SystemBurn: Principles of Design and Operation * Release 3.0

Jeffery A. Kuehn, Stephen W. Poole, Stephen W. Hodson,
Joshua K. Lothian, Matthew B. Baker
Jonathan D. Schrock

The Extreme Scale Systems Center
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831
http://www.csm.ornl.gov/essc

kuehn@ornl.gov, spoole@ornl.gov, hodsonsw@ornl.gov,
lothian@ornl.gov, bakermb@ornl.gov
schrockj@ornl.gov

June 19, 2013

Abstract

As high performance computing technology progresses toward the progressively more extreme scales required to address critical computational problems of both national and global interest, power and cooling for these extreme scale systems is becoming a growing concern. A standardized methodology for testing system requirements under maximal system load and validating system environmental capability to meet those requirements is critical to maintaining system stability and minimizing power and cooling risks for high end data centers. Moreover, accurate testing permits the high end data center to avoid issues of under- or over-provisioning power and cooling capacity saving resources and mitigating hazards. Previous approaches to such testing have employed an ad hoc collection of tools, which have been anecdotally perceived to produce a heavy system load. In this report, we present SystemBurn, a software tool engineered to allow a system user to methodically create a maximal system load on large scale systems for the purposes of testing and validation.

*This work was supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory. This work was also supported by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL). ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725.

Contents

List of Figures

List of Tables

1 Introduction

In navigating the path forward in extreme scale computing, power and cooling have become central concerns. Moreover it is not sufficient to rely on vendor estimates of power and cooling requirements, since these requirements are based on typical start-up and operation of small scale test-stands and include sufficient "padding" to cover variability in actual components. Thus, power and cooling estimates for systems-at-scale may be over-provisioned for transient response peaks in both power and cooling which do not overlap in time, i.e.. the highest power draw is typically at the instant of system start-up, during which the system is nominally at ambient temperature, requiring no cooling, and hence, the power required for cooling is at a minimum. Of more concern are the situations in which the distribution of cooling capacity is poorly matched to the actual thermal load of the running system, creating situations which could result in damage to extremely expensive equipment or hazards to operations staff. Knowledge of the actual power and cooling loads for a system in its actual operational environment permits accurate provisioning of power and cooling capacity. Furthermore, operational risks and hazards can be significantly reduced if it is possible to accurately assess the correctness and sufficiency of cooling capacity distribution for the system. A methodology which permits re-assessment as other systems are added or replaced in the environment is key.

2 Vision

Traditional benchmarks probe only a few of a machine's hardware features at a time, which doesn't give a good overall feel for how the system acts under all kinds of stress. Conversely, SystemBurn has been engineered to place a system under composite stress. Under user-control, the induced stress can be either purely synthetic, to stress the system's operating envelope, or a composite load intended to provide some emulation of a real application or suite of applications, to examine "typical" operating conditions. SystemBurn was created to allow the user flexibility when choosing which hardware components are to be stressed and how that is to occur.

SystemBurn provides a capability for the creation and control of multiple concurrent artificial system loads. These loads can be used to test a system under a variety of stress states and to maintain those stress states in a controlled fashion allowing for collection of physical measurements of various parameters of interest on the system and its environment, including but not limited to temperatures, air flows, and power draws, both per node and for the overall environment. These data sets can then be used to determine which loads cause the most stress, in terms of component temperature, power drawn, heat generated, etc. Thus, with purpose-crafted loads, the actual peak run time power requirement and thermal load can be measured, providing data for the assessment of sufficiency of power and cooling capacity and the data required for the reallocation of power and cooling in excess of the actual requirement, rather than relying on power and cooling estimates from the vendor. From the data gathered using SystemBurn it will be possible to compile power and temperature profiles for specific tasks. These profiles have the potential to eventually trace the tasks running on a machine simply by looking at the profiles. Through the use of multiple load modules, each of which stress the system in a specific way, it is possible to assemble loads representing an extreme system stress, or loads which perform similarly to some single large application.

3 Design and Development

Phase 0 of the SystemBurn Project created a proof-of-concept prototype which demonstrated the ability to track physical response to various computational loads placed on a single server [1]. It was prototyped using a combination of UNIX Shell, Perl, Python, and C, using existing standalone benchmarks, and thus was not suitable for use on MPP systems or systems at extreme scale. The prototype was tested on a variety Linux servers and nodes of a small Linux cluster, to demonstrate the correspondence between various workload tests and thermal response. Monitoring was strictly external to the benchmark and required manual correlation of workload profiles to temperature profiles.

