follows: Consider, for example, the static memory layout of the following FORTRAN common block (assuming `real*4` variables):

```
COMMON a1(512,512),a2(512,512),a3(512,512)
```

presented in Table 2 and suppose that the data-structure transformation consists of an 8-byte external padding of array "a1". This padding would mean that arrays "a2" and "a3" would have to be shifted by 8-bytes, leading to the "padded" memory layout presented in Table 3. This external padding transformation is equivalent to inserting "dummy" variables between "a1" and "a2" that would occupy a total of 8 bytes. This would correspond to the following FORTRAN common block:

Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 **IEEE**

```
COMMON a1(512,512), dummy1, dummy2, a2(512,512),a3(512,512)
```

Start address	*End address*	*Data*	*Type*	*Dimensions*
0x20001dbc	0x20101dbc	a1	real*4	(512,512)
0x20101dbc	0x20201dbc	a2	real*4	(512,512)
0x20201dbc	0x20301dbc	a3	real*4	(512,512)

Table 2: Static memory layout example

Start address	*End address*	*Data*	*Type*	*Dimensions*
0x20001dbc	0x20101dbc	a1	real*4	(512,512)
0x20101dc4	0x20201dc4	a2	real*4	(512,512)
0x20201dc4	0x20301dc4	a3	real*4	(512,512)

Table 3: Padded memory layout

Internal padding is slightly more complicated, as address computation involves arithmetic with the dimensions of the array. Suppose, for example, that the first dimension of array "a1" in the previous example is increased from "512" to "513" elements, and assume, without loss of generality, that the memory layout is column-major, as is the layout produced by Fortran codes. As illustrated in Figure 3, the data belonging to the second column of "a1" would have to be shifted by 4 bytes, the data belonging to the third column of "a1" would have to be shifted by "4+4" bytes, and so on. In addition, the arrays "a2" and "a3" would have to be shifted by "512*4" bytes each. Considering again Table 2 as an example, the internal padding transformation of increasing the dimension of array "a1" by one row would involve building the new memory layout to take into account the shift of "a2" and "a3", as well as, re-computing the correct offsets within the padded data structure "a1". The corresponding memory layout is presented in Table 4.

The procedure to compute the new offset of any given element in an m-dimensional array "A", uses a weights vector $\boldsymbol{w=(1,d_1,d_2,\ldots,d_{m-1})}$, where $\boldsymbol{d_k}$ is the k^{th} dimension of "A", such that if $\boldsymbol{I=(i_1, i_2, \ldots i_m)}$, where $\boldsymbol{0 \le i_k < d_k}$, is an index vector of "A", the scalar product of $\boldsymbol{I \bullet w}$ is the integer offset of the element A($i_1, i_2, \ldots i_m$) within the array. For the example illustrated in Table 4, we have $\boldsymbol{w = (1,512)}$ for the original data structure and $\boldsymbol{w = (1,513)}$ for the padded version.

Start address	*End address*	*Data*	*Type*	*Dimensions*
0x20001dbc	0x201025bc	a1	real*4	(513,512)
0x201025bc	0x202025bc	a2	real*4	(513,512)
0x202025bc	0x203025bc	a3	real*4	(513,512)

