Dhara: A Service Abstraction-Based OS Kernel Design Model

Dharanipragada Janakiram, Hemang Mehta , S J Balaji
Dept. of Computer Science and Engineering
IIT Madras
Chennai, India
e-mail: djram@iitm.ac.in, {hemang, sjbalaji}@cse.iitm.ac.in

***Abstract*—Traditional procedural operating system (OS) kernels sacrifice maintainability and understandability for optimum performance. Though object oriented (OO) kernels can address these problems upto a certain extent, they lack the layered approach of services and service compositions. We present a new kernel design model Dhara, that raises the level of abstraction from objects and procedures to services. The service model of Dhara is richer in abstractions than current web service model and paves the way for building a new distributed OS kernel. Dhara conceives an OS as being constructed by multiple stacks of services containing several layers of abstracted services. A key research challenge we envisage in building such model is automatic service compositions of kernel services which can provide desired QoS. A kernel built using Dhara can easily be customized using composed services to derive optimal performance for different applications such as databases. A prototype is developed using Linux kernel as a case study by applying the design concepts of Dhara. We show that overhead of implementation of Dhara is 5% to 15%, which is reasonable, considering the advantages of new design and increased capacity of the hardware in recent times.**

***Keywords*-Kernel Design; Service Abstractions**

I. Introduction

The contemporary procedural operating system kernel designs emphasize on performance which makes them difficult to maintain. The advent of new hardware architectures makes them increasingly complex and difficult to understand. Object oriented (OO) kernels have tried to address this issue by common extensible interfaces and different implementations. However, objects are not inherently organized in a layered approach which can be used to create compositions. Our previous work OO wrappers for Linux kernel[8], introduces class-like abstractions around subsystems. The OO wrappers are cleaner abstractions than procedures and attempt to provide loose coupling among kernel components.

Procedural, object oriented and wrapper based kernels traditionally employ a generic set of policies that cater to the needs of different applications. The applications may perform suboptimally under this assumption[20]. On the contrary, applications can perform better if they can recommend the suitable policies. For example, if the virtual memory is managed according to the recommendation of the applications[7], page faults can be reduced. Similarly, Earliest Deadline First (EDF) scheduling scheme[5], which is not a standard scheduling policy in Linux, can provide real-time guarantee for a given task regardless of the behavior of other processes.

The suboptimal performance of applications can be attributed to following facts: First, the present implementation of policies is transparent to the user applications. Secondly, even if the predicted resource access pattern can be passed on to the kernel, it may lack the suitable policy to meet the needs of the applications. An application specific policy can not only handle an access pattern more effectively, but also can provide user applications with finer control over the functionality using tweakable parameters such as queue size of pages in case of FIFO. Thus, multiple policies can significantly enhance performance of user applications.

Another issue with the traditional operating systems is that, although they provide same services to the applications (such as memory, process, timer, network, etc.), they have to be re-engineered for different categories of devices. An OS kernel that starts with bare minimum and keeps building by integrating more specific services based on the complexity of hardware and the needs of the applications can greatly ease up the system engineering process.

These issues can be addressed if kernel objects are raised as abstract services and are exposed to user applications using a layered and secured approach. We propose Dhara, a new service abstraction based kernel design model. Dhara presents richer abstractions than traditional web services which can be used to form automatic service compositions as per the need of the applications. A calculus for service compositions[15] is employed here to capture various types of compositions such as length or width wise chaining. Using the proposed service model the OS kernel design can be improved which can be applied to a distributed OS kernel as well.

The research contributions of this work are as follows:

- We present a model for kernel design, Dhara which encompasses a generic framework for raising kernel functionality to service abstractions (SA). It unifies the abstractions for specifying the kernel services under a common framework.
- We propose *Multilevel Service Stack* which gives a structured organization to the kernel services with repositories to locate them. It reduces the interdepen-

978-2-9541-8100-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICECCS.2012.16

dency among the subsystems, making them easy to maintain with changing hardware.
- We express interactions of kernel services using a new service abstraction design pattern and service compositions.
- We describe a working prototype of Dhara with Linux kernel as a case study. The prototype design features process scheduling, memory management, interrupt and timer management subsystems as services.

II. Motivation

The usage of service abstractions for redesign of OS kernel gives rise to many concerns. This section explains our motivations and tries to answer these questions.

A. Extending an OS Kernel with Service Abstractions

The benefits of multiple kernel policies for resource management are well known[5], [7]. However, in order to introduce a new policy, following issues have to be addressed:

1. Insertion of policy specific code and data
2. Functioning of multiple policies without interfering with each other
3. Sharing of a common resource by the policies
4. Ability of user applications to discover and use available services
5. Switching between similar services

The issues of high coupling in kernels like Linux are well known and studied extensively[16], [22], [18], [21]. Our efforts of adding new policies to Linux kernel also encountered similar obstacles. For example, the `runqueues` data structure is the most abstract data structure of scheduling subsystem in Linux kernel which stores all runnable processes. However, it is tightly coupled with policy specific structures of fair scheduling and real time scheduling policies, `fair_rq`, and `rt_rq`. Similarly, in page replacement, list of inactive pages is tightly coupled with other data structures, making it difficult to derive abstractions for new policy. Thus because of tight coupling, there are high chances that a programmer may miss changing some part of the kernel code that gets affected due to the new policy. This may lead to severe errors which could be hard to locate. In addition, for every new policy, new APIs might be required by the applications to use them. For example, APIs for using EDF scheduling included 3 new system calls[5]. This situation raises need of a generic framework to introduce new policies and a common set of abstractions to use them.

