MODA A Framework for Memory Centric Performance Characterization

Sunil Shrestha
University of Delaware
Newark, DE
shrestha@capsl.udel.edu

Chun-Yi Su
Virginia Polytechnic Institute
Blacksburg, Virginia
sonicat@vt.edu

Amanda White
Pacific Northwest National Laboratory
Richland, Washington
amanda.white@pnnl.gov

Joseph B. Manzano
Pacific Northwest National Laboratory
Richland, Washington
joseph.manzano@pnnl.gov

Andres Marquez
Pacific Northwest National Laboratory
Richland, Washington
andres.marquez@pnnl.gov

John Feo
Pacific Northwest National Laboratory
Richland, Washington
john.feo@pnnl.gov

ABSTRACT

In the age of massive parallelism, the focus of performance analysis has switched from the processor and related structures to the memory and I/O resources. Adapting to this new reality, a performance analysis tool has to provide a way to analyze resource usage to pinpoint existing and potential problems in a given application. This requires (1) memory trace collection with minimal perturbation of the application's behavior; (2) data management of multiple gigabyte and terabyte size trace files; (3) efficient data analysis and visualization of traces; and (4) the introduction of the target architecture's memory model into the analysis module for a truly memory-centric view. These features enable an application developer to anticipate algorithmic and structural resource bottlenecks on a small scale before a full scale roll out into production.

This paper provides an overview of the Memory Observant Data Analysis (MODA) tool, a memory-centric tool first implemented on the Cray XMT supercomputer along with the solution to above mentioned challenges. Throughout the paper, MODA's capabilities have been showcased with experiments done on matrix multiply and Graph-500 application codes.

1. INTRODUCTION

The multi-core revolution aims to achieve performance gains from one generation to the next by exploiting ever more thread- and data-parallelism available in current applications. Contrary to the performance gains achieved by superscalar design improvements in the past, multi-core systems require changes in programming models, system software and software tools, in order to harness the new found levels of parallelism. Tools, in particular, are required to automatically identify opportunities for parallelism, from the compilation phase up to the post-execution analysis phase. Current performance analysis tools tend to be control centered, a legacy that has its origin in the previous generation of architectures. These tools are good in pinpointing threads and code regions that are culprits of performance bottlenecks but provide little detail on the resources involved in the program execution. For example, current performance analysis tools can determine if a code section is memory bound by measuring CPU utilization and memory references over a specified time interval. These tools will attribute the performance bottlenecks to a set of threads but will not provide further details on the memory subsystem beyond caches.

This type of analysis was not necessarily a problem for previous architectures where resources like memory, network and I/O components were tightly coupled to a single or few cores in a bijective fashion. Identifying a resource problem, as in our previous example, automatically meant that the culprit could be identified as well. Yet with the advent of multi-core systems, to achieve some level of system balance, resources needed to be replicated as well. In the case of memory, that means multiple memory subsystems attached over multiple channels to a pool of cores and(or) processors. In addition, muticore systems created an execution paradigm shift from being compute bound to memory, bandwidth and I/O bound. Control-centric analysis in a such case can identify a memory bound problem, but won't provide clues about the root causes.

Similary, cache analysis provides information about cache hit/miss and the volume of traffic going in and out of the network. However, these metrics are not enough to pinpoint network congestion or memory bank conflicts. Moreover, other memory characteristics, like sharing a memory line, going back and forth between memory pages, etc, can cause great performance degradation. These architectural hotspots can be exacerbated by algorithmic bottlenecks which do not become apparent until certain resource preset limits, (e.g. page sizes, network buffer sizes, etc) are reached. Without an architectural memory and network model in their designs, processor-centric performance tools may overlook

these pathological cases for smaller runs that will only become symptomatic once a scale-up/out is performed. Resource centric performance analysis tools are therefore a good addition in the tool arsenal to identify potential over- and under-subscribed resources early on. In particular, we will focus in this paper on the discussion of a novel memory centric performance analysis tool.

As a target for our resource-centric tool development we chose the Cray XMT [2]. Several reasons contributed to that choice: applications developed for this architecture tend to exhibit large amounts of concurrency, irregular memory access pattern and use large amounts of memory capacity. Memory-subsystems are shared across processors, and data caches are non-existent due to the lack of spatial and temporal locality for irregular applications. Each processor supports 128-way SMT concurrency in hardware and fine-grained synchronization in memory. Up to 8192 of these processors can be arranged into a 3D Torus connected with a high speed Cray XT network. In such an environment, a memory-centric performance analysis tool can be developed at large enough scale to gauge early on its analysis usefulness and at the same time identify any detrimental impacts on the application under test.

This paper showcases a memory-centric performance tool called the Memory Observant and Data Analysis Framework, or MODA [8] for short. It is designed to reveal existing and potential algorithmic and architectural resource hot-spots by means of a sophisticated memory model. The tool helps to identify performance degradation factors at a small scale where debugging and performance analysis is more manageable. Salient features of this tool include (1) a memory trace collection with minimal perturbation of the application's behavior; (2) data management of multiple gigabyte and terabyte size trace files; (3) efficient data analysis and presentation of traces; and (4) the introduction of the target architecture's memory model into the analysis module for a truly memory centric view.