Table 4: Memory layout after internal padding

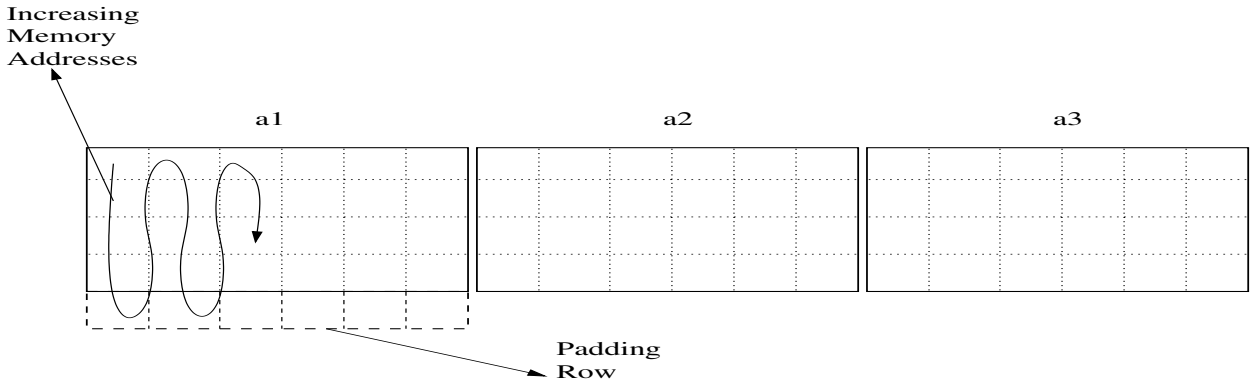


Figure 3: Memory layout changes due to internal padding

Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 **IEEE**

3.3 Repository of Statistics and Query Tool

The statistics collected during the simulation phase are organized into a repository, with the purpose of allowing for both a "*data-centric*" and "*control-centric*" retrieval, providing users the flexibility of looking at performance metrics based on both data and control-flow selections interactively. The interactive query module provides interface to the user to select (i) a subset of the statistics, (ii) a reduction operation on them, and (iii) a means to display of the result. As illustrated in Figure 4, statistics are selected by providing a desired sub-space of the following 3-dimensional space. Along one dimension, all data structures are laid out. One can specify a list of desired arrays or array subsections. Along the second dimension, the control flow through loops, functions and files is laid out. One can select segments of the locus of control from this. Finally, along the third dimension, all the performance metrics are laid out. These include loads, stores, misses etc. Thus selecting a sub-space [{a,b,c[*,1]}, {foo}, {loads}] will give the number of loads issued from the function "foo" to the arrays "a" and "b" and to the first column of "c". Currently the reduction operators are addition and max/min. In addition, facilities are provided for computation of derived metrics, such as cache hit or miss ratio. The results can be displayed as tables, histograms, pi-charts, graphs etc.

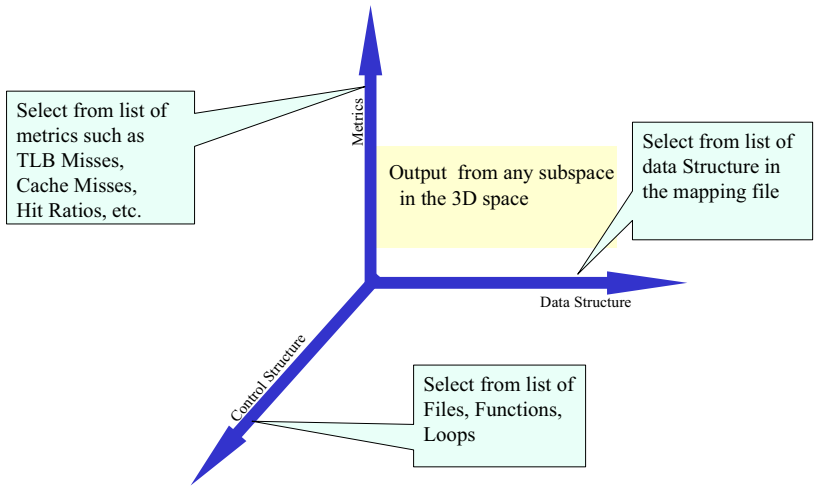


Figure 4: Sigma 3D view of memory analysis

4. Experimental Results

In this section we discuss our experiments to validate the SIGMA approach. In Section 4.1 we validate the performance metrics obtained with SIGMA against data obtained using the hardware performance counters. In Section 4.2 we validate the padding algorithm using a synthetic kernel that generates a large number of TLB misses and compare the performance metrics obtained after using the padding algorithm with the numbers obtained with a padded version of the same kernel.