The design of any software system with service abstraction has qualities such as loose coupling, dynamic service publishing, discovery and usage, provision of QoS using service compositions, load balancing and fault tolerance using multiple implementations of the same service, etc. These features can help solve the issues raised above for extending a kernel, which is the primary motivation for the proposed model Dhara.

B. Comparison of Service Abstractions With Other Design Techniques

There are many design techniques which are in practice and can be used for kernel redesign. Object orientation (OO) can address the issue of tight coupling in procedural code of the kernels. In contrast to objects, services represent an architecture style and are more abstract than OO. The increased level of abstraction from objects to services allows them to be described, discovered and used in a more elegant manner. In Component Object Model (COM), every entity is considered as an object, identified by a global unique identification (GUID). All classes with GUIDs are listed in a registry, which is the case with Windows OS. Here the registry is a global namespace and corruption in registry may lead to severe side effects. In our design, restricted access to registry is granted using system calls. Furthermore, only specific kernel policies are published in the registry, limiting its size. Usage of Aspect Orientation (AO) has been explored in the field of OS[11]. It is indeed a good design technique as it allows policy specific and non-specific attributes to evolve independently. Nonetheless, kernel functionality can be more naturally modeled as independent services interacting with each other and forming compositions, rather than aspects. AO also lacks the framework of publish, find and bind. Hence, runtime discovery of entities and binding with them is easier with services than aspects.

III. Generic Framework of Dhara

This section describes the generic framework of Dhara that makes it possible to uniformly integrate different services of the subsystems with multiple implementations. We first explain how Dhara describes service abstractions in the context of kernel.

A. Service Abstractions in Context of Kernel

The conventional web services run in the service provider's execution environment upon invocation. Contrary to this, kernel services manage resources of the same system in which they are invoked. Moreover, the web services are invoked by peer web services. On the other hand, the kernel services exposed to the user application layer are only viewed and selected by the user applications but are invoked by kernel. Hence, Dhara defines kernel services differently from conventional web services. Services in Dhara are modeled as first-class entities that raise the level of abstraction of procedures or objects by introducing metadata. Thus kernel services can be referred at a higher level using the metadata and can be decoupled from their actual implementations.

The kernel is viewed as a large collection of such services by Dhara. The challenge is to identify loosely coupled services and to portray their complex interactions correctly.

However, in systems such as OS kernels, some level of interdependence among the modules is required. For example, a scheduler needs a timer to keep track of the runtimes of the processes or any executable unit requires scheduler to get CPU-time. In this case it becomes difficult to divide the system into clear-cut services. Dhara attempts to address this issue by categorizing services into component, composite and orchestrated services. The component services act as basic units of operation in a given subsystem which form a composition or participate in an orchestration. For example, enqueuing a task in runqueues, reading a disk block, generating a timer tick, etc. are component(independent) services. On the other hand, network communication is a composite service composed of components corresponding to each layer in TCP/IP stack since they all manipulate a common socket buffer. The orchestrated services can be used to depict interactions among services, where the output of one service is not used as input of another. The main service orchestrates the component services to complete its execution. An example of this is process execution service, which requires services like memory allocation, deallocation, scheduling, page replacement, timed execution, etc. Thus, the proposed service categorization can be used in the form of nesting of composite and orchestrated services to model complex service interactions.

The kernel services need to be modeled in detail to facilitate easy composition and orchestration. The structure of the service repository and metadata should provide simple interfaces for service discovery and usage. The services should be organized according to the levels of abstraction ranging from hardware to user for better understanding of the kernel. These aspects of Dhara are explained in the subsequent parts of this section.

B. Service Model

The service model of Dhara is inspired from OO based semantic model for the services[9] which models services by the notion of **service types** and **service instances**. Service types model service instantiation in terms of profile, description and state data of the service. They can be used to create operational specification of services called service instances. Service instances are created by instantiating individual components of service types. For example, metadata is an instance of description, capabilities are instances of profile and data members are instances of state data of the service type. In addition, a service instance provides a grounding which is an endpoint and serves as the address of the service instance.

Kernel services naturally fit in this service model. The abstract service types can be used to model policy hierarchy. Service instances created by instantiating kernel service types serve as the tangible services that can be published in a registry. For kernel service model, metadata is enough for modeling a service instance as they are local to a system and its detailed addressing is needless.

The structured service model enables composing services, based on the service type specifications and the interface requirements. Multiple independent variants of the same service can co-exist in the system allowing different service compositions. The service composition offers significant leverage compared to other design principles because of the diverse behavior composed services can exhibit.

C. Service Abstraction Pattern for OS Kernels

Dhara features a general design pattern for constructing the kernel service repository. The Service Abstraction Pattern (SAP) unifies kernel services of diverse interfaces. It provides abstractions for introducing new services and for easy service discovery. SAP builds on the service model using `ST_Desc` (Service Type Descriptor) and `SI_Desc` (Service Instance Descriptor) as entities and relationships such as 'generalization' and 'containment'. The root of the registry, `SOA_Root`, contains a list of service type descriptors. It mainly supports two methods, one to register a service type and the other to find a service type reference. Every service type has to register itself with `SOA_Root` before registering any service instance. A service type registration requires following metadata for first level of service discovery which is stored in its `ST_Desc`:
Subsystem: Name of the subsystem that service type belongs to (e.g. process management, memory management).
Capability: The actual task that the service can perform (e.g. process scheduling, page replacement).
ST_Reference: A reference to an instance of `SOA_Root` that can polymorphically point to any service type.