Due to all these features, MODA is designed to fill a gap realized by the lack of tools that pinpoint contention on the memory system, which has been open since many/multi core designs came to dominance.

The paper is organized as follows: Section 2 presents background and related work. Section 3 presents the framework of the MODA tool. Section 4 provides an overview of the experimental testbeds. Finally, Section 5 shows the conclusions and future work for the given tool.

2. BACKGROUND

The advent of massive multi-core/threading processing interacting with shared resources, paired with novel programing and execution-model paradigms poses new analysis challenges. Determining or predicting shared resource bottlenecks from a thread perspective is now more cumbersome. In light of our efforts to develop resource-centric tools, we briefly assess the capabilities of some prominent performance tools.

In its current form, the Cray XMT ships by default with the Apprentice-2 tool-suite[10] that consists of a compiler analysis tool, a block-profiler and a trace analyzer. It provides a trap-centric analysis of outstanding events. However, this might overlook pathological behaviors that are only seen in large data sets and its predeterminate thresholds limits scalability and the data sets that can be run on them.

Several post-execution analysis tools are presented next. *Scalasca's CUBE*[6] presents large-scale profile and trace data over a 3-dimensional data space that spans metrics, code segments and process/physical topology. Analysis is driven by the metrics, the program phases or the process/topology. *TAU's PerfExplorer*[4] introduces hierarchical analysis through clustering and event filtering. Correlation analysis helps to find performance bottlenecks. *Rice's HPCToolkit*[11] stands out for its sophisticated logical call path analysis, necessary to assign performance attribution in great detail to code segments.

Next, the *Totalview*[3] framework has some memory and data centric functionality; however, this is mostly geared towards logical debugging (e.g., memory leaks, dangling pointers). Another salient feature includes the tool's grouping capabilities to enable an hierarchical analysis approach. However, these groups are still built in a processor-centric fashion. Other examples include *Jumpshot*[15] and *DEEP/MPI*[13] that show some similarity to the tools above by addressing visualization scalability aspects.

Worth mentioning are novel environmental data capturing capabilities for *PerfTrack*[7]. Here, shared resources are power and thermal machine envelopes.

Finally, we have mainstream tools such as VTune, METRIC and ThreadSpotter. *Intel VTune* [5] provides a mechanism to do thread profiling, hardware event and stack sampling. It can also provide information at instruction level which can be used to find out pipeline stalls and analyze thread performance. *METRIC* [9] traces partial memory accesses which is used for memory hierachy simulation and cache analysis. This helps in characterizing the application's memory usage but its analyis doesn't go beyond caches. *ThreadSpotter* [12] helps to analyze memory bandwidth, latency, data locality, thread communications and detect performance issues.

In general terms, the above mentioned tools exhibit process/thread centric performance analysis/visualization. Resource-centric performance analysis is scant or non-existent. We believe that a new breed of resource-centric tools will provide new performance analysis insights, especially in multi/many core architectures where a control centric view might not prove sufficient.

3. MODA FRAMEWORK

An overview of MODA is shown in Figure 1. MODA is designed to instrument and monitor applications by creating traces that are subsequently analyzed and visualized to reveal their memory usage pattern.

Under the MODA framework, analysis can be done beyond the virtual address space. For all instrumented memory addresses, MODA can present memory usage patterns in terms of time, processors, streams, network nodes and memory banks. This is invaluable information that can be used to

pinpoint the source of performance degradation. Unlike processor centric tools (Section 2) which would only show if the application is memory bound, MODA presents a clear picture of execution evolution with a precision down to individual memory banks.

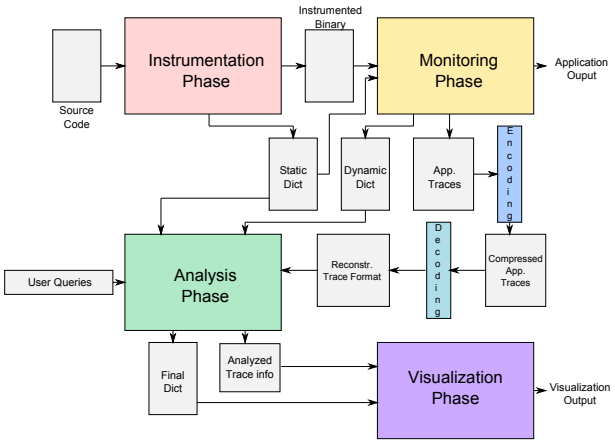


Figure 1: A High Level Overview of the MODA Framework

3.1 Instrumentation Phase

During MODA's first phase, there are three main steps that must be taken to prepare the application binary.

In the first step, using the Cray XMT's capabilities to set user-defined memory traps [2], the framework marks (i.e. instruments) certain data structures to be monitored during program execution. Static and global variables selection is done by extracting variable information using an ELF reader. Similarly, heap variables are instrumented by replacing calls to malloc and free with MODA's allocator wrapper functions to record memory usage during execution. Later on, this information is mapped to the collected traces so that a bottleneck's source can be linked to a source-level data structure.