4.1 Validation using hardware performance counters

In order to validate the numbers generated by SIGMA, we used a set of micro-benchmarks that allows us to precisely estimate the expected number of loads, stores, level 1 load and store misses, and TLB misses, based on the description of the micro-architecture. We ran these benchmarks on an IBM Power3 and using an interface to access the hardware counters [9], we counted the actual numbers for each of the metrics above, and compared with the values obtained with SIGMA. In addition, we instrumented the SPEC Swim benchmark [10] to access the hardware performance counters and again compared the results with the numbers obtained with SIGMA.

4.1.1 Validation using micro-benchmark

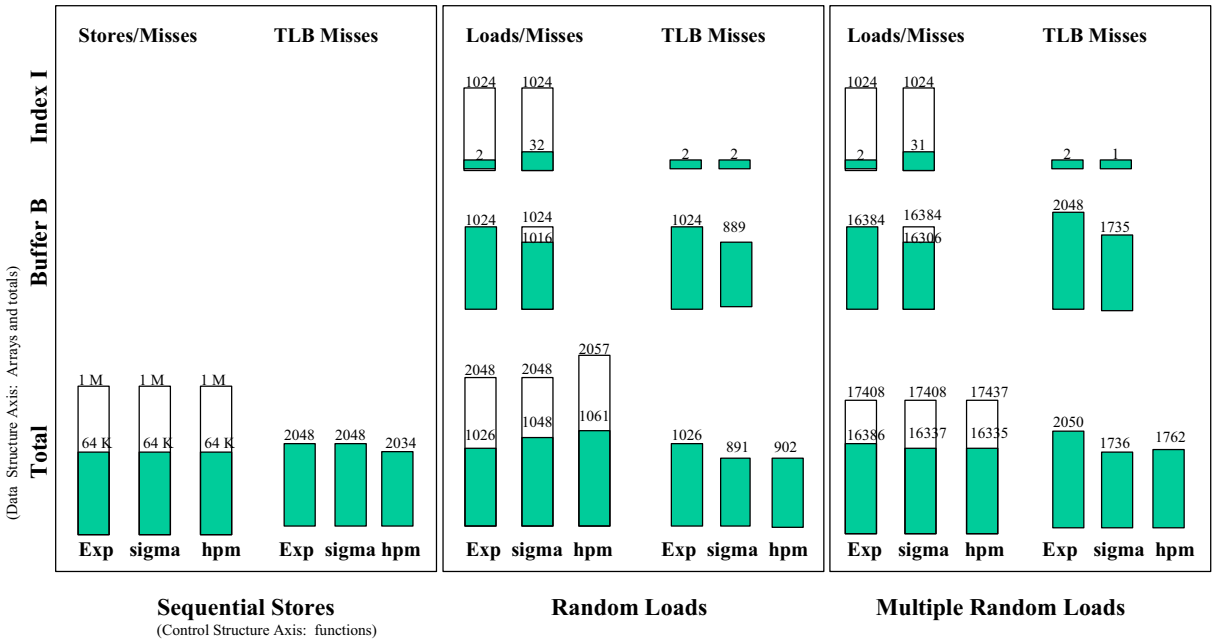
Our micro-benchmark is composed of three functions: *sequential stores, random loads, and multiple random loads*. For each of these functions we use an array, B, of n=1 million double precision words (8 bytes). In addition, in func-

Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 **IEEE**

tions random loads and multiple random loads, we use an array, *I*, of 1K integers (4 bytes), containing random indices from the range *(1..n)*. For each of the functions we expect the following results:

