RCRTool: Design Document
Version 0.1

TR-10-01

Allan Porterfield, Rob Fowler, Min Yeol Lim

February 12, 2010

RENCI Technical Report Series
http://www.renci.org/techreports

RCRTool: Design Document
Version 0.1

Allan Porterfield
akp@renci.org
Renaissance Computing Institute

Rob Fowler
rjf@renci.org
Renaissance Computing Institute

Min Yeol Lim
mylim@renci.org
Renaissance Computing Institute

February 12, 2010

Abstract

RCRTool, **R**esource **C**entric **R**eflection **Tool**, will allow application programmers to better understand resource contention between multiple threads of a single application or between simultaneously active applications sharing varying levels of hardware. The improved knowledge of how the entire system is performing will be available to applications and runtimes for dynamic performance tuning. This document provides some of the motivation and the initial design of the entire system including access of hardware and OS performance counters, system modeling with that data, API that allow access to the data by runtimes and applications, and a data logging facility for post-run analysis.

The design attempts to allow the same tool to be used with a future single shared address node (with tens of cores) and with a distributed memory system with tens of thousands of nodes and hundreds of thousands of cores. The difference between these systems, should be contained by difference in what parts of the system are watched for potential bottlenecks and the granularity of available dynamic feedback.

At the center of RCRTool will be the RCRdaemon. It will have several jobs, including watching the hardware and OS for performance bottlenecks using performance models. RCRTool will supply some models, but mechanisms for the user to add their own will exist. RCRdaemon will also be responsible for transmitting the current state of the system to applications and the OS for dynamic tuning. A third function of the daemon will be logging the information for post-execution analysis.

1 Introduction

As computers grow bigger and more complicated, there is a trend in the hardware to add more resources that are shared by multiple threads. At the node level, Intel has chips with 8 cores and systems (nodes) with 64 cores (counting HyperThreading), AMD has announced and publicly demoed 12 core chips and 48 core systems (nodes). In both of these systems, the cores share resources that use to be considered private (caches, memory controllers and DIMMs). When computer systems are made up of multiple nodes, all of the additional hardware (networking and IO) is shared across some or all of the nodes. Systems are being designed and built with hundreds of thousands of cores, if each core can use even 0.1% the total available bandwidth (network or IO) the overall system cannot support all cores simultaneously using the resource. Thread performance will not only depend on what the thread is doing, but what every other thread in the application is doing and what every co-executing application on the system is doing [7, 4, 1, 10]. Understanding thread performance now requires information about the currently available hardware resources more than ever before.

Current performance tools [15, 11] effectively locate performance problems within a single thread of execution, by using hardware counters to determine what resources a particular thread is using. This is effectively a first-person view of performance. As we move to systems where more and more is shared, this view will not be effective in locating performance problems. If the problem is caused by other threads attempting to share some resource a first person view will be ineffective. What is needed is a overall view of total system performance and some understanding of how the available machine is dynamically changing during the course of an execution, or a 3rd person view of performance.

RCRTool, **R**esource **C**entric **R**eflection **Tool**, will allow application programmers to better understand resource contention on modern computer systems, from single nodes to large systems. Understanding of resource sharing by multiple threads of a single application or between simultaneously active applications is required to understand application and system performance. Improved knowledge of how the entire system is performing, will be available to applications and runtimes for post-execution and dynamic performance tuning. The RCRTool system has to access hardware and OS performance counters, allow system modeling with that data to detect possible contention, supply an API allowing dynamic access by the OS and applications, and allow logging for post-run analysis.

The design attempts to allow the same tool to be used to locate performance problems in a single shared address node (with tens of cores) and distributed memory systems with tens of thousands of nodes and hundreds of thousands of cores. The systems will have different resource contention issues, but these should be handled by using different resource contention models within RCRTool and by the use of time scales appropriate to the system being monitored. The goal is to build a flexible framework in which resources models specific to a particular system can be easily inserted and system specific resource contention issues can be identified.

RCRTool will provide a daemon process that watches for various potential hardware and OS resource contention issues. When the daemon detects an active performance issue, it will

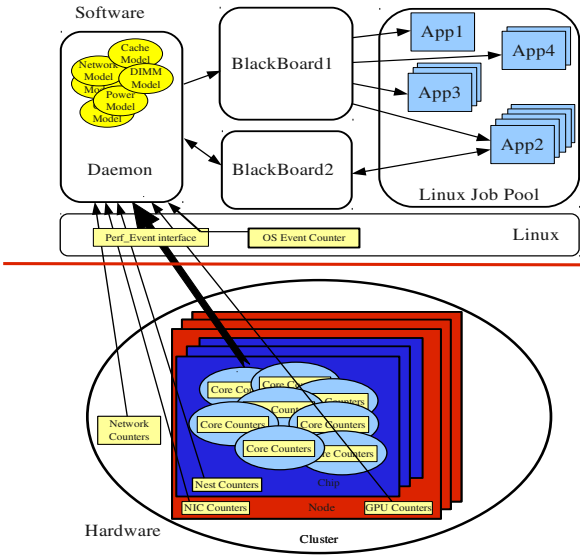

Figure 1: RCRTool Overview

focus on that issue and determine all of the active threads or applications that are actively contributing. RCRTool will also provide an interface to access that information by current applications, to enable tuning to the current available hardware configuration.

RCRTool is a resource contention *identification tool*, it should be a important tool for performance tuning once functional. RCRTool is not initially intended to fix the problems once detected. That is left for the application programmer. Fixing the contention problems is beyond the current scope of RCRTool. Information gathered will be useful the process of tuning applications to the dynamically available hardware. We intend to explore those possibilities once a functional RCRTool exists.

This is a working document. We expect it to have NOTE s and TODO s embedded within it as a means to make sure that information about implementation issues and open problems are not lost. As implementation progresses, we expect the type and number of items open will evolve over time.

2 Overview

One way of thinking about what RCRTool tries to do, is it describes the dynamically changing computer that is available to each program (or thread). The changes are only interesting to the program when it is trying to use more of a resource than is currently available. Up until

recently, comparing the amount of a resource an application needed against the amount of a particular resource available was enough to detect problems (swapping because it used more memory than available, or low cache hit rates). As systems grow, performance is limited by resources which can be shared by multiple applications (or threads). Large systems have all active applications share one network and file system. An applications that has a lot of message traffic may see it's performance reduced if it is co-scheduled with an second application that communicates along the same links. Likewise, two applications with intense sporadic IO usage, can see 'random' slow IO when they both attempt read/write at the same time. A programs performance is not just a factor of the program but also the environment it executes in.

RCRTool provides information about the actual environment that an application sees during execution. The information is more than just what applications are co-scheduled with it, but what portions of the hardware resources are actually available at any given time. To provide this information RCRTool needs to understand the computer system to detect dynamic over-use of any resource. This provides a view of the computer resources available during a particular application execution. It must relate that to portions of the application, so that a programmer can understand the actual performance.