In the second step, MODA binary-rewrites the first few lines of the user-level trap handler segment to redirect the runtime to the highly optimized monitoring kernel.

During the final step, the framework creates two identical versions of the executable code. Temporal (a.k.a. statistical) sampling is achieved by enabling tracing in one version of the code and by disabling it in the other. Sampling is turned on or off by jumping between the versions and updating the respective control structures (see Section 3.2.1 for more details).

3.2 Monitoring Phase

Upon execution of the instrumented binary, the framework initializes its runtime components. The framework instruments available data structure information from the previous phase. In this way, MODA may select a subset of data structures to monitor. Together with temporal sampling, this feature gives MODA the ability to fine tune the tradeoffs between accuracy and incurred monitoring overhead in both spatial and temporal dimensions.

Next, the framework allocates tracing buffers in low latency memory for each possible parallel entity. In the next stage, a group of helper threads known as "reconstruction streams" are created to consume any newly created traces. Then, the framework starts executing the application code.

During the application's execution, if MODA's allocator wrapper is called the new data structure is instrumented when its memory is allocated and de-instrumented when the memory gets de-allocated.

When an instrumented data structure is accessed by the application's code, it causes a user-level trap. The runtime trap handler recognizes the trap and passes it to the monitoring kernel. Trace events are generated and stored in the trace buffer which is read by the reconstruction streams in parallel. In an effort to reduce trace event message sizes, the tracing message format exploits the reduced dynamic range of an event's descriptive attributes in relation to a previous event, including time, code and data addresses of a memory reference. Posting difference information on occasion can yield a compression factor of three per message.

Since the reconstruction streams run in "parallel" with application streams, their data movement and manipulation has minimal perturbation to the system (*Feature 1*). The ratio between application versus reconstruction streams is an important tuning parameter for the MODA framework. Too many application streams will overwhelm reconstruction streams quickly and too few will slow down the application significantly. To overcome this problem, MODA implements a special type of temporal sampling that helps prevent buffer overflow.

3.2.1 Statistical Temporal Sampling

As described in Section 3.1, the duplicated executable schema is used to allow temporal sampling. One approach investigated in the current MODA framework is to use the executed instructions count as a parameter to enable or disable sampling: i.e number of instructions executed in the original and the duplicate version of the code determines average trace sampling rate TSR. For example, $TSR = 1$ would mean, only instructions from the original version of the code with tracing enabled are executed. Similarly, $TSR = 1/100$ would mean, for every trace enabled execution, we execute 100 trace disabled instructions.

$$ReadRate_{RS} > FillRate_{MK} \quad (1)$$

$$FillRate_{ii} = F_{max} * IIR * MRF * LSD \quad (2)$$

$$FillRate_{c} = F_{max} * LMA * RLS \quad (3)$$

$$X = min[FillRate_{ii}, FillRate_{c}] \quad (4)$$

$$FillRate_{MK} = \frac{X*TOS*TNS}{(TSR*TNS+(1-TSR)*TOS)} \quad (5)$$

In order to investigate TSR, we developed a conservative parametric model of the fill rates for each "Parallel Trace Buffer". Equation (1) states the requirement that the reconstruction streams read rate $ReadRate_{RS}$ has to exceed the monitoring kernel fill rate $FillRate_{MK}$ driven by the application. Equations (2) and (3) derive the theoretical fill rates from the vantage of two constraints: Equation (2) computes

TSR	$FillRate_{MK}$	Overhead Factor
0	0.5	2
1/100	0.411	∼2.43
1/25	0.268	∼3.72
1/10	0.158	∼6.3
1	0.022	45

Table 1: Overhead given an average tracing rate. The formulas presented in Equations (1 - 5) were used to get these numbers with the following values: $TOS = 1/45$, $TNS = 1/2$, $IIR = 1/21$, $MRF = 3$, $LSD = 1/2$, $LMA = 1/100$, $RLS = 1/5$ **and** $X = 1MHz$

the maximum memory instruction issue rate $FillRate_{ii}$ as a function of the processor's frequency F_{max}, the thread's maximum instruction issue rate IIR, the thread's maximum capability to have multiple memory references in flight MRF and finally the thread's maximum memory reference instruction i.e., Load Store density LSD. Equation (3) determines the rate constrained by the memory subsystem $FillRate_c$, parametrized by local memory access latency LMA and a conservative penalty factor for average remote latencies RLS. Plugging in the numbers reveals that the $FillRate_{MK}$ is limited by the memory subsystem in the form of X as shown in equation (4). Finally, in equation (5) the $FillRate_{MK}$ is expanded by considering the slowdown attributed to the MODA framework itself. Here, the TSR modulates the overhead incurred by either recording the event with trap overhead per sample (TOS) or discarding the event with trap overhead per non-sample (TNS).

