inTune: Coordinating Multicore Islands to Achieve Global Policy Objectives

Priyanka Tembey Ada Gavrilovska Karsten Schwan
Georgia Institute of Technology

Abstract

Multicore platforms are moving from small numbers of homogeneous cores to 'scale out' designs with multiple tiles or 'islands' of cores residing on a single chip, each with different resources and potentially controlled by their own resource managers. Applications running on such machines, however, operate across multiple such *resource islands*, and this also holds for global properties like platform power caps. The *inTune* software architecture meets the consequent need to support platform-level application requirements and properties. It (i) provides the base *coordination abstractions* needed for realizing platform-global resource management and (ii) offers *management overlays* that make it easy to implement diverse per-application and platform-centric management policies. A Xen hypervisor-level implementation of inTune supports policies that can (i) pro-actively prepare for increased or decreased resource usage when the inter-island dependencies of applications are known, or (ii) re-actively respond to monitored overloads, threshold violations or similar. Experimental evaluations on a larger-scale multi-core platform demonstrate that its use leads to notable performance and resource utilization gains: such as a reduction in the variability across request response times for a three-tier web server by up to 40%, and completion time gains of 15% for parallel benchmarks.

1 Introduction

As systems integrate more heterogeneous resources and scale out to many cores, it is imperative that their diverse resource managers coordinate to achieve platform-global properties and/or to run applications efficiently. In this paper, we present inTune: a set of system-level coordination abstractions and mechanisms that let resource managers interact with each other to achieve global properties and make it convenient for applications to interact with them via standard interfaces.

To sustain 2x performance growth with every chip generation, many-core architectures are evolving beyond high core counts and complex memory hierarchies to 'scale-out' architectures, such as with the 'tiles of cores' approach [17], to directly address scalability concerns for hardware cache coherence and thermal design power (TDP) bounds [18, 36]. Similar scalability concerns with routing a single global clock across the chip within limited power bounds and at ever increasing core frequencies, may lead to emergence of multi-clock sets of cores mapped to independent clocks and voltage/frequency domains [19, 30].

Hardware designs with such on-chip networked tiles of independent, disaggregated nodes give rise to software architectures structured as clusters of multiple, independent resource managers on a single chip [14, 15]. With the emergence of core and memory heterogeneity [24, 2] supported by specialized runtimes or custom device drivers (e.g., GPGPU platforms), these independent managers may also be heterogeneous. Further, even without hardware heterogeneity, multiple functionally different resource managers (e.g., real-time vs. throughput-oriented CPU schedulers) may manage spatially multiplexed CPU core-sets on a single platform to meet the requirements of highly diverse applications, like high throughput web services vs. real-time services like VOIP [22].

Systems with multiple distinct resource manager entities have been built to support heterogeneity in hardware resources [20, 29] and to enhance scalability [7, 15, 14]. Efficiently running applications on such systems, however, requires dealing with two fundamental issues.

I. *Maintaining global properties.* Applications may operate across multiple resource domains, and their resource managers schedule application components independently unaware of overlaying end-to-end application

TRIOS'13, November 03 2013, Farmington, PA, USA Copyright 2013 ACM 978-1-4503-2463-2/13/11$15.00.
http://dx.doi.org/10.1145/2524211.2524213

properties. How can applications obtain and maintain their desired levels of performance given the resource managers' independent scheduling decisions? Coordinating independent schedulers has been shown important for high-throughput graphics codes, between CPU threads and their GPU activities [16], and for parallel systems so that program threads avoid undue levels of timing variation for synchronization [15]. How to additionally maintain platform-level properties, like maintaining system-wide power caps across proposed independent voltage/frequency domains of cores [30] using coordinated power allocation [27, 28]? Stated in terms of systems functionality, what are the system-level coordination abstractions and supports needed to let independent managers interact to obtain these goals and maintain these properties?

II. *Diverse and independent managers.* Consider applications running on a single platform with multiple resource managers, with their own scheduling policies and maintaining their own threading or resource abstractions. How do they interact with these managers, via which interfaces, and/or can managers hide such complexity by coordinating with each other on an application's behalf?

Figure 1 shows an example that maintains global properties for a multicore platform on which different sets of software-partitioned cores are managed by independent and functionally different CPU scheduler entities. In this case, scheduler diversity serves to meet different functionality requirements, i.e., the need for real-time vs. throughput-centric core scheduling. Yet as typical for larger scale web applications, a single such code spans multiple of these core sets and their managers and so, it becomes difficult to maintain its desired end-to-end properties. From this example, we derive the following objectives for our research: (i) the need for standard abstractions and interfaces that provide applications or higher level policy managers the means to efficiently and conveniently interact with a platform's diverse resource managers, and (ii) the need for interaction – *coordination* – across different managers to provide to even a single application the composite levels of service needed to run it with the performance it requires.

The *inTune* systems software presented in this paper provides an efficient framework for managing applications and their performance properties on many-core systems organized as multiple resource sets, termed *resource islands*, with their own *island managers*. Our research makes the following three key contributions.

1. inTune implements ***resource islands***, as a system-level abstraction that places a subset of resources under the control of a specified resource manager. With islands, one can partition multicore hardware for reasons of scalability, e.g., the 'tiles' in the Intel SCC chip, for reasons of hardware heterogeneity, e.g., for commodity vs. accelerator cores, or to enable functionally different schedulers using software partitions. The example studied in this paper are islands created as software partitions for multiple CPU sets managed by distinct scheduler functions, to suit the varied needs of server and parallel applications using a multicore platform.

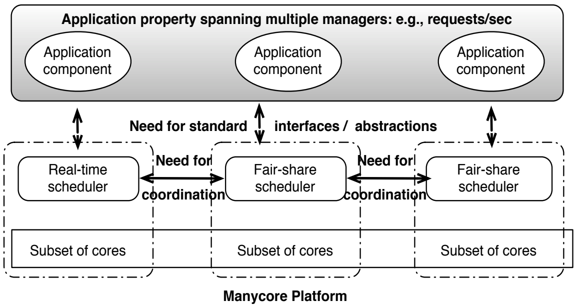


Figure 1: How to maintain global properties in multi-resource-manager manycore platforms?

inTune islands are dynamic partitions of multicore resources, so that within the limits defined by underlying hardware capabilities (e.g., cache coherence, or architectural incompatibilities), they can be flexibly sized to meet applications' elastic resource demands.

2. The ***inTune framework*** – offers system-level functionality that provides APIs and base mechanisms for inter-island coordination. Its mechanisms permit island managers to independently manage their resources, but then: (i) extend island-level resource managers, by pairing them with per-island *coordinators* that then interact with each other (and with their respective 'native' resource managers) using inTune APIs to achieve platform global properties, thus (ii) abstracting away from applications the heterogeneous resource management implementations of different islands.

3. ***Management overlays*** – are inTune's system-level representations of application and platform properties. They (i) encapsulate and maintain state that defines the specific property of interest (e.g., performance requirement of an application or platform utilization cap), (ii) define the linkages across the inTune coordinators involved in carrying out actions that serve to obtain and maintain the property, and (iii) can use proactive or reactive methods for policy implementation by invoking inTune coordination mechanisms.

The inTune abstractions and methods in this work are implemented with the Xen hypervisor [6], for multicore platforms with independent sets of CPU islands managed by distinct scheduler functions. The hypervisor-based implementation serves the purpose of exploring potential platform enhancements without reliance on specific guest OS features or functionality. inTune's methods and principles, however, can also be implemented in other system infrastructures, including in the operating system [23], or in island-based microkernels with island resources managed by library OSs [12].

We next present technical evidence of the importance and utility of the inTune approach for efficiently running applications on multi-island platforms (see Section 2). This is followed by Section 3 describing the inTune architecture and its components. Section 4 explains the inTune implementation, followed by experimental evaluations in Section 5. Section 6 summarizes prior related work with conclusions and future work appearing at the end.

2 Motivation for Islands and inTune

In order to highlight the use of islands and inter-island coordination to efficiently run an application across systems comprising multiple resource islands on a single hardware platform, Figure 2 demonstrates the performance effects seen for a representative parallel application, Fluidanimate from the PARSEC application suite when run in different configurations as follows. Fluidanimate, like other parallel codes, operates in multiple CPU-intensive phases implemented via explicit barrier synchronization. For this code, it is important to minimize its performance variability in each phase, so that all of its threads simultaneously reach the barrier.

