Cloud Management for Energy and Performance

Overview

In this project you will implement an energy- and performance-aware scheduler for a cloud environment. The scheduler will be implemented in the context of a cloud simulator. The simulator takes as input a file that contains descriptions of servers and workloads. The file defines the number of servers, their types, their energy consumption, and performance and memory characteristics. The simulator also reads the workload specifications from the input file. The specifications define the number of workloads, and for each, its statistical distributions, the specification of the SLA, the required VM image, and the processing and memory requirements on a specific machine type. You will be given the object files of the simulator, along with two header files that define the programmatic types and function interfaces. Your task is to modify the file Scheduler.cpp with your implementation to minimize the overall energy consumption during a run, all while maintaining SLA requirements. You may not change the .h files. You will do a comparative study with four different algorithms. You may use any algorithms that you wish, including, but not limited to, the algorithms discussed in class. However, at least *one* algorithm must be one that you researched from the literature. You may have to modify or enhance these algorithms or may also invent your own algorithm(s) since none of the algorithms in the literature or discussed in class are presented at a level of detail that enables a full-blown implementation.

The output of your work will be:

- A set of input files that you invent, testing specific scenarios in a cluster including aspects such as workload surges (to test the algorithm's SLA compliance under stress), low-intensity workloads (to test the algorithm's ability to save energy), different machine and configuration files (to test the algorithm's ability to handle a heterogeneous environment), and any other aspects that you would like to include. You may also put files that are aimed at debugging specific scenarios in a short simulation run. A sample input file will be provided to you, showing you the syntax. The first deliverable of the project will be *one* input file suggested from each group. The TAs and instructors will select the best files and share them among the entire class. Students whose files are selected will be rewarded accordingly. Additional input files will be provided later during the project for the ultimate testing for performance and energy. We may also use our own input files for grading purposes.
- The modified code for Scheduler.cpp and any other files that you wish to include.
- A short report showing the actual performance under different workloads and a description of the algorithms that you used. The report should describe the

algorithms and specific performance and overall energy consumption of each. The report should end with a comparative study.
- For the top four performing projects in the class, an in-class short presentation showing the algorithm(s) and overall implementation effort. The top three performing projects will receive an additional bonus.

You will need at the beginning to spend some time reading through this document and familiarize yourself with the simulator interface. You should not need to know the inner workings of the simulator. Some approximations were made to promote robustness, but that should be invisible to you. What you need to focus on is:

- The choice of the algorithms that you will implement.
- Tracking the states of the system components and tasks using the information that the simulator provides during a run, and actuating based on this information to promote energy saving and/or performance optimizations.
- Testing and reporting the energy and performance results.

Grading:

The project will be graded for a total of 20 points. The grade distribution is as follows:
- Submission of input file: 2 points.
- Submission of code: 14 points.
- Submission of report: 4 points.

Bonuses:
- Top three performers: *up to* 3 points.
- Input file shared with the class: *up to* 2 points.

Bonus points will go to offset points missed in the two exams. They may not count toward your project grade.

Logistics

Deadlines:

- Your input file suggestions are due on February 28th at 5:00PM.
- Your code is due on March 28h at 5:00PM. This is a FIRM deadline.
- Your report is due on March 31st.

The amount of time allocated to the project exceeds the time that is needed. The purpose here is to give you enough flexibility to manage your effort. Please be aware that waiting until the last possible minute to start is likely to cause problems. Pace yourself and start early.

Groups:

The project can be implemented in groups of two. Please sign up for groups on Canvas in the appropriate group pages.

Extensions:

No extensions will be given. There is more than a month worth of time for you to plan. Do not wait until the last minute. Exceptions will be made when, in the instructor's judgement, you are facing a situation of documented force majeure in your life.

Coding:

The simulator is written in C++. It will be most straightforward to implement your work in C++. However, you can use any programming language that you wish, provided that you are responsible for implementing the translation between your favorite language and C++. And yes, any programming language will be acceptable (even Python!). However, you will be responsible for all the additional logistics if you do not implement your project in C++. For health-related reason, the instructor will not be able to help you debug your code if written in a language that he doesn't like (e.g., Python).

Internet Tools and LLMs:

The use of Internet tools such as ChatGPT, Co-Pilot, etc. is permitted. However, you must make the necessary attributions in your final report and in the code that you borrow. Be a scholar and do not plagiarize the fruits of labor of these tools. LLMs are people too, or at least they are behaving like people, aren't they?

Bugs and Reporting:

The simulator has been tested. You should not have problems in general. However, the instructor is only human, and humans do not write perfect code. If you suspect a strange behavior, you need to alert the instructor directly and immediately. Every effort will be made to fix problems promptly. From time to time, we may refresh the simulator object files to reflect bug fixes and other changes. You may also report a need for a specific feature that the simulator doesn't provide. Every effort will be made to evaluate your proposed need and respond appropriately given the constraints of time and available bandwidth and potential impact of changing the interface file on other groups.

