Helios: Heterogeneous Multiprocessing with Satellite Kernels

Edmund B. Nightingale
Microsoft Research

Orion Hodson
Microsoft Research

Ross McIlroy[†]
University of Glasgow, UK

Chris Hawblitzel
Microsoft Research

Galen Hunt
Microsoft Research

ABSTRACT

Helios is an operating system designed to simplify the task of writing, deploying, and tuning applications for heterogeneous platforms. Helios introduces *satellite kernels*, which export a single, uniform set of OS abstractions across CPUs of disparate architectures and performance characteristics. Access to I/O services such as file systems are made transparent via remote message passing, which extends a standard microkernel message-passing abstraction to a satellite kernel infrastructure. Helios retargets applications to available ISAs by compiling from an intermediate language. To simplify deploying and tuning application performance, Helios exposes an affinity metric to developers. Affinity provides a hint to the operating system about whether a process would benefit from executing on the same platform as a service it depends upon.

We developed satellite kernels for an XScale programmable I/O card and for cache-coherent NUMA architectures. We offloaded several applications and operating system components, often by changing only a single line of metadata. We show up to a 28% performance improvement by offloading tasks to the XScale I/O card. On a mail-server benchmark, we show a 39% improvement in performance by automatically splitting the application among multiple NUMA domains.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance

General Terms

Design, Management, Performance

Keywords

Operating systems, heterogeneous computing

[†]Work completed during an internship at Microsoft Research.

SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.

1. INTRODUCTION

Operating systems are designed for homogeneous hardware architectures. Within a machine, CPUs are treated as interchangeable parts. Each CPU is assumed to provide equivalent functionality, instruction throughput, and cache-coherent access to memory. At most, an operating system must contend with a cache-coherent non-uniform memory architecture (NUMA), which results in varying access times to different portions of main memory.

However, computing environments are no longer homogeneous. Programmable devices, such as GPUs and NICs, fragment the traditional model of computing by introducing "islands of computation" where developers can run arbitrary code to take advantage of device-specific features. For example, GPUs often provide high-performance vector processing, while a programmable NIC provides the opportunity to compute "close to the source" without wasting time communicating over a bus to a general purpose CPU. These devices are not cache-coherent with respect to general purpose CPUs, are programmed with unique instruction sets, and often have dramatically different performance characteristics.

Operating systems effectively ignore programmable devices by treating them no differently than traditional, non-programmable I/O devices. Therefore device drivers are the only available method of communicating with a programmable device. Unfortunately, the device driver interface was designed for pushing bits back and forth over a bus, rather than acting as an interface through which applications coordinate and execute. As a result, there often exists little or no support on a programmable device for once straightforward tasks such as accessing other I/O devices (e.g., writing to disk), debugging, or getting user input. A secondary problem is that drivers, which execute in a privileged space within the kernel, become ever more complicated as they take on new tasks such as executing application frameworks. For example, the NVIDIA graphics driver, which supports the CUDA runtime for programming GPUs, contains an entire JIT compiler.

Helios is an operating system designed to simplify the task of writing, deploying, and tuning applications for heterogeneous platforms. Helios introduces *satellite kernels*, which export a single, uniform set of OS abstractions across CPUs of disparate architectures and performance characteristics. Satellite kernels allow developers to write applications against familiar operating system APIs and abstractions. In addition, Helios extends satellite kernels to NUMA architectures, treating NUMA as a shared-nothing multiprocessor. Each NUMA *domain*, which consists of a set of CPUs and co-located memory, runs its own satellite kernel and independently manages its resources. By replicating kernel code and by making performance boundaries between NUMA domains explicit, Helios removes the kernel as a bottleneck to scaling up performance in large multiprocessor systems.

Satellite kernels are microkernels. Each satellite kernel is composed of a scheduler, a memory manager, a namespace manager, and code to coordinate communication between other kernels. All other traditional operating system drivers and services (e.g., a file system) execute as individual processes. The first satellite kernel to boot, called the *coordinator kernel*, discovers programmable devices and launches additional satellite kernels. Helios provides transparent access to services executing on satellite kernels by extending a traditional message-passing interface to include *remote message passing*. When applications or services communicate with each other on the same satellite kernel a fast, zero-copy, message-passing interface is used. However, if communication occurs between two different satellite kernels, then remote message passing automatically marshals messages between the kernels to facilitate communication. Since applications are written for a message-passing interface, no changes are required when an application is run on a programmable device.