Phase 1 of SystemBurn (the current version) advances the concept by extending coordination across multiple nodes of clusters and MPP systems. Phase 1 is a hybrid distributed/multi-threaded parallel benchmark implemented in C, with MPI (to coordinate loading and aggregate state between nodes) and Pthreads (to monitor, schedule, and execute loads within a node). The lightweight, low-level multi-threading enabled by Pthreads is used to fork multiple threads of execution on either single or multiple processor core(s). Each thread has responsibility for managing one task concurrent with all other activity on the node. Most of these threads execute a load module that targets a specific hardware component or emulates the stress induced by a particular algorithm. Two other threads are critical to the design: (1) a node monitoring thread, which is responsible for collecting data on the node on which it is running and reporting this to (2) the scheduler and coordination thread which handles communication and scheduling, including the aggregation of error events and monitor data across the nodes, and the load and placement specifications.

The user specifies the loading and placement using a simple structured specification syntax language to describe the size and type of the loads to be run by the worker threads, and the placement of worker threads within a node. This simple specification is replicated across the plans within a particular job. Heterogeneity can be achieved by running several such jobs with different load specifications concurrently.

The Phase 1 prototype supports thread placement and temperature monitoring capability when running on newer versions of the Linux kernel. For older kernels, the prototype code recognizes the missing facilities and disables in-site monitoring. A small library of prototype loads was developed to flesh-out the design of the load library's API's. The resulting load library is highly modular and easily extensible by the end user. Phase 1 is capable of running a variety of modular load stresses across multiple scales of computer systems. It has thus far been tested on laptops, servers, clusters, Cray XT systems, and IBM BlueGene systems.

For future Phases, SystemBurn will continue to pursue the goal of putting extreme (abnormal) stress on the system while measuring parameters of interest, e.g. temperature, fan speeds, load on the system, IPS and performing periodic data integrity checks. We hope to use SystemBurn to characterize how performance and accuracy are affected by extreme heat, determine the physical limits of the chip, and determine whether or not a particular chip is capable of meeting the demands of extreme-scale software applications. Eventually, we hope to determine if we can drive a chip to such abnormal levels that excessive heat may result in a loss of numerical accuracy or may possibly cause physical damage to the chip, e.g. destroying elements or the entire chip. We hope to leverage this knowledge to address weaknesses in chip design. As with the CPU impact, we intend to be able to eventually exercise every element of a system, e.g. networking, I/O, and accelerators. If it is required, we will code elementary operations in the assembly language of the target machine. This will only be done if we find that we can not generate enough of a workload to severely

stress the elements under test. In addition we plan to use SystemBurn to determine from an algorithmic point of view, how the computational complexity of a variety of algorithms will affect power load on current and future systems. We will also use the results from SystemBurn to determine potential workload distributions as well as power signatures to be used in our power analytics.

4 Implementation

The following section gives a description on the techniques and tools used to implement SystemBurn.

4.1 Threads

To generate parallelism on the node level PThreads are used. This allows for a very light weight way to to bind a load module or a set of load modules to a specific CPU set or even a specific CPU. SystemBurn generates three different types of threads:

Scheduler Thread A single scheduler thread is spawned per node. This thread reads the load file and distributes the different load modules to the worker threads. The scheduler thread also assigns the worker threads to the proper CPU cores according to user input from the load file. This thread is also responsible for communication with other nodes through MPI calls including the communication test load.

Monitor Thread A single monitor thread is spawned per node. On systems with newer kernels, this thread is in charge of collecting temperatures from all of the CPUs on the node and updating a thermal state structure which is referenced by the scheduler thread. If the monitor thread notes a temperature which exceeds the configured maximum, it can "declare" a thermal emergency and re-assign the worker threads to a "sleep" load.

Worker Thread Worker threads are spawned to execute the different load modules specified by the load file. The number of worker threads that are able to be spawned is limited by the `MAX_WORKER_THREADS` value in the configuration file. If the load file specifies more load modules than available worker threads, then only the first load modules up to the maximum number of worker threads are loaded. The rest of the modules are discarded.

4.2 Communication and Control

While SystemBurn uses PThreads for most of the "heavy lifting" parallelism, a message passing style framework is used for control, coordination, and communication between nodes. This function is served by MPI or SHMEM. All of SystemBurn's communication is executed within the context of the scheduler thread on each node to accommodate those MPI implementations which do not permit multiple PThreads to concurrently call MPI communication routines. MPI or SHMEM is used for the following tasks:

- broadcasting of command line and configuration file input
- broadcasting of load information
- collection and reduction of temperature data

- collection and reduction of error flag data
- collection and reduction of load performance data
- a simple communication load to stress the nodes' NICs

4.2.1 Rationale:

MPI was initially chosen to maximize the initial portability and market acceptance with the intention that the simple communications constructs used could be easily substituted with OpenSHMEM equivalents by conditional compilation, once OpenSHMEM is available and stable. The system was then extended to use SGI and Cray style SHMEM, which closely resemble the OpenSHMEM API.

4.3 Error Flags

An error flag system has been implemented in SystemBurn as a way of tracking possible system failures while running the benchmark. If running SystemBurn causes a total system failure, the information printed to the terminal while running the benchmark should be enough to pinpoint what module, or combination of modules, caused the hardware failure. Through the periodic printing of error flag statuses across the machine, as well as current loads, the user has a brief, yet detailed, overview of the current benchmark progress on the machine.