Figure 1 shows two sample service types of `SOA_Root`, `Page_Rep_Service` and `Sched_Service`. The former corresponds to page replacement mechanism of memory management subsystem while the latter represents service of scheduling subsystem. Both service types are generalizations of their service instances and contain list of their service instances. The service instances in this case are page replacement and scheduling policies respectively. The service instance descriptors contain following metadata for second level search of a particular service:
Name: Name of the service instance (e.g. FIFO process scheduling, LRU page replacement etc).
Working: Short description of how the service instance works or operates.
Attribute Value List: A list of key-value pairs that are service instance specific and necessary for service instance to function. These key-values may be helpful to service consumer in selecting a service instance. For example, if a service instance of page replacement has a fixed queue size, it can be published with attribute-value as `<"queue_size",100>`.
Keyword List: Central dictionary of keywords that characterize the service instance.

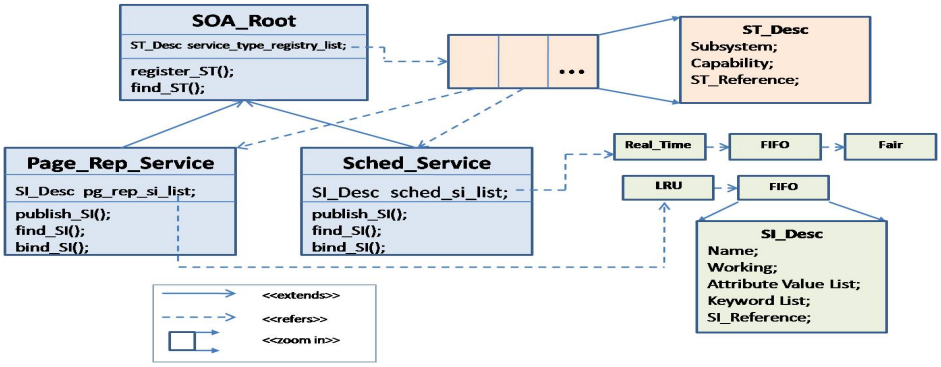

Figure 1. Service Abstraction Pattern for OS kernels

SI_Reference: Service instance object pointer is a generic pointer that refers to the actual functional unit in the subsystem. It is useful in binding the service instance with a process.

Publish-Find-Bind Paradigm: *Publish*, *find* and *bind* are the three programming abstractions which Dhara exposes to the kernel developers. Dhara allows publishing services at compile time as well as runtime. It is done in two phases: registering a service type and registering a service instance. The former involves specifying initial common state, profile functions and description of the service type. The latter includes defining policy (instance) specific structures and policy specific functions, helper functions, and metadata for the service instance. The *publish* abstraction initializes the descriptors of service types and instances in the registry. It also resolves the possible conflicts of different policies over the resource they manage at service type level.

Unlike *publish*, service discovery is carried out at runtime by peer kernel services or user applications. Services are discovered in case of failure of the current service, or simply because the needs of the invoker change. Numerous match making techniques can be used in this case, but they are out of scope of this work. However, the abstraction and the registry structure (due to the key value pair descriptors and hierarchical architecture) are generic enough to allow the service programmers to use the suitable service discovery method.

The user applications use *bind* to recommend using a particular service as a policy to the kernel, or kernel services can directly invoke other services at runtime. The service requester is required to provide the necessary parameters at this time. If the service requester is already bound with a similar service, safe unbinding precedes binding with the new service instance. The publish-subscribe model of Dhara is further illustrated in Section IV.

D. Multilevel Service Stack

The large number of kernel services of all subsystems and their relationships (component, composite or orchestrated) create a complex web, making it difficult to understand their interactions. In order to maintain coherency and structure among the kernel services, Dhara organizes the kernel services in Multilevel Service Stack(MSS) architecture. The architecture is constructed based on the level of abstraction: the lowest level being hardware devices and the highest being user applications. The services belonging to a layer are published in a registry corresponding to that layer using the SAP pattern described earlier. The higher level services try to discover services of a lower level at runtime and bind with them.

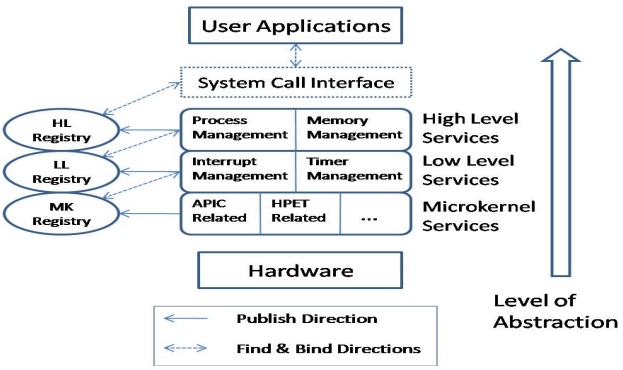

Figure 2. Multilevel Service Stack of Dhara depicting service abstraction layers of kernel

The high level service layer (HLSL) comprises of services that user applications can use to improve their performance, such as the page replacement or scheduling policies. Hence, services of process and memory management subsystems belong to this layer. The low level service layer (LLSL) offers the services of interrupt and timer management (e.g. generating scheduler ticks and enabling or disabling interrupts) which can be abstracted out as LLSL services since HLSL kernel services interact with them, not the applications. Finally, hardware or device specific services such as read, write and other services related to APIC

(Advanced Programmable Interrupt Controller), setting and starting HPET (High Precision Event Timer), etc. are encapsulated in the microkernel layer.