In a heterogeneous environment, the placement of applications can have a drastic impact on performance. Therefore, Helios simplifies application deployment by exporting an *affinity metric* that is expressed over message-passing channels. A positive affinity provides a hint to the operating system that two components will benefit from fast message passing, and should execute on the same satellite kernel. A negative affinity suggests that the two components should execute on different satellite kernels. Helios uses affinity values to automatically make placement decisions when processes are started. For example, the Helios networking stack expresses a positive affinity for the channels used to communicate with a network device driver. When a programmable network adapter is present, the positive affinity between the networking stack and the driver executing on the adapter causes Helios to automatically offload the entire networking stack to the adapter. Offloading the networking stack does not require any changes to its source code. Affinity values are expressed as part of an application's XML metadata file, and can easily be changed by developers or system administrators to tune application performance or adapt an application to a new operating environment.

Helios uses a two-phase compilation strategy to contend with the many different configurations of programmable devices that may be available on a machine. Developers compile applications from source to an intermediate language. Once an application is installed, it is compiled down to the instruction set of each available processor architecture. An additional benefit of an intermediate language is that it can encapsulate multiple implementations of a particular feature tuned to different architectures. For example, the `Interlocked.CompareExchange` function requires the use of processor-specific assembly language instructions. Any process that uses the function has all supported versions shipped in the intermediate language; the appropriate version is then used when compiling the application to each available architecture.

We built Helios by modifying the Singularity [13] operating system to support satellite kernels, remote message passing, and affinity. We implemented support for satellite kernels on two different hardware platforms: an Intel XScale programmable PCI Express I/O card and cache-coherent NUMA architectures. We offloaded several operating system components, including a complete networking stack, a file system, and several applications to the XScale programmable device by adjusting the affinity values within application metadata. We improved the performance of two different applications by up to 28% through offloading. On a mail-server benchmark, we show a 39% improvement in performance by splitting the application among multiple NUMA domains.

We discuss the design goals for Helios in the next section and we describe the implementation in Section 3. Section 4 evaluates Helios, Section 5 discusses related work, and then we conclude.

2. DESIGN GOALS

We followed four design goals when we created an operating system for heterogeneous platforms. First, the operating system should efficiently export a single OS abstraction across different programmable devices. Second, inter-process communication that spans two different programmable devices should function no differently than IPC on a single device. Third, the operating system should provide mechanisms to simplify deploying and tuning applications. Fourth, it should provide a means to encapsulate the disparate architectures of multiple programmable devices.

2.1 Many Kernels: One Set of Abstractions

Exporting a single set of abstractions across many platforms simplifies writing applications for different programmable devices. Further, these abstractions must be exported *efficiently* to be useful. Therefore, when determining the composition of a satellite kernel, the following guidelines were followed:

- *Avoid unnecessary remote communication.* A design that requires frequent communication to remote programmable devices or NUMA domains (e.g., forwarding requests to a general purpose CPU) would impose a high performance penalty. Therefore, such communication should be invoked only when a request cannot be serviced locally.

- *Require minimal hardware primitives.* If a programmable device requires a feature-set that is too constrained (e.g., an MMU), then few devices will be able to run Helios. On the other hand, requiring too few primitives might force Helios to communicate with other devices to implement basic features (such as interrupts), which violates the previous guideline. Therefore, Helios should require a minimal set of hardware primitives while preserving the desire to do as much work as possible locally.

- *Require minimal hardware resources*. Programmable devices that provide slower CPUs and far less RAM than general purpose CPUs should not be prevented from running Helios.

- *Avoid unnecessary local IPC.* Local message-passing is slower than a system call. Therefore, resources private to a process (e.g., memory) should be managed by a satellite kernel and accessed via a system call. However, if a resource is shared (e.g., a NIC), than it should be controlled by a process that is accessible via message-passing, and therefore available to processes executing on other satellite kernels.

Satellite kernels, which form the basic primitive for managing programmable devices, were designed with each of these criteria in mind. First, satellite kernels minimize remote communication by initiating it only when communicating with the namespace or when transferring messages between kernels to implement remote message passing. All other communication is provided either via a system call, or through local message-passing.

Second, in addition to CPU and DRAM, satellite kernels require three basic hardware primitives: a timer, an interrupt controller, and the ability to catch an exception (i.e., trap). Without these primitives, Helios could not implement basic services, such as scheduling, directly on the programmable device. We believe these requirements are quite reasonable; although the current generation

of GPUs do not provide timers or interrupt controllers, Intel's next generation GPU (the Larrabee [29]) will provide all three hardware primitives required to run a satellite kernel. We expect these three primitives will appear on more programmable devices in the future, providing new platforms upon which to run Helios.

Third, satellite kernels have minimal hardware resource requirements. Helios runs with as little as 128 MB of RAM on a TI OMAP CPU running at 600 MHz. We believe Helios could run on as little as 32 MB of RAM and a few hundred MHz CPU with additional tuning. However, we have not yet tested Helios on a programmable device that was so resource constrained. A small resource footprint allows Helios to "scale-down" while still providing the benefits of local resource management and a single set of OS abstractions.