On a 12-core hyperthreaded x86 Westmere processor platform, we execute copies of Fluidanimate codes inside each of three virtual machines with 512MB memory and configured with different number of VCPUs for the following two scenarios: (1) when the platform is underloaded and there is no CPU contention i.e., each VM is configured with 4 VCPUs (the left set of bars in the figure), and (2) when the platform is overloaded, such as in the event of a higher degree of workload consolidation i.e., each VM is configured with 8 VCPUs (the set of bars on the right). For each scenario, we compare mean application completion times along with observed deviation in (i) the default – the *NoIslands* case running native Xen credit scheduler, to (ii) a case where CPU resources are partitioned into smaller cells – the *Islands* case (with six cores each) – and (iii) a case where islands are elastic because there is explicit coordination across their individual schedulers, the goal being to achieve island right-sizing during an application's CPU-intensive phases – the *IntuneIslands* case.

The figure shows performance improvements with a higher number of application threads in the *overloaded* scenario for all three cases. However, we observe high amounts of variability in Fluidanimate completion times in the *NoIslands* case, particularly in the *overloaded* configuration. Use of partitioned CPU-sets in the *Islands* case lowers these levels of performance variability – which suggests that even at these moderate scales, *islands are important*. However, in the *overloaded* configuration in the *Islands* case, due to insufficient CPU resources or due to lack of knowledge of workloads' worst-case requirements, application performance may actually deteriorate due to infeasible static right-sizing of islands (as denoted by higher completion times). This implies that *coordination across islands is critical* to more flexibly size islands by trading resources, in order to meet application performance needs. This is shown via both improved completion times and lower variability values in the *IntuneIslands* case.

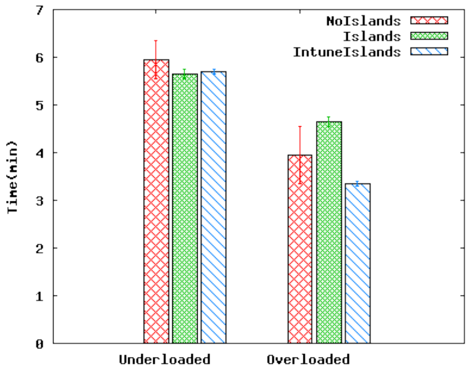


Figure 2: Islands reduce performance variability and at increased loads, coordination helps in island right-sizing.

In general, Section 5 shows that performance advantages gained from using inTune abstractions can be striking – leading to a reduction in the variability across request response times for a three-tier web server by up to 40%, and completion time gains of 15% for parallel benchmarks.

3 inTune Architecture

The *inTune* software architecture is an efficient framework for implementing platform-level and application-centric management for future many-core systems comprised of multiple, distinct *resource islands*. Platform-global properties are met using the following key design components of the inTune architecture:

3.1 Resource Islands

Figure 3 illustrates the main inTune components for a representative four-island, general-purpose, multi-core NUMA platform. A resource island is a system level representation of a subset of the resources (e.g., CPU, memory) present on the multicore platform in Figure 3, which in this case, can be initialized to naturally match

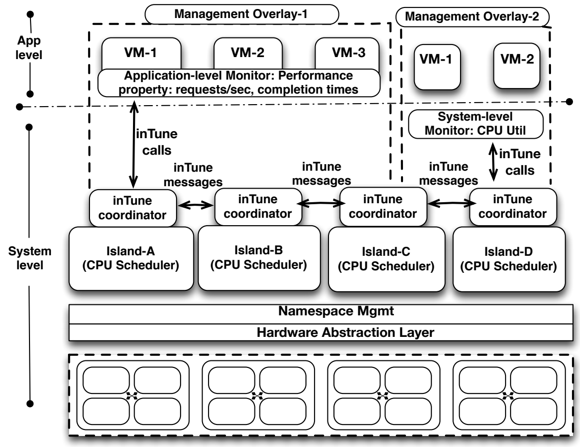


Figure 3: inTune architecture components.

the multi-socket (4 sockets) nature of this machine. Each socket however can be configured to host multiple islands. The CPU resource within each island is managed independently by a scheduler (e.g., credit-based, best effort, or real-time), determined at the time of island creation, and is not visible to other islands.

For island creation, a boot kernel initially performs discovery of resources, namely, enlisting the CPUs, memory, and devices in the global namespace of the entire platform. A single hypervisor image booting on all cores performs this task. Using this globally available shared namespace, CPU islands are created by grouping subsets of CPUs, assigning a single CPU scheduler to each, via a global platform manager entity (e.g., the privileged domain in our hypervisor-based model – not shown in the figure for clarity). The global manager also updates the shared namespace with mappings between islands and resources they manage. Memory islands may be defined by assigning NUMA memory node mappings to an island in a NUMA multicore platform. The global platform manager also instantiates a *coordinator* thread entity per resource island, primarily to interface with other islands in the system.

3.2 Management Overlays

Achieving platform-level properties, like power caps, and application properties, like requests/sec for multi-tier web services, requires actions that span multiple resource islands. Island resource managers, however, are unaware of those, and so, will schedule application components in an isolated and independent manner. This encourages scalable operation, but can detrimentally affect application performance (see Section 2) and prevent meeting platform needs. Responding to this issue, *management overlays* (i) encapsulate and maintain the state that defines desired global platform or application properties, at system-level, and in addition, they (ii) *actuate* inter-island coordination methods to achieve their specific properties.

In our hypervisor-based implementation, application-centric overlays (see Overlays-1 and 2 in Figure 3) comprise (i) the collection of component VMs used by a single application, (ii) the performance property (e.g., requests/sec, response times) specified at overlay-creation time, (iii) performance monitor threads implemented at the system-level and/or application-level that periodically monitor resource management state pertaining to the performance property, and finally (iv) system interfaces that the monitor threads use to request resource (CPU, memory) reconfigurations in order to maintain this property. These requests are then translated into inter-island coordination actions at the system-level.

Creating overlay components. A global platform manager creates an application-centric overlay by first creating the application VMs and mapping them to resource islands. Overlay VMs are *registered* using a unique overlay identifier with each constituent island. Further, each constituent island is made aware of the mapping between overlay VMs and other peer islands of the overlay. Affinity of application VMs to islands is determined by island properties, including available number of cores, memory size, and scheduler function. Islands are queried for these properties to decide initial VM-island mappings. An individual VM within an overlay may not span island boundaries to preserve isolation of the VM and the island resource management. An island's resources, however may be shared by multiple distinct overlays (see Island-C in Figure 3 hosting VMs of Overlay-1 and -2). The overlay monitors are then started at the system level (Overlay-2 in Figure 3) or application-level (Overlay-1 in Figure 3) to periodically monitor performance state, and to request reconfiguration of resources for component VMs. Finally, the platform manager also creates the necessary inter-island communication channels among islands within an overlay (e.g., amongst Islands-A, B and C in Overlay-1 and between Islands C and D in Overlay-2 in Figure 3).

The global platform manager is hence primarily responsible for creating the various components of overlays and passing relevant overlay state to constituent islands. inTune's current design hence delegates the runtime overlay management overhead of (i) interfacing with the overlay monitor for its resource re-adjustment requests, and (ii) translating those requests into subsequent actuations of inter-island interactions, to island coordinators instead of a centralized global manager. We make this design choice favoring scalability of overlay management for future manycore platforms, however inTune's architecture does not prohibit more centralized implementations.

Example overlay properties and their management. Overlay-1 in Figure 3 is comprised of the three VMs

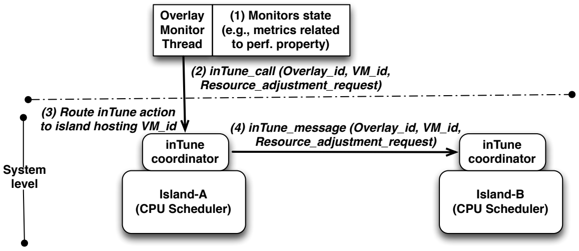

Overlay Monitor Thread
(1) Monitors state (e.g., metrics related to perf. property)
(2) inTune_call (Overlay_id, VM_id, Resource_adjustment_request)
(3) Route inTune action to island hosting VM_id
(4) inTune_message (Overlay_id, VM_id, Resource_adjustment_request)
System level
inTune coordinator
Island-A (CPU Scheduler)
inTune coordinator
Island-B (CPU Scheduler)

Figure 4: Overlay management actions using inTune API.

representing Web (VM-1), Application (VM-2), and Database (VM-3) tiers, each running in separate islands - A, B and C. The property implemented by this webserver overlay is to maintain the 99th percentile of its response time latencies within a pre-defined upper bound. In its current realization, the overlay relies on the webserver VM (VM-1) to use intelligent application-level request classification [5], and to monitor/detect changes in request traffic patterns. Based on such data, the overlay's internal policy then advises *tuning* of resource allocations for the backend server VMs (VMs 2 and 3), or it may request to increase its own resource allocations. Under heavy request loads (such as during request spikes) and/or with workload consolidation, driven by the web server's observations, it may also request for *urgent, boosted* scheduling of the backend server VM to avoid undue request processing lags.