The MSS sees kernel layers as pools of services, where each combined with a registry creates an abstract view of that layer. This empowers the higher layer to dynamically reconfigure the services of the layer beneath it according to its need. This compartmentalization allows replacement of services of a specific layer without hampering functionality of the others. It is easy to adapt to evolving hardware as the nearest level to hardware, the microkernel layer, abstracts outs its actual working and only exposes services to the higher level. It can be argued that the microkernel layer is similar to Hardware Abstraction Layer of Windows NT or I/O Kit of Mac OS X, but they lack the framework to compose kernel services, that essentially user applications can use. The MSS can allow selectively choosing services at each level, thereby stripping down a generic operating system kernel for specialized devices such as hand held computers, smart-phones, sensor motes, etc. The MSS architecture can be tailored to build any large and complex system like compiler, database system, middlewares, etc. in a structured way.

The basic premise of Dhara is that kernel services should be added at each layer of MSS only as needed. Hence, services in HLSL may not be aware of the existence and usage method of some of the runtime additions in LLSL. The services in HLSL will not be able to form compositions if the metadata of those services are not available. The registry serves this purpose and stores service metadata in an organized fashion. For example, suppose only one Fair scheduling policy is available in HLSL initially. If later FIFO is added, it is published in a registry for the applications to discover. The unmatched requests of a service can be redirected to a default policy that can work suboptimally for almost all cases. MSS thus justifies the need of registry for services in the kernel.

Design Choices for MSS: The two alternative design choices we consider are as follows: The first approach is same as the one explained using Figure 2. Here, a layer can use services of only immediate lower layer. This approach provides clear separation of services and ensures that user applications have a transparent view of the HLSL services. However, consider a scenario where each layer offers a certain Service Level Agreements (SLA) to the layer above it. The HLSL may not meet its SLA to user applications if a critical low level service is not available. The user application may not be aware of unavailability of that service and fails to understand why SLA was not met. Thus, in case where SLAs have to be met tightly, this option may not be advisable.

The second approach is to empower user applications to compose end to end service. If an LLSL service is not available, user application would expect degradation of SLA. However, the design becomes very complex and expects user applications to be aware of LLSL services. The service composition also takes long time, since every time user application would have to search through registries of lower and microkernel level services. Therefore, for the sake of simplicity and because of lack of need of tight SLAs, we have selected the first design choice.

IV. Case Study: Linux Kernel

In this section, we present a case study of Linux kernel by applying the design principles of the proposed model Dhara in order to verify practicality and feasibility of this approach.

Linux kernel is a complex system consisting of numerous services. However, from a user application's perspective, the key services are CPU time and memory which are managed by scheduling and paging. These services in turn depend on timer and interrupt for periodic and aperiodic scheduling, generating page faults, etc. Occasionally, the application performs I/O operations; which boil down to services of interrupt, timer and device drivers. The Multilevel Service Stack of Dhara allows these services to be separated by their level of abstraction and into high level, low level and microkernel services.

A. Scheduling Policies Raised to Services

We have raised the scheduling policies to services using the service model of Dhara as an example of high level service. Its objective is to address the problem of inability to extend the service interface and to achieve loose coupling among policies. The scheduling service type is `Sched_Service` which is built around 'runqueues', a list of all runnable processes. Its service instances correspond to the actual policies, real time (RT) and fair share (FAIR). The state of the service instance is an instance of the policy specific data structures.(e.g. priority queue for RT). Service instance profile includes functions (e.g. enqueue, dequeue tasks) required by the policy to operate. The objects of these classes serve as grounding of the service instances. The description of the service type and its instances is provided at the time of publishing the services.

We have created a new scheduling service instance, FIFO, to demonstrate easy introduction of a new policy by extending the service type. Linux kernel provides FIFO as scheduling policy but only under real time class. However, a non-real time FIFO policy could be useful for non-interactive and long batch jobs. Our FIFO policy works at low priority as compared to FAIR and RT policies. The key issue here is how to add it without disturbing the working of existing scheduling policies.

Policy Manager: We have introduced a new service called policy manager to ensure smooth integration of a new policy. With multiple policies, it is difficult to decide the order among the policies in which they will be probed for the next runnable task. The policy manager resolves the conflict

between different policies using a notion of global priority. The global priority is associated with every policy at the time of publishing. At the time of scheduling, the policy manager visits the policy instances in the order of global priority from high to low. This way the tasks in FIFO policy do not interfere with tasks in real time policy. Thus, the CPU time is first divided among policies and then among the processes. We have to note here that fairness cannot be guaranteed for tasks across policies as they are scheduled using different mechanisms. However, the local priority of tasks can be honored within a policy by the service programmer.