How RCRTool will work is shown in Figure 1. Information about system resource utilization is generated by the hardware and is available to the user though a variety of hardware counters. These counters can be accessed though (and supplemented by) the OS. The performance counter community is undergoing a transformation as it moves from Linux kernels patched with *perfmon2* [6], to the `Perf_Event` [17] which is integrated with Linux kernels newer than 2.6.31. RCRTool will use the `Perf_Event` interface to obtain information from the hardware and the OS about resource usage. The `Perf_Event` interface is flexible and expandable and where possible RCRTool will provide it's information to the application as extensions to that interface.

The heart of RCRTool will be a daemon that reads the various counters, models potential bottlenecks, and performs in-depth analysis when potential contention is detected. It will multiplex though the counters looking for different sources of resource contention. When contention is detected, it will be examined more closely to see if any application is seeing increased latency or decreased utilization. The active applications will be probed for their current locations. If the application (or thread) provides a location, the daemon will use that information to determine where within the applications is the set of activities that causes the problem. This data can be provided either dynamically or though a log for post-execution analysis.

The RCRTool daemon can provide data to the OS scheduler, to a system tuning application or tuner, or to an application using Blackboard_1. The amount of data produced during a long run will greatly overfill any pre-allocated memory space. Two flavors of the API will be required. The first is for post-execution analysis. The biggest problem will be the sheer volume of data produced. Techniques such as wavelet compression [8], and aggressive elimination of data during normal execution will be used to limit the data saved for post-execution analysis. The second flavor of API will be designed for interactive use by the OS and applications. It will focus on meters either displaying instantaneous values, or max/min values over

a short time interval. The meters will be accessible by external users to determine the dynamically available system resources during a programs execution. Finding the right point on the information/program security tradeoff will be addressed. RCRTools purpose is to provide information about what other jobs are doing, but care to make sure no private information can be extracted from other users is critical. To the extent possible both of these interfaces will be provided as extensions to the `Perf_Event` interface, to reduce usage effort and potentially ease adoption into Linux at a later date.

Blackboard_2 will provided by RCRTool to allow the applications to communicate with the daemon providing information about a thread's current program counter and information about thread status. This information will allow better post-execution relationships between resource usage and program state. The daemon will use this information to better pinpoint resource contention. The improved measurements will be made available to the programmer using the same interface as before.

Data on Blackboard_1 can only be written by the daemon and is readable by all applications. Blackboard_2 has data potentially read and written by any application or the daemon. It's data is expected to read by the daemon potentially impacting any later information that the daemon writes back to Blackboard_1 or 2. Where Blackboard_1 has security concerns, Blackboard_2 has less since any data was volunteered by the application.

The overview presented so far, works for hardware coherent shared memory nodes and distributed memory systems, systems can range from tens of nodes to millions. The environment does have an effect on the capabilities of the tool. In a distributed memory system, the Blackboards will be implemented on top of a PGAS area and will have substantially more overhead to access and keep current. This will result in more monitoring usage, larger time-steps and will require greater data compaction and greater need for knowledge about co-resident applications/threads. A shared memory node can use the hardware provided shared memory and allows interactive use with greatly reduced latency. The reduce latency should allow for information provided to productively influence parallel grain size and parallel scheduling decisions.

3 Performance Counters

The `Perf_Event` interface is a new Linux kernel-based subsystem that provides a framework for performance analysis. It monitors both software (software counters and tracepoints) and hardware events (using Performance Monitoring Unit (PMU)). Previous performance counter frameworks such as *perfmon2* only provided monitoring of hardware events. The software events are designed to monitor operating system behavior, and include fixed, pre-defined events such as page faults, CPU scheduling actions, and kernel memory utilization. The tracepoints are customizable events in the kernel and allow the user to define additional software events. By examining hardware and software events at the same time, RCRTool can build a "Big Picture" of what is going on within the system. The `Perf_Event` framework is incorporated into Linux kernel versions 2.6.31 and later.

Tracepoints provide a hook that calls a user defined function (probe) at runtime inside the kernel. When a new tracepoint is added in `Perf_Event` , a directory with that name is created in the *debugfs* file system, and relevant files within the directory are created. A tracepoint can be then be activated and deactivated using the *debugfs* interface on the fly. When active, each tracepoint results in a user defined function being called. Tracepoints can reside at various important locations inside operating system including drivers. In addition to being used by RCRTool, the tracepoints can be used for accounting. When used properly, monitoring software events will be complementary to hardware counters. In RCRTool, these software events will be used in conjunction with hardware counters to figure out various resource contentions issues effectively and efficiently.

The number of physical counters within the PMU limits the number of simultaneous events that can be monitored. This restriction is dependent on the hardware (CPU) used. For example, Intel Core i7 on-core PMU has 3 fixed counters and 4 generic counters and off-core (or "uncore") PMU has 1 fixed counter and 8 generic counters. This may not be enough to simultaneously count all desired counters, since there exist more than 100 performance counters provided by the vendor. Although, not all counters are significant to contention detection, monitoring large number of counters concurrently will be often required. In addition, certain events cannot be measured at the same time, if they conflict with each other within the PMU. Therefore, RCRTool will need to multiplex some hardware performance events onto the PMU counters. There may be penalty in accuracy for multiplexing depending on the underlying hardware and the granularity of a monitoring interval. The multiplexing can be easily implemented by grouping a set of counters or by using some API libraries such as PAPI, it is not known that how much the accuracy will drop in multiplexing. Understanding and addressing this concern, is an early implementation issue.

3.1 Cores, Sockets, SMPs, and beyond

As multi-core architectures become popular in modern microprocessors, monitoring the hardware performance counters grown more complicated. A system with multiple microprocessors in different sockets requires monitoring more hardware activities, like those involved in the communications between sockets. As the number of cores grows, the RCRTool daemon will have more hardware counters to monitor at both the core and socket levels. As each core and socket usually has its own PMU, the RCRTool daemon will have to control many PMUs. Managing them efficiently is a challenging issue in designing the RCRTool. For instance, the Intel Core i7 micro-architecture provides two PMUs: on-core and off-core. The on-core PMU can monitor performance counter events on the core it resides, whereas the off-core PMU (called "uncore") monitors the hardware events of the resources shared by all cores/threads on a socket. Special rules and restrictions exist to handle sharing of the "uncore" counters by multiple PMUs:

- trace events cannot be traced back to a particular core or thread to figure out were the action originated

- there is no privilege level filtering at the user or kernel level.
- multiple cores can not access the uncore PMU at the same time.