- **Sequential stores:** This function makes the following assignments: *B[k] = c, for k = 1..n*. Since we are accessing 1M double precision words (8M bytes) sequentially and the IBM Power3 has pages of 4K bytes, we would expect to have 2K page faults, which should generate 2K TLB misses. The IBM Power 3 cache line has 128 bytes, so that each page has 32 cache lines. Hence, this function should generate 64K store misses.
- **Random loads:** This functions performs the summation: *s = s + B[I[k]], k=1..1024*. In this case, we are accessing 1K random elements of the array, *B*; therefore, we expect to have 1K TLB misses and 1K L1 load misses. However, given that we are dealing with random entries, these numbers should be viewed as upper bounds. The larger the array *B* gets, the closer will be the measured value from this bound.
- **Multiple random loads:** This function is an extension of *random loads*, which performs the following summation: *s = s + B[I[k]+32*j], j=0..15, k=1..1024*. For each randomly accessed page, we access 16 consecutive entries in the array B, with a stride of 32 double precision words, which is equivalent of two cache lines (256 bytes), thus foiling any attempt to prefetch successive lines ahead in the hardware. Hence, we expect to have one cache miss for each load, with a total of 16K L1 load misses and 2K TLB misses overall. The second TLB miss per load is due to the fact that a page boundary is crossed during the span of accessing 16 double precision elements at a stride of 32. Again, these numbers should be viewed as upper bounds as we are dealing with random indices.

Figure 5 depicts the results of the micro-benchmark. Along the horizontal axis, the three functions, *Sequential Stores, Random Loads,* and *Multiple Random Loads* are shown. Along the vertical axis the data structures, Buffer *B*, Index *I*, and their sum total are shown. At each point in this two-dimensional space, the metrics of load/store counts, load/store misses and TLB misses are shown. Each metric has three values: an expected value estimated by observing the code, the value obtained from sigma simulation and the value measured by HPM. For load/stores, the height of a bar is the total number of loads/stores and the shaded area within shows the cache misses. For TLB, only misses are shown. In all cases, the measured values were close to the expected numbers. We observe, however, that there



Note: For loads/stores, the height of each bar is the total number of loads/stores and the shaded height gives the miss count within that. For TLB, only miss counts are shown. Also note that the heights of the bars are not to scale.

Figure 5: Expected, simulated, and measured metric values from the micro-benchmark

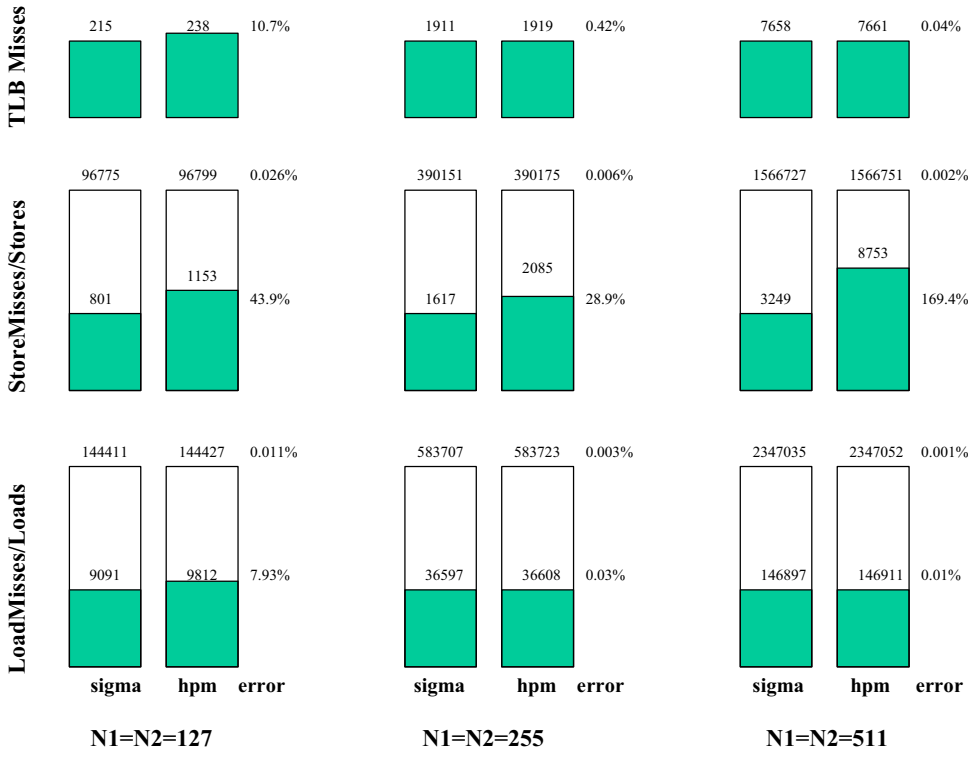
Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 **IEEE**