Publishing and Usage: The publishing of a scheduling service instance starts with extracting values from attribute-value list of `SI_Desc` supplied as an input. This includes mandatory (e.g. global priority) and policy specific parameters. In case when the new global priority is conflicting with existing priority, it is reduced by a fraction and is inserted immediately after the existing service instance by the policy manager. Initializing description of service instance with the metadata concludes instantiation of the service. The discovery of scheduling service is based on keyword matching, where keywords describe the service instances. Each keyword is assigned a weight value to indicate its importance. These values are aggregated for all the keywords matched. The service with highest score is returned as a match. In case there is no match, the default service is returned. The binding procedure first checks if the requested service instance is available. It then checks the current scheduling instance of the process. If there is already one assigned to the process, the task is dequeued from the existing scheduling policy. The reference to the new scheduling instance is stored in `task_struct`, which is a per process structure. Once the assignment of new priority value and the new scheduling class is finished, the task is enqueued on new scheduling service.

B. Redesigning Memory Management with Services

The current paging mechanism is designed around the lists of active and inactive pages to store pages based on their access frequency. The swapper swaps pages out of the inactive list when a particular threshold is reached. Pages belonging to all the processes are managed by a single LRU policy. Integrating a new page replacement policy with the existing mechanism is a non-trivial task raising concerns like:

- Splitting memory among multiple policies fairly
- Making page replacement policy work on per process basis
- Facilitating a process to switch between policies

Dhara provides abstractions that makes it easy for policy developers to cater to the concerns raised. The service type for page replacement policies is `PR_Policy`. To demonstrate the introduction of new policies, we have instantiated `PR_Policy` service type into `PR_FIFO` and `PR_Perm`. The former follows First In First Out algorithm for page replacement, while the latter pins the pages once they are in memory, until the process ends.

To divide the memory among all policies, special counters are added in the service state to keep track of the total number of pages present in corresponding lists. They are used to ensure that a particular policy does not allocate too many pages and the overall system remains fair. If a process wants more pages than the threshold, pages of that process are moved under LRU policy. Another challenge is to make policies selectable on per process basis. The pages of different processes are segregated within the list to ensure that a page of the same process is swapped out to accommodate its new page. In case of PERM, page list division is used to impose maximum usage threshold. The per process page list was realized by adding `start_list` and `end_list` pointers in `task_struct`. The same structure also stores maximum usage limit for PERM policy and size of queue for FIFO, which are policy specific configurable parameters.

We have introduced a novel capability of `PR_FIFO`, `trigger_swap()` which can be used to invoke page swap daemon aperiodically. If a user application is aware of pages it refers and the threshold on FIFO queue size, it can use `trigger_swap()` to clean its page list at the right time.

Polymorphism using service type as abstract type answers the final question of facilitating a process to switch between policies (service instances).

Following points summarize how LRU, FIFO and PERM policies treat their pages:

- LRU policy serves a group of processes, while FIFO and PERM policies serve processes individually.
- The way resources are shared among the service consumers: LRU has a single *pool* of resources and it adds pages from all processes in it until the pool is filled. FIFO and PERM services maintain a per process *bucket* of pages and limit the size of individual buckets. This can be observed at other places in the kernel: A synchronization service is an example of group handling service. A ‘pipe’ service that facilitates inter process communication handles communication on individual basis.

C. Low Level Services

As defined in the Multilevel Service Stack, the low level services are the set of services which are used by high level services. We present a redesign of timer and interrupt services as low level services of the Linux kernel.

Background: Peter et al.[14] have studied the timer subsystem of Linux extensively. It was observed that every timer set by an application is directly managed by the timer wheel of the kernel, increasing the load on kernel. Moreover, each timer deadline is considered to be a hard one. For example, consider an expected arrival of 3 network packets after t+30, t+31, t+32 milliseconds. The present design treats

each timer as a hard deadline and each expires at the precise time. Each timeout generates an interrupt whose service adds to the performance degradation. However, if the packets do not arrive by *t*+30 ms, one can say with some confidence that all three may not come by *t*+32 ms. Similarly the packet with *t*+29 ms timeout can be held till *t*+30 ms. Thus, all four timeouts can be triggered at *t*+30 ms with +/- 2 ms precision incurring overhead of only one interrupt service. Hence, Peter et al. express the need of applications specifying that 'at *t* +/- *x* time or anytime after *t*, code *p* should execute' to the scheduler. It should be the duty of the scheduler to handle this request in the above mentioned manner.

New Design: Our design attempts to address the concerns raised by Peter et al. by raising timer and interrupt as services. These services aid the scheduling service in handling timers set by the applications. The proposed design works as follows: The applications register their timers in the format mentioned here with the scheduler. The scheduler maintains the expiry times of timers on per process basis and the points of execution. When a tick happens, the timer service invokes interrupt service. The interrupt service executes the corresponding service routine. Here, instead of maintaining entries in timer wheel for all application timers, it only invokes scheduling service. The scheduling service updates the elapsed times and schedules the tasks whose timers have expired from the specified execution points. Here the scheduler orders the execution of service subroutines of the expired timers in order of the process priorities from high to low. Thus, the timer and interrupt services are hidden from the user applications and are represented by scheduling only.