Finally, satellite kernels are designed to manage a very small number of private resources: memory and CPU cycles. Satellite kernels expose APIs for memory management, process management, and thread management. Satellite kernels also contain code to bootstrap communication with the namespace, but all other operating system services, including device drivers, execute as processes and use message passing for communication.

As a result of following these guidelines, satellite kernels provide a small and efficient platform for using programmable devices. Satellite kernels simplify developing applications for a heterogeneous platform by making the abstractions that are available on a general purpose CPU also available on a programmable device or a NUMA domain.

2.2 Transparent IPC

Our second design goal was to provide transparent, unified interprocess communication, independent of where a process or service executes. Two problems must be solved to meet this design goal. First, an application must be able to name another application or OS service that is executing somewhere on a machine. Second, applications written to work when executing on the same kernel should continue to work if the two ends of a message-passing channel are executing on two different satellite kernels.

Helios meets the first requirement of this design goal by exporting a single, unified, *namespace*. The namespace serves multiple purposes. Drivers use it to advertise the availability of hardware on a machine and satellite kernels advertise their platform type, presence, and any available kernel services (such as performance counters) that are available via message passing. OS services (e.g., a file system) and arbitrary processes use the namespace to expose their own functionality. For the sake of brevity, we refer to any process that advertises itself in the namespace as a service. The Helios namespace allows an application to depend upon a service without knowing ahead of time where the service will execute.

Helios meets the second requirement of this design goal by implementing both local message passing (LMP) and remote message passing (RMP) channels beneath the same inter-process communication abstraction. If a service is local, then the fast, zero-copy, message-passing mechanism from Singularity [6] is used. However, if the service is located remotely, Helios transparently marshals messages between an application and the remote service. To the application, execution proceeds as if the service were co-located on the same satellite kernel.

Remote message passing allows an application to execute on an arbitrary platform while still communicating with other services upon which it may depend. The "network" of memory and PCI buses that connect satellite kernels is very robust compared to traditional LAN or WAN networks. Therefore traditional distributed systems protocols (e.g., TCP) are not required. Instead, each satellite kernel implements RMP using the communication primitives available to it. For example, the satellite kernel that executes on the XScale I/O card uses an ADMA (asynchronous direct memory transfer) controller to implement RMP, while a satellite kernel that executes on a NUMA domain uses `memcpy` instead.

One might think that routing among many kernels would be complex. In fact, routing is quite simple since there are no multi-hop routing paths between satellite kernels. When the coordinator kernel starts a satellite kernel it establishes a point-to-point connection with all other kernels in the system. This approach has worked well for a small number of satellite kernels, but if in the future machines have a very large number of programmable devices or NUMA domains, an alternative network topology may be required.

The namespace and RMP allow applications that depend upon sharing an I/O service to continue to function as if the service were executing on the same satellite kernel as the processes using it. For example, two processes that share state through a file system can continue to do so, since an incoming connection to a file system looks the same whether RMP or LMP is used.

2.3 Simplify Deployment and Tuning

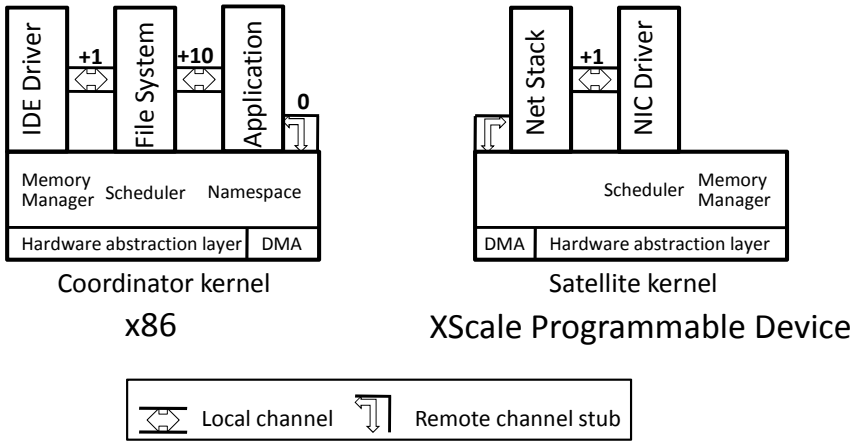
The third design goal for Helios is to simplify application deployment and tuning on heterogeneous platforms. A heterogeneous platform adds three new constraints that make tuning process performance more difficult and therefore makes the task of determining where a process *initially* executes important. First, heterogeneous architectures make moving processes between devices quite difficult; therefore the most practical strategy is to choose a device, put a process on it, and leave it alone. Second, the lack of cache coherence results in remote as well as local message passing. If an application is written expecting fast, local message passing, and is executed on a device where its communication to other processes is remote, then performance could suffer dramatically. Finally, an application might *prefer* a device with certain performance characteristics if it is available.