Project files

The following files are available in GitHub at the following URI:
https://github.com/anish-palakurthi/cloudsim_eec
The repository contains the following files:

Object files: Init.o, main.o, VM.o, Task.o, Simulator.o, and Machine.o.

Source file: Scheduler.cpp (this is the file that you need to modify to implement your algorithm(s).

Include files: Interfaces.h, SimTypes.h, and Scheduler.hpp.

Build files: Makefile.

Sample Input file: Input.md

To access the project files, clone the repository into your hub. You must use the laboratory machines for this purpose. Read the README file for instructions on how to build the project files and how to run the simulator.

Getting Started

The following approach is recommended:

1. Access the project files and build the simulator. Use the supplied Scheduler.cpp and the sample input file to play with the simulator and get familiar with it.
2. Change some of the parameters in the Scheduler.cpp to see the differences that various energy management policies will work.
3. Identify the algorithms that you would like to implement.
4. Implement the algorithms separately and use unit testing to ensure your code is working.
5. Integrate with the simulator files.
6. Write the report.

Report Structure

The report should be short. It should identify and describe the algorithm(s) that you are using. Then, it needs to show the performance (SLA, energy consumed, and running time) for various input files. Please do not exceed five pages.

Simulation Interface

The simulation presents an interface to enable the implementation of algorithms in the Scheduler module. The following is a description of the interface grouped by functionality.

Virtual Machine Interface

Down Calls:

extern VMId_t VM_Create(VMType_t vm_type, CPUType_t cpu);

This function creates a virtual machine with a specific operating system and a CPU type. This is likely among the functions that the Scheduler should call during initialization. Additionally, virtual machines can be created and torn down during the regular simulation as appropriate. The function returns an identifier that can be used later to manipulate the virtual machine. This function raises an exception if a combination of virtual machine type and the CPU is not feasible. At this point, the following virtual machine types and CPU combinations are available:

- `LINUX`: This VM is valid for all CPU types.
- `LINUX_RT`: This VM is valid for all CPU types.
- `WIN`: This VM is valid only for CPU types `ARM` and `X86`.
- `AIX`: This VM is valid only for CPU type `POWER`.

These constants are defined in the file SimTypes.h as follows:

typedef enum { LINUX, LINUX_RT, WIN, AIX } VMType_t;

This function does not generate any exception.

extern void VM_Attach(VMId_t vm_id, MachineId_t machine_id);

This function attaches the virtual machine to a specific hardware machine. The hardware machine is identified by a unique identifier (machine_id). This step is necessary to enable the virtual machine to start accepting tasks and running them on the hardware. When the VM is attached to a hardware machine, some memory overhead is allocated to enable the VM to work (currently, about 8MB). The scheduler should choose a hardware machine with a CPU that is compatible with the VM's required CPU (which was selected in VM_Create(). Otherwise, an exception is generated. Also, an exception is generated if the virtual machine is attached to a hardware machine that is currently not ready (in some form of sleep mode or is completely powered off). The parameter validation routines require that the VM's and machine's identifier be valid, otherwise an exception is generated.

extern void VM_AddTask(VMId_t vm_id, TaskId_t task_id, Priority_t priority);

This function adds a task identified by a unique task identifier to a particular virtual machine (that was created using VM_Create()). This is the only way to run tasks in the simulator. The scheduler cannot run individual tasks on the machine hardware directly. The function also specifies a priority. The priority will be assigned to the task and will affect the task scheduler in the simulator. The simulator uses a 3-level priority scheme, and uses strict priority-based scheduling. The parameter validation routines require that the VM's identifier and task's identifier be valid, otherwise an exception is generated.

The priorities are defined as:

```
typedef enum { HIGH_PRIORITY,  MID_PRIORITY, LOW_PRIORITY } Priority_t;
#define PRIORITY_LEVELS 3,        // System has 3 levels of priority,
```

Note that the priority of the task can be changed later (see the task management interface). Priorities must be carefully managed: If you have tasks with stringent SLA requirements consider running them at a higher priority. However, keep in mind that starvation could occur to tasks of lesser priority on the same VM.