There are several different types of error flags which SystemBurn tracks.

- Memory Allocation Failures
- Affinity Mask Failures
- Module Calculation Errors
- Module Communication Errors
- Module Specific Errors

4.4 Program Flow

4.4.1 SystemBurn Scheduler Thread(s): (src/systemburn.c:main()):

- Initialize MPI/SHMEM
- All scheduler threads initialize global variables from command line options
- ROOT node scheduler thread (MPI/SHMEM rank 0) reads and broadcasts the configuration file to all ranks.
- All scheduler threads initialize the local error flag system
- All scheduler threads initialize the local performance data gathering system
- All scheduler threads spawn a single monitor thread on their node
- All scheduler threads spawn multiple worker threads on their node (initial load: "sleep")
- For each load do the following:

  - ROOT scheduler thread reads and broadcasts the next load description
  - ROOT scheduler thread logs the new load to standard out
  - All scheduler threads update the plan structures for the worker threads on their node according to the new load
  - While the new load runs:
    * All scheduler threads on all nodes run the communication test (if it is enabled)
    * ROOT scheduler thread checks time and broadcasts continuation, completion, and output flags
    * All scheduler threads perform reduction and output of temperature state and error flag state at intervals

- Upon completion of all loads, stop all of the worker threads and clean up their traces in memory
- Sleep function: Sleep to get temperatures back to idle level
- Perform global reductions on gathered performance data and calculate simple statistics
- ROOT node logs performance data to standard out
- Finalize communication framework
- Exit

4.4.2 SystemBurn Worker Threads: (src/worker.c:WorkerThread())

- Each worker thread consists of the following loop:
  - If the scheduler passes a NULL load plan, terminate this worker thread
  - If the scheduler passes a new load plan:
    * Adjust thread affinity as required (newer kernels only)
    * Log performance data from the previous load plan
    * Clean up the previous load plan
    * Install the new load plan
    * Initialize the new load plan
    * If the initialization fails switch to the "sleep" plan
  - run the load

4.5 PAPI

If the Performance API (PAPI) library is installed on the system being used, SystemBurn can be configured to collect and report the system counters provided by PAPI. The desired counters must be specified in the plan modules that are to be used. The templates, `new_module.c` and `new_module.h` in the `planlib` directory, provide the format for incorporating the PAPI counters into a module. As many counters as desired can be added to the `PAPI_COUNTERS` structure; however, there is a global limit on the number of counters actually recorded and reported. This limit, `TOTAL_PAPI_EVENTS`, is in `src/performance.h` and can be altered as needed. Additional PAPI documentation, including a list of the available counters, can be found at `http://icl.cs.utk.edu/papi`.

4.6 Functions

- initialization.c

  int initialize Primary initialization phase for the SystemBurn benchmark. This function opens several input files and an output log, using the inputs information to set the program's global variables.

  int setLogName Handles the details of assigning a log file name obtained from the command line to a variable for later use.

  int setLoadNames Stores the names and paths of the load files in an array of strings.

  void freeLoadNames Frees the memory associated with the names of the loads.

  int setVerbosity Sets a global flag variable, dependent on the -v command line option, to determine what information is printed to the terminal during a run.

  void printHelpText Prints help information to the terminal when the -h option is given on the command line.

  int bcastConfig Performs a one-time broadcast of global configuration data to all MPI processes.

  int initConfigOptions Opens the configuration file and assigns the values in it to global variables.

  int parseConfigFile Parses the configuration file.

- load.c

  int bcastLoad Broadcasts a single load assignment to each MPI process.

  int initLoadOptions Opens a single load file to be parsed.

  void printLoad Prints the current load information to the terminal.

  void printSchedule Prints the name of the input affinity schedule enum value to the terminal.

  void freePlan Frees memory allocated within a LoadPlan structure.

  void freeSubLoad Frees memory allocated to a SubLoad structure.

  void freeLoad Frees all memory that was dynamically allocated to a load structure.

  schedules setSchedule Takes an affinity schedule name as a string and returns the enum value.

  int parseLoad Parses the load file.

  keyword keywordCmp Accepts a character string input and returns the associated load file keyword.

  int runtimeLine Parses a single load file line headed by the keyword `RUNTIME`.

  int scheduleLine Parses a single load file line headed by the keyword `SCHEDULE`.

  int subloadLine Parses a single load file line headed by the keyword `SUBLOAD`.

  int allocSubload Allocates memory to store a single sub-load.

  int planLine Parses a single load file line headed by the keyword `PLAN` and populates a loadplan structure with the data.

  assignPlan Inserts a populated plan structure into the currently active sub-load structure.

int maskLine Parses a load file line headed by the keyword `MASK`.

int assignMask Inserts a CPU set into the active sub-load structure.

int assignSubLoad Assigns the active sub-load structure to a load structure.