Using these formulas with the current values given by the Cray XMT architecture gives the results in Table 1. This tells us that using sampling can reduce the overhead of the framework substantially and prevent the overflowing of the tracing buffers. In other words, if we sample every 100th trace event ($TSR = 1/100$) theoretically produces an additional 2.43x overhead as compared to the non traced execution. The dominant factor in this case is TNS. However, sampling every memory fetch ($TSR = 1$) would yield a 45x overhead, determined by the dominant factor TOS. Hence, reducing TNS and TOS are primary goals in the design of the MODA framework (*Feature 1*).

3.3 Data Post-Processing

During post processing, the MODA framework collects the messages recorded during the monitoring phase. However, the amount of data collected can easily reach the upper gigabytes and terabytes data range. An efficient way to organize and transfer these traces across computer systems is to take advantage of data properties to organize and compress them so that they can later be decoded for analysis and visualization.

3.3.1 Encoding and Decoding

In order to take care of the massive data size, we introduce the Parallel Compression Encoder/Decoder (PCED) system (*Feature 2*). PCED takes advantage of lack of interdependencies between trace chunks to achieve embarrassingly parallel encoding and decoding. It aims to preserve maximum possible memory reference patterns in a manageable size trace file.

PCED's current version implements byte-stream encoding/decoding but does not fulfill redundancy optimization component yet.It exhibits a compression ratio of 3 .65bit/Byte on average. In practical terms, this means that MODA can compress a 4 GB trace file to 1.8 GB.

3.3.2 PCED Framework

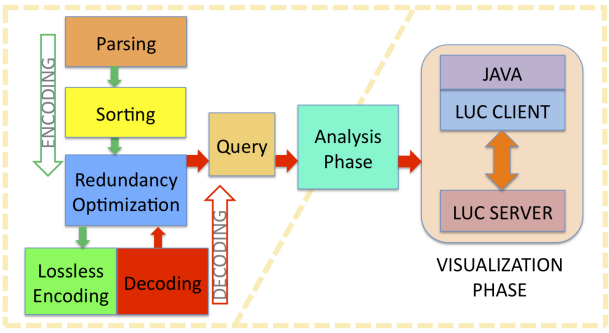


Figure 2: Trace File Manipulation: The encoding/decoding pathways and visualization

A high level overview for the PCED system can be found in Figure 2. PCED begins with a parsing component that recreates all the necessary data from the tracing messages produced by the reconstruction streams. Following the parsing stage, the PCED system sorts the reconstructed trace data in time order. Sorting helps to bring similar traces in symbol-level closer which is helpful to find more redundant information. The sorted data is then passed to a redundancy optimization step which eliminates redundant information and is followed by our lossless encoding component that uses an Adaptive Arithmetic Coding algorithm[14].

Our choice of PCED over other algorithms stems from PCED being embarassingly parallel and making near-optimal predictions using small chunks of uncompressed traces. PCED achieve speed up to 37.26 MByte per second which makes it more applicable to online interactions with the high speed monitoring kernel.

3.3.3 Visualization

In order to enable a human analysis of the memory traces generated by MODA, the visualization phase permits visual introspection of memory usage patterns. This visualization phase uses a Java-based GUI which allows users to extract saved memory trace files produced by MODA and plot memory access patterns in terms of time, processors, streams, processor nodes or memory banks. Other features include zooming in and out of chart regions, visual tips of data set properties and distribution of accesses according to source code level data structure(*Features 3,4*).

MODA implements a client / server interface which provides a set of remote procedure calls (RPCs) to handle the visualization requests. The client receives the user requests from the GUI and forwards them to the server over a high speed network. The server runs on the XMT computation nodes. It serves different requests from the user and asks the analysis engine to compute the result and send it back to client. Figure 3 shows an example of the GUI which displays the