D. Microkernel Services

According to the Multilevel Service Stack of Dhara, the architecture dependent and device driver services are microkernel services. We have raised timer source as a service to illustrate microkernel services of the Linux kernel. A timer source is a hardware device which generates an interrupt periodically based on its precision or resolution. In a given computer system, multiple such sources are available. Linux kernel uses each one of them at a point of time. However, only one source remains active at a time. For example, to generate scheduling tick, only low resolution timer is used. The need may arise when the timer service needs to provide finer granularity to the scheduler. Considering such scenarios, timer source is exposed as a service to the timer service. Thus, based on the need of a higher level service, the timer service can choose appropriate clock precision from multiple sources which are made available as service instances in the registry. Other examples can include GPU and CPUs as computing services, network interfaces as services, interrupt controller (local for CPU cores and global) as services, etc.

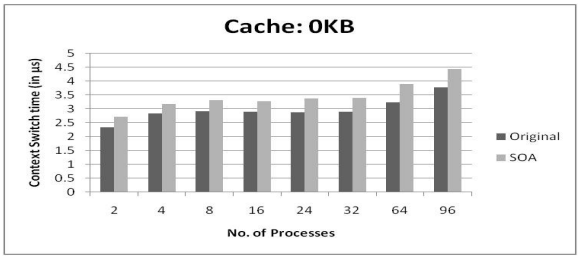

Figure 3. Context switch latency comparison for SOA enabled and original kernel using lmbench tool with 0KB cache footprint of process.

E. Implementation-Related Issues

The objective behind programming with services is to raise the level of abstraction from an object or a procedure. However, loose coupling among the services demands isolation and the ability to reuse interface to create new variants of a service. The same should be reflected in the implementation as well. The data encapsulation of C++ provides better isolation among objects. C++ facilitates easy *extension of interface* through inheritance, thereby increasing configurability of the kernel. The concepts corresponding to service type hierarchy and service instance in C++ are base class, derived class and their instances that represent inheritance. Thus, the service model is easy to implement as it naturally translates to C++. Hence, we decided to use C++ in the core kernel as a research challenge.

V. Performance Study

In this section we discuss the performance study of the new design that we have proposed for Linux kernel.

A. Experimental Setup

The major concern of any structural change in the kernel is its impact on performance. We evaluated scheduling and memory management subsystems to observe the cost and benefits of having multiple policies through the redesign. We used MOOL[8] kernel (originally Linux kernel 2.6.23) on Intel Pentium 4 machine with 3.4 GHz, 1 MB L2 cache processor and 1 GB of RAM. The reason of using MOOL is the presence of C++ runtime support in it. Though the kernel version we have used is not latest, the core kernel features like design of scheduler, page replacement, etc. are the same. The actual policies (CFS, RT, LRU) for them and how they are integrated with the kernel also remain unchanged.

B. Performance of Scheduling Service

We have used micro and macro benchmarks to measure the performance overhead in the scheduler. Lmbench[12] is a micro-benchmark suite that allows measuring context switch latency of a scheduler. It uses a ring of processes that communicate using pipes and pass a token in the ring. The token passing triggers a context switch from one process to another and the benchmark measures the time taken for

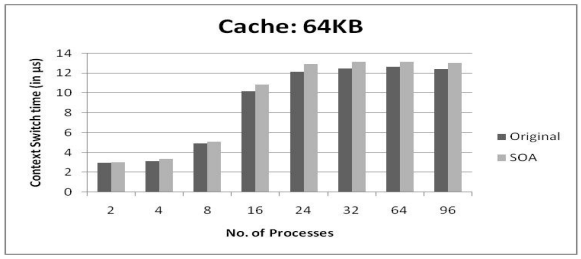

Figure 4. Context switch latency comparison for Dhara enabled and original kernel using lmbench tool with 64KB cache footprint of process.

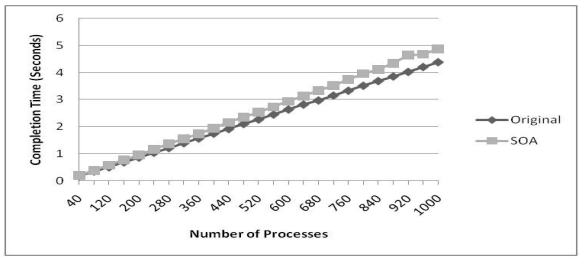

Figure 5. Comparison of completion time of 40 to 1000 processes under hackbench benchmark tool for DHara enabled and original kernel.

the context switch. Lmbench allows to specify size of data cache a process would need for its computation. We varied the number of processes from 2 to 96 and cache size of each process from 0 KB to 64 KB in our experiments. Figure 3 shows the context switch time in microseconds for Dhara enabled V/s original kernel version 2.6.23. The results shown are averaged over 100 runs. We deduce that as number of processes increase, so does the context switch latency for both Dhara and original kernel. Since the new implementation uses polymorphism, each call to `schedule` service requires resolution of virtual functions. Figure 4 illustrates similar experimental results, but with cache size as 64 KB for the processes. The key observation here is the sudden increase of latency after number of processes crosses 16. The machine used for the experiments in this case had 1024 KB of cache. When the number of processes crosses 16, the total size of cache needed also crosses 1024 KB, since each process has 64 KB of cache requirement. The difference in the context switch latency between original and Dhara enabled kernels was found to be 0.47 microseconds, regardless of the cache size needed by the processes. Thus, the overhead varied from 5% to 15% depending upon context switch time and hence the process size.

