A Unified Scheduling Model for Precise Computation Control

Michael Frisbie
frizz@ittc.ku.edu

17 May 2004

Outline

- Introduction
- Related Work
- Implementation
- Evaluation
- Conclusions
- Future Work

Introduction

Computer system designers wish to directly specify the computational semantics they desire with as much precision as possible.

Traditionally, computer systems have been divided into two separate use categories:

- General-Purpose
- Real-Time

A single set of system characteristics is no longer appropriate for every application.

A configurable platform is needed.

Properties of a Configurable System

Satisfaction of any given constraint may require precise control over any arbitrary set of system resources.

Threads are not the only computational elements managed by most operating systems. Computational elements present in the Linux operating system include:

- hardirqs
- softirqs
- tasklets
- processes

The Unified Scheduling Model

The scheduling policies controlling every computational element of the system are implemented as an explicit scheduling decision function (SDF).

The semantics of each SDF are a configurable system property.

Each SDF is associated with a set of computations, and this association is represented in a structure known as a group.

Linux interrupt processing has been restructured in order to place it under configurable scheduling control.

Related Work

CPU scheduling research has generally been restricted to the scheduling of threads, and almost all modern operating systems utilize a priority scheduling algorithm in their thread scheduler.

Advantages:

- Any given thread scheduling semantics can be represented by specifying a particular priority arrangement.
- Priority schedulers are efficient, both in terms of speed and space.
- Priority schedulers are easy to implement.

Disadvantages:

- The ease with which any given semantics can be translated into a priority arrangement can vary greatly.
- Priority Inversion
- Dynamic priority assignment introduces unnecessary overhead.

Co-Resident Operating Systems

This architecture allows for a real-time executive to coexist with a general purpose operating system.

The real-time kernel can meet its timing constraints, and the general-purpose kernel can operate with little modification.

One example, RTLinux, places a small, real-time operating system underneath the Linux kernel.

RTLinux intercepts interrupts and treats the Linux kernel as its idle thread.

Computations are assumed to either have strict timing constraints that cannot be violated or no timing constraints at all.

Other Work

POSIX soft real-time standard

UTIME increases the temporal resolution with which timer interrupts can be scheduled.

LynxOS and TimeSys encapsulate interrupt service routines as threads.

HLS structures execution domains in a similar hierarchical manner.

Implementation

The unified scheduling model consists of a set of components and permits the system designer to configure them into a hierarchic decision structure that is responsible for deciding which computation should be executed on a particular CPU.

Each computation is represented.

A scheduling decision function (SDF) is associated with a particular set of computations to form a group.

Groups can be linked to form a hierarchy.

Existing Scheduling and Execution Mechanisms

Some computations possess their own context while others can execute in any context.

Processes can be scheduled actively or lazily.

Hardirqs are scheduled “as fast as possible.”

Softirqs, tasklets, and bottom-halves are all deferrable functions.

Scheduling and interrupt latency are metrics of performance.