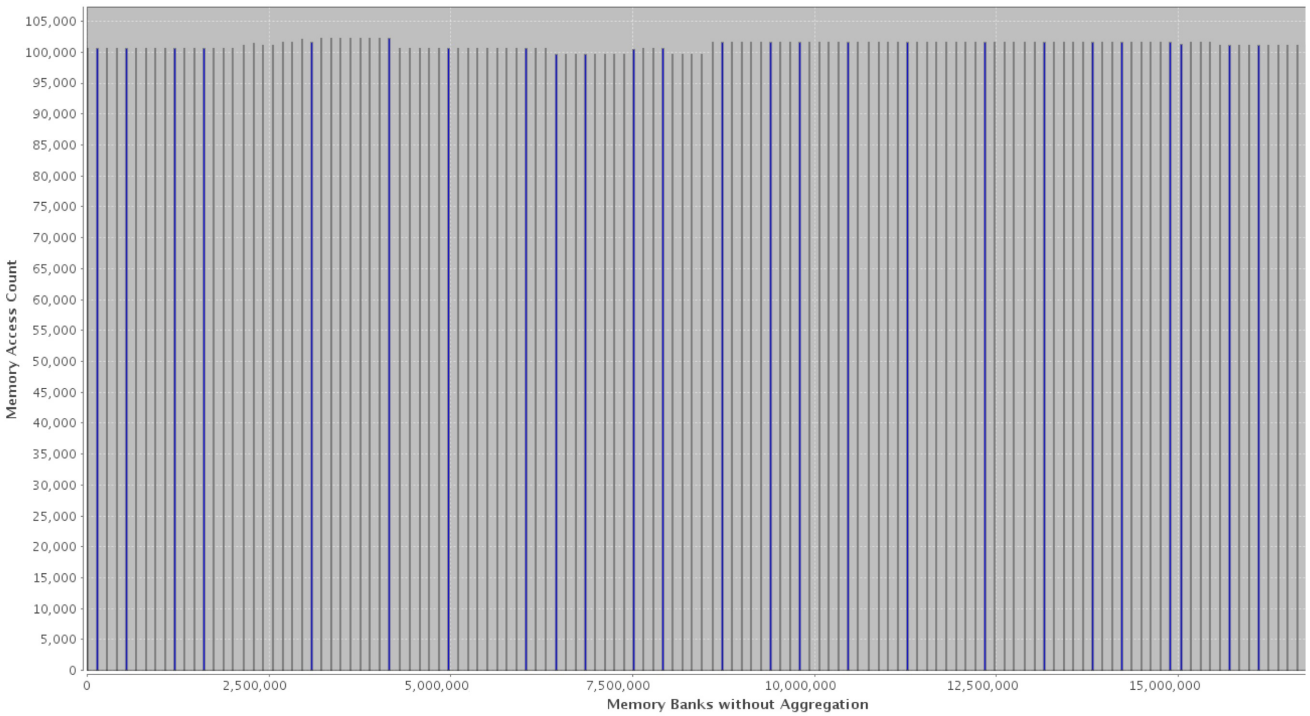


Figure 3: An example of the GUI: It shows the data distribution in Y-axis over the XMT memory banks shown in X-axis for a matrix multiply of 256 by 256. Memory banks are uniformly filled across network nodes(*Features 3,4*)

memory bank usage on the Cray XMT architecture for a regular application (in this case, Matrix Multiply).

3.4 MODA Memory Model

One of the main characteristics of MODA is its awareness of the architectural memory model (*Feature 4*) as demonstrated in Figure 3. During the postprocessing phase, the data structure information (that are collected during the instrumentation and monitoring phases) is compressed to the encoded file. When the next phase starts (visualization), the MODA visualization server reads this information and the traces and uses the machine address translation mechanisms to map it to physical addresses. Afterwards, a mapping (using the machine's memory characteristics, like bank sizes, pages, number of DIMMs, etc) of the physical addresses to the architecture's memory structures takes place. Afterwards, the results are returned to the GUI for visualization.

4. DISCUSSION

In this section we exercise our MODA tool with two example applications on our Cray XMT target platform. The configuration used in this study supports up to 128 processors, 1 Terabyte of shared memory and runs at 500MHz frequency. On the software side, we selected two applications: Matrix Multiply and Breadth-First-Search (BFS). Both applications are compiled with the XMT C compiler version 6.5.0 with all automatic parallelization features enabled.

We show memory access counts for the memory subsystem, variables and over-time to provide information about resource usage. We also detail cost overheads for each MODA component.

4.1 Monitoring Kernel Collection Overhead and Sampling

In the XMT architecture, an un-instrumented memory operation on average is around 700 cycles. In case that a memory cell is activated for tracing, the sophisticated piece of code incurs 45x overhead. However, with temporal sampling this overhead can be significantly reduced (as shown in table 1). Moreover, there are several optimizations that are applicable to the monitoring kernel.

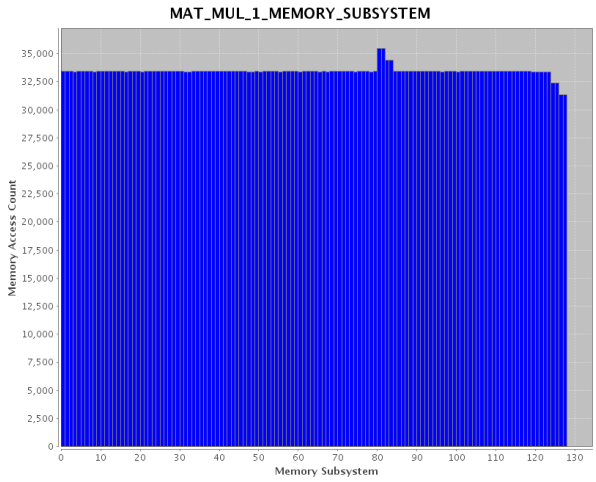
4.2 PCED Performance

The main components of the PCED sub system are the parallel encoding and decoding. These phases are composed of units that run in parallel and consume the given data. On average, an encoder unit (arithmetic encoder unit) will take 7.10 seconds to encode an input file of size 15.1 GB with 128 parallel processors. The decoder path on average will take 8.49 seconds per unit (arithmetic decoder unit) to decode the produced encoded file.

