MODA: A Memory Centric Performance Analysis Tool

Joseph B. Manzano
Pacific Northwest National Laboratory
joseph.manzano@pnl.gov

Andres Marquez
Pacific Northwest National Laboratory
andres.marquez@pnl.gov

Guang G. Gao
Department of Electrical and Computer Engineering
University of Delaware
ggao@capsl.udel.edu

Abstract

From one processor generation to the next, the mismatch in processing speed vs. memory and network access speed is exacerbating. The arrival of multi-core technologies accelerates this trend. The main mismatch drawback is that an increasing number of parallel applications suffer from latency and bandwidth bottlenecks incurred by accessing these shared resources. There is a need in the computer science community to identify these bottlenecks early in the development phase to better manage these scarce resources. Existing performance analysis tools emphasize a processor centric view that limits their capability to adequately analyze resource bottlenecks. Tools are required that can analyze resource access streams by taking into account the organization of the shared resource architecture structures.

This paper introduces the MODA memory centric performance tool prototype. MODA is designed to instrument, collect and analyze streams of memory requests. The tool provides a dynamic application behavior view of memory requests as they access various memory organizational constructs. MODA enables the detection of algorithmic and architectural resource conflicts that can greatly degrade performance. Moreover, possible scalability deterrents can be found predictively at relatively small data set scale.

Careful management of shared resource bandwidth will be one of the future multi-core technologies. The Cray XMT is ideally suited to serve as a test platform to analyze shared resource contention at massive scale. The initial prototype tool has been implemented for the Cray XMT multiprocessor and it has been used to analyze the behavior of small benchmark examples.

1 Introduction

Nowadays, multi-core gives a wealth of computational power with reasonable power consumption. However, the way to extract performance from these designs is still a challenge. Although system software and applications are making great progress, software tools are lagging behind. This is not to say that there are not excellent parallel tools out there (for a few examples, please refer to section 2). Nevertheless, these tools still put too much of their efforts on a processor centric view of a given application / architecture combo. This has been the way of thinking in performance analysis for a long time.

Especially due to the multi-core paradigm, the memory and network components of a machine are becoming critical resources. Several solutions have been proposed to decrease latency such as memory hierarchies and multi threading. Larger or multiple interconnect fabrics and different network topologies are continuously being proposed and tested to increase bandwidth utilization. However, due to the ever increasing number of cores on a chip, available bandwidth has to be rationed very carefully. Moreover, there are the architectural limitations of the memory and network themselves like memory physical boundaries, router's buffer length, etc. Due to these factors, possible architectural bottlenecks might appear. Complicating the issue, certain algorithmic bottlenecks might not become apparent until certain resource limits are reached. A bottleneck, or hot-spot, occurs when a resource is oversubscribed by an application or job. In the case of the network, an architectural bottleneck might represent a routing node in which its buffers are full. In

the case of the memory, this might be contention on the same memory node, DIMM, rank or page. This memory behavior may not be apparent to processor centric tools and it might not even appear until a large data set is used (e.g. when bank segments, or other memory structural barriers, are crossed).

Among the several parallel designs, the Cray XMT is a very special type of shared memory machine. It consists of massive multi threaded processors with no data caches, fine grained synchronization in memory and a large global shared memory, all arranged in a 3D torus network. As a highly parallel multi threaded machine with shared memory, it provides a great opportunity to study future massive multi-core shared memory machines. These systems will have an ever increasing number of cores and a pool of shared memory. Moreover, the Cray XMT can be used to experiment with communication and shared memory to a scale that is not possible in current shared memory machines. Due to these characteristics the Cray XMT can be used as a prototype for future shared memory machines. Moreover, the XMT uses a very light UNIX derivate OS. This OS is optimized for parallel execution environment and introduces very little noise. This means low OS noise and jitter. These features make the XMT a very attractive machine for this tool and these memory centric studies in general. A more complete description of the architecture can be found in [2].