Figure 4 shows how overlay resource allocation requests are propagated to its constituent islands. Upon observing events pertaining to the maintenance of its property, the overlay monitor initiates an inTune API call to tune or urgently adjust resource allocations that are then executed entirely at the system-level. Application-level overlay monitors specify this request in the form: *inTune_call(Overlay-id, VM-id, resource_adjustment_request)* (specific inTune calls are further explained in Section 3.3). This assumes that the overlay monitor knows how the application components are mapped to overlay VMs and their identifiers. Overlay monitors however need not be aware of underlying islands, as these requests will always be directed to the local island, which will further actuate inter-island coordination on behalf of the overlay, if necessary. An overlay monitor's resource reconfiguration request of the form *(Overlay-id, VM-id, resource_adjustment_request)* is then translated into an inter-island coordination message of the form: *(Overlay-id, Island-id, VM-id, resource_adjustment_request)* at the island-level by the island's local inTune coordinator using knowledge of island -to -overlay VM mapping initiated at overlay creation time.

The application-centric overlay described above uses application-level monitoring to implement proactive policies for maintaining properties expressed in terms relevant to the application. More generally, overlay policies can also be based on system-level monitoring, and they can use reactive vs. proactive methods for actuation. Overlay-2 in Figure 3, for instance, uses observation-based monitors in the Xen hypervisor, which we have added to more conveniently gather per-VM performance and resource utilization state (i.e., using performance counters).

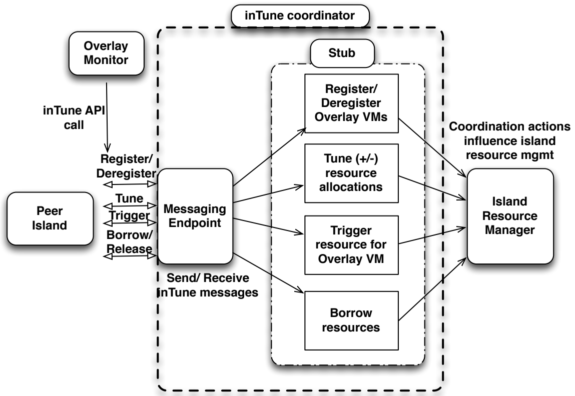

inTune coordinator
Overlay Monitor
inTune API call
Stub
Register/ Deregister Overlay VMs
Tune (+/-) resource allocations
Trigger resource for Overlay VM
Borrow resources
Coordination actions influence island resource mgmt
Island Resource Manager
Register/ Deregister
Tune
Trigger
Borrow/ Release
Peer Island
Messaging Endpoint
Send/ Receive inTune messages

Figure 5: inTune coordinator and inTune messages.

We next describe the inTune coordination API and the inTune coordinator design.

3.3 Coordination with inTune

In order to obtain and maintain global overlay properties distributed across multiple distinct islands, island-local events in an overlay must be able to influence resource management actions in remote islands within the overlay, and external events must be able to influence local island management. With inTune, any actuated coordination action directed toward a remote island is propagated via explicit communications. Such message-based coordination ensures that inTune can operate both across strongly (via shared memory) and weakly (via PCI) connected resource islands.

Message channels are used for communication across island coordinators. Their platform-dependent realization may use message queues implemented in shared memory regions or DMA operations across the PCIe interconnect (e.g., as for the prototype x86-IXP platform used in our prior work [34]). Each active overlay uses an independent set of transmit and receive message channels (see Section 3.2) between pairs of all its constituent islands, the purpose being to cleanly isolate different overlays' coordination actions. These message channels are created at overlay creation time by the global platform manager. Each constituent island is also initiated with routing state to reach other islands in the overlay, and differentiates between sets of message channels by their overlay identifier.

An ***inTune coordinator*** (see Figure 5) is responsible for (1) interfacing with the overlay monitor that requests resource re-adjustments to maintain platform- and/or application-level policies leading to inter-island coordination actions. (2) carrying out the necessary explicit communications across islands, to coordinate *advisory* coordination actions across islands, and (3) implementing stub-like functionality for an island, so as to translate remote messages into local resource management actions. To achieve this, each island's coordinator implements the following set of abstractions, illustrated in Figure 5:

1. **inTune coordination messages** – implement the inter-island coordination mechanisms supported in inTune, summarized in Table 1. As seen in Section 3.2, the occurrence of certain events in an overlay (e.g., a peak in client requests at the front-end web server, or when an overlay exhausts its CPU resource) leads to resource re-adjustment requests from overlay monitors. These requests are translated into inTune coordination messages at the island-level. *The subsequent coordination resulting from sending/receipt of these messages amongst islands of an overlay along with their self-reconfiguration serves to accomplish the overlay's resource adjustment requests, and hence the overlay performance property.* inTune supports the following messages – *tune, trigger, borrow* and *release* to let an island convey coordination actions that may influence resource allocation decisions in peer islands, in response to its local overlay requests.

Motivation behind inTune API. Inter-island coordination APIs in future islands-based systems should possess the following properties: (i) The API needs to be *expressive* enough to successfully translate resource adjustment requests from application and/or platform overlays, so as to meet their properties. (ii) The API should be *generic* and should support a *unifying* set of abstractions to be applicable to heterogeneous resource managers of varied resource types (e.g., CPU, memory). (iii) The API actions should be implemented with low overheads for those resource types.

In choosing to support the specific messages – *tune, trigger, borrow, release* in the current inTune coordination API, we account for the range of resource adjustment request types that overlays may use (for ease of expression), the properties of resource-types managed by islands (for generality) and the overheads of implementing them. We classify resource adjustment requests as those that pertain to (i) increase/decrease or *tuning* of resource shares for overlays over subsequent time periods of application runtime and, (ii) urgent provisioning or *triggering* of resource shares for time-critical use. *Tuning* of resource shares is limited by island resource capacity. Additional resource adjustment beyond an island's capacity may necessitate *borrowing*, and then *releasing* of resources from other islands. Urgent provisioning of resources may be requested in a time-shared preemptible system, and is translated into a *trigger* message. We believe the API comprising of *tune, trigger, borrow, release* is also generic and can be applied to varied resource types such as CPU and memory. We next discuss coordination messages in further detail and consider their overheads and generality when applied to different resource types.

When an island uses *tune* coordination messages to influence a remote island's management, it permits one island to non-preemptively request dynamic adjustment – *tuning* – of the resource shares assigned to a virtual machine in an overlay in a remote island for a specific resource. This makes it straightforward to increase/decrease a particular resource allocation given to a remote VM, without specifying actual 'amount' parameters. This also enables *tune* to non-intrusively integrate external events with an island's resource management decisions, without complete knowledge of the algorithm used by the island's resource manager. A successful *tune* action for the CPU resource at the remote island involves increasing/decreasing CPU shares of a particular overlay VM along with modifying CPU scheduler state in the remote island. Similarly, *tuning* of memory resources may add/release memory pages for the overlay VM using *ballooning* techniques [35], along with appropriate changes to the VM's page tables. A successful *tune* call may be followed by additional such calls from the overlay monitor, if allocations are still insufficient. The remote island sends a *negative* acknowledgement if it fails to carry out the *tune* action.

trigger, on the other hand, has pre-emptive semantics. If permitted, a *trigger* requests immediate execution of some overlay VM in the remote island. *trigger* may be used to deal with time-critical runtime decisions, where island run-queues are reordered to boost a VM's VCPUs. Implementing *trigger* for the CPU resource involves possibly pre-empting the current VCPU task, saving its state and reordering scheduler run-queues to boost the target VM's VCPUs. Preempting memory is a heavy-weight operation which may involve saving and restoring a VM's address space. *Triggering* memory resource hence has practical limitations.