LoadPlan * planCopy Accepts a loadplan structure, produces a copy and returns a pointer to the new copy of the loadplan.

int sumArr Adds the integer members of an array and returns the sum.

plan_choice setPlan Takes a plan name in string form and returns the enum value.

char * printPlan Takes a plan enum value and prints the name to terminal.

- monitor.c

void CheckTemperatureRange Checks the temperatures of the system. Currently this function is only set to work with Linux systems with ACPI * enabled.

void CheckTemperatureRange Checks the range for the temperature readings.

void EmergencyStop Kills everything immediately to preserve the system. This function is called if the system state exceeds the operating parameters.

void * MonitorThread Sleeps in a loop periodically waking up to check the state of its node. Will periodically output a state description.

void StartMonitorThread Starts the monitor thread.

void printTemp Prints the current state of the minimum, mean, and maximum temperature values for the entire system.

void reduceTemps Gathers temperature data from every MPI process and calculates the minimum, mean, and maximum temperatures among all nodes.

- schedule.c

int WorkerSched Assigns modules to individual worker threads as dictated by the load file.

void MaskBlock Sets the affinity mask for the CPU sets in a block type fashion.

void MaskRoundRobin Sets the affinity mask for the CPU sets in a round-robin fashion.

void MaskSpecific Sets the affinity mask from each individual sub-load's CPU set, as specified by the user.

void ZeroMask Initializes the affinity masks for each thread.

void SetMask Sets the mask for the CPU sets based on the selected option in the load file.

- systemburn.c

int main Main function for the SystemBurn benchmark.

- utility.c

void EmitLog Prints different messages to the screen with time stamps as well as information about the tread calling the function, i.e. what type of thread, what thread ID number and MPI process.

void EmitLog3 Same as EmitLog but with the ability to print more information to the screen if needed.

- worker.c

int InitPlan Sets a worker threads pointer to the `fptr_initplan` function of its current plan. This allocates memory for the current plan to run.

int runPlan Sets a worker threads pointer to the `fptr_execplan` function of its current plan. This executes the plan.

void * killPlan Sets a worker threads pointer to the `fptr_killplan` function of its current plan. This cleans up the memory used by the current plan in preparation for a new plan.

void StartWorkerThreads Sets up affinity masks for CPU sets. Spawns worker threads and starts them with sleep plan.

void StopWorkerThreads Tells all of the worker threads to finish in preparation for ending the SystemBurn benchmark.

void * WorkerThread Continuously checks to see if a new plan has been issued by the scheduler to the worker threads. If a new plan is issued the old plan is cleaned up and the new plan is initialized and executed. This function also collects flags from initialization and execution of plans.

void initWorkerFlags Initializes the workers flag counters to zero.

void reduceFlags MPI Reduces all of the flag arrays to the ROOT MPI process.

void printFlags Prints updates on the current flag counts across the system to the terminal.

5 Usage

Running SystemBurn on a given machine is a four step process.

Build: Build the SystemBurn suite. This requires running the `configure` script to set up the suite for your system (or manually modifying `Makefile.in` to point to the appropriate libraries and compilers for the target system and defining the appropriate macros in `config.h`). Once the build is configured simply type:

- `make clean`
- `make`

Config: Create a configuration file or modify the default file, `systemburn.config`. Configuration section of this document contains detailed instructions on how to do this.

Load: Create a load file or modify the default load in `systemburn.load`. This includes determining which modules, or plans, to load and how to load them on the architecture. The Load Files section of this document contains detailed instructions on how to do this.

Run: Determine which nodes will be running SystemBurn. If a homogeneous node distribution of the load is desired then setting up a single MPI tasks per node will be sufficient. If a heterogeneous node distribution of the load is desired then the following steps must be taken:

- Create all the different loads to be used. This will require multiple load files (one for each different load).
- Set up a batch script for the system which will run different instances of SystemBurn on a set number of nodes. (Each instance of SystemBurn will be given one of the load files set up in step 1).
- Run SystemBurn either from the command line for the homogeneous node distribution or with the batch script for the heterogeneous node distribution.

5.1 Command Line Interface

SystemBurn is designed to be run using a command line interface and has several options and other arguments that control run-time behavior. Also, because SystemBurn is based on a message passing style framework, using MPI or SHMEM, for inter-node communication (see the Implementation section), if running multiple nodes is required, the appropriate runtime command, such as `mpirun`, should be used to execute SystemBurn. For general use, a call to run SystemBurn should have options occurring first and any number of load files as non-option arguments. It should look like this:

```
./systemburn [OPTIONS...] [LOAD FILES...]
```

The following command line options can be used with SystemBurn:

-c #bytes Enable running the inter-node communication load with a message size of `\#bytes`. Initial testing suggests that sizing the messages just below the MPI "Eager Limit" (usually 64kb) is reasonable.

-f "config file" This option is used to specify a configuration file, when it is used it requires a file name as an argument. If this option is not used, the default configuration file, `systemburn.load` is used.