In this paper, we propose a Memory Centric tool, which reveals existing and potential algorithmic and architectural hotspots. Such a predictive tool is useful because it allows a programmer to reason about contention at the development phase at small scale, reducing the likelihood of encountering contention surprises at larger scale where debugging and performance analysis is more onerous. It is classified as memory centric because it concentrates on the behavior of memory locations (e.g. variables on the high level source code) with respect with the underlying memory subsystem. Such analysis reveals certain hot-spots that may not be apparent to other tools until very large data sets and a large number of threads are used. As far as the authors know, no other performance analysis tools have tackled the contention problem from a resource centric approach

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 testbed from a software and hardware angles. Finally, Section 5 shows the conclusions and future work for the given tool.

2 Background

Most existing performance analysis tools are processor centric. Program execution evolution is captured and presented by these tools from the perspective of processes/processors/cores or threads. These tools excel under the assumption that the use of shared resources is most often temporally disjoint due to architectural or algorithmic enforcement. Under this scenario, shared resource use, such as memory or network access, tends to be concentrated in well-defined program-phases. Hence, analyzing shared resource use can be accomplished by analyzing particular phases in thread execution.

The advent of massive multi-core processing interacting with shared resources such as the memory hierarchy or the network subsystem introduces new analysis challenges. Shared resource usage between multiple threads appears now to be "seemingly" random and more difficult to attribute to specific phases. Determining or predicting shared resource bottlenecks from a thread perspective is more cumbersome. In light of our efforts to develop resource centric tools, memory centric in this paper, we briefly assess the capabilities of some prominent performance tools.

In its current form, the Cray XMT ships by default with the Apprentice-2 (v. 4.x) tool-suite[7] that comprises a compiler analysis tool, a block-profiler and a trace analyzer. A salient feature of the Apprentice-2 suite stems from the Cray XMT's unique execution model: Memory references are trapped if they are re-dispatched multiple times because they violate the program's intended causal order or because they encounter hotspots. These trap events are reported in Apprentice-2 to help the programmer identify algorithmic or architectural performance bottlenecks. Trap events provide some memory centric analysis functionality. The drawback is that trapping is expensive and thresholds are predetermined before the run, limiting scalability assessments for varying data-sets and architecture configurations.

Scalasca's CUBE (v. 3.x)[1] presents large-scale profile and trace data over a 3-dimensional data space that spans metrics, code segments and process/physical topology. Scalasca's supported programming models include OpenMP, MPI and hybrid OpenMP/MPI codes. OpenMP constructs can be automatically instru-

mented with the OPARI source-to-source tool. Analysis is driven by the metrics, the program phases or the process/topology.

Like Scalasca, Vampir (v 5.x)[6] can monitor MPI, OpenMP and hardware counter events. Hierarchical trace views on a lightweight client that can be served from the production environment are supported, hereby avoiding large data transfers. Trace views are process driven.

The Paradyn's MRNet (V. 2.x) toolsuite[9] also addresses data transfer requirements. Hierarchical analysis is supported by creating focus groups. Hierarchical performance analysis is aided by a performance consultant. The consultant refines its search for performance bottlenecks by recursively testing hypothesis.

TAU's PerfExplorer (v. 2.x)[4] introduces hierarchical analysis through clustering and event filtering. Statistical tools such as correlation analysis help the search for performance bottlenecks.

As with the previous tools, Totalview[3] supports the MPI/OpenMP programming models. The Totalview framework has some memory and data centric functionality that is mostly geared towards logical debugging. Also of interest are the tool's grouping capabilities to enable an hierarchical analysis approach. Groups are built out of process elements.

Jumpshot (v. 4.x)[10] and DEEP/MPI[8] show some similarity to the tools above by addressing visualization scalability aspects.

Worth mentioning are novel environmental data capturing capabilities being developed for PerfTrack[5]. Here, the shared resources are power and thermal machine envelopes.

In general terms, above mentioned tools exhibit process/thread centric performance analysis/visualization. Resource centric analysis is scant or non-existent. We believe that a new breed of resource centric tools will provide new performance analysis insights.