tuning a VM's resource shares is limited by the island's resource capacity. *borrow* and *release* are further used to flexibly re-size islands, i.e., to request resources from an external island and add them to the pool of the local island, and to then either release them voluntarily or upon request. When resource borrowing is caused by some specific management overlay, the first attempt is to borrow resources (e.g., CPU cores) from other islands in the same overlay (e.g., in the same multi-tier web application). If a 'borrow' request is not satisfied within an overlay, current policy is to borrow resources beyond the overlay from the island that has the maximum available capacity. To determine idle capacity, previous research offers many design alternatives, including continued distribution of resource freelists across all islands [9]. To avoid incurring the overheads of such periodic updates, our current design resorts to explicit resource utilization queries to other islands when local utilization exceeds some limit. *Borrowing* resources to re-size islands is limited by hardware properties such as cache coherence and core architecture. In our current platforms where islands are software partitions of homogeneous cores, borrowing of cores is a relatively low overhead operation that involves (i) freeing up cores by migrating running tasks to other cores in the lender island, and then further (ii) updating borrower island state to reflect addition of cores to its capacity. We evaluate inTune overheads in Section 5.2.3. Finally, the *register* and *deregister* actions are used to notify the inTune co-ordinators of an overlay component's entry into or exit from an overlay.

2. **Messaging endpoint** – via which the coordinator communicates with peer coordinators. A messaging endpoint is set up during island creation. Routing information about the other island coordinators in an overlay is made available to the endpoint at the time of overlay creation (see Section 4). The endpoint drives the sending and receipt of coordination messages on the messaging channels connecting coordinators. Since an island can be part of multiple overlays, the messaging endpoint also implements an *arbitration algorithm* when dequeuing coordination messages from multiple overlays (round robin and priority-based in current design).

3. **Scheduler stub** – is a set of functions that translate the information carried in inTune messages to appropriate actions in the target island, and vice versa. For instance, a *tune* call pertaining to the CPU resource will make the stub adjust the CPU shares of VMs according to its own credit system. Such actions will be in accordance with the management state and constraints in the target island.

Arbitration. When islands are part of multiple overlays, an island's inTune coordinator may receive requests with conflicting control actions, e.g., that compete with other overlay components for an island's limited resources, or conflict with the island's platform-centric objectives (e.g., its CPU caps). The current inTune coordinator supports two methods for arbitrating across conflicting requests: (1) a simple round-robin method to execute incoming requests, and (2) a priority-based method that executes requests in order of decreasing overlay priority. Based on these arbitration methods at the island-level, the coordinator executes a 'winner' request and sends back negative acknowledgements to conflicting requests. Arbitration can be further optimized, for instance by including the use of historical information to deal with repeated oscillations (e.g., core ping-ponging between two islands due to borrow-release). Under certain conditions, like repeatedly declined (i.e., 'nack'd') requests (five successive ones in the current prototype), arbitration decisions are forwarded and handled by the global platform manager. Finally, an overlay's use of one resource in an island may need to be arbitrated against other resource managers. For instance, a CPU borrow request may need to be arbitrated against the overlay's potential memory bandwidth use, as it may interfere with other overlays sharing the memory bandwidth. Arbitration across two application overlays competing to borrow a third island's CPU resources is described in Section 5.3.

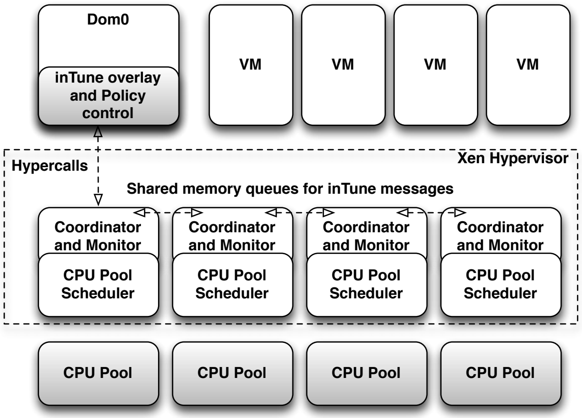


Figure 6: inTune components in Xen hypervisor.

4 Implementation

Islands and inTune have been implemented in the Xen hypervisor, on two different platforms: 8- and 12-core x86-based machines. The hypervisor is extended to construct multiple scheduling islands (see Figure 6) comprised of pools of CPU, each with a separate, possibly functionally different, CPU scheduler to manage the allocation of the island's cores. The initial configuration of islands, and scheduler assignment is performed by a global manager running as a user-level process in the privileged domain, Dom0. Beyond island creation its actions include defining inTune overlays by creating and mapping application VMs to islands. After overlay initialization, the Dom0 manager is needed only for (infrequent) global arbitration across multiple overlays. All of its actions, initiated from Dom0, are implemented as hypercalls.

Two periodic threads, implemented as Xen *softirq* contexts on the first CPU in every island, are responsible for per-island monitoring and as the per-island inTune coordinator respectively. A monitor thread periodically (every 30ms – which is the Xen scheduling epoch)

tune (resource_type, increase_request/decrease_request, overlay_id, island_id, vm_id) request increase/decrease of resources allocated to a given overlay VM; the actual change to resource allocation is determined locally at the remote island
trigger (resource_type, overlay_id, island_id, vm_id) request immediate boost to resources allocated to a given overlay VM;
borrow_preferred (resource_type, overlay_id, island_id, request_amount) request to temporarily gain exclusive access to resources currently managed by target remote island.
borrow_any undirected borrow request to multiple islands and then choose based on available utilization.
release (resource_type, overlay_id, island_id) release resource to lender island.

Table 1: inTune Coordination messages.

calculates island CPU utilization as part of system-level observations of overlay progress (Section 3.2). Any ***inTune coordinator*** implementation is inherently manager-specific, because it must translate inTune messages into actions understood by the island's resource manager. One of our current implementations is a coordinator for the Xen Credit scheduler. Regarding the inTune messages described in Section 3.3, a *tune* message translates to tuning the CPU cap allocation, which is a parameter that is already supported by the credit scheduler. A *trigger* results in boosting a VCPU to the front of a CPU runqueue, but only if it has sufficient credits to spare in the current scheduling epoch. We added this functionality to the Xen Credit scheduler. For resizing island resources via the *_borrow* mechanism, the implementation sends resource utilization queries to all islands within its overlay, and then to other islands, and borrows from the one having maximum available capacity. The coordinator thread polls for incoming inTune messages every 30ms and, based on feedback from the local monitor maintaining island utilization state, applies external control actions locally, if possible. A negative acknowledgement message is sent back to the messaging island in case the control action cannot be applied due to local constraints.

Management overlays. Management overlays encapsulate the platform-level or application-centric properties and run the methods used for obtaining these properties. When creating an overlay, the global manager in Dom0 first retrieves island information (e.g., current load in islands and available resource capacity). It then performs a basic matching of overlay resource requirements to underlying island resources (i.e., admission control) and updates state regarding the overlay's mapping to islands. The constituent island coordinators are then notified by *registering* the overlay and overlay VMs, establishing pairwise message channels among the corresponding islands, and initiating the island messaging endpoints with routing state (shared memory pointers) to reach remote islands in the overlay. Message channels are implemented as producer-consumer message rings. The global manager also initiates hypervisor-level monitors that may be distributed across islands and maintain island-local monitoring state like island utilization, but it assumes monitoring within the application to be initiated by the application itself. A combination of such hypervisor-level monitors and/or application-level monitoring periodically updates and checks overlay state pertaining to the overlay property to be attained. An example application-level monitor keeps track of incoming request streams for a web-server overlay, and in reaction to observed traffic patterns, requests resource re-allocations to backend components. This particular monitor implementation is simple and attributes to an additional 50 lines of code to the existing web-server code used in Section 5.2.1.

To trigger overlay operations in response to changes to application- or platform-level property, we add inTune coordination hypercalls to overlay monitors that invoke local island coordinators, as was shown in Figure 4. Xen hypercalls to local islands will include the inTune directive (tune, trigger, borrow) and an additional overlay VM identifier, for which resource allocation needs to be adjusted. In our simplified current overlay implementation, we assume that the overlay monitor exactly knows its resource type needs while specifying an inTune request. Our application-level monitors are implemented in the same language runtime as the application (see Section 5.2). However, we then use language-specific extensions to invoke the xen-level 'C' inTune hypercall API.