was a small variance when collecting the numbers with the hardware performance counters. This variance occurs due to several factors, including effects of the operating system and the fact that the HPM utility was not designed for collection of data with code regions that have small granularity. We also notice that while the HPM toolkit provided the values for the metrics accumulated over all variables in each function, with SIGMA we were able to collect the same data, specific for each variable.

4.1.2 Validation using the SPEC Swim Benchmark

For our second validation test we instrumented the SPEC Swim benchmark to collect hardware counters at a function level, in order to compare with the results obtained with SIGMA. We ran three different problem sizes to verify if differences in measurement were scaling with the problem size, or were just random noises. For each run, we executed three time-steps, so the function CALC3 would be executed only once, in the second iteration. This approach would also reduce the effects of operating system and cold misses that would occur in the first time-step. The metric values for function CALC3, one of the most time consuming function in the program, are presented in Figure 6, for the problem sizes defined by N1=N2=127, N1=N2=255, and N1=N2=511, respectively.

From Figure 6 we observe that the differences in measurements for most metrics are just random noise in the measurements. They did not scale with the problem size. In fact, some of these differences can be attributed to the environmental perturbations that affect the hardware counters, as described in Section 4.1.1. There is no easy way to filter them out. For the larger problem size, we observed that all but L1 store misses had differences less than 0.1%. The large difference in L1 store misses is due to the subtleties involved in the micro operations for a store operation involving store queues, which may determine as to when a store miss is detected. The current simulator implements each memory operation as atomic and does not model any timing.



Note: Error is computed as percentage of the difference between sigma and hpm values, with sigma value as the base
Also note that the heights of the bars are not to scale.

Figure 6: Metric values for function CALC 3 in swim, for various problem sizes

Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 IEEE

Finally, as shown in Table 5, which presents the metrics for the function CALC3 with the larger problem size, we observe that a major advantage of the SIGMA approach over the collection of hardware performance counters, is that it relates the analysis to data structures as defined by the user. All the metrics obtained from Sigma are broken down and given for each array structure, so one can investigate the influence of each array in the performance of the program. We omitted them in Figure 6, as we are comparing with the output from hpm, which collected the metrics for the entire CALC3 function. The row in Table 5 labeled as "*unknown"* indicates memory addresses that were not declared as global variables by the programmer[2]. This includes local variables, as well as stack variables generated by the compiler for temporary use. Although we have not discussed here, the detection of a large number of temporaries can be helpful in the identification of sections of code that could need restructuring, in order to help the compiler to avoid the generation of an excessive number of temporaries.

SWIM CALC3 N1 & N2 = 511		Loads			Stores			TLB misses
		Total	L1 misses	TLB misses	Total	L1 misses	TLB misses	(total)
SIGMA	POLD	261121	16355	514	261121	542	509	1023
	VOLD	261121	16355	514	261121	541	508	1022
	UOLD	261121	16354	513	261121	541	508	1021
	PNEW	260100	16253	508	0	0	0	508
	VNEW	260100	16252	508	0	0	0	508
	UNEW	260100	16255	508	0	0	0	508
	P	261121	16353	514	261121	542	509	1023
	V	261121	16352	514	261121	541	508	1022
	U	261121	16353	512	261121	541	508	1020
	scalars	3	0	0	0	0	0	0
	unknown	6	5	3	1	1	0	3
	Total	**2347035**	**146897**	**4605**	**1566727**	**3249**	**3050**	**7658**
HPM		**2347052**	**146911**	**----**	**1566751**	**8753**	**----**	**7661**
Difference		0.001%	0.01%		0.002%	169.4%		0.039%