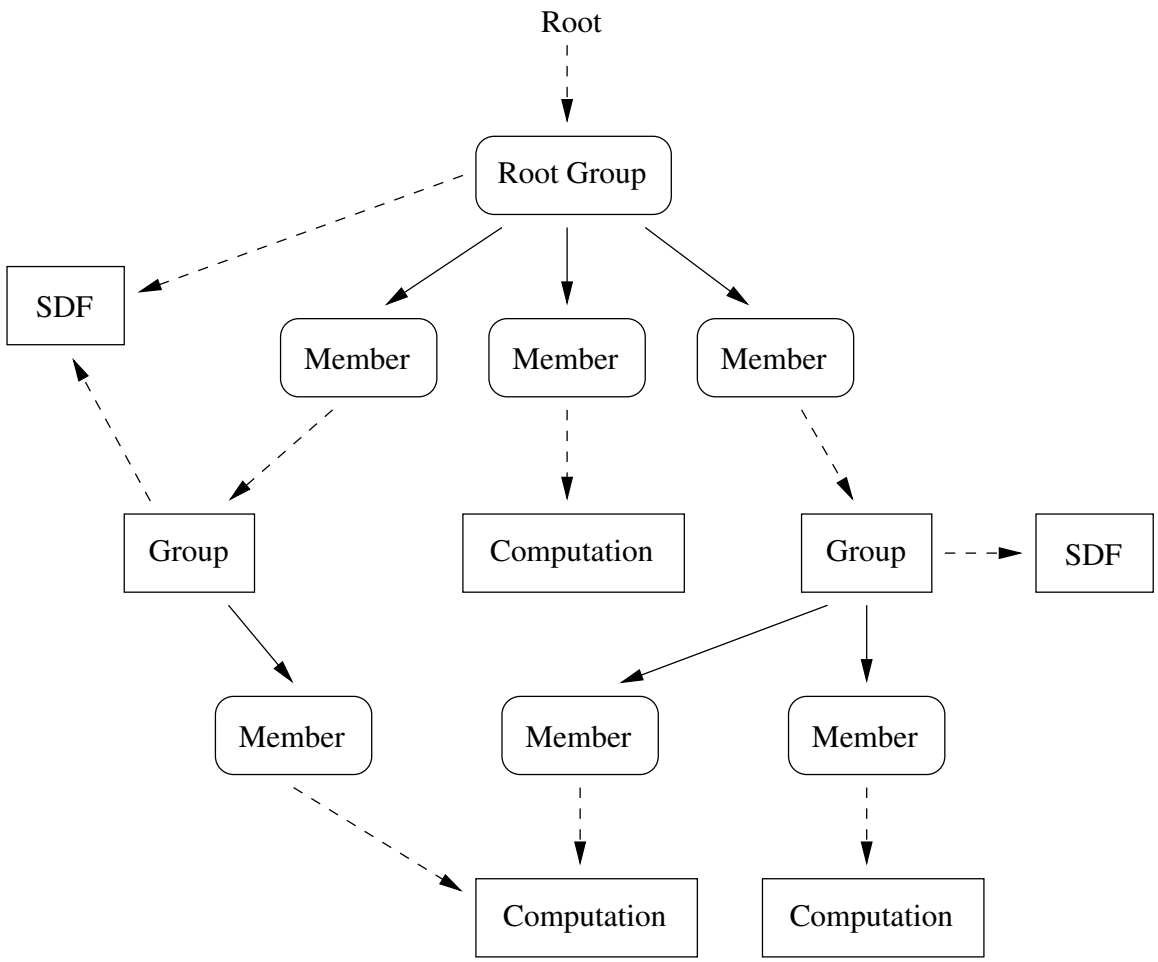
The Scheduling Hierarchy

The scheduling hierarchy is employed by first calling the decision function that is associated with the group at the root of the hierarchy.

Decision functions associated with other groups may be called.

The decision of the root group is returned to the calling function.

This function is responsible for properly executing the computation that is chosen by the scheduling hierarchy.


Root
Root Group
SDF
Member
Member
Member
Group
Computation
Group
SDF
Member
Member
Member
Computation
Computation

Scheduling Decision Functions

Each SDF has a unique name and a reference count.

The decision function chooses which member to execute, but does not actually execute it.

The parameter setting function is used to communicate data that is used by the decision function.

The member removal function is used to inform an SDF that one of its members has been removed.

Groups

Each group has a unique name and numeric identifier.

A pointer to an SDF determines the scheduling semantics that the group will impose on its members.

A pointer is provided which can be used to store arbitrary data.

A reference count is stored for each group.

A list of members is maintained for each group.

Members

Each member contains a pointer to the entity that the member represents, such as a process, hardirq, softirq, or group.

An integer used to indicate the type of this entity.

Members also contain a pointer to data that can be considered during the execution of a group's SDF.

Group Creation

This function accepts a group name and an SDF name.

It first verifies that a group with the given name does not already exist and that the corresponding SDF exists.

A new group is allocated and stored in the global group data structure.

The SDF reference count is incremented and the group's reference count is initialized to zero.

If no root group exists, then the new group becomes the root group.

Joining or Leaving a Group

These functions accept a group identifier, a member identifier, and a member type.

They confirm that the group identifier refers to a valid group and that the computation or group that is specified by the member identifier and member type pair exists.

Existing membership is tested.

A new member is allocated and appended to the group's member list or the existing member is removed from the group's member list and deallocated.

If the member is a process, then the group is appended to or removed from the process's group list. Otherwise, if the member is a group, then its reference count is incremented or decremented accordingly.

Group Parameter-Setting

An SDF often needs data to be supplied by the user in order to make its decisions.

This function accepts a group name, setting number, member identifier and type, and a value pointer along with the size of the data it references.

It first verifies that the specified group exists.

It attempts to locate the member that is denoted by the member identifier and member type pair.

The presence or absence of this member is noted and the parameter-setting function of the SDF that is associated with the group is executed.

Group Destruction

This function accepts only a group name.

It first verifies that a group with the given name exists and that its reference count is zero.

The member list must be empty.

The corresponding group is removed from the global group data structure and deallocated.

The SDF usage count is decremented.

If the group is the root group, then the root group is set to NULL.

Atomic Hierarchy Creation

It is often not adequate to sequentially create a hierarchy through a series of group joining.

The ability to specify a hierarchy atomically has been added.

This is accomplished by submitting an array of atomic hierarchy nodes.

Each of these nodes specifies the relationship between a parent group and a set of child groups.

This function only specifies a hierarchical relationship and does not actually create the groups.

Writing an SDF: The Decision Function

```
1  member decision_function(previous_task, this_cpu, group) {
2       choice = PASS;
3       for (each_member_in_group_member_list(group)) {
4            if (member_is_ready(member)) {
5                consider_member_data(member.data);
6                if (this_member_is_better(member, choice)) {
7                     if (member.entity_type == group) {
8                          member = run_decision_func(previous_task,
9                                                     this_cpu,
10                                                    member.entity);
11                    }
12                    if (member != PASS)
13                         choice = member;
14               }
15          }
16      }
17      return choice;
18 }
```

Writing an SDF: The Parameter-Setting Function

```
1  int set_param_function(group, setting, member, value, size) {
2      switch(setting) {
3          case SETTING_NUMBER_ONE:
4              if (member == NULL) {
5                  return error;
6              }
7              if (member.data == NULL) {
8                  allocate(member.data, size);
9              }
10             copy(member.data, value, size);
11             break;
12         case SETTING_NUMBER_TWO:
13             copy(group.data.variable_number_one, value, size);
14             break;
15         default:
16             return error;
17     }
18     return 0;
19 }
```

SDF Registration and Unregistration

The creation of an SDF is not complete until the SDF data structure is specified and accessible.

An SDF is useless unless it is associated with one or more groups within the scheduling hierarchy.

In order for groups to associate themselves with a particular SDF, the SDF must first register itself with a global data structure that provides a mapping between SDFs and their unique names.

SDF registration is done in the kernel module initialization routine.

If an SDF is no longer being used, then it can be removed from the global SDF storage structure.

Computational Elements

The unified nature of the scheduling model establishes the ability of the scheduling hierarchy to consider a wide range of computations.