Hackbench[17] tool creates user specified number of groups of processes, each with 20 clients and 20 servers. Each client sends a message to each server in that group. We executed this test for number of groups varying from 2 to 50, i.e. 40 to 1000 processes. The tests were run for 100 times and the results were averaged out for plotting.

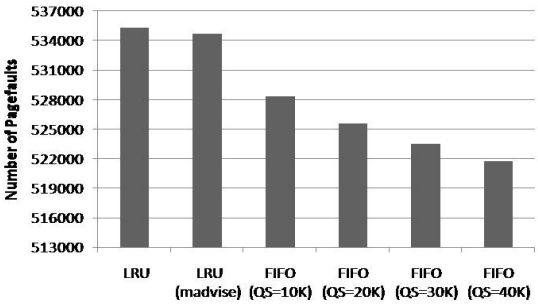

Figure 6. Page fault comparison of the test program for under FIFO and LRU policies.

Figure 5 depicts the overall completion time of 40 to 1000 processes in seconds. A reasonable overhead of 12.21% was observed due to service abstractions.

C. Performance Evaluation of Page Replacement

We conducted three experiments for page replacement policies. The first experiment measures the overhead of Dhara kernel over the normal kernel in terms of page replacement using LRU policy. The second experiment shows the improvement of using FIFO policy over LRU policy for a specifically designed test program. The third experiment compares the performance of original kernel with *madvise* system call against FIFO implementation under Dhara kernel.

The process completion time in Dhara kernel increases by 1.688% due to the introduction of polymorphic calls in page replacement as the result of first experiment. The number of pagefaults experienced by the test program using LRU policy in Dhara kernel and normal kernel remains almost same. This is because the implementation of service abstractions does not alter the program access pattern of pages.

To show the advantages of multiple page replacement policies, we have written a test program in C which benefits from FIFO policy. The test program allocates an array of size 1.05 GB(greater than RAM size). The elements in the array are accessed from start to end sequentially, and are accessed again to bring the pages back from the swapspace. The test program also uses the `trigger_swap()` abstraction to transfer excess pages to inactive list. We used `time` utility to measure the program execution time and the number of page faults generated. The results shown in Figure 6 are averaged over 5 runs. It can be seen that the number of pagefaults in FIFO implementation under Dhara kernel comes down by 7032 for FIFO queue size 10K, as compared to LRU. The reason behind this observation is that the page reference pattern is sequential and suitable to FIFO implementation as compared to LRU. FIFO queue size is a configurable parameter that can be specified by the test program at bind time. By varying FIFO queue size from 10K to 40K, we were

able to reduce number of pagefaults to 13620 as compared to LRU.

In the final experiment, we used `madvise` system call to instruct the kernel to treat the pages sequentially under LRU in original kernel. The pagefaults came down by 645 as compared to LRU, but still were not able to match the reduction in pagefaults under FIFO in Dhara kernel. This shows `madvise` will not give the same level of performance increase as compared to FIFO.

D. Discussion

The overhead observed in the experiments is mainly because of the implementation and is not inherent to the proposed architecture. This conclusion can be drawn from the fact that only possible overhead in the model is introduced by invocation of `find` and `bind` functions. However, these calls are made at the beginning when a process starts execution. Once all the necessary kernel services are attached to the process, they are repetitively used without invoking `find` and `bind` again.

The overhead of implementation can be attributed to usage of C++ as polymorphic calls are used extensively. The developers of a large scale system may be willing to pay this price to gain on modularity. However, in performance-oriented systems this may not be the case. In fact, the proposed framework is independent of implementation as the key aspects are service abstractions and related functions which facilitate discovery and binding. Hence it is possible to strike a balance between performance and degree of modularity by choosing the implementation language. For example, the higher level services can be implemented using C++, while lower level services can be written in C and so on.

VI. Related Work

We discuss how prior works differ in terms of platforms, motivation/objectives, design/features and implementation. We also compare Dhara with other known kernel design models in terms of extendability, ease of usage and degree of decoupling they provide.

Lutz et al.[19] view OS as a distributed environment in which computation capability and other resources are provided as services. This paper, however, does not define an explicit service model nor defines publish-find-bind abstractions in any way.

Milanovic and Malek[13] propose to achieve availability at OS level using service orientation in OS. The key emphasis is on continued availability of the service. Our approach is more towards redesigning kernel using service abstraction to reduce coupling and provide better configurability.

Liphardt et al.[10] explore how an SOA based OS can be used in a wireless sensor network. The open architecture of this system allows the services to be added and removed at runtime (migratable services). However, the core features like scheduling are defined as non-migratable services and they remain fixed throughout the lifetime. On the other hand, we provide effective mechanism to manage core kernel services as well.

Loadable Kernel Modules (LKM), Virtual File System (VFS), hierarchical scheduler[6], etc. are some of the popular mechanisms for making a kernel modular. LKM can be used to load device drivers at runtime, but it cannot introduce core policies like new page replacement or scheduling. Similarly, VFS can provide a common interface to only preexisting file systems. Hierarchical scheduler can provide multiple policies by having a process manage the scheduling of its children in a suitable way. Each child in turn can manage its children in different ways. However, this method incurs a significant overhead at lower levels.