-l "log file" In the future, this option will be used to specify a file in which to log data produced during run time.

-n # of loads This option allows the user to clearly specify the number of non-option arguments to use as load files, using the specified number or all arguments, whichever is greater. Without this option, every non-option argument will be treated as a load file.

-t This option enables calculation error checking in loads that have that capability. The default behavior, without this option, disables error checking.

-v "output level" This option is used to specify the level of output SystemBurn will produce.

- Level 0, the default, produces basic, scalable output that displays the current load at initialization, periodically displays a summary of temperatures and errors that have occurred during runtime, and gives a summary of load performance at completion.
- Level 1 is not as scalable, with every MPI process printing status and performance output in addition to level 0's output.

- Level 2 adds output from each MPI task's sub-threads.
- Level 3 adds additional debugging information to the output.

-h This option displays the help text for SystemBurn, with a usage template and a description of each command line option.

An example of a call to SystemBurn:

```
mpirun -np 4 ./systemburn -c 64000 -v0 -f system.conf \
           systemload1.load systemload2.load
```

5.2 Configuration Files

The configuration file, `systemburn.config`, contains several important values which control the SystemBurn suite's overall behavior. A note on syntax: all characters on a line after a `#` symbol are treated as comments, and each parameter is associated with a specific keyword that must precede the value on that line; if one of the keywords is not used, a default value will be substituted.

Maximum Number of Worker Threads Keyword: `WORKERS`. Default: 16 threads. This value specifies the maximum number of execution threads per MPI process to create when scheduling the current load. If the number of module loads in the load file exceeds the number given here, then the entire module load will not be scheduled, but only the first modules up to this maximum number.

Thermal Relaxation Time Keyword: `REST_TIME`. Default: 10 seconds. This is the time period for which the code will idle before beginning the set of loads and then again after completing the set of loads. The intent is to allow the idling system to come into a baseline thermal equilibrium before and after the loads. It should be noted that the loads should also run long enough for the system to come into a stressed thermal equilibrium for an accurate max temperature. This value can be set to 1 for quick testing. For small systems which have little impact on their environment values between 30-300 seconds are generally sufficient. Recommendations for values for larger systems which do affect their environment have yet to be developed but will likely run much longer to allow machine room power and cooling to also reach equilibrium.

Emergency Shutdown Temperature Keyword: `MAX_TEMP`. Default: 70 degrees Celsius. This value specifies the maximum temperature a node can reach before SystemBurn shuts itself down. This feature is only enabled on newer kernels which support temperature monitoring.

Monitor Thread Checking Period Keyword: `MONITOR_FREQ`. Default: 1 second. This specifies how often the monitor thread retrieves temperature data from the system. This feature is only enabled on newer kernels which support temperature monitoring.

Monitor Thread Output Period Keyword: `MONITOR_OUT`. Default: 10 seconds. This specifies how often the monitor thread will print its temperature information, if output from threads other than the main thread have been allowed (see the verbosity command line option, above). This feature is only enabled on newer kernels which support temperature monitoring.

An example configuration file appears as follows:

```
# systemburn.config

# This file contains the default configuration
# information for systemburn. Comments are
# denoted by a '#' sign, and the needed numbers
# are described by the comments below:

# Number of worker threads:
WORKERS 16

# Thermal Relaxation time: (seconds)
# This is the time period for which the code will idle before
# beginning the set of loads and then again after completing the
# set of loads. The intent is to allow the idling system to
# come into a baseline thermal equilibrium before and after the
# loads. It should be noted that the loads should also run long
# enough for the system to come into a stressed thermal equilibrium
# for an accurate max temperature.
# This value can be set to 1 for quick testing.
# For small systems which have little impact on their environment
# values between 30-300 seconds are generally sufficient.
# Recommendations for values for larger systems which do
# affect their environment have yet to be developed but will likely
# run much longer to allow machine room power and cooling to also
# reach equilibrium.
REST_TIME 10

# The following are only useful on newer systems that support
# internal temperature monitoring via the kernel's /sys and
# /proc interfaces. Generally, this requires that the appropriate
# Linux kernel module (k8temp, k10temp, coretemp) or equivalent
# be loaded into the kernel.

# Emergency Shutdown Temperature: (degrees Centigrade)
# Over Temp protection requires access to the processor temperature.
# A few important notes:
# on AMD systems (esp k8) the
MAX_TEMP 85

# Temperature Monitoring Frequency (in seconds)
# This determines how often the internal core temperature data is
# checked/updated and how often we check for an overtemp emergency.
MONITOR_FREQ 5

# Monitor thread ouput period:
# This is how often we summarize and print the temperatures.
```

```
# Note that the summary requires an MPI_REDUCE or equivalent
# and can thus be expense on larger system partitions. Adjust
# the output frequency accordingly:
# Small systems: 15-60 seconds
# larger systems:
MONITOR_OUT 15
```