For these reasons, careful attention has to be paid when detecting contention in the shared resources. RCRTool will need to build a mechanism to correlate on-core events with off-core events and to trace uncore events back to on-core actions when detecting the contention. For example, L3 cache events are monitored in off-core PMU's, since the cores in a socket share a L3 cache in Core i7 microprocessor, determining which threads are generating low hit rates or are occupying an excessive cache footprint will be desired.

3.2 Rates and Utilization

To analyze multiple performance counter values, they often need to be converted into rates for direct comparison. Event rates indicate how many events occur during a set unit of time, even when their monitoring durations are not identical. To increase the accuracy of elapsed time, RCRTool will use the Time Stamp Counter (TSC). The TSC is a 64-bit register present on all x86 processors that counts the number of CPU clock cycles. Since the TSC increases at the rate of the processor cycle speed, it provides the basis for high-resolution measurements. The TSC is invariant, it runs at a constant rate in all ACPI P-state, C-state, and T-state, even though the CPU cycle rate is changing.

The utilization for each sampled event is calculated as $\frac{EventRate(t)}{EventRate_{max}}$. The maximum event rate is the upper bound achieved when the resource is fully loaded (utilized). It is not simple to get the maximum event rates for all the performance counters. One possible approach is to derive the theoretical limit for whatever resource is being counted. The theoretical bounds can be calculated from architectural specification in hardware such as maximum CPU speed and maximum memory bandwidth. Although this gives an absolute upper bound, it may not be practical to compute for all counters and may be of little relevance because it is not achievable by any program. Another approach is to get empirical bounds by running micro-benchmarks or existing benchmarks that can stress the selected resource/counters and take the maximum value observed. Although this approach is technically sound, it requires non-trivial amount of work to write the benchmarks and to ensure whether they are achieving the maximum performance. A combination of micro-benchmarks and observed resource peaks will be used by RCRTool to determine maximum performance of each modeled resource. Each resource will also have a contention level defined, at which latency to use the resource is climbing faster than throughput.

For some resource contention issues, the usage rates will not be sufficient to determine the actual problem. Ratios between similar usage counters will also be required (Cache Miss Rate, Memory Reads vs Memory Accesses, etc). Computing these ratios accurately requires that the counters be gathered simultaneously. The RCRTool will generate several resource usage ratios in addition to the usage rates, to help the models accurately identify resource contention and its sources.

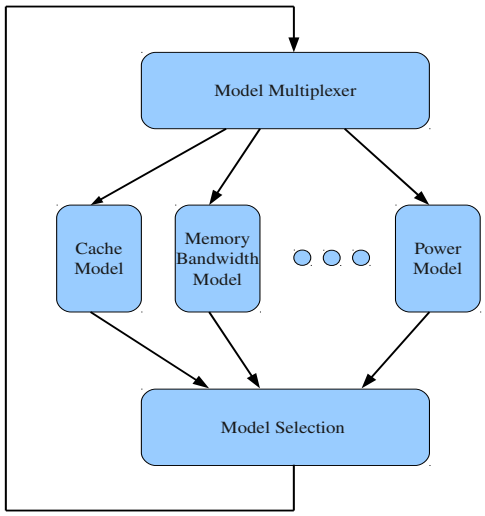

Figure 2: RCRTool Daemon Structure

4 RCR Daemon

The center of the RCRTool is the RCR Daemon. The RCR Daemon is responsible for gathering the counters, determining whether resource contention exists, relating contention with current application activities, creating entries to be logged, compressing any entries to be logged, and providing dynamic resource usage information for any interested applications. Various resource contention/performance models will be embedded within the daemon. Each of the models will be responsible for detecting a particular resource contention ranging from network contention, to shared cache contention or power contention.

Figure 2 gives the outline for the daemon code. Depending on the number of counters needed and that the hardware can simultaneously provide, the daemon will need to multiplex over different sets of counters on every outer loop iteration. This will distort the initial detection of bottleneck, but by focusing on that single problem after initial detection, the effects of the initial distortion should be minimized. Each pass though the outer loop will check all performance models relevant for the current set of counters and determine if any of them requires more detailed observation. The models will be updating any meters supplied on the Blackboards. As long as no problems are detected, little or no logging will occur.

When a potential resource contention is noted by one of the models, the daemon will ensure that model gets more detailed testing. The counter set will not be changed, so that several consecutive values maybe evaluated to determine actual resource contention exists. If resource contention is found, the daemon will look for the activities contributing to the problem. RCR

Daemon will examine the current location of each application, either though Blackboard_2, though the OS or by forcing a trap within each executing job.

Each model is responsible for determining the elements of a resource contention using a set of common tools to determine the current application state. When doing this more detailed analysis, one or more log entries can be generated and passed to a logging sub-system responsible for compaction and IO. Each model after completing a detailed pass will release execution and the daemon will look for other possible sources of contention. RCRTool assumes that multiple sources of resource contention may be occurring simultaneously and that the first detected may not be the easiest way for the programmer to see the actual problem.

The initial models will focus on memory bandwidth utilization, cache utilization, network utilization, and power consummation. The first three are clearly looking for over-used resources where performance could be improved by reducing demand. Over-use of any of them can result in decreased total throughput, due to cache or DIMM page capacity misses or due to increased backoff within the resource. The newest Intel systems change the clock rate depending on the many factors including current chip temperature and power utilization. Tuning a system to make maximal use of this feature will be difficult particularly in the presence of DVFS. A flexible clock rate will complicate power an performance tuning. RCRTool will include a model to estimate power utilization, to help the programmer determine where more power is used than actual more performance gained.

5 RCRTool Application API

RCRTool provides data to running applications and to post-execution tools. These interfaces for how the data is gathered and distributed, to a large extent, will determine how useful a tool is developed. The interfaces need to be flexible, easy to use and be able to handle both logging and dynamic reporting of system performance. To examine multiple applications, RCRTool will exist (at least partially) inside the Linux kernel. To simplify the Linux adoption process, the dynamic performance interface will be an extension to the existing `Perf_Event` interface. The logging interface should handle the creation of the log and data retrieval of both raw and compacted logs. Because at various points in the application's life not all forms (dynamic/logged) of the data will be available, the interfaces must have graceful failure modes. Failure to check return codes should not cause applications to crash.

RCRTool will test, by default, for a set of possible resource contention issues. These tentatively include DIMM bandwidth contention, cache footprint contention, network contention and power utilization. Models for each of these issues is embedded with the daemon and is regularly checked. As part of RCRTool installation on a system or during RCRTool execution additional models may be added. These models can check for system specific resources (for instance, inter-socket communication on a multi-socket system) or an software specific metric (reported OS queue depth). The ability to instantiate new models will give RCRTool flexibility to detect new types of performance problems on new systems as they appear.

The initial definition of RCR API will use a subset of C++. As development progresses, the interface will likely move to a more template form or revert to ANSI C.