Helios meets this design goal by allowing processes to specify their *affinity* for other processes. An affinity value is written in the metadata of a process, and is expressed as a positive or negative integer on a message-passing channel. When a process expresses a *positive affinity* value it represents a preference to execute on the same satellite kernel as another currently executing process. Positive affinity hints that two processes will benefit from communicating over a fast, zero-copy message-passing channel. Processes may also use positive affinity to express a *platform preference* by listing a positive affinity value for a satellite kernel executing on a particular programmable device.

When a process expresses a *negative affinity* value it represents a preference to execute on a different satellite kernel than another currently executing process. Negative affinity hints that two processes will benefit from non-interference and is therefore a means of avoiding resource contention. Processes may also use *self-reference affinity*, where a process expresses negative affinity for itself, to ensure multiple instances of a process execute independently.

Since affinity is expressed via metadata, it is very easy to tune the performance of a set of processes by changing affinity values. Developers can readily experiment with different placement policies, and system administrators or users can add or change placement policies to adapt to environments not considered by developers (e.g., new devices). In our experiments, we use affinity to quickly determine policies that maximize performance.

Helios does not strive to optimally map an arbitrary graph of processes to a graph of active processors on a system. Instead, Helios attempts to strike a balance between practicality and optimality by choosing the satellite kernel where a process will execute based



This figure shows a general overview of the architecture of the Helios operating system executing on a machine with one general purpose CPU and a single programmable device. Applications co-located on the same kernel communicate via a fast, statically verified, message-passing interface. Applications on different kernels communicate via remote message-passing channels, which transparently marshal and send messages between satellite kernels. The numbers above the channels are affinity values provided by applications to the operating system. Helios uses affinity values as hints to determine where a process should execute

Figure 1: Helios architecture

upon the affinity values a process expresses and the location of the other processes with which it wishes to communicate. We note that affinity does not prevent a process from harming its own performance. The values are only hints, and we assume that the fact that they are easily modified will allow poorly designed affinity policies to be easily remedied.

2.4 Encapsulate Disparate Architectures

The last design goal for Helios is to efficiently encapsulate a process that may run on multiple platforms while preserving the opportunity to exploit platform-specific features. Helios achieves this design goal by using a two-phase compilation strategy. Applications are first compiled into the common intermediate language (CIL), which is the byte-code of the .NET platform. We expect applications to ship as CIL binaries. The second compilation phase translates the intermediate language into the ISA of a particular processor. Currently, all Helios applications are written in Sing#, compiled into CIL, and then compiled again into different ISAs using a derivative of the Marmot [7] compiler called Bartok.

As an alternative, one could ship fat binaries, which would contain a version of the application for each available platform it supports. Packaging an application using CIL has two advantages over fat binaries. First, a developer that uses fat binaries must choose ahead of time which platforms to support and fat binaries will grow in size as the number of ISAs supported by an application increases. Second, CIL already contains infrastructure for efficiently supporting multiple versions of a method. This feature allows an application to take advantage of device-specific features if they are present, while still functioning if these features are missing. For example, an application could have one process that executes large amounts of vector math. If a GPU were present, the calculations would be accelerated, but if it were not, the process would still function using a general purpose CPU. Helios already uses this functionality in libraries that support applications, such as code to implement `InterLocked.CompareExchange` and code that implements an `Atomic` primitive. The two-phase compilation strategy also means that an older application could run on a new programmable device without modification, as long as a compiler exists to translate from CIL to the new instruction set.

3. IMPLEMENTATION

Figure 1 provides an overview of Helios running on a general purpose CPU and an XScale programmable device. Each kernel runs its own scheduler and memory manager, while the coordinator kernel also manages the namespace, which is available to all satellite kernels via remote message passing. In the example, an application has a local message-passing channel to the file system, and a remote message-passing channel to the networking stack, which is executing on a programmable NIC. The numbers above each channel describe the affinity the application or service has assigned to the channel. Since the file system and networking stack have positive affinities with their device drivers, they have been co-located with each driver in a separate kernel. The application has expressed positive affinity to the file system and no preference to the networking stack, therefore the application runs on the same kernel as the file system.

3.1 Singularity Background

Helios was built by modifying the Singularity RDK [22] to support satellite kernels, remote message passing, and affinity. We begin by providing a brief overview of Singularity.

Singularity is an operating system written almost entirely in the Sing# [6] derivative of the C# programming language. Applications in Singularity are composed of one or more processes, each of which is composed of one or more threads. Threads share a single address space, while processes are isolated from each other and can only communicate via message passing. Applications written for Singularity are type and memory safe. The operating system relies on software isolation to protect processes from each other and therefore all processes run in the same address space at the highest privilege level (ring 0 on an x86 architecture).