Mircokernel[1] projects various kernel functionality as user level servers which communicate with each other using messages. The core kernel is very small and handles the messaging among servers. Unlike microkernel, Dhara comprises a hierarchical architecture for the kernel. The service model and abstractions of Dhara make the system easily extensible. Dhara advocates an architectural style based on service abstractions, which can be easily applied to existing and new kernels alike.

Infokernel[2] promotes the idea of exporting low level state of the system and allowing the applications to control the policies usually implemented by the OS. This work shares similarity with ours in terms of objective. In Infokernel, applications have to give specifics of the policy (e.g. list of pages to be swapped out, not the policy), which may be difficult for applications to decide.

Exokernel[4] and K42[3] have previously tried to provide such extendability to the operating system, but such approaches require the programmer to be aware of the services present or to use an existing library. Dhara unifies all kernel services under a common framework. We provide kernel developers with abstractions of publish, find and bind to introduce kernel services and corresponding system calls to the application programmers respectively. Finally, service composition and orchestration features of Dhara can be automated to choose the best combination of kernel services transparently from the applications.

VII. Conclusion and Future Work

This paper presents how service abstractions can be applied to OS kernels to improve their design. We highlighted the importance of multiple kernel policies and issues of OS kernels in introducing and using them. Dhara, the proposed design model enables developers to easily introduce new services and facilitate user applications to locate and use them at runtime. We provide an implementation of the proposed design. Our experiments show that the overhead caused by the new design is reasonable. Besides, newly introduced custom services perform better than existing ones

for their target applications. A future direction is to extend the service abstractions of kernel transparently to the legacy user applications. This will allow them to benefit from multiple policies, yet remaining oblivious to them. Another direction is to extend Dhara for a kernel with distributed services for multicore and cloud environments.

VIII. Acknowledgments

The authors would like to acknowledge Department of IT, Government of India for financially supporting this work. The authors thank Srinivas, Rakesh and Giridhari of IITM for their suggestions in design of page replacement policy framework design.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[2] A. C. Arpaci-dusseau, R. H. Arpaci-dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with infokernel. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 90–105. ACM Press, 2003.

[3] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[4] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[5] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino. An edf scheduling class for the linux kernel. In *Proceedings of the 11th Real Time Linux Workshop (RTLW)*, page 8 pp., 2009.

[6] B. Ford and S. Susarla. Cpu inheritance scheduling. In *Proceedings Of The Second Symposium On Operating Systems Design And Implementation*, pages 91–105, Seattle, Washington, USA, 1996.

[7] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. Technical report, Stanford, CA, USA, 1991.

[8] D. Janakiram, A. Gunnam, N. Suneetha, V. Rajani, and K. V. K. Reddy. Object-oriented wrappers for the Linux kernel. *Software Practice & Experience*, 38(13):1411–1427, 2008.

[9] A. Kumar, A. Neogi, and D. Janakiram. An OO based semantic model for service oriented computing. In *Proceedings of the IEEE International Conference on Services Computing*, SCC '06, pages 85–93, Washington, DC, USA, 2006. IEEE Computer Society.

[10] M. Lipphardt, N. Glombitza, J. Neumann, C. Werner, and S. Fischer. A Service-Oriented Operating System and an Application Development Infrastructure for Distributed Embedded Systems. In *17th GI/ITG Conference on Communication in Distributed Systems (KiVS 2011)*, pages 26–37, Dagstuhl, Germany, 2011.

[11] D. Lohmann, W. Hofer, W. Schröder-Preikschat, and O. Spinczyk. Aspect-aware operating system development. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011*, pages 69–80, 2011.

[12] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[13] N. Milanovic and M. Malek. Service-oriented operating system: A key element in improving service availability. In *Proceedings of the 4th international symposium on Service Availability*, ISAS '07, pages 31–42, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] S. Peter, A. Baumann, T. Roscoe, P. Barham, and R. Isaacs. 30 seconds is not enough!: a study of operating system timer usage. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 205–218, New York, NY, USA, 2008. ACM.

[15] V. Rajani, A. Kumar, and D. Janakiram. ξ-calculus: A calculus for service interactions. In *IEEE International Conference on Services Computing*. IEEE Computer Society, 2010.

[16] V. Reddy and D. Janakiram. Cohesion analysis in Linux kernel. In *Proceedings of the 13th Asia Pacific Software Engineering Conference(APSEC)*, pages 461 –466, Bangalore, India, December 2006.

[17] R. Russell. Hackbench: A new multiqueue scheduler benchmark. *http://lkml.org/lkml/2001/12/11/19*, December 2001.

[18] S. Schach, B. Jin, D. Wright, G. Heller, and A. Offutt. Maintainability of the Linux kernel. *Software, IEEE Proceedings*, 149(1):18 –23, Feb. 2002.

[19] L. Schubert and A. Kipp. Principles of service oriented operating systems. In *Proceeding of The Second International Conference on Networks for Grid Applications (GridNets)*, pages 56–69, Beijing, China, 2008.

[20] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981.

[21] L. Yu, S. R. Schach, K. Chen, G. Z. Heller, and J. Offutt. Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD. *Journal of Systems and Software*, 79(6):807–815, 2006.

[22] L. Yu, S. R. Schach, K. Chen, and J. Offutt. Categorization of common coupling and its application to the maintainability of the Linux kernel. *IEEE Transactions Software Engineering*, 30:694–706, October 2004.