5.1 Adding Models

RCRTool allows new resource models to be added. The potential scope of these models is unknown, so the interface should be as generic as possible. Each model defines an evaluation function and a set of events as inputs. Each model returns a short vector of information. The return vector includes the value(s) to be posted on BlackBoard about the utilization of this particular metric. It will also contain the model's recommendation on whether a problem exists and how soon the model would like to be rerun. The last part of the return vector will be any log entry that the model produced. The logging will be handled outside of the models to allow easier handling of multiple and/or no active logs.

5.1.1 Implementation

Functions to create, install and execute resource contention models are included. To see a sample model definition see Section 6.3.2 Each model requires a set of input counters to be specified and returns a vector of results.

Counter Definition This class is used both in defining the counters used by a model and returned by the model with an utilization values for the BlackBoard. The counter id definitions will be defined by `Perf_Event` with possible extensions defined by RCRTool. The value field will contain the latest observed utilization for this counter. The union will allow both floating point and integer values to be returned.

```
enum RCRCounters {
     uint64  Perf_Event_id;
     union _RCRCounterValue{
        double  floatValue;
uint64  intValue;
     }
};
```

Daemon Return Enumeration the enumeration is used as input to the model selector to decide which models to run on the next iteration.

```
enum RCRDaemonModelResult {
  RCR_OK,         // no resource contention
  RCR_CONTENTION, // possible resource contention
  RCR_BUSY,       // definite resource contention
  RCR_LOG,        // resource contention + event to be logged
};
```

Model Return Values the return class allows the daemon to decide whether to rerun this model, outputs the instantaneous utilization for the blackBoard and a log entry (possibly NULL).

```
class RCRModelResult{
  enum RCRDaemonModelResult modelResult;
  RCRCounter                blackBoardResult;
  RCRLogEntry              *logResult;
}
```

Definition define a model and its required input counters. The length of the RCRCounter array will be hardware specific (for most AMD processors 4, Intel allows more but rules exist against some combinations). If invalid combination specified RCRModelDef will return `false`.

```
RCRModelResult (*func)(int, RCRCounters[]);

boolean RCRModelDef(func, numOfCounters, RCRCounter[]);
```

Execution The daemon will call each model with a frequency that will depend on the amount of resource contention detected by that model previously (contention results in calls more often) and the other active models (contention elsewhere results in fewer calls).

Each model execution is responsible for determine whether resource contention currently exists and updating a blackboard meter for that resource. In most cases, identifying the thread or threads responsible will not be possible within the model, but when possible the model should make that identification.

If an execution of the model takes an excessive amount of time it will be aborted (using setjump/longjump back to the model selector). If a model fails consistently, the model selector will prevent additional calls.

5.2 Daemon to Blackboard

Information for use by applications must be posted on the Blackboard. The daemon thread will update the simple counts, whenever that counter is selected by the multiplexer. For information computed by the models, the counts will be updated with the value return by the model in `RCRModelResult.blackBoardResult`.

5.2.1 Implementation

Time Scales The time scale is a 32 bit value in microseconds. The low 2 bits are used to determine the measurement type to be taken (average, sum or peak). By using microseconds and stealing the lower two bits, the minimum value is 4 microseconds and the max is over 4 million seconds or between one and two months.

NOTE: Should we go to different time units or a longer(64 bit) value?

```
enum RCRTimeType{
   RCR_INSTANEOUS = 0,   // most recent value
   RCR_SUM = 1,          // total over requested time span
   RCR_AVE = 2,          // average over requested time span
   RCR_PEAK = 3          // highest value during time span
}
union RCRTimeScale {
  enum RCRTimeType  type;
  uint32_t val;
};
#define RCRType(x) ((enum RCRTimeType) (x && 0x3))
#define RCRTime(x) ((uint32_t)(x && 0xfffffffc))
```

Initialize Meter Define a new meter, by both type and period; If multiple time periods for the same value are needed, define multiple meters RCRMeter returns the meter number to use when writing values to the meter. RCRMeterReset returns FALSE if reset failed for some reason.

```
int meterNum RCRMeter(TimeScale t);   // get new meter
boolean RCRMeterReset(int meterNum);  // reset meter history
```

Post updates the meter with new instantaneous value.

```
boolean RCRMeterUpdate(RCRCounter c); // update meter (which
                                      //    meter part of RCRCounter)
```

5.3 Blackboard to User

The blackboard is the communication vehicle from RCRTool (really the daemon) to the user. Information transfers should look both like standard counters and meters where data over defined time periods can be examined.

The goal is to build on top of existing interfaces as much as possible. `Perf_Event` uses standard IO calls and the debug file system (*debugfs*). RCRTool will provide access to the counters and meters through *debugfs*. RCRTool will define a subdirectory within the debugfs and use the counter numbers as file names within that directory.

The data can be retrieved using standard IO calls or by using the macros (functions?) defined here to process the file contents.

NOTE: No macros/functions are going to be defined until initial models become operational.

5.3.1 Implementation

Functions for each defined meter a file will be created (/debugfs/RCRTool/meter_#). That file will contain the most recent values (current, peak during time slice, system defined maximum) for each meter. The file will be read-only except to the daemon and will contain three _RCRCounterValue and the current time scale for values meant to be displayed as meters or a single _RCRCounterValue for values meant to be displayed as counts.

```
// get the value of meter for specified timescale
class {
  _RCRCounterValue  curMeter;
```

```
  _RCRCounterValue  recentPeakMeter;
  _RCRCounterValue  sysMaxMeter;
  RCRTimeScale      timeScale;
}RCRMeterValue;

class {
  _RCRCounterValue  curCount;
}RCRCountValue;

RCRMeterValue * RCRGetMeter(int MeterNumber);
RCRCountValue * RCRGetCount(int MeterNumber);
```

5.4 Daemon to Logging Subsystem

In addition to dynamic feedback to application given on the Blackboard, RCRTool will provide post-execution information in the form of a log. The logging system will be responsible for compacting the log to minimize actual IO during execution of an application. Ideas similar to wavelet compression in [8] will be used.

RCRTool can have a execution spanning many applications, intermediate logs will be possible. An application can have a log of the system issues during its execution

5.4.1 Implementation

Class A log entry starts with 16 bits meant to be ease visual identification of log boundaries (0x1111?). RCRTool will allow 65536 different log entry types. The `LengthInWordsOfLogEntry` field allows each large but limited size log entries (64K words).