5.3 Load Files

The load file for SystemBurn has been designed with flexibility in mind. During execution, a single load is run at a time, with the same load running on each MPI process. Each load contains a run time value and is also subdivided into one or more sub-loads, each of which contain some number of plans. Each sub-load is also associated with a CPU set mask, some subset of the CPU cores available to the MPI process running the load. With this association, plans contained in a sub-load with a given CPU set can be required to run on only the CPUs in that set. An example use of this capability is running SystemBurn on a dual socket system. Two sub-loads could be defined, with CPU sets that placed each sub-load on its own socket. Then, a third sub-load could have a CPU set encompassing both sockets. These CPU sets can be managed at two levels: individually, with custom masks for each sub-load; or at the load level, by dividing the available CPU cores up based on the number of sub-loads and a user specified pattern (block or round robin). More detailed description of these load parameters are given below.

The load file allows the user to set the following general parameters for each load:

Run Time This allows the user to specify how long they would like an individual load to run for.

CPU Set Type This parameter has three different options:

1. Block: This sets the CPU masks such that each sub-load is assigned a set of adjacent CPU cores. For example: A node with 12 CPUs and 3 CPU sets will have an affinity mask which looks as follows:

   ```
   SET     CPU(s)
    0      0  1  2  3
    1      4  5  6  7
    2      8  9 10 11
   ```

2. Round Robin: This sets the affinities in the classic round robin distribution. For example: A node with 12 CPUs and 3 CPU sets will have an affinity mask which looks as follows:

   ```
   SET     CPU(s)
    0      0  3  6  9
    1      1  4  7 10
    2      2  5  8 11
   ```

3. Sub-load Specific: This option allows the user to specify exactly which cores are to be put in each CPU set. This allows for a very low level of control when it comes to which CPUs run a specific set of modules. To use this affinity type the individual masks for each CPU set must be specified within the associated sub-load. This can be used to set more unbalanced loads:

```
SET     CPU(s)
 0      8 9
 1      1 2 3 4 5 6 7
 2      0
```

In addition to the general parameters that apply to the entire load, the load can also contain any number of sub-loads, each possessing several other pieces of information associated with that sub-load:

Copies A single number in each sub-load determines how many distinct copies of that sub-load should be run as part of running the full load. This allows the file to be more concise when repetition occurs in the load.

CPU Set Mask This is a list of CPU cores that should be included in the sub-load's CPU set (if the load's CPU set type is sub-load specific). If there are any copies of the load, a CPU set mask should be specified for each copy.

Also, each sub-load will contain one or more plans, each of which has its own parameters:

Copies For the same reason that sub-loads can specify copies, a single parameter in a plan specifies the number of copies of that plan are in the sub-load.

Load Module Ultimately, most of SystemBurn's work is performed by load modules (see the next section, Modules). Each plan contains the name of a single module for it to run.

Load Module Parameters Each load module takes some kind of input parameter(s). Thus, each plan requires values that serve as the input parameters for the specific module used. Note: plans accept a parameter for the amount of memory the plan should use. This can be specified by the size followed by the byte prefix; i.e. MB, KB, GB ... ; m,k,g,p,t, and e work as well.

The amount of customization available within the load files should allow SystemBurn to emulate a vast array of different application stresses. The run time parameter can be altered to run several different modules in a short amount of time, or a few modules for a very long period of time. The number of CPU sets and the different affinity types allow the user to bind threads to specific CPUs or to have unbound threads float over the entire node. A combination of these 2 extremes is also possible.

The syntax of the load files is based on a collection of keywords, each specifying a certain function. Each keyword is then followed by the necessary arguments (if any) on the same line.

LOAD_START This keyword signals the beginning of a load entry within the file. It can be replaced with an opening brace: `"{"`.

RUNTIME Specifies a run time value (in seconds) for the load as an integer immediately following the keyword.

SCHEDULE Specifies a CPU set type from the three options described above, using these keywords: `BLOCK`, `ROUND_ROBIN`, or `SUBLOAD_SPECIFIC`.

SUBLOAD Acts as a header to a sub-load, indicates the number of copies of the next sub-load as an integer following the keyword.

SUB_START Begins a sub-load entry, within a load entry. It can be replaced with an opening square bracket: `"["`.

PLAN Starts a plan entry. Immediately followed by the number of copies, the module name, and all module parameters.

MASK Specifies a CPU set, and is followed by any number of integer core numbers to be associated with the current sub-load. This keyword is only used when `SUBLOAD_SPECIFIC` is the CPU set type. Also, If a sub-load has multiple copies, the `MASK` keyword must be used multiple times: one for each copy.

SUB_END Ends a sub-load entry. It can be replaced with a closing square bracket: `"]"`.

LOAD_END Signals the end of a load entry, and can be replaced with a closing brace: `"}"`.