An overlay implementation need not be island-aware, as further translation of a VM identifier to an island hosting the VM is performed at the local island, using initial state known at overlay creation time. Inter-island interactions are performed directly by island coordinators.

Since an island may be a part of multiple overlays, arbitration may be needed to mediate across multiple policy overlays; we currently use round-robin and priority-based arbitration. Upon successive negative acknowledgements the global manager is invoked for veto control.

5 Experimental Evaluation

Jointly, inTune resource islands, inter-island coordination, and management overlays serve to realize application- and system-level policy objectives on scale-out multicore systems. This is demonstrated with experimental evaluations performed on two platforms: (1) a 8-core hyperthreaded dual socket Nehalem x86 host and (2) a 12-core hyperthreaded dual socket Westmere x86 host, both running the Xen 4.0 hypervisor. Experiments use the RUBiS enterprise benchmark, PARSEC and Sequoia parallel benchmarks, Apache Web servers, and Map-Reduce codes. A *platform efficiency* metric is used to evaluate inTune's resource management mechanisms. The metric measures application performance against resource utilization, where a higher *platform efficiency* value using inTune signifies its benefits.

5.1 Meeting Global Platform Properties

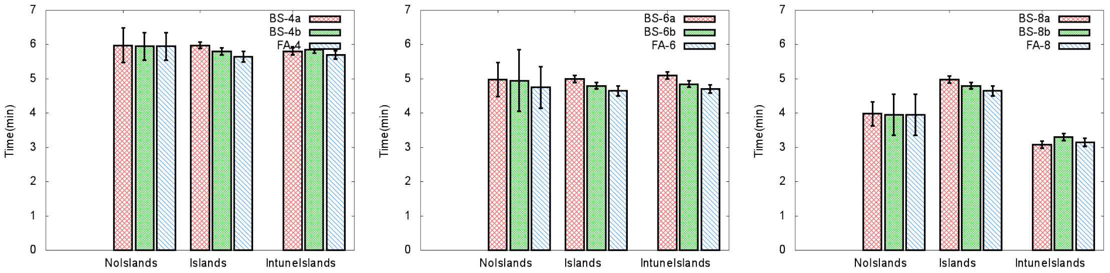
Consider a platform CPU cap, as an example of a specific platform level property. We achieve this global property as a sum of per-island CPU caps,

Experiment setup. We run the inTune prototype on a 12-core hyperthreaded Intel X5660 Westmere machine, which for this example, is initially configured into four islands of three cores each. We use parallel benchmarks from the PARSEC suite to generate CPU load within each island. In three islands, we deploy one PARSEC application per VM; we leave the first island for Dom0 execution. Given our virtualized environment, three possible relationships may hold for the total number of VCPUs(nVCPU) and total number of PCPUs(nPCPU): (i) nVCPU < nPCPU, (ii) nVCPU = nPCPU, or (iii) nVCPU > nPCPU. For brevity of presentation, we report a case in which the number of PCPUs in an island is kept constant and equal to 6, changing the number of VCPUs for each deployed VM as 4,6, and 8 for the three cases. The number of PARSEC application threads inside the VM is always equal to the number of VM VCPUs, to avoid oversubscribing within a VM. Every CPU island is managed by a separate Xen credit scheduler and has its own inTune coordinator. Given that the platform has 24 threads, the maximum CPU utilization is 2400. The global property is to maintain the CPU cap at 2000, by specifying an initial platform-wide overlay with a per-island CPU cap of 500.

Coordination is needed to achieve platform properties like CPU caps. We first compare the *coordinated* management approach (labeled *IntuneIslands*) with the native Xen scheduler (*NoIslands*) with respect to their abilities to maintain platform-wide properties. The *NoIslands* case is the default Xen case, where a single Xen scheduler manages the entire platform. We observe that with inTune coordination, we can continuously ensure the platform-wide CPU cap of 2000, whereas with the default case, this property is violated. In this experiment, coordination among islands is triggered in *reaction to events*. The platform overlay in this case creates local monitors for each island, to track its local CPU cap (refer to Section 3.2). When a local monitor detects a violation of its local cap, this indicates that the island has exceeded its resource capacity, and so, the monitor actuates the local inTune coordinator's *borrow* mechanism to borrow a core from a peer island. Overlay monitors in both lender and borrower islands, then, recalculate their local CPU caps without any central controller intervention. The cap enforcement is subject to availability of idle capacity on the platform. Note that the parallel applications used in this experiment exhibit collective phases of computation, followed by relatively idle phases. With inTune, even without knowledge about such application behavior, solely by monitoring CPU utilization, we are able to detect the presence of idle capacity on islands, so that this capacity can then be made available to islands with greater CPU demand. As stated earlier, we conclude from the experiment that *coordination is a necessary means for obtaining desired platform-level properties.*

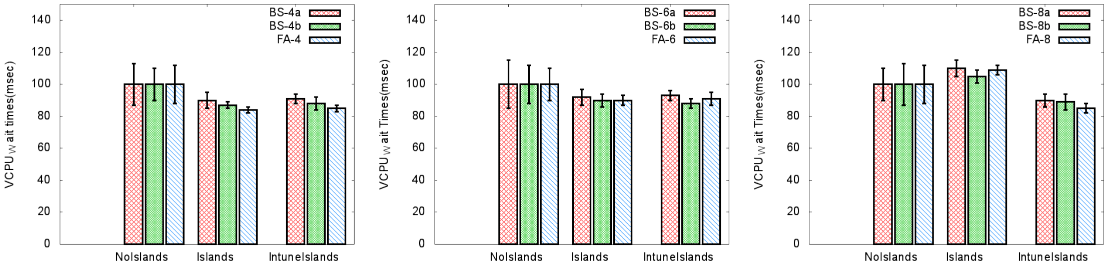
Utility of island elasticity. Figures 7a, 7b, 7c show the mean completion times of the three benchmark applications along with standard deviation bars for all three nVCPU:nPCPU configurations. The first and third set of bars in each figure correspond to the *NoIslands* and *IntuneIslands* cases described above. The second set of bars shows times for 'uncoordinated islands' (*Islands*), where no coordination takes place across the islands' CPU managers. The following obversations can be made about the results shown in the figure. When nVCPU<=nPCPU, in Figures 7a and 7b, we observe that island creation reduces the level of performance variation seen in the default *NoIslands* case by close to 15%, and it also shows small gains in average completion times. *IntuneIslands* shows similar performance as *Islands*, as the use of *borrow* is never triggered for the case of nVCPU<=nPCPU. However, when nVCPU>nPCPU (see Figure 7c), the use of uncoordinated islands may result in performance degradation, since we are *statically* partitioning the platform, thereby restricting the CPU resources available to applications. In fact, as seen in the *IntuneIslands* case in Figure 7c, with the use of inTune's mechanisms for *borrowing* and *releasing* resources, flexibly resizing islands reduces performance variation from 10% to less than 0.2% and achieves close to 16% improvements in mean completion time compared to the centralized *NoIslands* case. From Figure 8, we see that with inTune, these gains in performance variation are primarily because of reduced VCPU_wait times for individual VMs in the *IntuneIslands* case compared to the *NoIslands* case, sometimes by up to 20%, as when nVCPU>nPCPU, indicating that re-adjustment of island size reduces runqueue lengths, thereby scheduling VCPUs in a more timely manner.

In Table 2, we show the aggregate performance in-



(a) VCPU<PCPU, nVCPU=4 (b) VCPU=PCPU, nVCPU=6 (c) VCPU>PCPU, nVCPU=8

Figure 7: PARSEC: average completion times. BS: Blackscholes FA: Fluidanimate



(a) VCPU<PCPU, nVCPU=4 (b) VCPU=PCPU, nVCPU=6 (c) VCPU>PCPU, nVCPU=8

Figure 8: PARSEC: average VCPU_wait times.

	NoIslands	inTune
Completion Times(min)	11.88	9.98
Avg CPU Utilization	938	1040
Platform Efficiency	8.97	9.63

Table 2: PARSEC – platform efficiency.