NOTE: I'm not sure what these will be used for yet, but seems like a good idea. May need to be a 32 bit field to make sure we never run out. Do I need an enum here?

```
class RCRLogEntry{
  uint16 LogID;                   // 0x1111
  uint16 DefinedZero;             // if we need a version number in the future
  uint16 TypeOfLogEntry;          // type
  uint16 LengthInWordsOfLogEntry; // size
  uint64 * LogEntry;
}
```

Functions create and terminate application specific logs and add an entry to all active logs RCRLogCreate creates a log for a particular process (pid) the call opens a file descriptor which is returned. RCRLogTerminate closes the log associated with this file descriptor (fd). RCRLogAddEntry adds the specified log entry to all currently open logs. This will allow to overlapping applications to control their own logs.

```
int RCRLogCreate(int pid);
boolean RCRLogTerminate(int fd);

boolean RCRLogAddEntry(uint16 Type, uint64 Size, int64 *LogEntry);
```

5.5 Logging Subsystem to User

Gathering post-execution information for the application programmer will be the responsibility of GUI tools[1]. The GUIs will need to process the log entries.

Currently there are two forms of log data that maybe available, raw and compacted. The actual log entries will be the same and the only difference to the GUI will be the volume of entries.

5.5.1 Implementation

Log Type whether the log is compacted and whether the log was for a single application execution is returned when the log is opened. It will also identify any failure opening the log.

```
enum RCRLogType{
   FAILED,
   RAW_APPLICATION_LOG,
   RAW_SYSTEM_LOG,
   COMPACTED_APPLICATION_LOG,
   COMPACTED_SYSTEM_LOG,
}
```

---

[1] designed after this infrastructure is at least partially implemented

Functions need to find the correct log to open and get an entry from the log

---

```
enum RCRLogType RCRLogOpen(int *fd, const char *);
RCRLogEntry *RCRLogNext(int fd);
```

---

6 RCR Models

A critical component of the RCRTool infrastructure are the resource models it has at its disposal. If these models are inaccurate or overly liberal, resource contention with in a system that causes major application slowdowns will not be detected. If the model are too conservative, too many false positives will be flagged and the programmer will be swamped in possible problems that each require examination to determine if in fact something needs to be done. It is our intent to start with a small set of models for x86 processor architectures and interconnects locally available. The architecture models should be useful on a wide variety of systems allowing effective use of RCRTool.

NOTE: As the initial models are implemented, this section should contain a careful example of what the interface, implementation and installation entails.

6.1 Cache Performance Model

A CPU cache is designed to reduce the average time to access memory by storing copies of the most recently used data from main memory. It is also used by prefetching hard and software to store copies of date likely to be used in the near future. Application performance can vary wildly, depending on cache hit rate. Improving cache performance is a frequent goal of performance tuning. Since modern processors have multiple levels of caches on chip and some caches are shared between cores on chip, RCRTool needs to define performance models appropriate for each cache.

One of the most popular metrics measuring cache performance is the cache miss ratio for each level cache. In general, a cache stalls execution until it is satisfied. The stalls cause a drop in performance normally visible by a corresponding drop in Cycles per Instruction (CPI). Each level of cache has a significantly increased latencies in CPU cycles. For example, Intel Core i7 microprocessor have per-core L2 caches and a shared L3 cache. Handling a L2 cache miss (L3 cache hit) takes between 35 and 74 cycles. If the reference misses in the L3 the resulting memory access takes between 180 and 320 cycles depending on memory location and type. In building the cache performance model, careful attention should be paid to inefficient cache utilization, especially for the higher level and shared caches. For Intel Core i7, the fraction of

CPU cycles spending in cache misses can be calculated as follows [20].

$$L3MissCycleRatio = \frac{MEM\_LOAD\_RETIRED.LLC\_MISS \times 180}{UNHALTED\_CORE\_CYCLES}$$

$$\begin{aligned} L2Misses = (&MEM\_LOAD\_RETIRED.OTHER\_CORE\_L2\_HIT\_HITM \times 74) \\ &+(MEM\_LOAD\_RETIRED.LLC\_UNSHARED\_HIT \times 35) \end{aligned}$$

$$L2MissCycleRatio = \frac{L2Misses}{UNHALTED\_CORE\_CYCLES}$$

When running multiple applications on different cores within a socket, there is a chance that one applications cache usage will cause cache missed for other application sharing the L3 cache. Most events in the shared must be monitored in a system-wide session, and detecting the origin of the events is not straightforward. This issue will be addressed in our model by modeling the lower level caches and computing cache footprints for the various applications.

6.2 Memory Bandwidth Model

As multi-core architectures are widespread in computing systems, they are more likely to have performance bottlenecks in memory accesses. As two or more threads may share a path to memory, it is easy to have contention for the shared path. Within a socket this contention will most likely occur within a memory controller, trying to access memory. Across sockets, the contention may still be at the memory controller, but it may also be for the interconnect path (AMD HyperTransport or Intel QuickPath). Monitoring the memory bandwidth at several levels will be an essential component in our tool.

Memory bandwidth expressed normally in bytes per second, is one a key metrics in measuring memory performance. The fraction of the observed rate by the maximum rate in specification can indicates how much memory contention is occurring at a given time.

The bandwidth between the last level cache and memory can be approximately estimated as the cache line size multiplied by the number of lines transferred divided by the elapsed time. The similar method can be applied to estimated the bandwidth between L2 and L3 caches. Using the performance counters on Intel Core i7 microprocessor, the bandwidth is formulated as:

$$MemoryBandwidthToL3 = \frac{UNC\_LLC\_LINES\_IN : ANY \times L3CacheLineSize}{ElapsedTime}$$

$$L3BandwidthToL2 = \frac{L2\_LINES\_IN : ANY \times L2CacheLineSize}{ElapsedTime}$$

6.3 Power Models

The power model is designed to estimate how much power the system is drawing and detect when the power is consumed exceeds either an average or a budget. RCRTool uses an accurate power estimation model built on top of hardware performance events. Since performance events are machine specific, different power models need to be developed for different microprocessors. RCRTool's power model will be able to work for all power states available on a system. This includes the settings both for dynamic voltage and frequency scaling and the number of active cores [12].

We first determine these counters empirically from the analysis of power and performance data collected when running benchmarks. A multiple linear regression method is used based on performance counters that are strongly correlated with power consumption. Previously, Lee *et al.* [19] proposed performance and power prediction models using linear and non-linear models. The relationship between power and performance events may depend on the particular events chosen as a predictor variables. RCRTool plans to explore non-linear methods to improve the power model, if necessary.

Since our models are to be built with the observed power and performance data, an internal power monitoring device, **PowerMon** described in [3] is used to gather power data. The PowerMon measures power consumption of six different power rails between power supply and motherboard. The voltage and current, and therefore the power consumption, are measured for each power rail and can be used to calculate power for specific components. The model can separately measure CPU, memory, PCI power consumptions via PowerMon data. The `Perf_Event` interface is used to obtain the performance values.

NOTE: The power model on multi-socket architecture may or may not be different. Currently, the power model is being targeted for single-socket, multi-core architectures.