Processes are unique among system computations in two ways:

- They possess their own context.
- Processes can be created and destroyed.

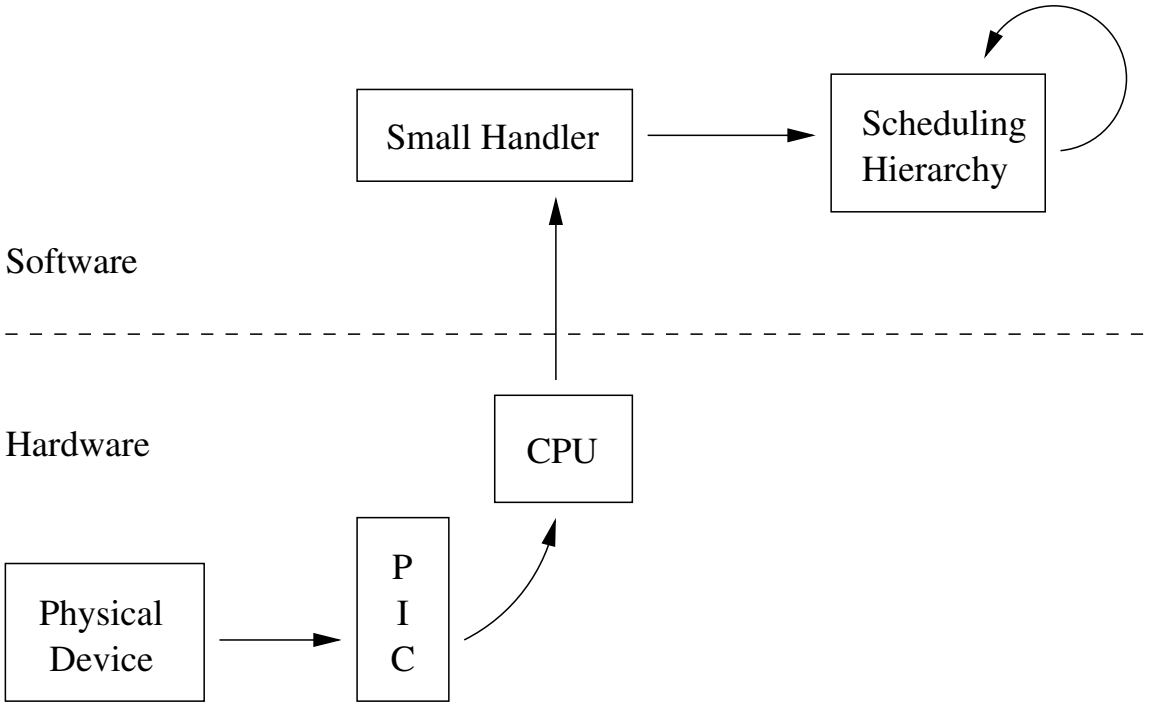
A placeholder member type is created in order to represent the standard Linux scheduler and all the processes that have not been explicitly placed under control of the scheduling hierarchy.

Hardirqs

Hardirqs have become a controllable computation by handling interrupt concurrency control and scheduling in software rather than hardware.

When a critical section attempts to disable interrupts at the hardware level:

- This fact is noted in a software data structure but not actually carried out on the hardware.
- When a subsequent interrupt occurs, it is marked in a separate data structure, and control is then transferred to the decision function.
- Based on the available data structures, the decision function can then decide whether to run the corresponding hardirq or postpone its execution until a later time.


Small Handler
Scheduling Hierarchy
Software
Hardware
CPU
P I C
Physical Device

Concurrency Control

Hardware Interrupt Flag	This flag should be disabled during access to any data that is used by the small irq handler.
Software Interrupt Flag	This flag must be disabled during access to any data that is used by a hardirq. This flag is not sufficient for protecting data that is accessed by hardirqs that ignore this flag.
Preemption Count	This counter variable is associated with each process. Whenever it is equal to zero, the current process may be preempted by another process. Asynchronous preemption cannot occur while an interrupt flag is disabled.
Spin Locks Read/Write Locks	Locks protect data that can be accessed concurrently by multiple CPUs. When a lock is acquired by a process, its preemption count is incremented in order to prevent the process from being preempted while holding a lock.
Bottom-Half Flag	This flag synchronizes access to bottom half data structures.
Interrupt-Specific Flags	A specific flag is designated for each hardirq and can be used as a replacement for the software interrupt flag.

Softirqs

The softirq scheduling algorithm is reliant upon the softirq execution mechanism.

The softirq decision function assumes that it is only called after the hardirq decision functions have expressed no opinion.

The softirq decision function first ensures that no other hardirqs or bottom-halves are executing. If this check fails, then RETURN is returned.

Otherwise, the highest priority pending softirq is determined by surveying the pending softirq bit field, and the corresponding softirq member is returned. If no softirqs are pending, then PASS is returned.

Adding New Computational Elements

The existing scheduling and execution code must be separated.

A data structure needs to be associated with the new computation type.

This structure must contain a unique identifier for the computation.

The internal functions of the unified scheduling model must then be modified to use this identifier in locating computations of the new type.

Also, a new computation type identifier must be designated for the new type, and the the internal methods must be modified to acknowledge it.

Code that calls the scheduling hierarchy must be modified to execute the member if it is chosen.