The function throws an exception if the task's operating system and CPU requirements do not match those of the virtual machine. It also throws an exception if the VM is currently on a machine that is in sleep mode, or if the VM is currently migrating to a new machine. That is, when the VM is migrating and until the system informs the scheduler that the migration is complete, no new task can be added. The parameter validation routines require that the VM's and task's identifier be valid, otherwise an exception is generated.

extern VMInfo_t   VM_GetInfo(VMId_t vm_id);

This function returns information about the VM. This can be called any time and returns the current status of the VM. The following shows the definition of the information returned by the function:

```
typedef struct {
  vector<TaskId_t> active_tasks;
  CPUType_t cpu;
  MachineId_t machine_id;
  VMId_t vm_id;
  VMType_t vm_type;
} VMInfo_t;
```

As seen, the information returned includes the CPU type required, the type of the VM (mainly the operating system), the machine on which the VM is attached, and a list of tasks that are currently running in the VM. This function generates an exception if the VM's identifier is not valid.

extern void   VM_Migrate(VMId_t vm_id, MachineId_t machine_id);

This function enables the scheduler to move a virtual machine (along with its tasks) from its current machine to the machine specified by the machine identifier (machine_id). Migration has a high overhead and should be used carefully. The new machine should be one of the same family as the current machine (same CPU type). An exception is generated otherwise. An exception is also generated if the destination machine is in sleep mode (not state S0) or if the VM is already migrating, or if either the VM's or machine's identifier is not valid.

extern void VM_RemoveTask(VMId_t vm_id, TaskId_t task_id);

This function removes a task from a VM. This is useful in case if the VM has too many tasks and some load balancing is desired. Careful allocation of tasks to VM's may make calling this function unnecessary. Exceptions are generated if either the VM's or task's identifiers are not valid, or if the task is not in the VM.

extern void VM_Shutdown(VMId_t vm_id);

This function is called to shut down the VM. This function frees up the resources that the VM has on the machine to which it is attached. The VM should not have any tasks still running in it. An exception will be generated otherwise. An exception is also generated if the VM identifier is not valid.

Upcalls:

Upcalls are the mechanism by which the simulator call the scheduler, either to communicate a result back or to alert the scheduler to new tasks (e.g., a new task).
extern void MigrationDone(Time_t time, VMId_t vm_id);
The simulator calls to alert the scheduler that the VM has been migrated successfully in response to a previous request to migrate task. Now, the VM is established in the new machine and can accept new tasks if necessary.

Task Management Interface

Down Calls:

The majority of the task management down calls are informational, that is, they are used to inquire about the parameters and the status of a task. Only one call is an actuator to set the priority of the task.

extern unsigned GetNumTasks();

This function returns the total number of tasks in the simulation. This includes tasks that haven't yet arrived, and tasks that have already completed. This function does not generate exceptions.

extern TaskInfo_t GetTaskInfo(TaskId_t task_id);

This function returns a collection of information about the task. It generates an exception if the task's identifier is not valid. The information returned by this function is as below:

```
typedef struct {
  bool completed;                    // Set to true if the task has completed and is no longer active
  uint64_t total_instructions;       // Total instructions necessary to run the task
  uint64_t remaining_instructions;   // The instructions yet to execute to complete the task.
  Time_t arrival;                    // When did the task arrive?
  Time_t completion;                 // The time when the task completed. Invalid for active tasks
  Time_t target_completion;          // The target completion for the task to satisfy the SLA
  bool gpu_capable;                  // Can the task benefit from a GPU (significant boost)
  Priority_t priority;              // Task priority. One of three levels.
  CPUType_t required_cpu;           // Specifies the expected CPU
  unsigned required_memory;         // How much memory is required by the task.
  SLAType_t required_sla;           // The type of service level agreements.
  VMType_t required_vm;             // The type of VM that is required by the task
  TaskId_t task_id;                 // The task's unique identifier
} TaskInfo_t;
```

extern unsigned GetTaskMemory(TaskId_t task_id);

Returns the memory required by the task. An exception is generated if the task's identifier is invalid.

extern unsigned GetTaskPriority(TaskId_t task_id);

Returns the priority assigned to the task. An exception is generated if the task's identifier is invalid.

extern bool IsSLAViolated(TaskId_t task_id);

Returns an indication if the task has violated its SLA. An exception is generated if the task's identifier is invalid.

extern bool IsTaskCompleted(TaskId_t task_id);

Returns an indication if the task has completed. An exception is generated if the task's identifier is invalid.

extern bool IsTaskGPUCapable(TaskId_t task_id);

Returns an indication if the task is capable to exploit a GPU. Assigning a GPU-capable task on a VM that is attached to a machine that supports GPUs will result in a significant performance boost. An exception is generated if the task's identifier is invalid.

extern CPUType_t RequiredCPUType(TaskId_t task_id);

Returns an indication of the CPU required by the task. An exception is generated if the task's identifier is invalid.

extern SLAType_t RequiredSLA(TaskId_t task_id);