Singularity supports a threading model similar to POSIX, where threads have contexts that are visible to and scheduled by the operating system. Further, threads have access to all the usual synchronization primitives available in C#. Since all processes execute in the same address space and rely on software isolation, context switches between processes are no more costly than context switches between threads. Further, Singularity does not require an MMU or a virtual address space. Virtual memory is, however, cur-

rently used as a convenient method of catching a null pointer dereference by trapping on accesses to page 0.

The kernel exports a system-call interface called the kernel ABI. It exposes calls to manage memory, processes, threads, the namespace, message passing, and calls for drivers to gain access to special hardware resources such as I/O ports. All other OS services execute as processes and are accessed via message passing. Further, the *only* method of inter-process communication available to a process is message passing; Singularity does not support shared memory, signals, pipes, FIFOs, System V shared memory, or any other traditional inter-process communication mechanism outside of message passing. Processes can communicate through I/O devices (e.g., a file system), but there is no explicit support for such communication.

Message passing in Singularity is designed to be fast. When a message is sent, only a pointer to the message metadata is passed to the receiving process. When the message is received, it can be used immediately (i.e., without making a copy) by the receiving process.

Zero-copy message passing is implemented using the Sing# programming language, compiler, and runtime. Sing# is used to write *contracts*, which describe what data will be passed across the two ends (endpoints) of a message-passing channel. The Sing# compiler then uses the contracts to generate code that implements the machinery to send and receive data on the channel. The contract ultimately defines a state machine, and the legal sequences of messages that can be sent or received within any given state. The contract also determines which party can send or receive a message (or response) at any given point in the state space.

Each Singularity process has two heaps. One heap is the .NET garbage collected heap, and the other heap, called the exchange heap, is used for passing messages and data over channels. Sing# uses compile-time analysis to guarantee that processes will not have dangling pointers, race conditions, or memory leaks related to exchange heap data. Sing# also guarantees that only one thread may write to an endpoint at any time, thus preventing any race conditions on the use of endpoints.

Finally, each process, service, and driver has a separate manifest that describes the libraries, channels, and parameters required for execution. Helios also uses the manifest to express affinity policies.

3.2 Helios Programming Model

Singularity provides fast message passing and context-switching to promote a modular programming model; an application is composed of one or more processes that communicate via message passing. Singularity is designed for a homogeneous, symmetric computing environment. Any process (or thread) can run on any CPU, and all CPUs are assumed to be equivalent.

Helios builds upon this model by running additional satellite kernels on programmable devices and NUMA domains. Since message passing is the only means for two processes to communicate, support for remote message passing ensures that processes written for Singularity continue to operate even if they are executing on different satellite kernels. Each satellite kernel manages its resources independently; resources from one kernel (e.g., CPU cycles or memory) are not available to processes executing on other satellite kernels. Further, a process cannot exist on more than one satellite kernel and therefore the threads within a process must all execute on the same satellite kernel. However, since applications are composed of one or more processes, applications do span multiple satellite kernels. A satellite kernel is fully multi-processor capable, and if a programmable device or NUMA domain is composed of more than one CPU, then multiple threads will run concurrently.

Programmable devices are not cache coherent with respect to other programmable devices or general purpose CPUs. The Helios programming model makes the boundaries between devices explicit by disallowing a process from spanning multiple kernels.

However, on a NUMA machine, NUMA domains are cache coherent with other NUMA domains on a system. Most operating systems allow processes to span multiple NUMA domains; this functionality comes at the cost of complexity within the operating system and complexity in the programming model for applications. In a NUMA architecture processors have access to memory that is local and memory that is remote. Local memory accesses are much faster than accessing memory in other NUMA domains. In our tests, we have found local memory have access latencies that are 38% lower than remote memory.

In a monolithic operating system, the OS and applications must be carefully tuned to touch local rather than remote memory. Further, as the number of CPUs increases, locks can quickly become bottlenecks and must be refactored to improve performance. Helios takes an alternative approach by treating NUMA domains as high-speed programmable devices; it therefore executes a satellite kernel in each NUMA domain. Processes are not allowed to span NUMA domains. In return for this restriction, memory accesses are always local and as the number of available processors increases, the kernel does not become a bottleneck because multiple kernels run in parallel. However, limiting processes to a single NUMA domain could hamper large, multi-threaded processes that are not easily decomposed into separate processes. Such processes could scale-out across NUMA domains if Helios supported a programming model similar to Hive [4], which allows threads to use CPUs and access memory located on remote NUMA domains.