3 Framework

The whole MODA framework is shown in figure 1. It is designed to instrument, collect, analyze and visualize information with minimal or no user help or interference. The main objective is to find bottlenecks in an application and to correlate them with variables in the source code, threads in the application and memory structures in the system. In its current state, it can correlate variables, threads and memory modules. However, extending this to multiple memory structures is under study. The framework consists of four components or phases that take care of each of the tool objectives (instrument, collect, analyze and visualize). Under the Cray XMT architecture, the tool takes advantage of the user-defined traps to collect runtime information. However, other approaches, such as binary rewriting, are being explored. Each phase will be discussed more in depth in the next subsections.

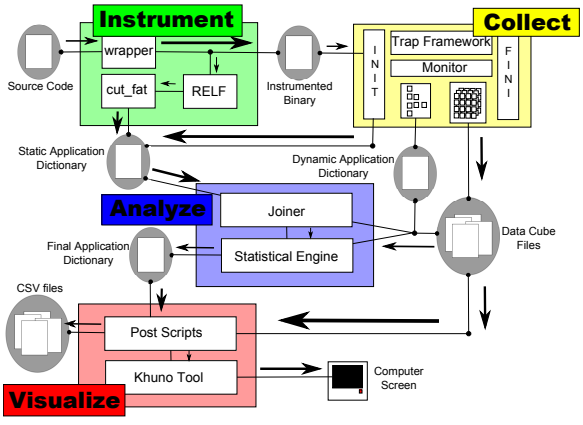


Figure 1: The MODA Framework

3.1 Instrument

The main component in this phase is a script that wraps around the Cray C Compiler. The wrapper will replace the application's main function with a framework defined function, compile the source and add the monitoring libraries before the final linking. Afterwards, it will call a homemade ELF tool (named RELF) which will read the symbol table. This produces a list of application names, sizes, virtual addresses and tags. In this context, a tag is a unique 64-bit value assigned to an address (or a range of addresses) which uniquely identify them on the framework. Moreover, for every tag there exists a corresponding variable (named or unnamed) in the source code. However, this list is *unclean* since it contains many of the runtime and monitoring variables. A small script will delete the extra variables and produce the application dictionary. In the framework, this dictionary is called the *Static Application Dictionary* since it is produced by the compiler.

3.2 Collect

This phase takes place while the application is running. It collects runtime and state information for the given variable list while the application is running in parallel. It depends on two structures: The *data cube* and the *Dynamic Application Dictionary*. The data cube is a 3-D table which will be filled with data from each running thread. The dynamic application dictionary is used to keep accesses of variables that were created during the execution of the application (i.e. heap allocated arrays).

When the application starts, it enters the framework-modified main function. This function will allocate nearby memory (for its data collection functions), set the trapping mechanisms for each variable contained in the static application dictionary and initialize the dynamic application dictionary. An entry in the dynamic dictionary is composed of a tag (which contains the processor id of whoever allocated it plus a unique identifier), the virtual address, the size of the block and an optional name. These entries will be registered to the monitoring framework at the time of their creation. When deallocated, these entries are not erased from the dictionary but marked as inactive. If the same memory range is reused then a new tag is assigned to the created address range. Currently, the framework will monitor all the heap allocated blocks created by the provided w_malloc function[1].

The data cube is used to save monitoring information about the running application. An entry in the cube can be seen as a 7-tuple consisting of the tag, operation data, hardware thread id, type of operation, time stamp and the address of the operation. All these entries are arranged by processor and hardware thread id in the cube[2].

When the application ends, it returns to the modified main. This function will clean up all the structures used by the monitoring framework and save the data cube and the dynamic dictionary to the disk.

3.3 Analysis, Post-processing and Visualization

In this phase, both dictionaries are joined together and some post-processing (i.e. endianness conversion) is applied to the data cube files. Visualization is joined with the analysis phase in the current version, but they will be two distinct phases in future iterations. During this phase, the information in the data cube is collected and organized into a master record which will be read by the visualization tool to obtain related information (threads versus accesses, memory module utilization of variables, etc). In its current iteration, it has two "portals". The first portal shows the variables that were used and their percentage of access in the whole execution. From this portal, the number of accesses per processor per variable can be obtained, as well as, the number of hardware streams from that processor that worked on that variable. The second portal is the processor / hardware thread one and shows the percentage of participation of each processor in the overall computation. Both portals have a text based user interface.