Returns the SLA required for this task. Four levels of SLA are defined for the tasks as follows:

```
typedef enum {
  SLA0,          // SLA requires 95% of tasks to finish within expected time
  SLA1,          // SLA requires 90% of tasks to finish within expected time
  SLA2,          // SLA requires 80% of tasks to finish within expected time
  SLA3           // Task to finish on a best effort basis
} SLAType_t;
#define NUM_SLAS 4
```

Dealing with SLAs require special care. A run is successful if the percentage of tasks that miss their deadlines is higher than the expected limit (e.g., 90% for SLA1). But deadlines also factor in with the SLAs. Tasks with SLA0 will generally have tighter deadlines than SLA1, which in turn has tighter deadlines than those with SLA2. Tasks with SLA3 are expected to complete on best effort basis. This is a good opportunity to save energy if managed properly.

An exception is generated if the task's identifier is invalid.

extern VMType_t RequiredVMType(TaskId_t task_id);

Returns the VM required by the task. An exception is generated if the task's identifier is invalid.

extern void SetTaskPriority(TaskId_t task_id, Priority_t priority);

This is the only actuating call in the task management interface. It changes the priority of the task specified by task_id to a new priority. This function can be called at any time. An exception is generated if the task's identifier is invalid.

Up Calls:

extern void HandleNewTask(Time_t time, TaskId_t task_id);

Called every time a new task arrives to the system. The scheduler should decide, according to the policy being implemented, which VM should host the task and what priority should be assigned to it. The time parameter gives the current simulation time (in microseconds). The task_id parameter uniquely identifies the task.

extern void HandleTaskCompletion(Time_t time, TaskId_t task_id);

Called whenever a task completes its run. The scheduler should adjust its data structure, and according to the policy being implemented, what action should be taken to balance the workload and reduce energy (e.g., migration, task reassignments, etc.).

extern void SLAWarning(Time_t time, TaskId_t task_id);

Called to alert the scheduler of an SLA violation concening the task specified by task_id. The time parameter indicates the simulation time. Reaction could include readjusting the current task distribution, activating more machines, etc., depending on the policy being implemented.

Energy Management Interface

Down Calls

extern uint64_t Machine_GetEnergy(MachineId_t machine_id);

This function returns at the time of the call how much total energy has been consumed by the machine specified by machine_id (in raw form). This may be useful for monitoring. An exception is generated if machine_id is invalid.

extern double Machine_GetClusterEnergy();

This function returns at the time of the call how much total energy has been consumed by the entire cluster, measured in KW-hr. This may be useful for monitoring. This function does not generate any exceptions.

extern void Machine_SetCorePerformance(MachineId_t machine_id, **unsigned** core_id, CPUPerformance_t p_state); // This is oriented toward dynamic energy

This call helps the scheduler performs some dynamic energy management (typically not done in practice). As a result of this call, all CPUs on the machine specified by machine_id will have their speed and power set to the specific P-state that is given by p_state. Note that because the scheduler has no visibility to the internal scheduling of tasks on the machine, it is no advisable to limit this call to just one or a subset of cores on the machine. This may lead to unsatisfactory results if a task with higher priority is scheduled on a machine with a high P-state. Therefore, all CPUs are set to p_state. The core_id parameter is ignored. An exception is generated if machine_id is invalid.

extern void Machine_SetState(MachineId_t machine_id, MachineState_t s_state);

This function sets machine S-state to the value specified by s_state. This is the most effective function to control energy consumption. The definition of the S states is as below:

```
typedef enum {
   C0,      // CPU is at state C0, in this case the power consumption is defined by the P-
states
   C1,      // CPU is at state C1 (halted but ready)
   C2,      // CPU is clocked gated off
   C4       // CPU is powered gated off, note: C3 is not supported
} CPUState_t;
#define C_STATES 4  // For C0, the power consumption is defined by the P-states
typedef enum {
   S0,      // Machine is up. CPU's are at state C0 if running a task or C1
   S0i1,     // Machine is up. CPU's are all in C1 state. Instantenous response.
   S1,      // Machine is up. CPU's are in C2 state. Some delay in response time.
   S2,      // S1 + CPUs are in C4 state. Delay in response time.
   S3,      // S2 + DRAM in self-refresh. Serious delay in response time.
   S4,      // S3 + DRAM is powered down. Large delay in response time.
   S5       // Machine is powered down.
} MachineState_t;
#define S_STATES 7
```

Upcalls

extern void StateChangeComplete(Time_t time, MachineId_t machine_id);
Called in response to an earlier request to change the state of a machine (Machine_SetState()). It takes time for the machine to transition between different states. Keep in mind that it is faster to shut down than to bring up, and that bringing a machine from the S5 state can take minutes (long).