3.3 XScale Satellite Kernel

The first platform to support satellite kernels is an IOP348 RAID development board. The board communicates with the host over an 8-lane PCI-Express interface and features a 1.2 GHz XScale (ARM v5TE) processor with its own chip interrupt controller, programmable timer, and clock. It also has a gigabit Ethernet controller, a PCI-X slot, a SAS controller, and 256 MB of DRAM.

To implement the satellite kernel, we wrote our own bootloader that initializes the card and then presents itself on the PCI-Express bus of the host PC. The bootloader first waits for the coordinator kernel to load on the host PC. The coordinator kernel then detects the device and loads a driver, which copies the satellite kernel image to the card and allows it to start executing. The satellite kernel is aware that it is not the coordinator kernel, and checks within the code ensure it does not, for example, start its own namespace and attempt to export it to other satellite kernels. The satellite kernel then requests any drivers that it needs from the file system via the namespace.

The satellite kernel runs the identical C# code-base as the kernel on a general purpose CPU (but compiled to an ARM v5 vs. an x86 ISA). Code that was written in assembly (e.g., code that implements a context switch) must be ported to the native assembly language of the programmable device. The XScale satellite kernel contains about 3,000 lines of ARM assembly language, and an additional 13,000 lines of ARM assembly is contained in the C runtime library.

Once the XScale I/O card completes booting, it starts a controller process and registers itself in the namespace of the coordinator kernel. This registration process is the mechanism that allows the coordinator kernel to launch applications on the XScale I/O card.

3.4 NUMA Satellite Kernel

NUMA machines comprise the second platform to support satellite kernels. We have implemented satellite kernels on a dual-socket AMD hyper-transport architecture. Each socket has its own NUMA domain, and each socket is populated with a dual-core processor. Each processor has an on-chip programmable interrupt timer that it can use for scheduling and also has access to shared clock resources. Device interrupts are received by I/O interrupt controllers (I/O APICS) that are configured to steer all device interrupts to the processors running the coordinator kernel. This avoids the issue of sharing access to the I/O APICS between NUMA domains. Although routing device interrupts to the coordinator kernel restricts the location of device drivers, it does not limit the location of the services that use them. For example, we have run a mounted FAT file system on a satellite kernel while the disk driver is executing on the coordinator kernel. We rely on the safety properties of C# to prevent wild writes and cross-domain accesses, and the memory manager of each satellite kernel is aware of only the RAM local to its domain.

When Helios boots, its bootloader first examines the ACPI system resource affinity table (SRAT) to see whether or not the machine has a NUMA architecture. If a NUMA architecture exists, the Helios boot loader enumerates the different NUMA domains, recording the physical addresses of the memory local to that domain and the processor identifiers associated with each domain. The bootloader then loads the coordinator kernel in the first NUMA domain.

Once the coordinator kernel has booted all of its processors the CPU with the lowest processor identifier in the next domain enters the bootloader and is passed the address range of its own local memory. Helios makes a copy of the page table and other low-level boot data structures into the local memory of the domain and then boots the satellite kernel image. Finally, as part of the implementation of remote channels, the boot loader takes a small part of the address space of each satellite kernel and marks it as an area available for passing messages. Once a satellite kernel within a NUMA domain completes its boot process it registers itself in the namespace and waits for work.

3.5 Namespace

Services are made available through a namespace, similar in spirit to the namespace provided by the Plan9 [25] operating system. Services register in the namespace so that applications can find them. A process passes one end of a pair of endpoints to the service, which uses the received endpoint for communication. When a message passing channel is established, the kernel connects each end of a message passing channel into the address space of the service and of the process that initiated the connection.

Services may execute on any satellite kernel, and Helios does not put any restrictions on the number or types of services that execute. For example, the boot file system is available at "/fs," while other file systems may be advertised at other locations in the namespace. A service need not be tied explicitly to a hardware device. For example, Helios provides a service that decompresses PNG images; the application registers itself in the namespace, and awaits connections from applications wishing to use the service. Traditional I/O devices (i.e., those that cannot support a satellite kernel), are represented by a single driver that communicates with one or more applications or services. For example, each non-programmable NIC has a single driver, registered as 'NICx,' where x is a monotonically increasing number, and the networking stack has multiple components (ARP, UDP, TCP, routing) that communicate with each NIC driver.

The coordinator kernel manages the namespace, and it serves three different roles for applications. First, registration allows an application to create an entry in the namespace, which lets other applications know a service is running and awaiting connections. During registration, the coordinator kernel checks to make sure that the entry in the namespace is available; the coordinator kernel then sets up a message passing connection between the namespace entry and the service awaiting connections. Second, binding allows a process to request that a message passing channel be established with a service advertised in the namespace. When a bind request is received, the coordinator kernel forwards the message to the service, which then establishes a new message passing channel directly with the process that requested a connection. Finally, entries in the namespace are removed either when the service advertising in the namespace requests it, or when the message passing channel into the namespace channel is closed.