Table 5: Metric values for the function CALC3 with problem size N1 and N2=511

4.2 Validation of the padding algorithm

In general, when a large number of arrays is accessed in a loop and the hardware counters indicate a large number of misses (e.g., TLB), it is not trivial to determine the exact cause of the problem - whether it is caused by accesses to a single array or by complex interactions between many array accesses. We ran the synthetic kernel summarized in Figure 7 on an IBM Power3 system to test the effectiveness of the padding algorithm. When running this program, we observed a poor performance, which was caused by a large number of TLB misses (35.6M TLB misses for a total of 44.8M accesses). We generated a trace file for the program and two SIGMA data repositories, one for the original program and the other using the filtering algorithm for internal padding of one extra row on all arrays. We then used the memory query tool to obtain the performance metrics presented in Table 6. In addition, we padded the program by hand and generated a second trace file and a third repository, in order to validate the numbers obtained with the filtering algorithm. We observe that the error of measurement between the numbers obtained when using the filter and the numbers obtained with the repository from the hand-padded program was less than 0.01%.

[2] The current simulator breaks down the accesses only by global variables. Work is in progress to break down further with local variables details.

Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 **IEEE**

```
      program array1
      integer ix, iy, ixy, id, jd, nx, ny, nxy, nd
      parameter (nx=512, ny=512), (nxy=nx*ny), (nd=3)
      real a1(nx,ny),a2(nx,ny),a3(nx,ny)
      real b1(nxy,nd),b2(nxy,nd),b3(nxy,nd)
      real c11(nxy,nd,nd),c12(nxy,nd,nd),c13(nxy,nd,nd)
      real c21(nxy,nd,nd),c22(nxy,nd,nd),c23(nxy,nd,nd)
      real c31(nxy,nd,nd),c32(nxy,nd,nd),c33(nxy,nd,nd)
      ...
      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
      ...
```

Figure 7: Synthetic kernel that generates a large number of TLB misses

	Total loads	*Total stores*	*Total accesses*	*TLB LD misses NO padding*	*TLB ST misses NO padding*	*TLB LD misses with padding*	*TLB ST misses with padding*
a1	2359296	3145728	5505024	2354697	2883072	4626	3375
a2	2359296	3145728	5505024	2359296	2753280	8037	202702
a3	2359296	3145728	5505024	2359296	2754048	198501	201505
b1	2359296	0	2359296	2359296	0	198495	0
b2	2359296	0	2359296	2359296	0	2310	0
b3	2359296	0	2359296	2359296	0	2310	0
c11	2359296	0	2359296	2359296	0	2313	0
c12	2359296	0	2359296	2359296	0	2313	0
c13	2359296	0	2359296	2359296	0	2313	0
c21	2359296	0	2359296	2359296	0	2313	0
c22	2359296	0	2359296	2359296	0	2313	0
c23	2359296	0	2359296	2359296	0	2313	0
c31	2359296	0	2359296	2359296	0	2313	0
c32	2359296	0	2359296	2359296	0	2313	0
c33	2359296	0	2359296	2359296	0	2313	0
total	35389440	9437184	44826624	35384841	8390400	435096	407582

Table 6: TLB misses generated from a SIGMA trace file with/without filters for internal padding

Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 **IEEE**

5. Related Work

Execution-driven simulators [11,12] were first developed for MIPS processors. They first developed the technique of modifying a program binary to collect performance statistics while running the program on the native processor. Subsequently, Torrellas *et.al*., [13] extended MINT to collect timing characteristics of Intel processors, which was ported to PowerPC processors (Augment6k) by Giampapa [14]. All these simulators were concerned with timing characteristics of program segments. They trapped memory access instructions and transferred control to a backend that can simulate desired memory architecture. Non-memory access instructions were run on the native processors and efficient techniques of estimating their timing were incorporated. Since our objective is not timing characteristics, we stripped Augmint6k to its barest essentials and incorporated the identification of blocks and offsets. Each memory access invoked a trace library routine that recorded the instruction identification and the address being accessed. For blocks that do not have memory accesses, special calls were made to the trace library to record the control flow properly. These enhancements enabled us to produce a trace that gives complete information to orchestrate the whole program on a modified architecture. Also none of the above simulators linked the memory references to symbolic data structures and subroutines in the source program. This was another crucial aspect of the SIGMA approach.