Machine Interface

Down Calls

extern CPUType_t Machine_GetCPUType(MachineId_t machine_id);

This function returns the CPU type of the machine. An exception is generated if machine_id is invalid.

extern MachineInfo_t Machine_GetInfo(MachineId_t machine_id);

This function returns a lot of information about the components and status of the machine identified by machine_id. The data structure is as follows:

```
  unsigned num_cpus;                // Number of CPU's on the machine
  CPUType_t cpu;                // CPU type deployed in the machine
  unsigned memory_size;             // Size of memory
  unsigned memory_used;              // The memory currently in use
  unsigned active_tasks;            // Number of tasks that are assigned to this machine
  unsigned active_vms;              // Number of virtual machines attached to this machine
  bool gpus;                   // True if the processors are equipped with a GPU
  uint64_t energy_consumed;          // How much energy has been consumed so far
  vector<unsigned> performance;        // The MIPS ratings for the CPUs at different p-state
  vector<unsigned> c_states;          // Power consumption under different C states
  vector<unsigned> p_states;          // Power consumption for cores at different P states.
Valid only when C-state is C0.
  vector<unsigned> s_states;          // Machine power consumption under different S
states
  MachineState_t s_state;           // The current S state of the machine
  CPUPerformance_t p_state;           // The current P state of the CPUs (all CPUs are set to
the same P state to simplify scheduling
  MachineId_t machine_id;            // The identifier of the machine
} MachineInfo_t;
```

extern unsigned Machine_GetTotal();

Returns the total number of machines in the simulation. No exception is generated.

Upcalls.

The machine interface does not include any upcalls.

Exception Handling and Error Reporting

Exception handling is used to alert the programmer to errors. The default action for exception handling is to write a message and stop the simulation. If you wish to change the default handling, you can capture the following exception signature in a try/catch structure as usual:

```
extern void        ThrowException(string err_msg);
extern void        ThrowException(string err_msg, string further_input);
extern void        ThrowException(string err_msg, unsigned further_input);
```

The exception handler has the following syntax:
throw(runtime_error(err_msg + further_input))
catch(**const** runtime_error & error)

In addition, the simulator can run with the -v [0-4] flag. This will turn on the audit trail. It is quite verbose and will generate a huge volume at level 4 that describes the innerworkings of the simulator (and will significantly slow down the simulation). Levels 0 to 3 are mostly reserved for you. You can use the following function if you would like to add a message to the audit trail. The audit trail will be very useful for debugging, especially if it needs to involve the instructor.

extern void SimOutput(string msg, **unsigned** verbose_level);

Simulation Concepts

The following are general concepts and caveats that do not fit elsewhere.

Scheduler initialization

After reading input and generating the parameters of the simulation, the simulator calls the function InitScheduler() to enable you to do the necessary initialization of your data structures. After this function returns, the simulation starts.

Scheduler period check

After every timer has been served, the upcall SchedulerCheck() is called to enable the scheduler to do the necessary continuous monitoring as being implemented by the cloud management policy.

Simulation completion

When the simulator completes its function, the last function it calls is the upcall SimulationComplete(). In this function, you should report the results of the simulation (energy consumed, total running time, and SLA compliance results).

Memory management:

Virtual machines and tasks consume memory. You should pay attention to the current memory used in the machine. Overcommitting tasks to a machine that does not have much free memory left will drastically reduce performance. The upcall v v**oid** MemoryWarning(Time_t time, MachineId_t machine_id) alerts the scheduler that the machine identified by machine_id is overcommitted.

Dynamic energy management:

You can adjust the speeds of the processors on a given machine. However, you cannot (and should not) try to do this on a single core. The cluster-level scheduler typically does not have access to the inner workings of the machine to exercise this level of control in real life, and so in the simulation.

Processor static states

The processor static states (C1 and above) are tied to the state of the machine and are adjusted according to the S-state that you specify. It is not possible for you to have direct control of the C-states.

Machine in sleep mode

While a machine is in sleep mode, it cannot accept requests for attaching VM's. Additionally, it is a very bad idea to put a machine that has tasks active in it to sleep.

Simulation report

At the end of the simulation run, the skeleton code provided with the project reports on the SLA compliance achieved, and the total energy that has been consumed.