Other information (like module information and variable memory distribution) can be obtained from this phase. This information will be used to describe the examples presented in this paper.

[1]The framework also provides the w_free function that must be used to deallocate the blocks. Otherwise, the framework will be left in an inconsistent state

[2]The hardware thread id, which in XMT terminology is called a stream, is saved twice since there is a possibility that the thread is switched to another hardware stream while waiting for a long latency operation.

4 Experiments

The current MODA toolset has been tested with a small set of examples. In this section, two small examples will be introduced and their behaviors will be analyzed using the tool visualization and analysis phases. However, both the hardware testbed and software examples should be explained and expanded.

4.1 Hardware Testbed

All experiments were run on a 64 processor Cray XMT system. In this system, the processors are arranged into a 3-D torus. Each processor consists of 128 hardware threads (or streams). Each stream can issue a LIW operation. However, they share the same LIW 21-stage pipelines, which clocks at 500 MHz. Under the used configuration, each processor has 8 GiB of DDR DRAM memory running at 200 MHz. The LIW pipelines can only accept one LIW instruction from each stream. Thus, each stream has to wait at least 21 cycles before issuing its next LIW instruction[3]. Given the 128 streams and enough parallelism, the pipelines can be kept fully utilized. Each stream is composed of a set of 31 general purpose registers, 8 target registers (Registers used exclusively for jumping), a set of 8 trap registers and a status register (called the stream status word which contains the stream program counter). As stated before, the XMT is a shared memory machine. It possesses a large amount of memory (in the order of a tera byte for a 128 processor configuration) which is accessible to everyone in the system. Every address in the machine goes to an scrambling and distribution phase that ensures that all accesses are uniformly distributed across all available memory nodes in the system. Hereby contention is significantly reduced but not fully eliminated. Moreover, the actual memory word contains special bits which are used for fine grained synchronization, pointer forwarding, synchronization traps and user defined data related traps. The XMT has a high speed memory buffer on each memory controller to accelerate common accessed variables. The network used in the XMT is the proprietary Cray SeaStar2 which has a 3-D torus topology and static routing[2].

Thanks to its memory features and massive threading processors, the XMT is a prime candidate to run irregular applications in parallel. A block diagram of the architecture can be found in Figure 2.

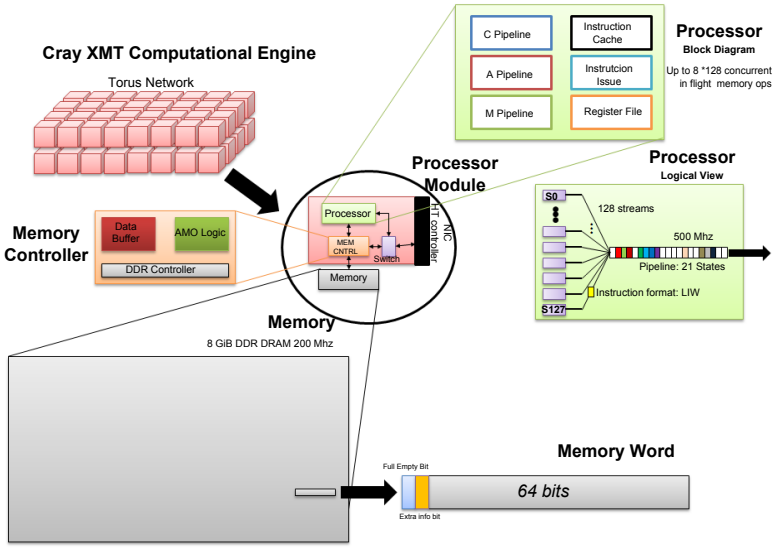


Figure 2: A logical view of the Cray XMT

4.2 Software Examples

Two small applications were selected to test the tool. These two applications (actually two implementations of the same application) are very well known. The first one is a very simple vanilla matrix multiply without