Below is a template for a load file used by SystemBurn.

```
LOAD_START
        # GENERAL INFORMATION
        RUNTIME <# of seconds>

        # Affinity Type
        SCHEDULE <BLOCK, ROUND_ROBIN, or SUBLOAD_SPECIFIC>

        # SUBLOADS (Specify different loads for different cpu sets)

        SUBLOAD <# of copies>
        # Number of cpu sets to run this subload on
        SUB_START
                PLAN <# of copies> <module> <module parameters...>
                ...

                MASK <cpu cores...>
        SUB_END

        ...

LOAD_END
```

Example Load File 1:

```
LOAD_START
        # 16 core system
        RUNTIME 60
        SCHEDULE SUBLOAD_SPECIFIC

        SUBLOAD 3
        SUB_START
                PLAN 2 GUPS 25m
                PLAN 1 DSTRIDE 1000000m
```

```
                PLAN 1 FFT2D 3000m

                MASK 0 2 4 6
                MASK 1 3 5 7
                MASK 8 9 10 11
        SUB_END

        SUBLOAD 1
        SUB_START
                PLAN 2 FFT1D  1000000Mb
                PLAN 1 GUPS 24M
                PLAN 1 GUPS 26m

                MASK 12 13 14 15
        SUB_END
LOAD_END
```

Example Load File 2:

```
{
        # simple load for a four socket system with 8 cores per socket
        # each socket runs 4 smaller PV1 loads and 4 larger PV2 loads
        RUNTIME 450
        SCHEDULE BLOCK
        SUBLOAD 4
        [
                PLAN 4 PV1  800m
                PLAN 4 PV2 1600m
        ]
}
```

5.4 Modules

SystemBurn is designed to run several different modules, or plans, simultaneously. These modules all do different types of calculations and stress different areas of the architecture. The user is able to load new modules at any time so as to fine tune the load. The modules which are included in this release of SystemBurn are:

DGEMM size A double precision matrix multiplication benchmark which will run to consume "size" bytes of memory.

RDGEMM size A double precision rectangular matrix multiplication benchmark which will run to consume "size" bytes of memory.

LSTREAM size Streaming integer vector operations run to consume "size" bytes of memory.

DSTREAM size Streaming double precision floating point vector operations run to consume "size" bytes of memory.

LSTRIDE size An integer load which accesses memory with changing stride, using "size" bytes of memory.

DSTRIDE size A double precision floating point load which accesses memory with changing stride, using "size" bytes of memory.

FFT1D size A 1 dimensional complex fast Fourier transform in a memory footprint of "size" bytes.

FFT2D size A 2 dimensional complex fast Fourier transform in a memory footprint of "size" bytes.

GUPS size Giga Updates Per Second - a random memory access benchmark on a table of "size" bytes. Note that "size" must be a power of 2, if it is not, it will be adjusted to the largest power of 2 which will fit within "size" bytes.

PV1 size A power hungry streaming computational algorithm on four arrays of 64bit values, which will operate with a memory footprint of "size' bytes.

PV2 size A power hungry streaming computational algorithm on one array of 64bit values, which will operate with a memory footprint of "size" bytes. This load was tuned to a quadcore Intel "Nehalem" processor, but may be suitable for loading multiple x86-64 cores until the memory system is saturated. It is intended to be run with a footprint large enough to require main memory access.

PV3 size A power hungry streaming computational algorithm on one array of 64bit values, which will operate with a memory footprint of "size" bytes. This load was tuned to a quadcore AMD "Instabul" processor, but may be suitable for loading multiple x86-64 cores until the memory system is saturated. It is intended to be run with a footprint large enough to require main memory access.

PV4 size A power hungry streaming computational algorithm on one array of 64bit values. It is intended to run in a smaller memory footprint which will be contained in L2 cache, not inducing main memory traffic.

CBA size A bit-twiddling load which will run within "size" bytes of memory.

TILT niter A bit-twiddling load with a small memory footprint, "niter" iterations at a time.

DCUBLAS device count threads A CUDA double precision load for GPUs. The load is sized automatically to memory available on the GPU. The GPU "device" number, the "count" of iterations per pass, and the number of GPU "threads" to be used, may optionally be specified. The defaults are device 0, count 8, and a thread count appropriate to the device hardware.

SCUBLAS device count threads A CUDA single precision load for GPUs. The load is sized automatically to memory available on the GPU. The GPU "device" number, the "count" of iterations per pass, and the number of GPU "threads" to be used, may optionally be specified. The defaults are device 0, count 8, and a thread count appropriate to the device hardware.

DOPENCLBLAS device count threads A OpenCL double precision load for GPUs. The load is sized automatically to memory available on the GPU. The GPU "device" number, the "count" of iterations per pass, and the number of GPU "threads" to be used, may optionally be specified. The defaults are device 0, count 8, and a thread count appropriate to the device hardware.

SOPENCLBLAS device count threads A OpenCL single precision load for GPUs. The load is sized automatically to memory available on the GPU. The GPU "device" number, the "count" of iterations per pass, and the number of GPU "threads" to be used, may optionally be specified. The defaults are device 0, count 8, and a thread count appropriate to the device hardware.