Units of measurements

To facilitate the innerworkings of the simulator, the following units are used:

- Time in microseconds
- Energy in KW-hr
- Performance in MIPS (million instructions per second)

Simulation Input

The simulator receives its input from a configuration file that contains a description of the classes of machines and the classes of workloads. The file consists of a few stanzas, each describing either a class of machines or a class of workloads. Comments can be inserted by preceding it with a #. A sample input file is shown below:

```
machine class:
{
# comment
    Number of machines: 16
    CPU type: X86
    Number of cores: 8
    Memory: 16384
    S-States: [120, 100, 100, 80, 40, 10, 0]
    P-States: [12, 8, 6, 4]
    C-States: [12, 3, 1, 0]
    MIPS: [1000, 800, 600, 400]
    GPUs: yes
}
task class:
{
    Start time: 60000
    End time: 800000
    Inter arrival: 6000
    Expected runtime: 2000000
    Memory: 8
    VM type: LINUX
    GPU enabled: no
    SLA type: SLA0
    CPU type: X86
    Task type: WEB
    Seed: 520230
}
```

Syntax of the input file and Description

- Each stanza must start with either "`machine class`" or "`task class`".
- `machine class`: Here are the possible fields and their interpretations-
    - Number of machines: The number of machines in this class that are installed in the simulated cluster.
    - CPU type: The type of the CPU. There are four types supported, including `ARM, POWER, RISCV, and X86.`
    - Number of cores: The number of cores per machine in this class. Every core can execute a task independently from the other cores. However, all cores share the memory of the machine they are installed in.

- Memory: The amount of memory in the machine in MB. All cores on this machine share this memory. If the memory requirements of the tasks that are currently scheduled on the machine exceed the amount of available memory, all the cores will experience significant slowdown.
- S-States: This array defines the power consumption of the machine under different S states. It is given in Watt. The figure does not include the power consumed by the cores. In the example above, you can expect that the machine (excluding the cores) consumes 120W at the state `S0`, and 10W at the state `S4`. These are the S states of the machine. They are defined as follows:
  - `S0:` Machine is running normally. All CPUs are in state `C0` if running a task or `C1` if idle.
  - `S0i1:` Machine is up. CPUs are in state `C1`. Machine can respond instantaneously to request.
  - `S1:` Machine is up, CPUs are in state `C2`. Some delay in response should be expected to move to either state `S0` or `S0i1`.
  - `S2:` Same as `S1`, except that the CPUs are in state `C4`. More delay in response should be expected to move to either state `S0` or `S0i1`.
  - `S3:` Same as `S2`, except that the DRAM is in self-refresh. Additional delay in response should be expected to move to either state `S0` or `S0i1`.
  - `S4:` Same as `S3`, except that the DRAM is powered down. Significant delay in response should be expected to move to either state `S0` or `S0i1`.
  - `S5:` Machine is powered down. Must be rebooted before being ready to run.
- P-States: This array defines the power consumption of the cores at different P-States. The array values are in Watt. In the example above, a core consumes 12W at `P0`, and 4W at `P3`. These are the P-states of the cores. Cores can be in different P states. Initially, all cores are initialized at `P0`.
  - `P0`: CPU at normal frequency.
  - `P1`: CPU at 3/4 frequency, 0.8 voltage.
  - `P2`: CPU at 1/2 frequency, 0.7 voltage.
  - `P3`: CPU at 1/4 frequency, 0.6 voltage.
- C-States: This array defines the power consumption of the cores at different C-states. The array values are in Watt. In the example above, a core in state `C1` consumes 3W, while it consumes no power at `C4`. Please note that the power consumption at `C0` is defined by the relevant number in the P-state array. The number included in the C-state array for `C0` corresponds to state `P0`. Cores can be in different C states. Initially, all cores are initialized at `C0`.
  - `C0`: CPU is running. In this case the power consumption is defined by the P-states

        - `C1`: CPU is halted but ready.
        - `C2`: CPU clock is off. Some delay should be expected in coming back.
        - `C3`: Unimplemented.
        - `C4`: CPU is powered down. Substantial delay should be expected in coming back.
    - MIPS: The MIPS is an array of numbers that define the performance of the core at each P-state. It is measured by the number of instructions per second and is given in MIPS (million instructions per second). It starts with the performance corresponding to `P0`, then `P1`, etc. In the example above, the CPU executes at a rate of 1000MIPS at state `P0`, and 400MIPS at state `P3`.
    - GPUs: This is a flag that indicates if the cores are enhanced with an accelerator. It can take the values of "yes" and "no".
- Task Class: The task class specifies parameters for the task class. The simulator reads these parameters and then it generates the actual tasks using a statistical distribution (assumes task inter-arrival time to be according to an exponential distribution and uses a uniform distribution for the rest).
    - Start time: This is the earliest time for a task of this class to arrive at the cluster. The time is measured in microseconds.
    - End time: Marks the last time for a task of this class to arrive at the cluster. After this time, no more tasks of this type show up. Measured in microseconds.
    - Inter arrival: The average interarrival time between consecutive tasks. The simulator uses this average to generate the task with an inter arrival time subject to a statistical exponential distribution. Measured in microseconds.
    - Expected runtime: This is the expected run time (assuming a processor of 1000 MIPS). It is measured in microseconds.
    - Memory: The memory required per task, in MB.
    - VM type: The type of virtual machine that is needed for this task. The following virtual machine types are implemented by the simulator:
        - `LINUX`: This VM is valid for all CPU types.
        - `LINUX_RT`: This VM is valid for all CPU types.
        - `WIN`: This VM is valid only for CPU types `ARM` and `X86`.
        - `AIX`: This VM is valid only for CPU type `POWER`.
    - GPU enabled: A flag that takes the values "yes" or "no". If enabled, then when placed on a core that has a GPU one should expect a significant performance boost.
    - SLA type: Service level agreement level. The following SLAs are defined in the simulation:
        - `SLA0`: This SLA requires 95% of all tasks to finish within 1.2 the expected running time, measured from the arrival time of the task.
        - `SLA1`: This SLA requires 90% of all tasks to finish within 1.5 the expected running time, measured from the arrival of the task.