6.3.1 AMD Phenom (Opteron)

The AMD Phenom and Opteron architectures support only 4 hardware performance counters, limiting our performance model to 4 simultaneous events. Using Spearman's rank correlations between performance counters and power consumption. the greater than 100 candidate performance events are ranked. Spearman's rank correlation describes the relationship between two variables without making any assumption about the relationship. A group is selected for RCRTool's power model from that ranking. Each groups coefficient of determination (R-squared) values is compared to determine the best fit to the regression model. The performance events with the highest coefficient of determination are RETIRED_UOPS, DISPATCH_STALLS, DATA_CACHE_ACCESSES, and RETIRED_MMX_AND_FP_INSTRUCTIONS:ALL. The majority of power on Phenom is used by the CPU and low level cache. The four events chosen all represent various CPU actions. Although highly correlated with memory power, events such as MEMORY_CONTROLLER_REQUESTS:ALL and CPU_IO_REQUESTS_TO_MEMORY_IO:ALL are a relatively small potion of overall power and because of the 4 counter limitation, are not incorporated to our model.

6.3.2 Intel Core i7

The Intel Core i7 provides 7 on-core performance counters (3 fixed counters and 4 selectable counters) and 9 off-core (or "uncore") counters (1 fixed counter and 8 selectable counters). To determine the best performance counters for the model, Spearman's rank correlation method [21] is again used. The correlation values were again obtained for more than 100 performance events, and the best selected. The CPU power has a high correlation with CPU events, such as INSTRUCTIONS_RETIRED and UNHALTED_CORE_CYCLES. Memory power is highly correlated with the memory events, like UNC_QHL_REQUESTS:LOCAL_READS. The Core i7 has 3 counters fixed: INSTRUCTIONS_RETIRED, UNHALTED_CORE_CYCLES, and UNHALTED_REFERENCE_CYCLES. Our estimation model is for total power consumption, so RESOURCE_STALLS:ANY and L1D_ALL_REF:ANY are selected to help model CPU power. The additional performance counters on the Intel, allow the power model to be extended to model memory power consumption. The "uncore" events UNC_LLC_HITS:ANY and UNC_QHL_REQUESTS:LOCAL_READS were selected for their correlation with memory power consumption.

The selected performance counters become the predictor variables in the estimation model together with CPU frequency. The CPU frequency varies during execution, explicitly, power management services (line DVFS and DCT) and implicitly with the new hardware Turbo-Boost technology [9]. To determine the effective actual frequency we can use:

$$BaseOperatingFrequency \times \frac{UNHALTED\_CORE\_CYCLES}{UNHALTED\_REFERENCE\_CYCLES}$$

on Intel Core i7 microprocessor. The *BaseOperatingFrequency* indicates the fixed standard bus clock frequency for all CPU power states. Our model uses $\frac{UNHALTED\_CORE\_CYCLES}{UNHALTED\_REFERENCE\_CYCLES}$ as the power predictor representing overall CPU frequency.

With the performance events selected for predictor variables, the power estimation model can be built using regression analysis. Traditional multiple linear regression is based on certain assumptions, such as a normal distribution of errors in the observed responses, if the distribution of errors is asymmetric or prone to outliers, model assumptions are invalidated. Robust regression is used to circumvent the limitations [18]. The robust regression assigns a weight to each data point. Weighting is done automatically and iteratively using the iteratively reweighted least squares process.

7 RCRTool Feedback API

The blackboard interface defined in Section 5.3 did not allow the user to write to it. This restriction allows RCRTool to track any sharing of information between applications better. However, the goal of assigning responsibility to individual sections of source code, can be greatly hindered by separation from the executable. RCRTool provides a second blackboard, which is writable by the application to facilitate information transfer about where an application

currently executes. This information is not required from the application but will greatly improve the quality of data returned.

Besides execution location, a runtime can communicate its current and future resource needs and desires to RCRTool how can pass scheduling suggestions on to Linux.

7.1 User to Blackboard

RCRTool initially will be expecting particular information about the execution location of an application. As the tool evolves, the allowed information will expand.

7.1.1 Implementation

Functions Allow user to create and terminate execution threads and update the current location of any thread.

```
boolean RCRStartThread(int id, int pc);  // new thread or application
boolean RCRUpdateThread(int id, int pc); // current thread location
boolean RCRCompleteThread(int id);       // thread completes - cleanup
```

7.2 Blackboard to User

Data on the writable blackboard will be accessed in the same manner as the read-only blackboard, see Section 5.3.

7.3 Blackboard to Daemon

The daemon will access the user-writable blackboard, in the same manner as any other user, see Section 5.3.

8 Use Cases

The application programmer's question is Why is the program running slow during this time period? A multitude of reasons can be the cause of slow performance during the course of an

execution. The serial code could just not be using the memory or the ALU efficiently. This is the type of problem where first-person analysis works well. First-person tools identify single thread bottlenecks and can locate the code segments that contribute to that bottleneck [16].

RCRTool tries to address transient and shared problems. For example, an application running one thread gets 95+% cache hit rate because of blocking within the application. When running multiple threads, sometimes the application gets 90% and the iterations take longer and other times 99% and the iterations are faster. Why and what can the programmer do to force the 'fast' case?

We have identified several cases, that we expect to see on shared multi-socket multi-core clusters and will be using them to guide our research and development. As the tool matures, and the systems get more complicated, we expect to add new use cases.

- Poor thread performance
- Poor shared cache performance
- Poor memory performance
- Poor network performance

Some of these problems are clearly related, for instance: poor memory performance can cause poor thread performance, but this separation allows us to talk about focusing on parts of the problem to identify or eliminate them as overall problems.

8.1 Poor Thread Performance

One common performance problem that can have a multitude of causes is having, on average, each instruction take multiple cycles. Most first person performance tools have multiple methods to compute the achieved cycles per instruction (CPI) for code segments within an application. When the CPI rises above a threshold (often 1), the application tuner knows which code segment to look at in order to understand if the high CPI is appropriate. It is often the result of the program/system not tolerating memory or floating point latency. Whereas the past serial systems had a limited number of ways to combat high CPI, the highly multi-threaded systems about to be available, have additional available options to reduce CPI. CPI vs. the number of threads will become a common tradeoff for performance tuning. The goal of RCRTool will be to guide the tuner (or runtime) to select the threading choice that will maximize performance.

The decision of whether to use SMT (like Intel's HyperThreading) or not for HPC type workloads has been workload dependent. For some workloads, it increases performance substantially, due to better ALU pipeline utilization. For other workloads, the contention for shared resource causes 2 threads to run slower than 1. One goal of RCRTool is to determine the correct number of threads to have active in each code segment.

The original motivation for SMT was to allow more memory accesses to be simultaneously outstanding from a single core to increase its ability to hide the memory latency. For some sorting applications and applications like HPC Challenge Benchmark Random Access[13], RCRTool will detect high CPI combined with a high cache miss **ratio**, and a low to medium memory controller access **rate**. By recognizing the high miss rate and the fact that more misses can be handled by the system, the tuner can see that the number of active streams can be raised.