The namespace is the only component of Helios that relies on centralized control. This was done for the sake of simplicity and because communication with the namespace occurs only when a process is started and when it establishes connections to other services. However, the centralized namespace does mean that messages sometimes travel further than they need to when initially establishing a connection between a process and a service. For example, if a process attempts to establish a connection to a service that is executing on the same satellite kernel it will first contact the namespace remotely. The namespace will forward the request, via a second remote message-passing channel, back to the service. When the service receives a request for a connection, the remote message-passing runtime recognizes the service and requesting process are executing on the same satellite kernel, and the connection is converted to a local message-passing channel. Therefore, in this particular case, two remote message passing channels are traversed in order to establish a local message passing channel between two processes that are executing together on the same satellite kernel.

3.6 Remote Message Passing

Implementing message passing across remote Singularity channels is more challenging than implementing message passing across local Singularity channels. As described in section 3.1, two processes communicating over a local channel share the same address space, exchange heap, and thread scheduler, so that the sender can write pointers directly into the receiver's endpoint without copying the pointed-to data. A remote channel, on the other hand, connects processes running on different satellite kernels, where each satellite kernel has its own exchange heap, its own thread scheduler, and its own address space. Therefore, the remote channel implementation must copy data between the different exchange heaps, and must use non-local data transfer and signaling mechanisms.

To manage this copying and signaling while preserving the original Singularity channel programming interface, Helios creates two *shadow endpoints* B' and A' for each remote channel between ordinary endpoints A and B. Endpoint B', which resides on the same satellite kernel as endpoint A, shadows the state of the remote endpoint B. (Similarly, endpoint A' shadows the state of endpoint A.) When a sender wants to send a message from endpoint A to endpoint B, it first sends the message locally to the shadow endpoint B'. Endpoint B' then encapsulates the message type and pointer information into a message metadata structure and places it into a FIFO queue connected to the receiver's satellite kernel. After the receiver dequeues this message metadata structure from the FIFO, the receiver pulls the updated state from endpoint B' to endpoint B and copies the exchange heap data pointed to by the message metadata structure, allocating memory to hold the exchange heap

data as necessary. Note that the exchange heap data is copied only once, directly from the sender's exchange heap into the receiver's exchange heap.

The lower-level details of the message passing depend on the heterogeneous system's underlying communication mechanisms. On a NUMA kernel, each kernel's page table maps all memory available to the NUMA machine (although the memory manager is aware of only the memory available locally within its own NUMA domain). The inbound and outbound message FIFOs are shared between domains, with one kernel pushing onto a FIFO and the other pulling from it. A dedicated thread in each kernel polls for inbound messages and handles the message reception process. Since a NUMA satellite kernel can translate the addresses provided in a message from another NUMA satellite kernel into addresses mapped into its address space, it can directly copy the data from a remote domain into its own.

On the XScale, our initial implementation used shared memory to pass message metadata structures and `memcpy` to pass message data between the PC host and the XScale card. However, direct memory accesses across the PCI-E bus stall the initiating CPU, and we therefore found that up to 50% of CPU time was spent in `memcpy` routines. To reduce direct shared memory access and to allow overlap between I/O and computation, we used the XScale's asynchronous DMA (ADMA) engine to transfer metadata and data. This reduced the number of CPU cycles used during a data transfer by an order of magnitude. Since the data transfer is asynchronous, we needed to determine when the transfer completes without resorting to polling. We therefore programmed the ADMA controller so that, after transferring the data to the receiver, it initiates a DMA from the memory of the XScale to a register *in the XScale's own messaging unit*. This DMA triggers an interrupt to the x86 when the transfer is complete, which notifies the satellite kernel on the x86 that it is safe to free the memory associated with the transferred data.

We further improved performance by adding a new allocator method that is DMA-aware. Normally, the allocator zeroes out pages to prevent information written by other processes from leaking. Since pages are immediately overwritten during a DMA, the DMA-aware allocator skips the step that zeroes out pages, which saves CPU cycles.

3.7 Affinity

3.7.1 Expressing Affinity Policies

Affinity policies are expressed in a manifest that accompanies a process executable. The manifest is an XML file that is automatically generated when a process is compiled into CIL, and it contains a list of message-passing channels the process depends upon. Message-passing channels are expressed as the names of Sing# contracts, which were used to generate the code that executes the details of message passing within each process. Although the initial manifest is generated by the Sing# compiler, the XML file is designed to be human readable and edited without necessarily making any changes to the binary that accompanies it. In fact, since affinity has no impact on a process once it begins execution, processes that are not developed with affinity in mind will continue to operate correctly even if affinity values are added to a manifest by a third party without any changes to the program binary.