Adding New Computational Elements: Computation Types

Process	Computations of this type are executed using a context-switch.
Hardirq	Computations of this type are executed by calling the computation function directly.
Softirq	Computations of this type are executed by calling the computation function directly.
Group	Members that are associated with this computation type are not chosen by the scheduling hierarchy. Instead, decision functions can directly execute the decision function associated with a member of this type.
Nothing	This computation type signifies a placeholder for the Linux scheduler. It signifies that the scheduling hierarchy has chosen nothing but that another scheduler may make a choice.
Pass	This computation type can be returned by decision functions to assert that they have no opinion about what to choose.
Return	This computation type can be chosen by a decision function in order to signify that no choice should be made by the hierarchy and that the current thread should continue execution by returning from the calling function.

Evaluation

Implement both existing and desired programming models using this new framework.

- KURT-Linux
- Pipeline Model
- E-Machine

Scheduling overhead and interrupt accuracy have been measured.

Explicit Plan Programming Model

The KURT-Linux programming model allows processes to be explicitly assigned intervals of execution.

Interval data structure:

- member_id
- member_type
- begin
- end
- processor
- instances
- period

Explicit Plan Programming Model: Decision Function

Consider the first interval in the schedule.

If the member associated with the interval can be executed, then it is selected for execution and a timer to signal the interval's end time is created.

If the first interval on the list has expired, then its beginning and ending times are updated based on its periodicity and remaining lifetime. The first interval is then reconsidered.

If none of the previous checks have succeeded, then a timer for the interval is programmed and the decision function returns without having an opinion about what member should be chosen.

Explicit Plan Programming Model: Parameter-Setting and Member Removal Functions

The *SUBMIT_EXPLICIT_PLAN* setting is used to submit a schedule.

The *SUBMIT_EXPLICIT_PLAN_WAIT* setting is also used to submit a schedule and block calling process until the schedule is finished or a signal is delivered.

The *SUSPEND_INTERVAL* setting allows a computation to suspend its execution during any intervals in which it is currently executing.

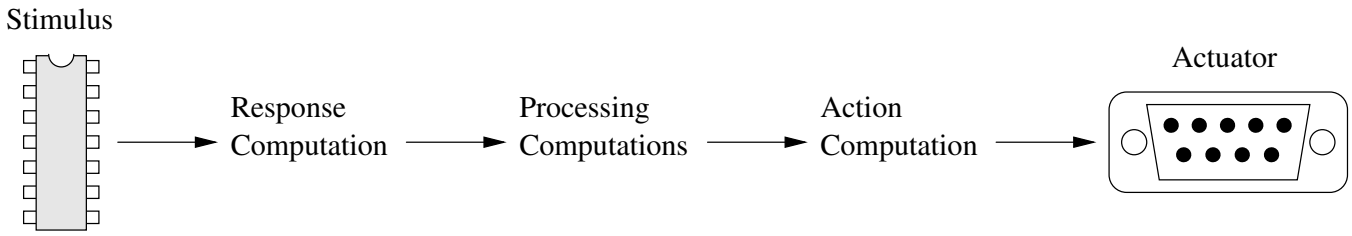
The member removal function removes any of the intervals remaining in the schedule that are associated with the leaving member.

Pipeline Model

Many real-time applications require multiple tasks to coordinate their execution.

A common model for this cooperation is the pipeline model.

Applications such as videoconferencing programs and industrial automation control can be represented as a pipeline of several tasks.



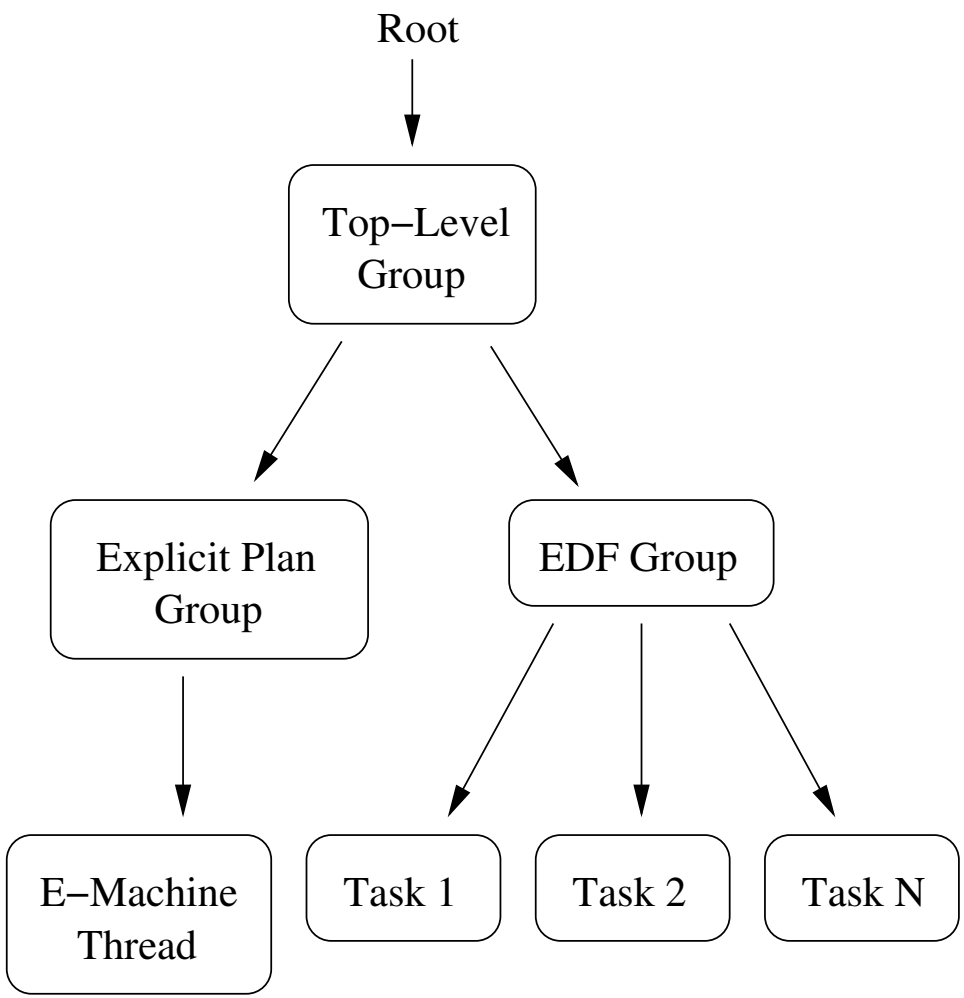
E-Machine

The E-machine is an embedded interpreter that is designed for real-time applications.