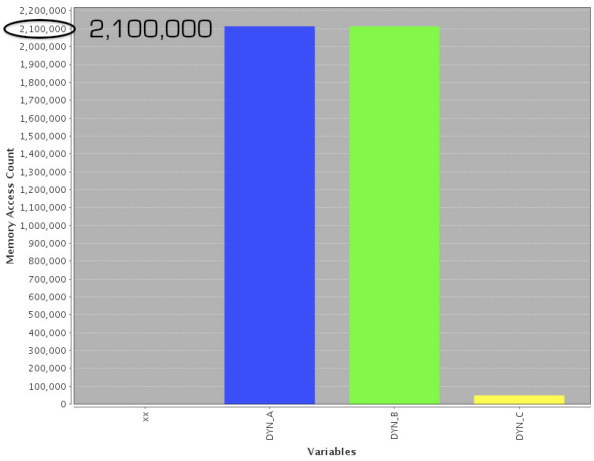
The highly efficient compression and decompression algorithms together with the high speed network provided by the system allows the interoperability between the GUI and the decoding part of the PCED's framework.

4.3 Selected Applications

We selected two different aplications which exhibit intrinsic characteristics suitable to exercise MODA. Matrix Multiply

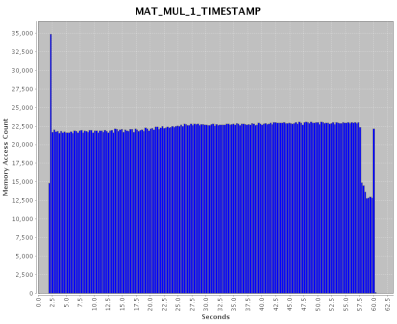


(a) AC (Y-axis) over Memory Subsystem (X-axis)

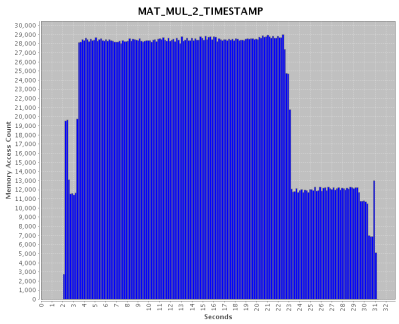


(b) AC (Y-axis) over Variables (X-axis)

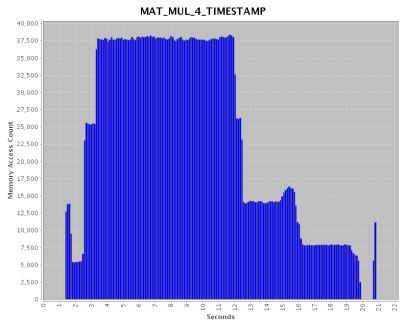
Figure 4: Memory Access Count (AC) for Matrix Multiply