[3]It might need to wait more if a memory operation takes a long time to return

Listing 1: Vanilla Matrix Multiplication

```
void MM(int start_i, int start_j, int size){
   int i, j, k;
   for(i = start_i; i < start_i + size; ++i)
      for(j = start_j; j < start_j + size; ++j)
         for(k = start_i; k < start_i + size; ++k)
            c[i][j] += a[i][k] * b[k][j];
}
```

Listing 2: Strassen Matrix Multiplication

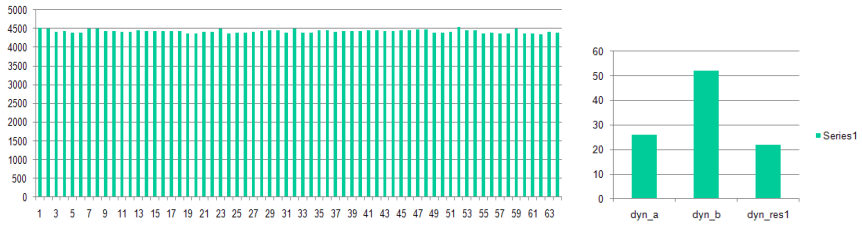
```
void STRASSEN(int start_i, int start_j, int size){
  if (size < threshold){ MM(start_i, start_j, size); }
  future STRASSEN(start_i, start_j, size/2);
  future STRASSEN(start_i + size/2, start_j, size/2);
  future STRASSEN(start_i, start_j + size/2, size/2);
  future STRASSEN(start_i + size/2, start_j + size/2, size/2);
  P1 = (A11 + A22) * (B11 + B22); /* Matrix operations */
  P2 = (A21 + A22) * (B11 + Z11); /* on the sub blocks  */
  P3 = (A11 + Z11) * (B12 - B22);
  P4 = (A22 + Z11) * (B21 - B11);
  P5 = (A11 + A12) * (B22 + Z11);
  P6 = (A21 - A11) * (B11 + B12);
  P7 = (A12 - A22) * (B21 + B22);
  C11 = P1 + P4   P5 + P7; /* Final Matrix Calculation */
  C12 = P3 + P5;
  C21 = P2 + P4;
  C22 = P1   P2 + P3 + P6;
}
```

any user enhancements. The Cray XMT compiler performs extremely well on this type of problems (regular loops) and can perform many optimizations on them. The second implementation is using the Strassen method to compute matrix multiplication. In the XMT, this application has been parallelized using futures. The pseudo code for both applications are given in code listings 1 and 2.

As you can see in listings 2, this implementation of Strassen has a hotspot on reading the Z11 block. Take in consideration that this factor will be shown later when the memory utilization and threads behavior are analyzed.

4.3 Testbed Parameters and Results

These two implementations were run with a small matrix (64 by 64 double precision elements) on 64 processors on the Cray XMT. Each of the following graphs shows total variable and memory node utilization during the whole calculation.

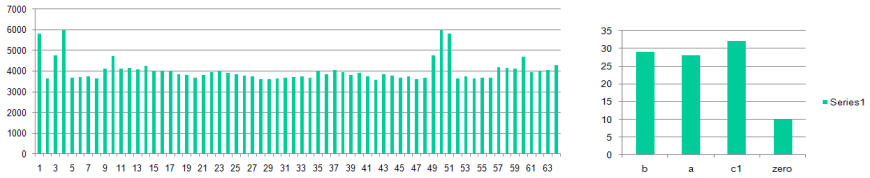


(a) Total Memory Node Utilization (b) Total Variable Accesses

Figure 3: Results for the Vanilla Matrix Multiply

Figure 4 shows the results for the Strassen run. Sub figure 4a the total number of accesses on each memory node and Sub figure 4b shows the total number of accesses for each of the used variables of the application. The X axis in these graphs represents the memory nodes in the system and the Y axis represents the number of accesses. Similarly, figure 3 and its sub figures represent the runs for the vanilla matrix multiply. Figure 5 shows the usage map for each variable in percentages for each memory node.