There have been some tools that access hardware performance counters. For Intel platforms, Vtune[15] is available. PAPI[7] provides a multi-platform interface to access hardware counters. However, these approaches only provide counters of data or sampling among code regions. In contrast, SIGMA provides detailed information about individual memory references, **and** the actual memory addresses being accessed.

Other systems have taken advantage of the flexibility provided by the hardware to add instrumentation of data-centric caches. ATUM [16] uses the ability to change the microcode in some processors to collect memory reference information. The FlashPoint [17] system uses the fact that the Stanford FLASH multiprocessor [18] implements its coherence protocols in software, allowing instrumentation to be added at this level. Buck and Hollingsworth [19] proposed using interrupt on overflow to sample the addresses of data cache misses, but this approach does not provide the level of detail provided by SIGMA. Mtool [20] provides information about the amount of performance lost due to the memory hierarchy, but only relates this information back to program source lines, not to data structures. A system with more similarity to the techniques in this paper is MemSpy [21], which provides data-oriented information as well as code-oriented, but it uses simulation to collect its data. StormWatch [22] is another system that allows a user to study memory system interaction. It is used for visualizing memory system protocols under Tempest [23], a library that provides software shared memory and message passing. However, the goal of StormWatch is to study how to adapt a memory system protocol to suit the application, rather than how to change the application to match the memory system. Because of this, the information provided is also different. This information includes what protocol events are taking place, what code is causing them, and how they are related.

6. Conclusions

In this paper we presented the Simulation Infrastructure to Guide Memory Analysis (SIGMA), a new data collection framework and family of cache analysis tools that provide detailed cache information by gathering memory reference data using software-based instrumentation. This infrastructure can facilitate quick probing into the factors that influence the performance of an application by highlighting bottleneck scenarios including: excessive cache/TLB misses, false sharing, and inefficient data layout. This framework can also assist in perturbation analysis to determine performance variations caused by architectural or program changes.

Our simulation infrastructure performs run-time trace compression that produces substantial reductions in trace sizes, as evidenced by the statistics presented in our experiments. The compression algorithm works particularly well with programs that have regular memory access pattern. In these cases, the compression rates were over 20000. With the applications from the SPEC benchmark, we observed an average of compression rate of three orders of magnitude.

Our validation tests using micro-benchmarks demonstrated that the performance metrics obtained with SIGMA are close to the expected values and comparable to the numbers obtained with hardware performance counters. Using the SPEC Swim benchmark as a test case, we observed that most of the performance metrics obtained with SIGMA were in the same range of the values collected with hardware performance counters, with the advantage that SIGMA provides information at a data structure level, as specified by the programmer. Such feature is not possible with the current state of the art tools that use the hardware counters. Finally, we described the use of filters for prediction of performance from perturbation analysis, such as padding of data structures, where we observe an error of measurement of less than 0.01%.

Proceedings of the IEEE/ACM SC2002 Conference (SC'02)
0-7695-1524-X/02 $17.00 © 2002 **IEEE**

References

[1] L. DeRose and D. Reed. “SvPablo: A Multi-Language Arquiteture-Independent Performance Analysis System”. In *Proceedings of the International Conference on Parallel Processing,* pages 311-318, August 1999.

[2] D. Reed, R. Aydt, R. Noe, P. Roth, K. Shields, B. Schwartz, and L. Tavera. “Scalable Performance Analysis: The Pablo Performance Analysis Environment”. In *Proceedings of the Scalable Parallel Libraries Conference,* IEEE Computer Society, 1993.