Computations are divided into to classes:

- Drivers
- Tasks

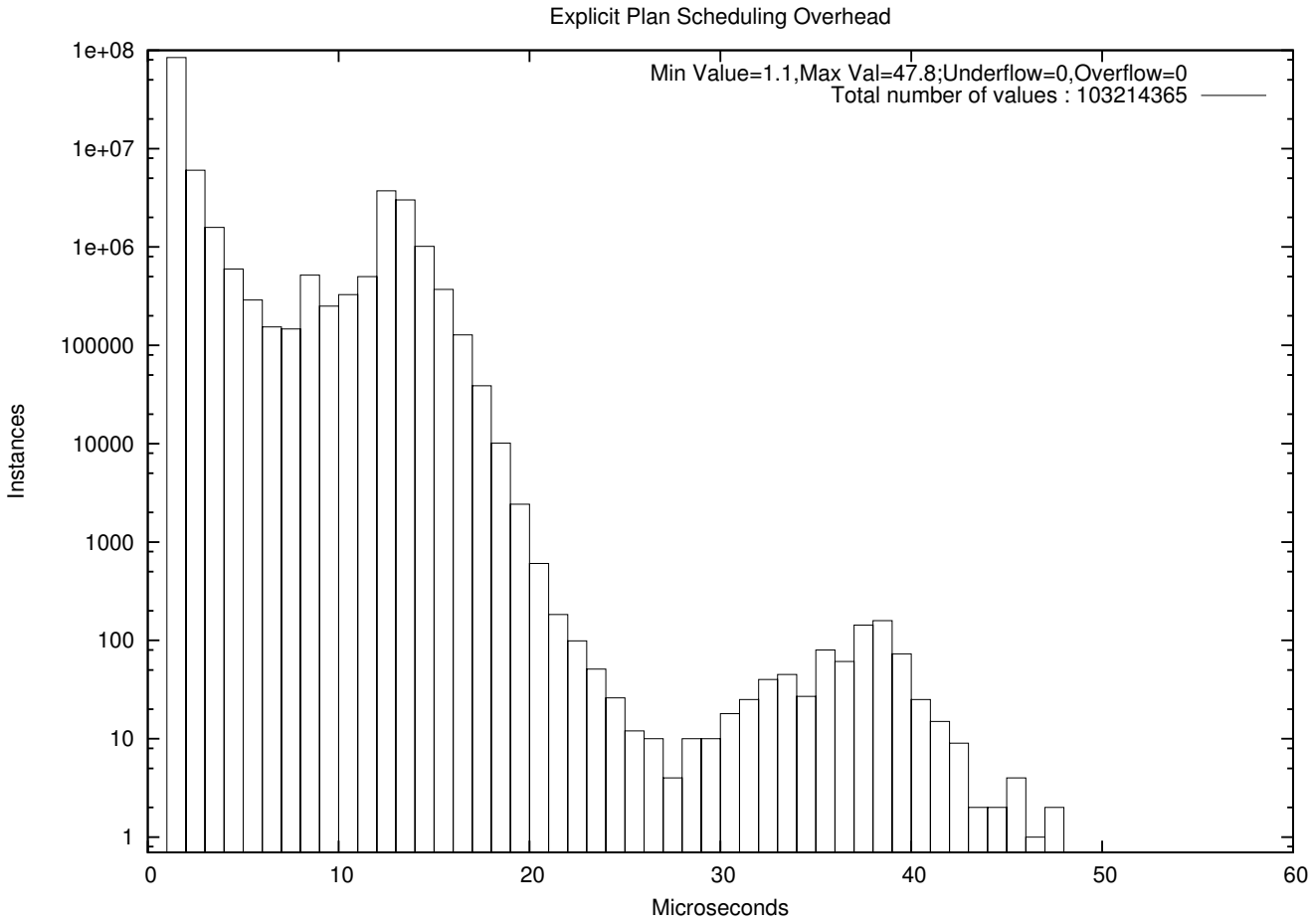
The E-machine consists of a single thread that is responsible for executing E-code. E-code can execute a driver, schedule a task, and schedule the execution of E-code in the future.