    - `SLA2`: This SLA requires 80% of all tasks to finish within 2.0 the expected running time, measured from the arrival of the task.
    - `SLA3`: This SLA requires all tasks to finish on a best effort basis.
  - CPU type: This field identifies the required CPU type for the task. The task must be run on a VM that runs on the expected CPU type. The CPU type values are the same as the ones specified for the core types in the machine class description above.
  - Task type: The type of the task. This is mainly a way to define general parameters during the task generation. It is ignored by the simulator.
    - `AI`: Task is compute-intensive and can benefit from GPU.
    - `CRYPTO`: Task is compute-intensive, short, but repetitive. GPU-enabled.
    - `HPC`: Task is compute-intensive, very long, and can benefit from GPU.
    - `STREAM`: Streaming workload. Compute-intensive, short bursts.
    - `WEB`: Short requests.
  - Seed: A value to initialize the random number generator. It ensures repeatable runs for the sake of debugging.

Simulation Run

The simulator reads the configuration and workload files, then starts generating requests as specified in the workload files. As requests arrive, the scheduler will be invoked to decides where to place them (on which server). As requests complete, the scheduler deallocates them and may induce activities that can aim at reducing the center's overall energy consumption or overall performance or both. Also, the scheduler is alerted when an SLA violation takes place.

In addition to the scheduler, the cloud software also has a hypervisor that supports virtual machine operation. A hypervisor runs on each server, and supports virtual machine creation, deallocation and migration. The hypervisor also schedules the various virtual machines that it hosts on the hardware, with round robin scheduling being the algorithm of choice with a time slice of 60 milliseconds. When a virtual machine runs, it schedules the tasks it has with a round robin scheduling algorithm with a time slice of 20 milliseconds.

The servers

A configuration file will be given to you specifying the servers. They will be rated by effective MIPS and memory capacity. The effective MIPS is the actual number of instructions that run after factoring out the effects of pipeline conflicts and cache misses. As a simplifying assumption, we will use the same figure for all workloads. The server also has an energy consumption profile defined by a number of figures according to the ACPI standard.

The workloads

A workload file will be given to you specifying the workloads. Each workload is characterized by a trio of statistical distribution functions that define the rate of request arrivals, amount of MIPS needed to execute and the size of memory for each request. The workload also specifies the type of virtual machine on which it must run. The workload also will specify the SLA requirements specific to that workload.

Your task

You will need to implement the following components:

1. The simulation infrastructure. This will be an event driven simulation. You need to implement the various queues typical in a simulation.
2. Probability distributions. Please use public sources as appropriate.
3. The scheduler algorithms. You will need to implement the scheduler to minimize overall energy consumption and meet the SLA requirements, subject to the constrains imposed by the workload. For instance, a request must be placed in a virtual machine of the type that the corresponding workload specifies, and only if memory allocated to the virtual machine permits the placement of the request. This

is the main part of the project, and you will need to document the algorithm in writing as part of your submission of this project.

4. The various routines of managing the virtual machines, including the creation, decommissioning, and migrating the virtual machines as needed. You will also need to simulate the scheduling of tasks on the CPUs within the virtual machines, to estimate the queuing effects due to scheduling.
5. The output:
    a. Implementation of the simulation.
    b. Documented results of the simulation with various seeds, showing the overall energy consumption, and overall performance measured by the average rate of performance per task as a percentage of the maximum response time stipulated by the SLA.
    c. Documentation of the scheduling algorithm that you implemented.