DOPENACCGEMM device count threads A OpenACC double precision load for GPUs. The load is sized automatically to memory available on the GPU. The GPU "device" number, the "count" of iterations per pass, and the number of GPU "threads" to be used, may optionally be specified. The defaults are device 0, count 8, and a thread count appropriate to the device hardware.

SOPENACCGEMM device count threads A OpenACC single precision load for GPUs. The load is not sized automatically to memory available on the GPU, since OpenACC provides no access to this information. The GPU uses "size" bytes, "device" number, the "count" of iterations per pass, and the number of GPU "threads" to be used, may optionally be specified. The defaults are device 0, count 8, and a thread count appropriate to the device hardware.

WRITE megabytes string A I/O load which writes "megabytes" to files with "string" as the basename (this can be a path).

SLEEP N Puts a thread to sleep for N seconds at a time.

Note: in most places a size in bytes is requested, the integer may have K, M, G, or T appended to indicate kilo, mega, giga, tera, (power of 2 based).

5.5 Adding New Modules

There are templates included in the `planlib` directory to aide in creating new modules. These files are called `new_module.c` and `new_module.h`. They contain the necessary format for creating new modules. The steps to add new modules to SystemBurn are as follows:

1. Make a copy of `new_module.c` and name the file accordingly.
   e.g. `plan_NewModule.c`
2. Do the same for the header file, `new_module.h`. This is where you will create all of the structures for the new module.
3. Be sure both files are in the `planlib` directory.
4. Be sure the names of all the functions and their prototypes match. Also, it is recommended that they fall into the same naming scheme as the existing loads. i.e. makeFFT1DPlan(*) (FFT1D is the name of the load.) However, this naming scheme is not necessary for the program to function. You need the 6 listed functions, as well as a plan_info struct.
5. If using PAPI, edit the PAPI variables at the top of the file to reflect the appropriate number of events you wish to track, the names of the events to track, and unit descriptions for these events. No other changes to PAPI specific functionality within the module should be needed.

6. In the make function, one of the parameters that it will receive will be the memory footprint size for the main data structure of your plan. You will need to convert this into the variables you need in order to correctly allocate memory.
   (a) A good example is `plan_dgemm.c`: in its make function, it must separate the given memory footprint into 3 identical 2D arrays of doubles. To do this, it divides by `3*sizeof(double)`, then takes the square root of the result. This results in a total memory usage of the value given in the load file.
7. Change the init, kill, and parse functions to use your structs and data structures.
8. Write whatever is needed to execute the exec function. It is preferable to use additional compute functions (declare the prototypes in the header) if the code is lengthy. The given template provides two sections that can be filled: an execution phase and an optional calculation checking phase. Also demonstrated is the use of performance timers, of which 3 are available and 2 are used.
9. Anywhere there is a potential fatal error (malloc, etc.) or a calculation error, be sure to have the load return flags to indicate if anything goes wrong. See existing loads for examples. This is actually done by setting the dummy variable, `ret`, to the flag value. To set `ret`, call the `make_error` function, with either one of the enum values(i.e. ALLOC for allocation errors) or the index value of your custom error messages.
10. Change the perf function to calculate the correct operation count for each of the timers used in the exec function. For example, the DGEMM plan performs on the order of `2*M*M*M` floating point operations per execution, where M is one dimension of the matrix. So, to calculate the total number of operations, this value is multiplied by the execution count of the module.
11. Add your plan_info struct into the .c file. This consists of:
    (a) Plan name (actually the `plan_choice` enum value you will set in the next step.
    (b) Name of your array of custom error messages (if none are needed, then you can just say `NULL`.)
    (c) Number of messages in your custom array (if you used `NULL` this is 0.)
    (d) Name of your make function.
    (e) Name of your parse function.
    (f) Name of your exec funtion.
    (g) Name of your init function.
    (h) Name of your kill function.
    (i) Name of your perf function.
    (j) An array of 3 strings containing the units to be associated with the performance timers. (`NULL` if that timer is not in use.
12. Edit the `planheaders.h` file as in the following steps:
    (a) Add the new module's name to the `plan_choice` enum. Be sure to place it at the end of the lists, but before the `UNKN_PLAN` value, which is used to denote the length of the enumeration.

(b) Add your `plan_info` struct to the `plan_list` array. Be sure it is in the same index position as your module is in `plan_choice`.

(c) Include the new header file in the section with all of the #include's. Also include any additional headers needed to run the new load.

(d) To run the new module add it into your load file using the name you gave it in your plan_info struct. Compile and run Systemburn.

Acknowledgments

This work was supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory.

This work was also supported by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL). ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Ladd, J. S., Lewkow, N. R., Poole, S. W., and Hodson, S. W. Systemburn: A total system benchmark. Internal design study report for a script-based prototype, Aug. 2009.