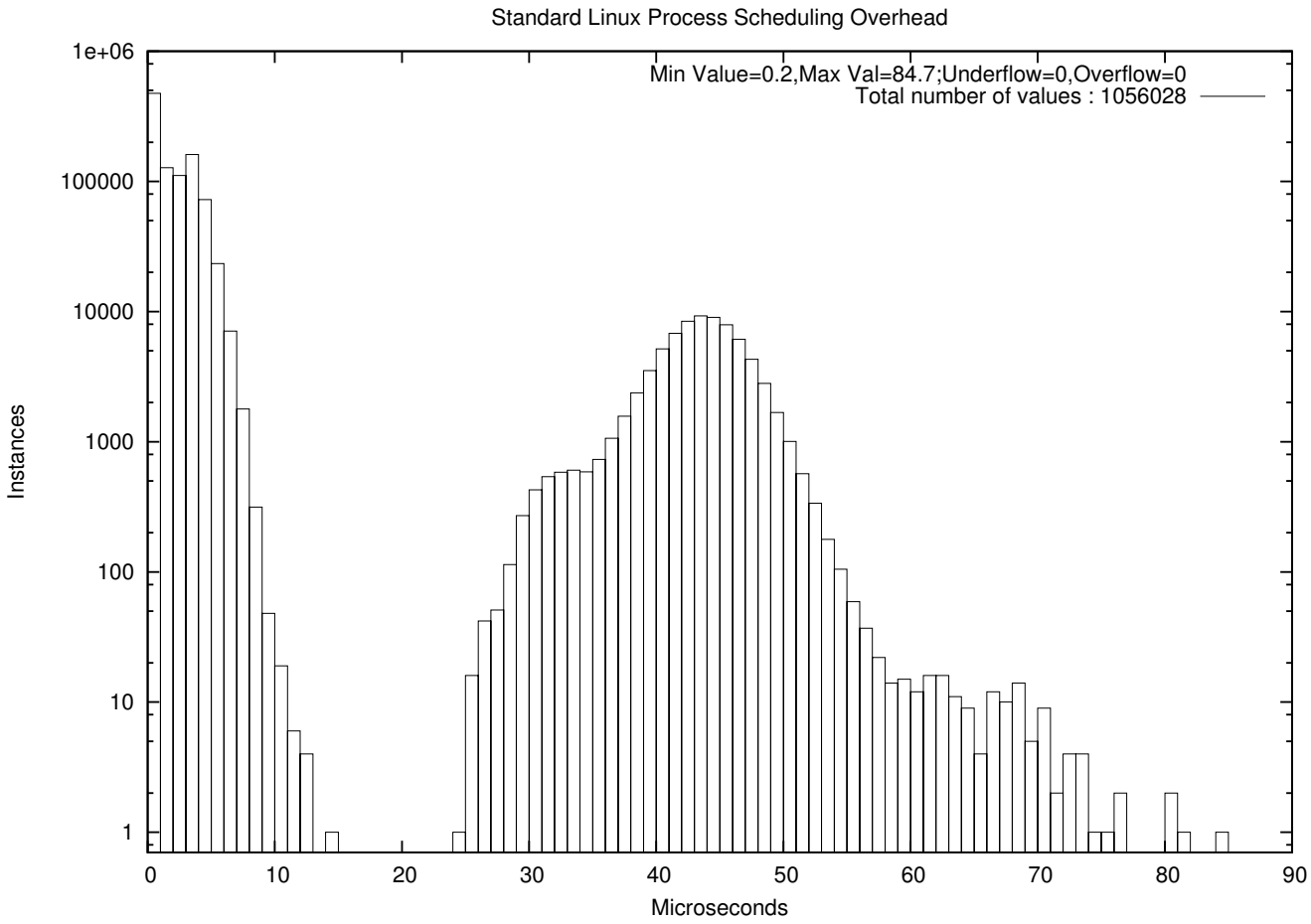

Root
Top−Level Group
Explicit Plan Group
EDF Group
E−Machine Thread
Task 1
Task 2
Task N