A second case where increasing the number of streams is productive, is where the threads are using only minimal shared resources. Benchmarks such as NAS EP[2] and Linpack[5] fall into this category. The ALU is very busy but the shared cache access **rate** is very low. The data is loading into private cache and reused very heavily. The very low rate of shared resource use will be the key to the tuner to increase the number of threads. Increasing the number of threads will improve performance, as long as none of the shared resource are over-subscribed (sum of cache footprints is less than shared cache size).

When share resources are fully (or over) utilized, the thread count can be reduced. Depending on the application, performance can actually increase two ways. The shared resource may start performing better under a lighter load (for example, reduced cache capacity misses) and the application may incur low overhead generating fewer threads. Using RCRTool, detecting an overused shared resource can be done by watching the **rate** at which it is being used and comparing that to the known peak rate of the resource. As long as the shared resource is running at its maximum rate the number of threads can be reduced.

NOTE: Some resources have different peaks depending on the operations being handled. Care with generating the 'peak' will need to be taken.

One example, will be watching the shared cache miss rate. If multiple threads are forcing data out of cache before it is reused, the overall performance of the system plummets. In this case decreasing the active thread count can substantially improve overall performance by decreasing the number of misses. If the number of misses and the miss rate does not change, the thread reduction will not increase execution time and may allow lower internal thread overheads.

A second version of the same problem, is likely to occur when shared arithmetic pipes are shared between threads, in the SMT it would be the shared ALU, for some new systems it is a shared GPGPU chip. A shared GPGPU will often have a queue of work to done on it. Maximizing the performance will be complicated, given the need to move data in and out of GPGPU memory around the computation. Performance can often be considerably lowered by reusing the previous data, if still valid, but in a shared environment that can only be done if the core does not release control. One core holding control, can effectively turn the system into a uniprocessor. RCRTool allows the GPGPU usage rate to be watched, allowing programmers to detect locations to improve GPGPU efficiency and utilization.

8.2 Poor Shared Cache Performance

First person application tuning often involves detecting poor cache performance over a region of code. Many techniques are used for improving cache performance like blocking and prefetching. As multi-core systems have become available most have added a cache layer which is on chip and shared by multiple cores. This shared cache presents new opportunities for performance problems.

We expect one of the most shared performance common problem on multi-core chips will be excessive shared cache footprints. On current AMD's and Intel the shared cache is between 6 and 8 MBytes and shared between 4 or 6 cores. If the data processed by a every core uses less than its fraction of the cache, all cores will benefit. However, if the sum of the cache footprints exceeds the shared cache size all suffer. If a single core has a cache footprint greater than the cache size, it will push all the other data out of the cache preventing other cores from reusing their data. So, if 5 of 6 cores use 1/2 GByte apiece of shared cache, no effective shared cache misses occur. But, if the sixth core is touching 10's of GBytes of data without reuse, not only will all of its references be misses but it will push the data being used by the other cores out causing them to miss also. A different scenario with the same problem, results if each core need 2GBytes of cache. Then only 2 or 3 cores can execute without causing cache capacity misses. Understanding and controlling cache footprints may be crucial to multicore performance.

Detecting poor cache behavior is an activity that first person performance debugging tools have studied aggressively and have good tools for. These problems are more variable, because cache performance depends on what else is executing or how many copies of the same thread are running. Sampling techniques on the number of cache misses works well for non-shared caches. In a shared cache, identifying the missing thread will be more difficult. Watching the cache miss rate and ratio should give RCRTool a good view of how the shared cache performance is performing.

The interesting question here that we will need to solve, is how to recognize two different cases. The first case is threads overfilling the cache, where running fewer threads results in much higher cache hit rates and better performance. The second case is where each thread individually overfills the cache and the number of threads need rises to try an mask the latency of going to memory.

NOTE: We have some ideas about watching the TLB rate and/or using the HT Assist probe cache features to allow us to detect the first case, but this is an open problem.

A second common problem in SMPs is false sharing. False sharing occurs when two threads are write data that is allocated into the same cache line although distinct. If the system has separate caches, like different sockets in a multi-socket multi-core system has, then each write requires a node wide coherence message to determine that the write is safe. This extra step, potentially transforms a cache only operation to multiple messages across the socket interconnect (HyperTransport or QuickPath). If false sharing occurs memory latency increases substantially and thread performance typically falls.

RCRTool can detect the possibility of false sharing by watch the number and type of hardware cache coherence operations traveling between sockets. If the number of intersocket operations, exceeds a quarter of the operations (really 1 over the cacheline length in words), RCRTool will start looking for false sharing between threads.

The other side of false sharing is actually a benefit and RCRTool should help identify when it is occurring, so that the programmer knows how the sharing is beneficial. If the same streams above are scheduled on cores sharing one cache, the overall number of cache misses can be greatly reduced over a serial execution. If a thread touches one word per cache line and the sum of cachelines is greater than the cache size, the thread runs at memory speed with every reference missing in cache. This can occur when a loop nest has one loop traveling rows through an array and the other loop columns. If the four threads that use a cacheline are co-scheduled, the first brings the data into the cache and the next three can use it without going to memory. The overall misses associated with the loop nest is quartered. Normally, this results in a significant performance increase.

NOTE: Recognizing should be possible at least in relation to the false sharing is bad case. It may be possible to see by the rate with which DRAM pages are being accessed. If 'good' false sharing is happening the cache hit rate will be high, the dram access rate will be high (as the one stream runs though bring in every cacheline) and the DRAM page miss rate will be low (should be striding though memory).

8.3 Poor Memory Performance

Memory bandwidth problems can be hard to detect. They will often masquerade as problems in the memory subsystem closer to the processor (cache). For this discussion, we're going to look for two different types of memory problem. The first is contention within the memory controller. Some of these have relatively simple causes and workarounds. The second type of problem involves reduced bandwidth from the DIMMs because of poor scheduling or poor locality.

When conducting memory bandwidth studies [14], several noticeable performance anomalies were noticed in the performance of multi-socket quad-core AMD Opteron systems. One was using 2 of 4 sockets with multiple GBytes of memory being accessed. As the number of concurrent reference increase from about 80 to 104, the performance fell 10-15%. As the reference count continued to increase above 104, the reduction gradually disappears. This was found to be a problem in the Linux memory scheduler being used. After splitting allocations evenly for the first 2G, all of the memory allocated between 2 and 3 GB was on one bank of memory (and one memory controller). At the point that 3GB was allocated (occurs at about 104 concurrent references for our study), 2/3% of the memory access were going through one controller and the other controller handled only 1/3%. The busy one could not go faster and the lower utilized controller did not use all of bandwidth available to it.