[3] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. “The Paradyn Parallel Performance Measurement Tools”. *IEEE Computer,* 28(11) pages 37-46, November 1995.

[4] B. Mohr, A. Malony, and J. Cuny. “TAU Tuning and Analysis Utilities for Portable Parallel Programming”. In G. Wilson, editor, *Parallel Programming using C++*, M.I.T. Press, 1996.

[5] J. Yan, S. Sarukkai, and P. Mehra. “Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs Using the AIMS Toolkit”. *Software Practice & Experience,* 25(4) pages 429-461, April 1995.

[6] M. Zagha, B. Larson, S. Turner and M. Itzkowitz. “Performance Analysis Using the MIPS R10000 Performance Counters”. In *Proceedings of Supercomputing'96,* November 1996.

[7] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. “A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters”. In *Proceedings of Supercomputing’00*, November 2000.

[8] R. Berrendorf, Heinz Ziegler, and Bernd Mohr. “PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors”. Research Centre Juelich GmbH, http://www.fz-juelich.de/zam/PCL/ Version 2.1, February 2002.

[9] L. DeRose. “The Hardware Performance Monitor Toolkit”. In *Proceedings of Euro-Par,* pages 122-131, August 2001.

[10] R. Sadourny. *“*The Dynamics of Finite-Difference Models of the Shallow-Water Equation”. In *Journal of Atmospheric. Sciences*, 32(4), April 1975.

[11] S. Herrod. “Tango lite: A multiprocessor simulation environment”. In Stanford University, Computer Systems Laboratory, Technical report, http://citeseer.nj.nec.com/herrod93tango.html. 1993.

[12] J. Veenstra and R. Fowler. “MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors”. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 201--207, Durham, NC, January--February 1994.

[13] A. Nguyen, M. Michael, A. Sharma, and J. Torrellas. “The augmint multiprocessor simulation toolkit for intel x86 architectures”. In *Proceedings of the International Conference on Computer Design*, 1996.

[14] M. Giampapa. “Augmint6k: The Augmint multiprocessor simulation toolkit for IBM PowerPC architecture”. IBM Internal Report, 1998.

[15] Intel Corporation, http://developer.intel.com/software/products/vtune/index.htm.

[16] A. Agrawal, R. L. Sites, and M. Horowitz, “ATUM: A New Technique for Capturing Address Traces Using Microcode”. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*. June 1986, pp. 119-127.

[17] M. Martonosi, D. Ofelt, and M. Heinrich, “Integrating Performance Monitoring and Communication in Parallel Computers”. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. May 1996, Philadelphia, PA.

[18] J. Kuskin, *et al.*, “The Stanford FLASH Multiprocessor”. In *Proceedings of the 21st International Symposium on Computer Architecture*. April 1994, Chicago, IL, pp. 302-313.

[19] B. Buck, and J. K. Hollingsworth, “Using Hardware Performance Monitors to Isolate Memory Bottlenecks”, In *Proceedings of Supercomputing'02,* November 2002.

[20] A. J. Goldberg and J. L. Hennessy, “MTOOL: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications”. *IEEE Transactions on Parallel and Distributed Systems*, 1993, pp. 28-40.

[21] M. Martonosi, A. Gupta, and T. Anderson, “MemSpy: Analyzing Memory System Bottlenecks in Programs”. In *Proceedings of the 1992 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. June 1-5, 1992, Newport, Rhode Island, pp. 1-12.

[22] T. M. Chilimbi, T. Ball, S. G. Eick, and J. R. Larus, “StormWatch: A Tool for Visualizing Memory System Protocols”. In *Proceedings of Supercomputing '95*. December 1995, San Diego, CA.

[23] S. K. Reinhardt, J. R. Larus, and D. A. Wood, “Typhoon and Tempest: User-Level Shared Memory”. In Proceedings of the *ACM/IEEE International Symposium on Computer Architecture*. April 1994.

0-7695-1524-X/02 $17.00 © 2002 **IEEE**