	Base (req/s)	RUBiS-2 (req/s)	RUBiS-3 (req/s)
Throughput	75req/s	110req/s	126req/s
Avg CPU Utilization	385	475	492
Avg session time(s)	473	454	440
Platform Efficiency	0.19	0.23	0.26

Table 3: RUBiS – Throughput Results.

formation, and the resulting platform efficiency for the nVCPU>nPCPU case. With inTune, average CPU utilization increases equivalent to one hardware thread, but this increased resource utilization at the appropriate time(s) also delivers lower completion times. The outcome is a higher platform efficiency value (1/ (CPU-Utilization*Completion-time)), thus meeting our global policy objective with application performance benefits.

5.2 Application Overlays

We next show how inTune's management overlays and system-level coordination abstractions help achieve performance properties for applications spanning across multiple islands, e.g., in scale-out platforms.

5.2.1 Ensuring Predictable Response Times for Multi-tier Web Applications

As a representative multi-tier web application, we use RUBiS, a well-studied auction website prototype modeled after eBay, comprised of an Apache webserver frontend, a Tomcat Servlets application server, and a MySQL database server backend, all deployed in separate Xen hardware virtual machines. Each component VM with three VCPUs is deployed in its own separate island of 2 cores each. Dom0 is deployed in a dual-core island of its own. A RUBiS client is deployed on a separate x86 dual-core host, with 2 GB physical RAM. The standard RUBiS benchmark client has two workload profiles: browsing (read) mix and bid/browse/sell (read-write) mix. From previous work [5] and our own profiling, we know that the browsing mix results in a large amount of web-server and application server processing with practically no database processing. For the read-write mix, database-intensive processing is initiated at the server end. We use this analysis to drive coordination policies that use inTune's methods.

Overlay setup and coordination methods. This particular realization of the RUBiS application overlay uses proactive policies to actuate inTune coordination by ex-

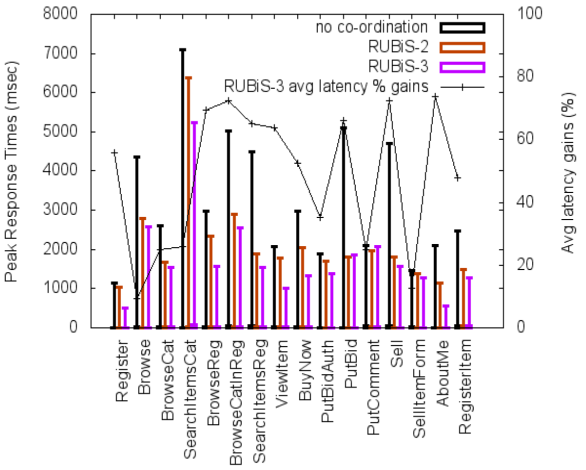


Figure 9: RUBiS Min-Max Response Times. Coordination helps in peak response latency alleviation.

ploiting the application knowledge gained from offline profiling (Overlay-1 in Section 3.2). We modify the Apache webserver to classify incoming client requests into bid and browse-type requests, and to also count the numbers of requests of each type every 30 seconds. Depending on the request traffic pattern in the previous sample, the webserver propagates this information via a *_tune* hypercall to the hypervisor-resident inTune coordinator on its island. The coordinator then sends a *_tune* coordination message either to the application or the database island, as appropriate. Resource reallocation within islands due to actuation of *_tune* progresses between three states: (1) no-coordination – a case when there is no coordination and resources are allocated under a *RUBiS-3/2* policy with all three application VMs capped at 1.5 cores within their 2-core islands, (2) *RUBiS-2* – a case when only inTune's *_tune* messages are used, and the initial allocation changes from 1.5 to 2 cores for the application or the database server; and (3) *RUBiS-3* – a case when both *_tune* and *_borrow* messages are used, and the CPU allocation further changes to a *RUBiS-3* one, with 3 cores for either the application or the database server island, depending on the incoming request pattern. As a result of the receipt of the *_tune* message, the remote island CPU scheduler may change its CPU allocations from *RUBiS-3/2* to *RUBiS-2*, and vice versa, or go a step further, and make a *_borrow* request, so its allocation can be changed to *RUBiS-3*. The *RUBiS-3/2* policy is representative of cases where webserver resource usage may be capped initially for lower client request rates, and subsequently adapted based on varying load requirements [10].

Elasticity: from 'tuning' to 'borrowing'. Figure 9 shows the min-max response times for different types of RUBiS requests in a read-write browsing mix workload with 480 simultaneous client sessions and the normalized average latency benefits of the *RUBiS-3* configuration over *RUBiS-3/2*. We experiment with varying numbers of simultaneous sessions, starting from 180. On our testbed, the baseline *RUBiS-3/2* configuration can sustain up to 360 sessions, beyond which a *_tune* coordination reconfigures the application and database resource allocations to *RUBiS-2* state. Upon increasing the number of sessions to 480, the application and database server reach maximum CPU utilization in their islands under a *RUBiS-2* allocation. Tuning works only up to capacity limits, so this is when a *borrow* request is used to get another core to go to *RUBiS-3* allocation.

We also observe that *RUBiS-2* and *RUBiS-3* exhibit reduced variation for almost all request types, sometimes by up to 60%, as in case of the 'PutBid' request type. The average response latency gains, also presented in Figure 9 (see right-side Y axis) for the *RUBiS-3* allocation as a percentage value decrease over the baseline *RUBiS-3/2*, show similar trends of decreased response times when the application and database servers receive appropriate CPU allocations for the corresponding request types, sometimes by up to 70%.

Note that for two of the request types, 'PutBid' and 'PutComment', response time latencies for *RUBiS-3* case are higher than that of *RUBiS-2*. We attribute this increase to the slight overheads experienced when borrowing and releasing cores (Borrow_preferred and Borrow_any mechanisms take 291 and 494 nanoseconds to complete), and the fact that our classification engine is not very sophisticated. However, even with this simple classifier, we are able to see substantial benefits due to the flexible resource allocation achieved via inTune's mechanisms. Table 3 shows additional performance metrics for the RUBiS benchmark, where the use of inTune-based coordination clearly results in improved performance and more efficient utilization of platform resources. The platform efficiency metric in Table 3 justifies the resulting higher CPU utilization with increased application throughput *and* lowered response time, thus demonstrating the importance of using inTune mechanisms for coordinated management. We conclude that *inter-island coordination plays an important role in regulating and optimizing the performance of web and datacenter applications*.

5.2.2 Latency-sensitive Codes

It is not just the aggregate resource availability that determines application performance, but in addition, it is important 'how' and 'when' applications' use of island resources is scheduled. We demonstrate this via an inTune-realized *coscheduling* policy for applications

	Uncoordinated		inTune	
	Time(ms)	%Var	Time(ms)	%Var
Completion Times (mpi_barrier)	1580	±6.0	1361	±2.93
Completion Times (BS_32_1)	4122	±0.2	4197	±3.2
Completion Times (BS_32_2)	4117	±0.1	4207	±3.1

Table 4: MPI collectives overlay.

running across multiple islands, making use of inTune's *trigger* mechanism.

Overlay setup. We use the MPI Barrier benchmark, part of the Sequoia parallel benchmark suite [32], which times MPI_barrier operations for a specified number of iterations across all processes in its communication world. We divide 16 MPI processes among 2 VMs, each with 8 VCPUs and 256M memory. In *inTune* parlance, this MPI world of 2 VMs becomes our application level overlay to which policy objectives may be applied. Each VM is assigned to a separate 4-core island on a 12-core hyperthreaded Intel Westmere machine. The remaining 4 cores are reserved for Dom0. Each island's CPU resources are managed by separate credit schedulers. We launch two compute-intensive Blackscholes benchmarks in separate 8 VCPU VMs, and assign each one to the created islands, so that every island has two VMs, one running MPI and the other running Blackscholes. Table 4 shows the barrier performance results of this experiment.

***Coordination: coscheduling using* trigger.** It is well-known that the performance of applications using collective operations like barriers is improved when participating processes are coscheduled. To achieve this, we permit the MPI application overlay (or the communication stacks they use) to explicitly initiate coordination, by providing them with hypercalls directed at the inTune coordinator. Using these hypercalls, the application can inform its island's coordinator to *trigger* all VMs in the MPI overlay. The local coordinator then sends appropriate *trigger* messages to islands that contain the other VMs to be coscheduled; using the routing information distributed at the time of overlay creation (Section 3).