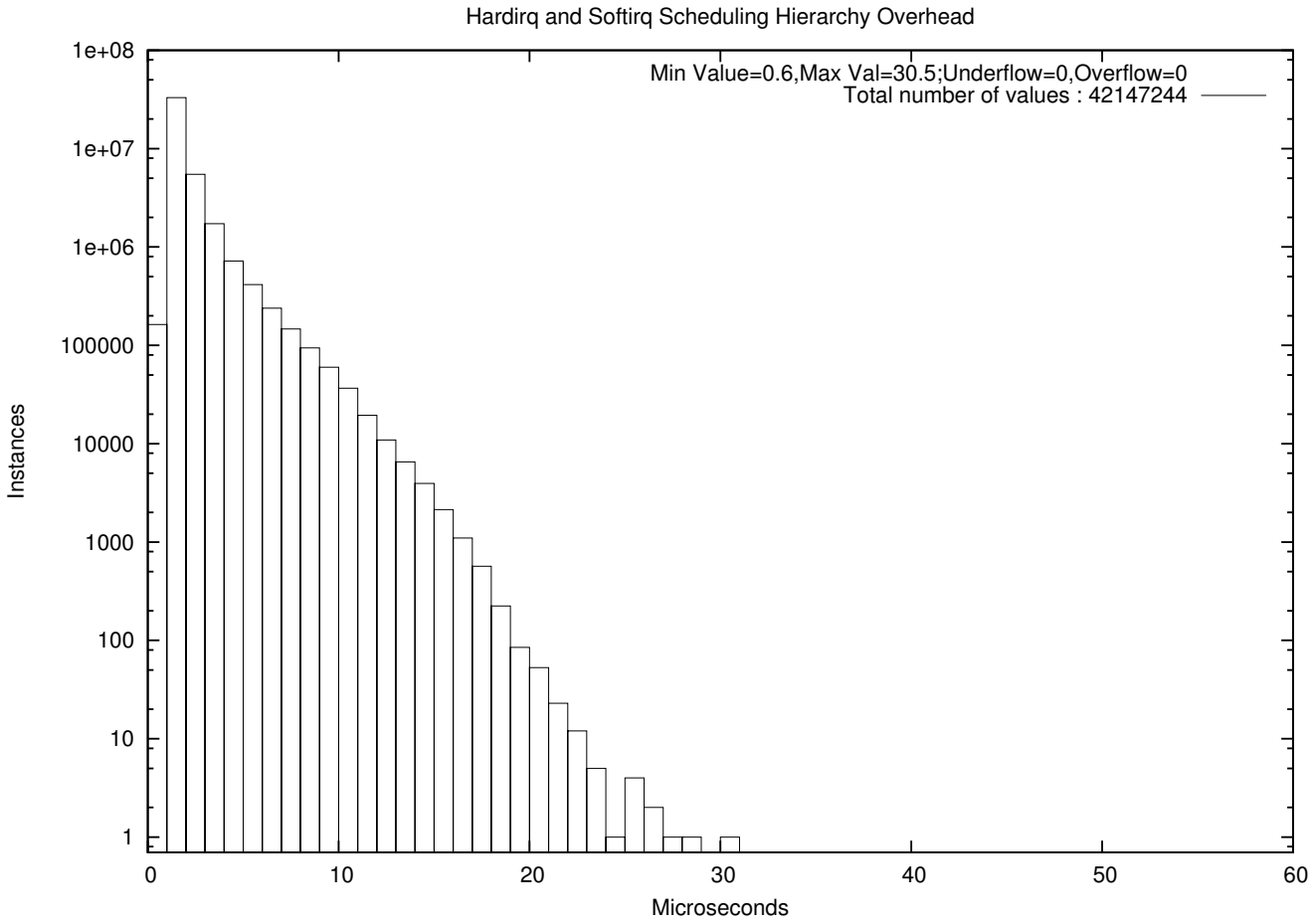
Scheduling Overhead

The overhead of executing the scheduling hierarchy depends on the structure of the hierarchy and the decision path that it takes.

These tests were performed over a period of 10,000 seconds on a computer system with a 200 MHz Pentium Pro processor. System load was generated by cyclically executing a two-process kernel compilation on the local hard disk through a secure shell connection.


Explicit Plan Scheduling Overhead
Min Value=1.1,Max Val=47.8;Underflow=0,Overflow=0
Total number of values : 103214365
Instances
Microseconds
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0
10
20
30
40
50
60


Standard Linux Process Scheduling Overhead
Min Value=0.2,Max Val=84.7;Underflow=0,Overflow=0
Total number of values : 1056028
Instances
Microseconds
1e+06
100000
10000
1000
100
10
1
0
10
20
30
40
50
60
70
80
90


Hardirq and Softirq Scheduling Hierarchy Overhead
Min Value=0.6,Max Val=30.5;Underflow=0,Overflow=0
Total number of values : 42147244
Instances
Microseconds
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0
10
20
30
40
50
60

Interrupt Accuracy

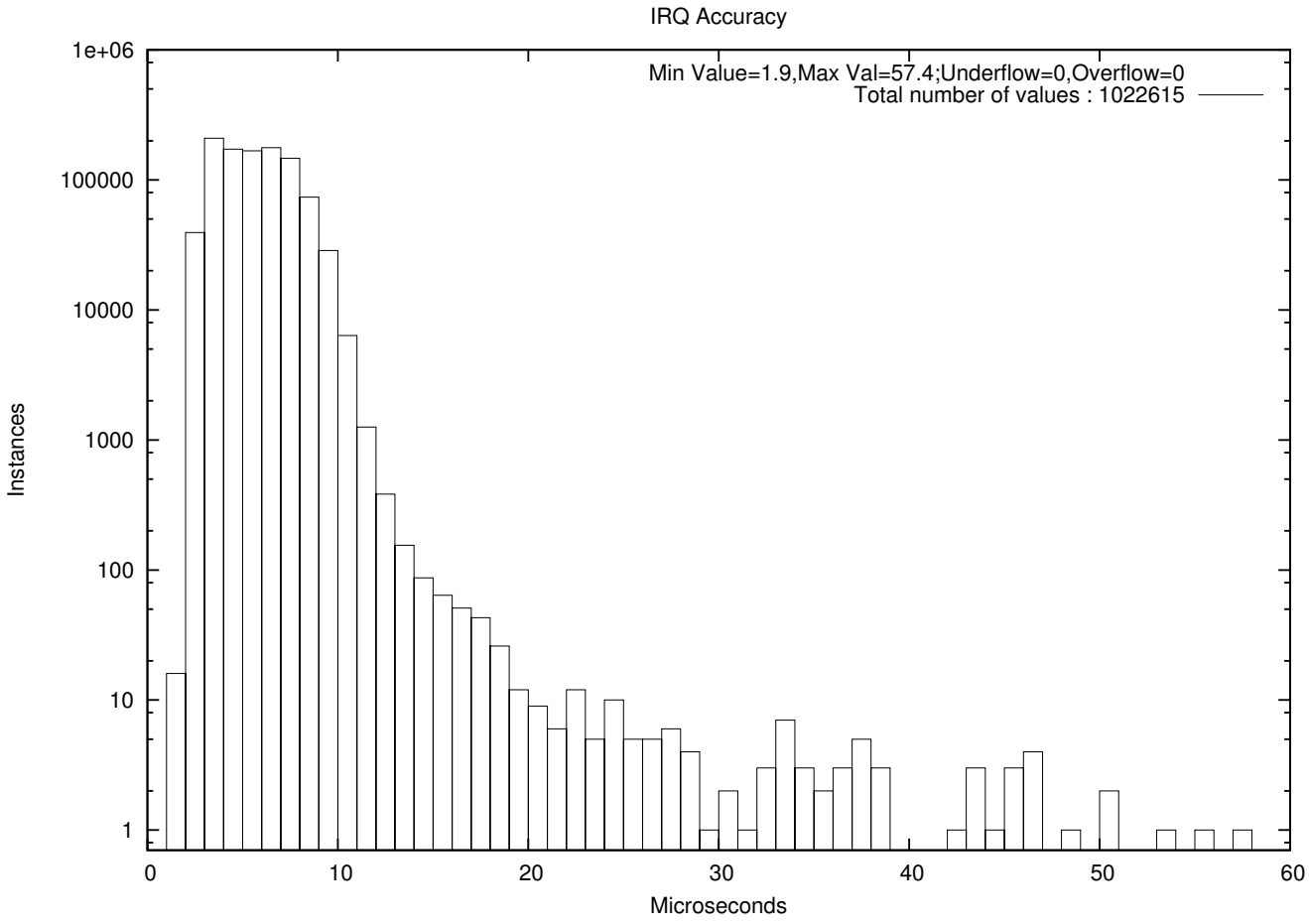
The modifications to interrupt concurrency control that have been made allow for interrupts to be delivered to the CPU more accurately.