RCRTool would help debugging related problems, by allowing a system view of memory allowing the inequality of memory locations to be recognized. On a multi-socket system,

RCRTool will be watching the number of memory references on each memory controller. If one controller has a much higher rate than the others, a memory imbalance exists. Since different threads are using the memory, this could be a natural effect of the application. On both AMD and Intel multi-socket systems, there exist counters which measure the cross socket traffic. In the case of true memory imbalance, the model will see that the HyperTransport or QuickPath links that connect to the socket with the memory are busy and the other links are relatively idle. Using the available counters RCRTool should be able to locate memory imbalances and tell the application where in the source it occurs.

The second type memory bandwidth contention, is closely related to cache contention. Each DIMM has a small number (2 DDR2 or 4 DDR3) of pages(1KB?) from which memory accesses can be processed. If a memory request is on one of the currently loaded pages, the value can be read immediately. If the page is not loaded, one of the current pages needs to be cleared, and a DIMM page read must occur before the access can be completed. This is practically a large cache, with a small number entries with very long line lengths. Traditionally, the page policy in uniprocessor systems worked well because of spatial and temporal locality in access patterns. In multi-core processors, however, independent access patterns of each core are easily interleaved, thus losing the locality and significantly reducing DRAM efficiency [22]. A page refill consumes 2 to 3 times the bus cycles of a page hit. For many memory chips, peak transfer speeds are reached only when page hits are being handled. Since the number of active threads exceeds the number of pages, without doing reordering, even stride one accesses by each thread could generate poor DRAM page performance.

To detect this type of performance problem, RCRTool would look at the memory controller counters provided by both AMD and Intel. For the AMD, the counters exist that give the number of hits, misses and conflicts(need to clear a page for a new page) operations sent to the DIMM. By looking at the rates of these counters (with knowledge about the maximum available), the model can look for high conflict rates and a low TLB miss rate. If a lot of new pages are being used without reuse on the pages, the threads footprint will be large and the the TLB will be missing on some of the accesses. RCRTool can identify the current thread locations as using a very large footprint. If the number of TLB misses is small, the pages are being reused but not effectively. RCRTool can identify the portions of the code where this is occurring, and suggest that fewer threads or a rearranging of the memory accesses could improve performance of this portion of the application.

8.4 Poor Network Performance

Detecting and identifying network and communication library contention will depend on many variable traits of a system. What interconnect and topology is present? Which communication library is being used? What is the threads layout within the network? We need a (several) network models that handle common hardware features and allow local specialization to handle features unique to any particular system.

The base question to be answered, will be the detection of network congestion. For most

network and network interface chips (NICs), there are counters that over time tell us the rate that this node has injected into the network and some indication of rejection rate. This may be the rejected packet rate or a packet queue length. Either one can tell us that a node could have generated more traffic. When a node can generate additional traffic and the actual rate is falling, the model can infer that somewhere in the network there is congested.

After congestion is detected, relating that back to what all of the cores on the node are doing will be important. Having looked at the whole system and detected a problem (3^{rd} person view), we can now use traditional 1^{st} person performance debugging tools views to determine each applications current location. This can either be interactive, the model/RCRTool forcing interrupts, or passive using information provided on the BlackBoard by each thread. By combining information from across nodes and the OS the model should be able to identify regions where the network is congested and potentially suggest better application allocations.

Computing physical communication patterns between nodes is an interesting performance debugging question, that should be possible given a whole system view. Depending on the number and types of counters provided by the NIC, it may be possible to determine actual traffic patterns within a MPI application and describe how that compares to the traffic pattern that the portion of the system the application was allocated can support. The long term goal would be to dynamically adjust the work on the nodes so that the use of the communication layer was optimized.

9 Change Summary

0.1 February 2010 - Initial version

References

[1] Sadaf R. Alam, Richard F. Barrett, Jeffery A. Kuehn, Philip C. Roth, and Jeffrey S. Vetter. Characterization of scientific workloads on systems with multi-core processors. In *In IISWC*, pages 225–236, 2006.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[3] Daniel Bedard, Min Yeol Lim, Robert Fowler, and Allan Porterfield. Powermon2: Fine-grained, integrated power measurement. Technical report, Renaissance Computing Institute, 2009.

[4] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic task and data placement over numa architectures: An openmp runtime perspective. In *IWOMP '09: Proceedings of the 5th International Workshop on OpenMP*, pages 79–92, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Jack Dongarra, Hans Meuer, Horst Simon, and Erich Strohmaier. Top 500. http://www.top500.org/list/2008/11/100.

[6] Stephane Eranian. Perfmon2: a flexible performance monitoring interface for linux. In *Linux Symposium 2006, Ottawa, Canada*, July 2006.

[7] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2):49–57, 2010.

[8] Todd Gamblin, Bronis R. de Supinski, Martin Schulz, Robert J. Fowler, and Daniel A. Reed. Scalable load-balance measurement for spmd codes. In *Proceedings of the ACM/IEEE SC2008 Conference on High Performance Networking and Computing, 2008, Austin, TX, USA*, page 46, 2008.

[9] Intel. Intel turbo boost technology in intel core microarchitecture (nehalem) based processors. 2008.

[10] Haoqiang Jin, Robert Hood, Johnny Chang, Jahed Djomehri, Dennis Jespersen, Kenichi Taylor, Rupak Biswas, and Piyush Mehrotra. Characterizing application performance sensitivity to resource contention in multicore architectures. Technical report, NASA Ames Research Center, 2009.

[11] Innovative Computing Laboratory. PAPI: Performance Application Programming Interface.

[12] Min Yeol Lim, Allan Porterfield, and Rob Fowler. Softpower: Fine-grain power estimations using performance counters. *Submitted*, 2010.

[13] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D. Takahashi. Hpc challange benchmark. http://icl.cs.utk.edu/hpcc/.

[14] Anirban Mandel, Rob Fowler, and Allan Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *2010 IEEE International Symposium on Performance Analysis of Systems and Software*, White Plains, New York USA, March 2010.

[15] John M. Mellor-Crummey, Robert J. Fowler, Gabriel Marin, and Nathan Tallent. Hpcview: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23(1):81–104, 2002.

[16] John M. Mellor-Crummey, Robert J. Fowler, Gabriel Marin, and Nathan R. Tallent. Hpcview: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23(1):81–104, 2002.

[17] Performance counters for linux.

[18] Peter J. Rousseeuw and Annick M. Leroy. *Robust regression and outlier detection.* 2003.

[19] Karan Singh, Major Bhadauria, and Sally A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37(2):46–55, 2009.

[20] Intel Software and Services Group. Using Intel VTune Performance Analyzer to Optimize Software for the Intel Core i7 Processor Family.

[21] C. Spearman. The proof and measurement of association between two things. *Amer. J. Psychol*, 15, 1974.

[22] Kshitij Sudan, Niladrish Chatterjee, Davied Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: Increasing dram efficiency with locality-aware data placement. In *ASPLOS: Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.