As seen from Table 4, with such coordination, barrier completion time is reduced by 13%, compared to the uncoordinated case, and we also observe reduced variation across consecutive barrier runs. Such *trigger*-based coordination could negatively affect other workloads, but the two Blackscholes VMs experience tolerable performance degradation (less than 4%), which is less than the cumulative advantage seen for the coscheduled MPI VMs. The slight variation for MPI VMs in the *inTune* case can be attributed to 'advisory' control design principle used by the coordinator; this means that an attempt to coschedule VCPUs may not always succeed for reasons of fairness within the credit scheduling algorithm. Note, however, that even the worst case performance remains better than the uncoordinated case. We conclude that *inTune interfaces and coordination support help performance-critical codes define relevant policies to improve their performance*.

5.2.3 inTune overheads

The overheads of our inTune implementation consist of: (i) messaging overheads – i.e., time to create an inTune message by assembling all relevant parameters, send the message to the target island by copying to the appropriate shared memory channel, and then receive the message, again by copying from the shared memory channel, in our implementation; (ii) arbitration overheads – i.e., time to arbitrate among multiple requests; (iii) scheduling of software interrupt at the target island to schedule the inTune coordinator thread, and finally (iv) executing the specific coordination action.

Considering messaging overheads and specific coordination action overheads, a *tune* call needs 212ns, *trigger* needs 293ns, *borrowing* in a two-island overlay requires 381ns, and from among all six islands (of two cores each) needs 594ns (due to additional message routing decision costs). By comparing to a 'null' message, which takes 150ns, we observe the enqueueing and dequeueing of inTune messages on the shared memory channel represent a dominant factor of total cost. These overheads can be reduced by leveraging future hardware support (e.g., like fast messaging support for control exchanges on Intel's SCC chip) or by better mapping inTune's explicit message-based communications to the underlying hardware interconnects [7]. Scheduling of software interrupts on our Westmere platform incurs additional overheads between 200-400ns. This only accounts for dispatch overhead, not potential cache misses, etc.

A potential variable cost is incurred due to arbitration across multiple overlays that may queue up within one scheduling epoch of the inTune coordinator (30ms). The total number of simultaneous resource readjustment requests at a coordinator is the sum of requests sent by each overlay mapped to an island within the previous 30ms interval. These may also include previously 'nack'd' requests. In order to place a bound on this sum, a limit may be applied to the total number of outstanding requests an overlay sends in a time-interval, while using techniques like exponential back off [33] to delay sending subsequent requests. We are still exploring such optimizations to better scale our arbitration methods as the number of overlays and islands scale.

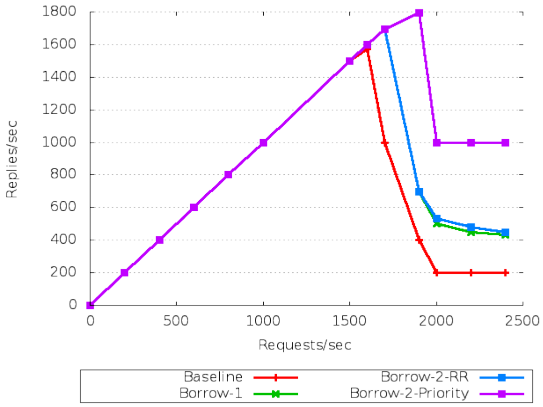


Figure 10: Apache Web server throughput scales with increasing number of cores, especially when its ‘borrow’ request is prioritized over other overlays.

5.3 Arbitration among Policy Overlays

In the presence of multiple management overlays, there may be multiple control requests to an island’s coordinator that (i) compete for the island’s available resources, or (ii) conflict with the island’s local objectives (e.g., local CPU caps). Such situations require arbitration, as demonstrated by the next use-case in which inTune arbitrates across competing application overlays attempting to borrow a third island’s CPU resources.

Arbitrating between two competing overlays. Consider the inTune prototype running on the 12-core Westmere system (with disabled hyperthreading), initially configured into 4 islands. Dom0 occupies the first island of 4 cores. The second island of 4 cores hosts one VM running the Blackscholes application (of 2 threads) from the PARSEC suite. The third and fourth islands have 2 cores each and host a 4-VCPU Apache v2.4 Web server application and a 4-VCPU Phoenix Map-reduce Word-count [31] application VM, respectively. Each application VM is configured with 2GB of memory, and runs Ubuntu Linux kernel version 3.0.2. The Apache Web server VM services incoming requests for a 14KB static image from an external client system running *httperf*, while the Word-count application VM processes a 1GB text file using four Map and Reduce threads. At the beginning of the experiment, the 4-core Blackscholes island is least utilized and has an available capacity of 3 cores. The Apache Web server island starts capping its CPU resource, and hence experiences a throughput degradation when the *httperf* request rate starts exceeding 1500 requests/second, as seen in the ‘Baseline’ configuration in Figure 10. The Word-count island also caps its CPU, and both the Web-server and Word-count islands attempt to borrow a CPU core each from the Blackscholes island. The Blackscholes island satisfies both requests, reconfiguring itself to 2 cores, and lending one core each to both requesting islands. ‘Borrow-1’ in Figure 10 shows the Web-server throughput scaling with an additional core – by 10% compared to ‘Baseline’ and sustains almost double the ‘Baseline’ throughput at higher request rates.

A further increase in the *httperf* request rate causes the web-server island to again start capping CPU, matched with a similar capping of CPU in the Word-count island, causing both islands to again send ‘borrow’ requests to the Blackscholes island. However, as the Blackscholes island has only one core available, its inTune coordinator needs to arbitrate between the two requests. With *round-robin* arbitration, the Blackscholes island satisfies the Word-count island’s request, denying a core to the Webserver island (see ‘Borrow-1-RR’ case in Figure 10). However, we noticed that the Word-count application does not gain much with an additional core (completion time reduces by 20% with the first borrow request, but improves by only an incremental 5% with the second borrowed core). Hence, we next experiment with *priority-based* arbitration, which prioritizes the Web-server overlay over the Word-count overlay, thereby lending the available core to the Web-server island, instead (see throughput gains and scaling for ‘Borrow-2-Priority’ case in Figure 10 – sustaining almost four times the throughput at higher request rates). We conclude that *arbitrating across competing overlay requests is a necessary and important function of the inTune coordinator*, enabling it to make better decisions while maintaining application and platform properties, especially in resource-constrained, consolidated systems.

Discussion. In the current implementation of our inTune prototype, the arbitration policy supported by the inTune coordinator (round-robin vs. priority-based) is configurable and can be chosen via the global platform manager at the time of coordinator initialization. As seen in Section 3, if an overlay’s requests are successively nack’d over five intervals of 30ms by an island’s inTune coordinator due to insufficient resources, the inTune coordinator invokes the global platform manager for overall arbitration. Current policy of the global platform manager is to initiate capping CPU usage of another overlay chosen at random, and to re-allocate available CPU resource to the ‘starved’ overlay. We are exploring more sophisticated arbitration policies for the platform manager that provide weighted performance degradation for all applications [26]. In the event of insufficient resources to provide acceptable performance levels to all applications, necessary interactions with higher-level schedulers and load balancers (at the datacenter level) should take place.

Finally, the overall fairness and stability of the inTune coordinator can be further enhanced by accounting for the *history* of previous decisions.

5.4 Summary of Evaluation

Evaluation use-cases demonstrate the utility of inTune, and address the two issues in managing the resources of scale-out platforms introduced in Section 1.

Maintaining platform-centric and application performance properties. Section 5.1 shows for multi-island many-core systems, a platform-wide property of CPU cap can be implemented, using inTune policy overlays and its *borrow* and *release* mechanisms. Sections 5.2.2 and 5.2.1 demonstrate the necessity of *trigger*, *tune*, and *borrow* for implementing application-specific overlay properties. In all such cases, trigger and tune are the initial mechanisms of choice, with borrowing and realizing used when capacity limits are reached. We can also claim, based on the results seen in these use-cases, that coordination using inTune (i) is important for managing platforms and meeting application performance requirements (supported by the consistently improved performance and lower performance variation we see with coordinated islands compared to the centralized Xen scheduler) and (ii) is sufficiently versatile for representing a variety of higher level policies for island-based platforms. In Section 5.3, we additionally demonstrate the ability of the inTune coordinator to arbitrate amongst multiple consolidated overlays. Arbitration will be particularly important in consolidated data center systems, to 'better' allocate resources when realizing multiple properties with resource-constrained platforms.