Appendix: Simulation Structure

Strictly speaking, this section is not mandatory and is provided here to give you a glimpse at how the simulation works internally. The structure below shows the different components:

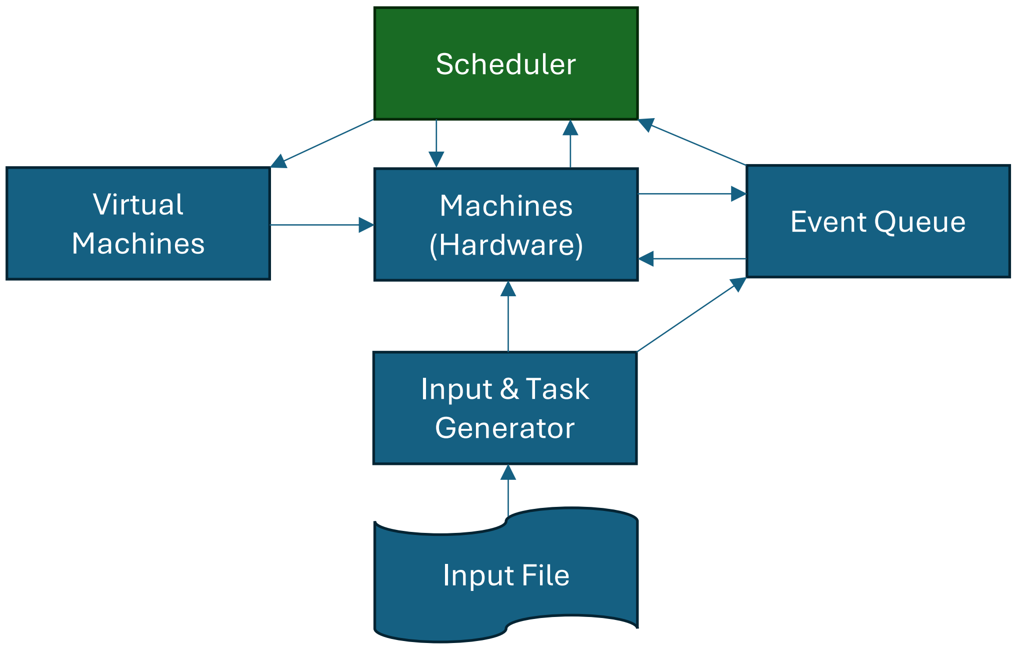

Figure 1. Simulator structure

The heart of the simulator is an event queue sorted by time. This implements a conventional event-driven simulation. Events on the queue are added by the other components, and after the initialization is complete, the simulator consists of a loop that takes the event in time order and calls the corresponding functions in the other modules to handle the events. There are four types of events:

- Task-related events, which signal the arrival and completion of tasks.
- Timer events, which are used to emulate the machine and enable the switching of tasks, metering of energy, etc.
- Migration events, which signal the completion of a scheduler-initiation migration of a virtual machine.
- Energy state events, which signal the completion of a state transition by a machine (S-state changes).

The simulator continues in the simulation loop until no more events are in the queue, at which point the simulation stops. It maintains a clock that advances with the occurrence of each event. For robustness, it is not possible to have a time warp (insertion of an event at a time less than the current simulation time), and it is not possible to strike out events. The current time of the simulation is maintained by the function "Now()", which is implemented by the event queue module. The implementation in C++ consists of a virtual event class from which the four classes above are derived. Strictly speaking, the energy state events are aligned with the timer events and as an implementation shortcut, they are handled by the timer events as well.

The “input and task generator” module is responsible for reading and parsing the input file. It creates the machines and tasks according to the parameters of the input file, and for each task, it generates the corresponding arrival event. After all tasks and machines are created, the module calls on the “Scheduler” modular to initialize itself, and then calls the simulation loop to start.

The “Machine” module contains the bulk of the simulated cluster. It has two main objects, namely “machine” and “CPU”. These emulate the processing and energy consumption of the machines that we described in the configuration file. The machine module monitors the tasks in the system and as long as the tasks have not finished, it will continue to generate timer events. The “machine” module is invoked by the timer events. The timer handler is where the tasks advance in their executions on the various simulated CPUs. The energy consumed is also updated in the timer handler.

The “Machine” module is also invoked by special calls from the virtual machine layer and the Scheduler. These calls implement special requests including:

- Attaching a virtual machine to a particular hardware machine.
- Changing the S-state of a simulated machine (to simulate energy saving).
- Migrating a virtual machine from and to a particular hardware machine.
- Various functions for monitoring to supply status information to the scheduler about machines and tasks.

Finally, the “virtual machine” module is the main interface to the scheduler module. It manages the task placement into machines. The virtual machine module is also the interface to perform migration.

The general implementation of these components uses an object-oriented approach with a flat object hierarchy (only CPU is embedded in the machine objects). The interfaces between the modules use regular function calls and not object invocations. This is done to simplify the implementation and avoids the problematic issues that result from circular inclusions of include files (a famous problem that C++ has).

GOOD LUCK!

START EARLY