This fact has been confirmed by measuring the accuracy with which a timer interrupt can be delivered to the CPU.

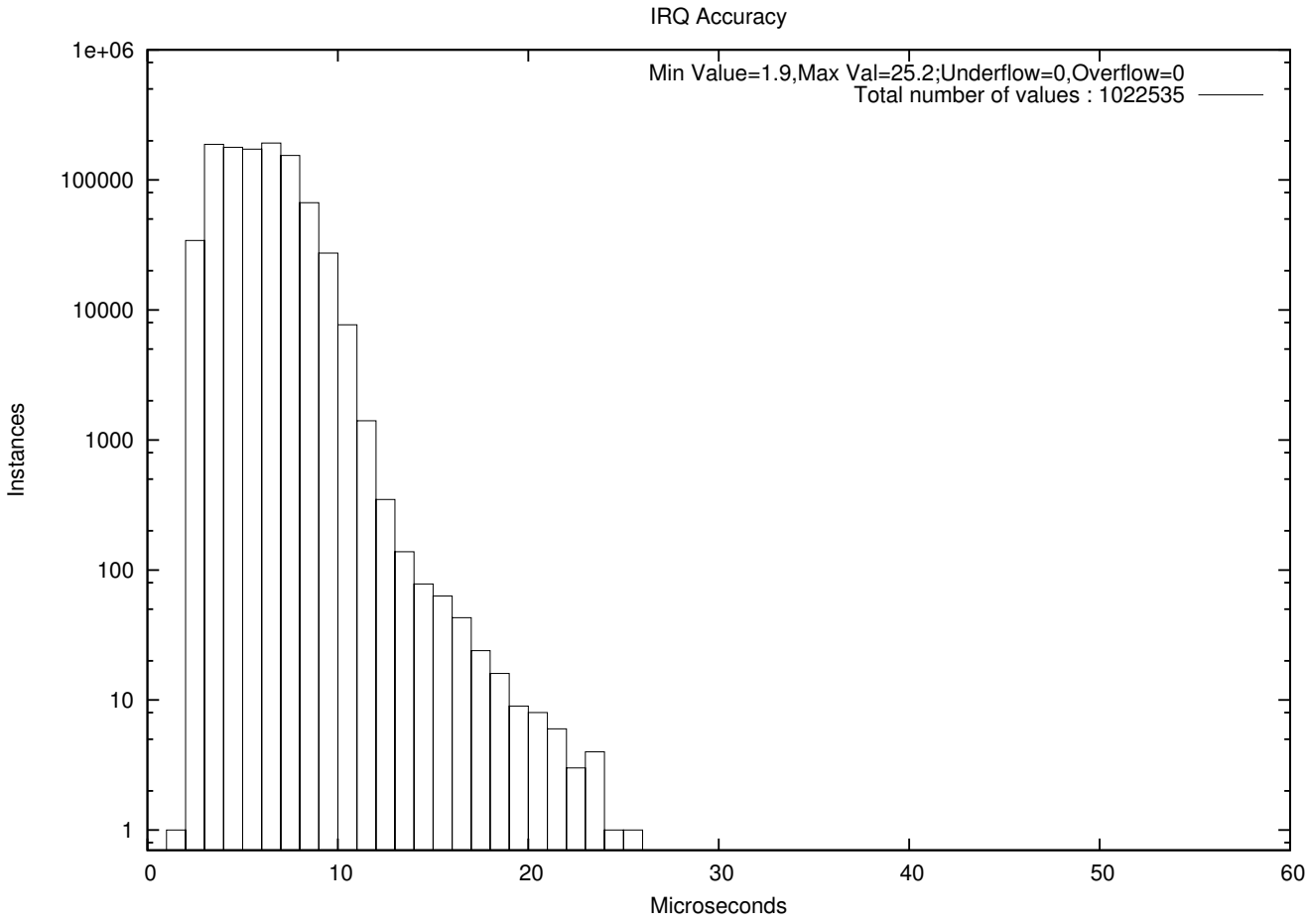
The difference between the scheduled time of a timer interrupt and its actual delivery has been measured.

These tests were performed under the same conditions as the scheduling overhead tests.

Interrupt Accuracy Without IRQ Modifications



Interrupt Accuracy With IRQ Modifications



Conclusions

The demand on computer systems to handle an increasingly wide range of applications with equally varied execution semantics creates a need for a fully configurable resource control mechanism.

Since control of system resources can be simplified to controlling all the computations that use them, a configurable computation scheduling system is needed.

The system must be able to unite all computations under a single scheduling mechanism in order to attain the precision that is desired by system designers.

Future Work

Proxy Execution

Computation Type Reduction

Control Path Consolidation

Questions?