Standard interfaces for diverse resource managers. With the coexistence of multiple, functionally different and/or heterogeneous resource managers becoming the norm in data center systems, we argue that rather than trying to design one scheduler to deal with all platform resources [8], using multiple scheduler islands and adding inter-island coordination interfaces is preferable. With this approach, custom interfaces and interactions exist only to 'translate' control actions between specific resource managers (e.g., real-time vs. credit-based schedulers) and the controllers with which they interact, whereas controllers interact via well-defined and -proven techniques based on the principles of online system control [25] and elasticity.

6 Related Work

Resource islands. The concept of resource islands used in Helios [29] has its roots in earlier work that includes Cellular-Disco [15], Hive [11], and K42 [21]. While Helios [29] uses satellite kernels to account for heterogeneous runtimes, Hive [11] uses resource-partitions for fault-containment, and K42 [21] uses them to exploit locality. The implementation of islands via hypervisors in our work is similar to the cell-partitions approach followed in the Cellular-Disco system [15], with the key additional contribution of general system-level support for abstractions and methods to coordinate these islands for achieving management goals. These abstractions permit arbitrary interactions amongst coordinating islands compared to fixed gang-scheduling in the Cellular-Disco system. Also, inTune accommodates cross-VM application dependencies and hence, coordinates VCPUs across VMs, not limited to a single VM. inTune complements micro- and exo-kernels [12], by building resource management abstractions on top of a minimal kernel using message passing and its contributions related to resource management apply to systems software in general: including island-based microkernels with island resources managed by library OSs [12]. Recent work includes the Tessellation OS [23] that creates space-time partitions (STPs) encapsulating applications and OS services within 'resource containers' [4] for performance isolation. STPs are complementary to our resource-level abstraction - 'resource-islands' where STPs may be time-multiplexed by OS-level island schedulers, similar to our hypervisor-based islands scheduling application VMs. In addition, applications need not be aware of the island abstraction, as STPs are exported to applications.

Messaging and coordinated scheduling. Communication primitives and messaging abstractions for resource management have been studied for grids and clusters – CCL [3] and Condor [13] being two widely-studied examples. In recent research, the use of message passing and RPC for inter-core communication has seen renewed interest in Barrelfish [7]. inTune adopts some lessons from such prior work in terms of its loose coupling across resource managers, but it extends the messaging based system of Barrelfish [7] with additional semantics focused on resource management. Also relevant to our work are the two-level scheduling problems discussed in previous research using scheduler activations [1], specifically in terms of our approach of exchanging only relevant and limited information about resource islands via relaying selective control, a similar approach is used by scheduler activations between user and kernel-level schedulers.

From multi-agent systems to inTune. The choice of the abstractions and coordination primitives supported by the inTune architecture is not accidental. Instead, since inTune is intended to enable and support coordination, its features are defined based on a substantial body of work in system controls and multi-agent systems. In particular, in reference to distributed control theory and multi-agent systems for power engineering [25], inTune coordinators are similar to intelligent agents in that they are: (1) *proactive*, initiating their own local control actions, (2) *reactive*, such that they can react to local monitored events; (3) *social*, such that they communicate management actions with other coordinators; (4) *autonomous*, such that they have local control and treat external control as 'advisory'.

7 Conclusions and Future Work

A key problem for future large-scale and heterogeneous multicore platforms is how to manage applications and their performance properties spanning across their multiple 'resource islands', each with their own resource managers and methods. InTune realizes an approach and framework for developing and evaluating future management methods, using as a guiding theme, coordinated management in which there are explicit interactions between different islands' resource managers taken on behalf of applications to maintain their properties. inTune lets islands cooperate using standard interfaces and mechanisms on behalf of applications hence abstracting away their implementation details.

inTune is implemented as an extension of the Xen hypervisor, and its management overlays using message-based coordination primitives are shown to operate on the Intel Nehalem x86-based manycore machine used in this paper. Experimental evaluations for both web and parallel applications demonstrate the utility and importance of the coordinated islands approach in meeting applications' end-to-end needs (with improved throughput, predictability, and lower response times) than when using a single homogeneous Xen hypervisor. In ongoing work, we are also exploring its use for meeting platform-level objectives like power budgets [28] and optimizing arbitration methods for stability and management efficiency.

Acknowledgements

We would like to thank our shepherd, Andrew Baumann and other anonymous reviewers for their useful insight and feedback for improving this paper. We would also like to thank Vishakha Gupta, and Dilma Da Silva for their comments on earlier versions of this paper and inTune in general. This research has been supported in part by the NSF via CCF-1161969, and by Intel Corp.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM TOCS*, 1992.

[2] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating System Implications Of Fast, Cheap, Non-volatile Memory. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, 2011.

[3] V. Bala, J. Bruck, S. Member, R. Cypher, P. Elustondo, A. Ho, C. tien Ho, S. Kipnis, and M. Snir. CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 1995.

[4] G. Banga and P. Druschel. Resource Containers: A New Facility For Resource Management in Server Systems. OSDI '99.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, 2003.

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[8] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to many cores. In *OSDI*, 2010.

[9] A. Butt, R. Zhang, and Y. C. Hu. A Self-Organizing Flock of Condors. In *SC '03*, 2003.

[10] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *IWQoS 2003*, pages 381–398. Springer, 2003.

[11] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-memory Multiprocessors. *SIGOPS Oper. Syst. Rev.*, 1995.

[12] D. R. Engler, M. F. Kaashoek, et al. Exokernel: An Operating System Architecture for Application-level Resource Management. In *SOSP*, 1995.

[13] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 2002.

[14] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *OSDI*, 1999.

[15] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 2000.

[16] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated Scheduling in Virtualized Accelerator Systems. In *Usenix ATC*, 2011.

[17] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. *ISSCC*, 2010.

[18] W. Huang, K. Rajamani, M. Stan, and K. Skadron. Scaling with Design Constraints - Predicting the Future of Big Chips. In *IEEE Micro*, 2011.

[19] A. Iyer and D. Marculescu. Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[20] V. Kazempour, A. Kamali, and A. Fedorova. AASH: An Asymmetry-aware Scheduler for Hypervisors. In *VEE*, 2010.

[21] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da, S. M. Ostrowski, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a Complete Operating System. In *EuroSys*, 2006.

[22] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting Soft Real-time Tasks in the Xen Hypervisor. In *VEE*, 2010.

[23] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *HotPar*, 2009.

[24] G. H. Loh. 3D-Stacked Memory Architectures for Multicore Processors. In *ISCA*, 2008.

[25] S. McArthur, E. Davidson, V. Catterson, A. Dimeas, N. Hatziargyriou, F. Ponci, and T. Funabashi. Multi-Agent Systems for Power Engineering Applications Part I: Concepts, Approaches, and Technical Challenges. In *IEEE Transactions on Power Systems*, 2007.

[26] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-Service in Large Disk Arrays. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*, 2011.

[27] A. K. Mishra, S. Srikantaiah, M. T. Kandemir, and C. R. Das. CPM in CMPs: Coordinated Power Management in Chip-Multiprocessors. In *SuperComputing*, 2010.

[28] R. Nathuji and K. Schwan. VPM Tokens: Virtual Machine-aware Power Budgeting in Datacenters. *Cluster Computing*, 2009.

[29] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *SOSP*, 2009.

[30] K. Niyogi and D. Marculescu. Speed and Voltage Selection for GALS Systems Based on Voltage/frequency Islands. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, 2005.

[31] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA*, 2007.

[32] `https://asc.llnl.gov/sequoia/benchmarks/#phloem`, 2008. ASC Sequoia Benchmark Codes.

[33] N.-O. Song, B.-J. Kwak, J. Song, and M. Miller. Enhancement of IEEE 802.11 Distributed Coordination Function With Exponential Increase Exponential Decrease Backoff Algorithm. In *Vehicular Technology Conference, 2003. VTC 2003-Spring. The 57th IEEE Semiannual*. IEEE, 2003.

[34] P. Tembey, A. Gavrilovska, and K. Schwan. A Case for Coordinated Resource Management in Heterogeneous Multicore Platforms. In *Workshop on the Interaction between Operating Systems and Architecture, WIOSCA*, 2010.

[35] C. Waldspurger. Memory Resource Management in the VMWare ESX Server. In *OSDI*, 2002.

[36] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner. Theoretical and Practical Limits of Dynamic Voltage Scaling. In *DAC*, 2004.