Figure 3a shows that the distribution of the accesses across memory modules is uniformly distributed, thus reducing contention at this level. The sub figure 3a shows that from the variables that were used, the



(a) Total Memory Node Utilization (b) Total Variable Accesses

Figure 4: Results for the Strassen Matrix Multiply

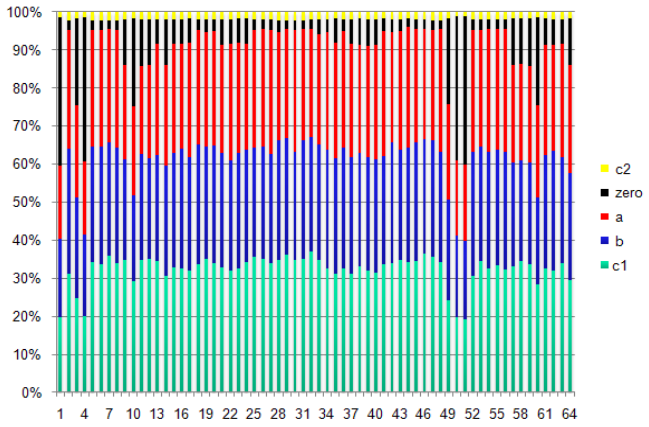


Figure 5: Variable Map Usage for the Strassen Case

b was used the most. This is due to the compiler optimizations applied to the given code. In this case, they are reusing the a and c matrices and loading b as it is needed.

Also of interest is what happens on the second set of figures. Figure 4a shows the bank distribution of the Strassen implementation. In this case, some of the banks are over-subscribed. This behavior (and its culprit) is (are) shown in figure 4b. The variable distribution map presented in this figure shows that the variable represented by zero[4] is the culprit of the strange behavior on the given memories modules. The figure shows that the access is highly irregular which suggests that certain elements of the block zero are the source of contention. This is an expected outcome of the computation due to the algorithm characteristics (using the zero block in many of the recursive calls continuously).

5 Conclusions and Future Work

Careful management of shared resource bandwidth will be one of the future key multi-core technologies. The Cray XMT is ideally suited to serve as a test platform to analyze shared resource contention at massive scale.

This paper presents a new memory centric performance analysis called MODA. It consists of four components and requires minimal user interaction. MODA was implemented on the Cray XMT architecture.

Preliminary application experiments have shown that MODA can be used to detect memory access pattern that can cause contention at the memory organizational level. A predictive tool with these capabilities is useful because it allows a programmer to reason about contention at the development phase at small scale. The likelihood of encountering contention surprises at larger scale is hereby greatly reduced. This is an advantage since debugging and performance analysis at large scale tends to be onerous.

Overall, the objective of this tool is to describe the application behavior from a shared resource perspective. Due to the increased importance of memory / network subsystems, tools like MODA can help identify several hotspots that might arise due to contention on shared resource subsystems. We believe that this

[4]In this context, this variable is a block of addresses

tool will help establish a set of new resource centric tool suites and to provide new insights into the field of massive parallel performance analysis.

Future work on this tool includes its optimization and several alternative methods to obtain monitoring information. Moreover, the current monitoring framework can be optimized by writing its critical sections in assembly and by-passing the whole XMT trap framework by static binary rewriting. A new GUI and more powerful analysis (e.g. aliasing analysis) are in the works for the analysis phase.

References

[1] Jlich Supercomputing Centre. Cube: Cube uniform behavioral encoding. `http://www.fz-juelich.de/jsc/kojak/components/cube/`.

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

[3] Chris 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] Kevin A. Huck and Allen 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] Karen L. Karavanic, John May, Kathryn Mohror, Brian Miller, Kevin Huck, Rashawn Knapp, and Brian 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.

[6] Wolfgang E. Nagel. Vampir: Performance optimization. `http://www.vampir.eu/index.html`.

[7] Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander 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.

[8] Crescent Bay Software. Deep/mpi. `http://www.crescentbaysoftware.com/deep_mpi_top.html`.

[9] Computer Science Department University of Wisconsin-Madison. The paradyn parallel tools project. `http://www.paradyn.org/index.html`.

[10] Omer Zaki, Ewing Lusk, and Deborah Swider. Toward scalable performance visualization with jumpshot. *High Performance Computing Applications*, 13:277–288, 1999.