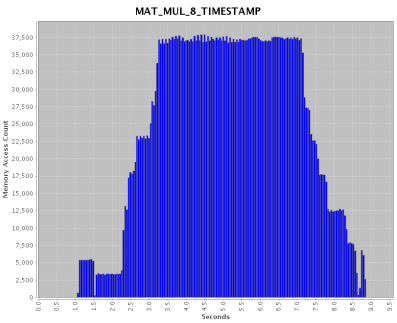
(a) 1P



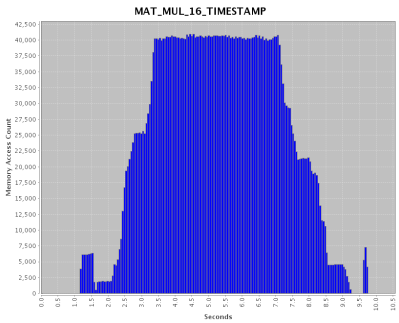
(b) 2P



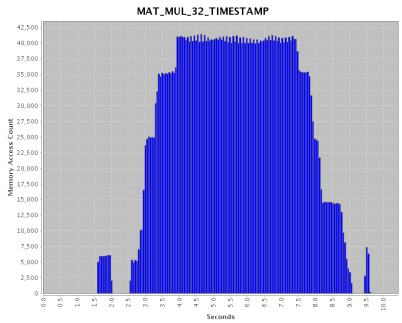
(c) 4P



(d) 8P



(e) 16P



(f) 32P

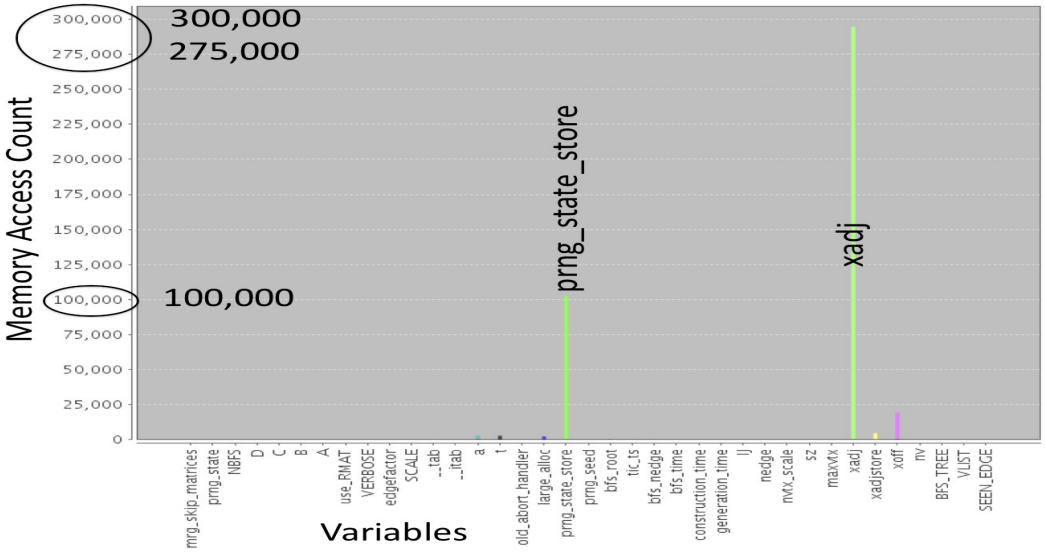
Figure 5: Memory Access Counts over Time (a-f) for Matrix Multiply

is a ubiquitous example of regular applications and Breadth-First-Search kernel retrieved from the Graph 500 benchmark [1] is famous for its irregular nature.

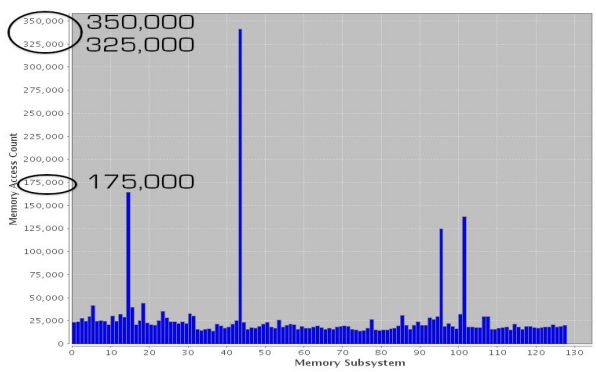
4.3.1 Matrix Multiplication

The first example we chose to exemplify MODA's capabilities is Matrix Multiply (using matrices of size 128 x 128) which was executed multiple times using a range from 1 processor to 32 processors – totaling up to 4k threads. This kernel serves also as a first step to validate MODA's behavior because its memory-access patterns are easy to predict with simple back-of-the-envelope calculations.

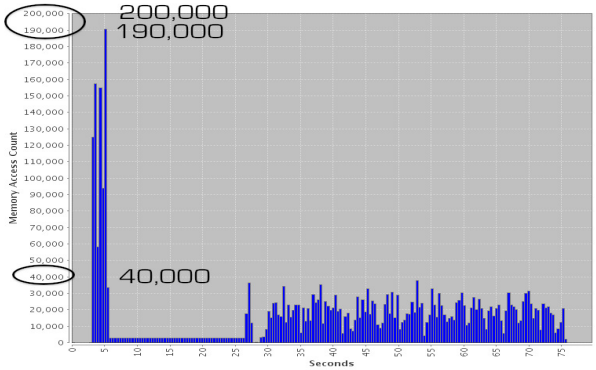
As shown in the Figure 4(a), there is a uniform distribution of memory accesses (Y axis) over the XMT's memory modules (X axis) for the one processor case. This is expected since the XMT has a hardware randomizer for memory addresses. Given the well-defined number of accesses per matrix element in our simple matrix multiply and the layout of these elements in contiguous address space, we expect to see this uniform distribution. Figure 4(b) shows the variable access count, represented by DYN_A, DYN_B and DYN_C.



(a) AC (Y-axis) over Variables (X-axis)



(b) AC (Y-axis) over Memory Subsystem (X-axis)



(c) AC (Y-axis) over Time (X-axis)

Figure 6: Memory Access Counts (AC) for Breadth-First-Search (BFS) using 32P

The radical reduction of acceses to the DYN_C variable is due to a simple optimization applied to the kernel code in which the partial results of the C matrix (DYN_C) are kept in a register for the lifetime of the inner-most loop. Overall, we observe good agreement between measured and expected pattern.

Figure 5(a) represents the Matrix Multiply's memory access count over time in seconds. Here, the first visible peak region occurs during matrix initialization. The next region shows almost uniform counts for the computation part of the matrix multiply (as it should). Finally, the last peak region represents verification code for the application which tests the result matrix for consistent results. Figures 5(b) to 5(f) show similar experiments for processors 2, 4, 8, 16 and 32 on the time domain. Curiously, performing our scaling experiments and visualizing memory access counts shows a pyramid pattern shaping up. It starts slowly with few accesses, ramps up as the parallelism increases and finally tapers off when some of the computations are done while some are still in flight. This behavior is the result of data starvation as the application kernel approaches completion and can be affected by changes in the scheduling policy.

4.3.2 *Breadth-First-Search(BFS)*

We examine the behavior of a BFS implementation for the XMT. To avoid experimental bias, we use a third party code provided by the Graph 500 benchmark suite [1].

Figure 6(b) shows the memory access count across the memory subsystem. As expected, the access pattern is highly irregular. We see very high peaks on memory banks 14, 43, 95 and 101. In conjunction with Figure 6(a), which represents variable accesses, we see that certain variables are being accessed more often than others. Two special cases are *prng_state_store* and *xadj*. Both variables are pointers to memory blocks used inside the traversal loops. Although the compiler should be able to analyze and keep these variables in registers, it fails to do so, creating a hotspot. A perfect example is the *xadj* variable that has around 290,000 accesses (confirmed through MODA and independent experiments).

By rewriting the code slightly we coaxed the compiler to promote *xadj* to the register file, hereby reducing the number of accesses to a staggering 4000. As an additional note, we identified that *xadj* resides on memory bank 43 (highest peak in figure 6(b)). In summary, by inspecting MODA's visualized output, we were able to reduce contention on this variable by around 72x.

Finally, Figure 6(c) shows the memory access count over time. Initial peaks appear during the initialization phase when the graph is generated. The plateau in the middle phase is due to graph randomization. Next, as the actual BFS traversal and verification phases are executed, the access counts start to exhibit a repetitive jagged hull-curve. This phenomenon can be explained with the way the 2-dimensional loop iteration space is traversed during the BFS search. The compiler was not able to collapse the loop iteration space to a single loop, yielding threads with variable workloads derived from varying neighbor-list sizes. This information in turn can be used by the algorithm designer or compiler writer to devise new approaches for mitigation.

5. FUTURE WORK AND CONCLUSIONS

Multiple efforts are under way to enhance MODA's capabilities even further. Under development is function instrumentation to better correlate the timeline of our traces with the evolution of the program execution. Function instrumentation will also allow us to keep track of local stack frame variables and their role in the performance gain or degradation over the program lifetime.

Thanks to the similarity of memory patterns in traces, we are implementing a redundancy optimization component in the PCED framework. This technique will have a higher compression ratio by using the concept similar to H.264 P-frame motion estimation technique.

Despite the benefits already extracted out of MODA just by analyzing two important kernels, MODA's visualization and analysis capabilities are still in their infancy. Automatic hotspot detection and 3-D visualization are just some of many new features envisioned. Finally, as mentioned in the introduction, we are currently investigating porting MODA to other architectures that might also derive benefit from a resource-centric analysis tool.

In summary, this paper has showcased the MODA framework: It gave a cursory overview into the design rationale and implementation of its components, exploiting parallelism opportunities in great number. The paper introduced the reader to MODA's potential by showcasing hot-spot detection uncovered in important kernel applications. We hope that this paper has shown the need for resource centric tools which are needed in the new many/multi core era.

6. ACKNOWLEDGEMENTS

This work was supported by the Center for Adaptive Supercomputing Software at Pacific Northwest National Laboratory. Special thanks to the very supportive staff at Cray, in particular Mike Rinkenburg, in charge of the XMT compiler.

7. ADDITIONAL AUTHORS

Additional authors: Kirk W. Cameron (Virginia Polytechnic Institute, Blacksburg, Virginia, email: cameron@cs.vt.edu) and Guang R. Gao (University of Delaware, Newark, Delaware, email: ggao@capsl.udel.edu)

8. REFERENCES

[1] The graph 500 list. http://www.graph500.org/.

[2] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM.

[3] C. Gottbrath. Eliminating parallel application memory bugs with totalview. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 210, New York, NY, USA, 2006. ACM.

[4] K. A. Huck and A. D. Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 41, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Intel. *Vtune Performance Analyzer Essentials*. Intel Press, 2005.

[6] F. Juelich. Cube: Cube uniform behavioral encoding. http://www.fz-juelich.de/jsc/kojak/components/cube/.

[7] K. L. Karavanic, J. May, K. Mohror, B. Miller, K. Huck, R. Knapp, and B. Pugh. Integrating database technology with comparison-based parallel performance diagnosis: The perftrack performance experiment management tool. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 39, Washington, DC, USA, 2005. IEEE Computer Society.

[8] J. B. Manzano, A. Marquez, and G. G. Gao. Moda: A memory centric performance analysis tool. In *In Proceedings of 11th LCI International Conference on High-Performance Clustered Computing*, Pittsburgh,PA, USA, 2009.

[9] J. Marathe, F. Mueller, T. Mohan, S. A. Mckee, B. R. D. Supinski, and A. Yoo. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems*, 29, 2007.

[10] M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz. Cray performance analysis tools. In *Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*. SpringerLink, 2008.

[11] Rice. Hpctoolkit. http://hpctoolkit.org.

[12] Roguewave, Rogue Wave Software, Inc. *ThreadSpotter*, 2011.

[13] C. B. Software. Deep/mpi. http://www.crescentbaysoftware.com/deep_mpi_top.html.

[14] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30:520–540, June 1987.

[15] O. Zaki, E. Lusk, and D. Swider. Toward scalable performance visualization with jumpshot. *High Performance Computing Applications*, 13:277–288, 1999.