Figure 2 shows a portion of a manifest from the Helios TCP test suite. Every process has at least two message-passing channels, stdin and stdout, connected to it by default. In Figure 2, stdin is represented in the manifest and has an affinity value of 0, which means that the process does not care where it executes in relation

```
<?xml version="1.0" encoding="utf-8"?>
<application name="TcpTest" runtime="Full">
<endpoints>
   <inputPipe id="0"  affinity="0"
    contractName="PipeContract"/>

   <endpoint id="2"  affinity="-1"
    contractName="TcpContract"/>
</endpoints>
```

Figure 2: Example process manifest with affinity

to the stdin process. The second message-passing channel listed in the example is a connection to the Helios TCP service. The process lists an affinity value of -1, which means the process expresses a wish to execute on a satellite kernel that differs from the kernel that is servicing TCP packets and sockets. The placement of the application is changed simply by changing the affinity entries in the manifest.

3.7.2 Types of Affinity

Affinity is either a positive or negative value. A positive affinity value denotes that a process prefers local message passing and therefore it will benefit from a traditional, zero-copy, message-passing channel. A negative affinity value expresses an interest in non-interference. A process using negative affinity will benefit from executing in isolation from other processes with which it communicates. The default affinity value, which is zero, indicates no placement preference for the application.

Affinity could be expressed as two different values. However, since preferences for local message passing and non-interference are mutually exclusive, Helios uses a single integer value to represent one preference over the other. Beyond its simplicity, affinity allows developers to express the dependencies that are important to maximize the performance of an application without having any knowledge of the topology of a particular machine.

Positive affinity is generally used to describe two different types of dependencies. First, positive affinity represents a tight coupling between two processes. These tightly-coupled relationships are sometimes between a driver and a process that uses it. For example, a NIC driver and a networking stack are tightly coupled. Receiving packets and demultiplexing them is a performance-sensitive task. Therefore the Helios networking stack expresses positive affinity with networking drivers to ensure that the networking stack and the driver will always execute on the same device. In this example, the driver exists on a device because the hardware requires it, and therefore the networking stack wanted to express the policy that "wherever a NIC driver executes, the networking stack should execute as well." Positive affinity is not limited to drivers, but is used to express a preference for fast message passing between any two processes that wish to communicate with each other.

The second use of positive affinity captures a platform preference. Platform preferences do not require any additional mechanism. Instead, they are expressed as preferences for the message-passing channels to other satellite kernels, which are used to launch processes remotely. The name of the service describes its platform. For example, a satellite kernel on a GPU may advertise itself as a "Vector CPU" while a typical x86 processor is advertised as an "Out-of-order x86." A process with a preference for a particular set of platforms can express a range of affinity values. For example, a process may express a higher affinity value for a GPU than an x86 processor, so that when both are available one will be chosen over

```
SelectSatelliteKernel()
{
  if platform affinity {
   find max affinity platform with at least 1 kernel
   keep only kernels equal to max platform affinity
   if number of kernels is 1 return kernel
  }

  if positive affinity {
   for each remaining kernel
    sum positive affinity of each service

   keep only kernels with max positive affinity
   if number of kernels is 1 return kernel
  }

  if negative affinity {
   for each remaining kernel
    sum negative affinity of each service

   keep only kernels with min negative affinity
   if number of kernels is 1 return kernel
  }

  return kernel with lowest CPU utilization
}
```

Figure 3: Process placement pseudocode

the other. Helios uses positive platform affinity values to cull the list of possible satellite kernels that are eligible to host a particular process.

Negative affinity allows a process to express a policy of non-interference with another process. Negative affinity is often used as a means of avoiding resource contention. For example, two CPU bound processes might want to maximize performance by ensuring they do not execute on the same satellite kernel. Alternatively, a process might want to ensure it does not suffer from hardware interrupts serviced by a particular driver.

By itself, negative affinity provides no guidance other than to attempt to avoid certain satellite kernels. Therefore, negative affinity may be combined with a positive platform affinity to guarantee a performance threshold. For example, a process with a negative affinity for another process with which it communicates and a positive affinity for any satellite kernel executing on an out-of-order x86 CPU ensures that it will execute in isolation from the other process only if there are other satellite kernels executing on high throughput x86 processors. Otherwise, the two processes will execute on the same satellite kernel. This policy prevents a process from being offloaded onto a CPU that is orders of magnitude slower than the developer intended.

There is a second use of negative affinity called *self-reference affinity*. If a process can scale-out its performance by running multiple copies of itself on different devices or NUMA domains, it can reference its own service and place a negative affinity on the communication channel it advertises in the namespace. When additional copies are invoked, Helios will ensure they run independently on different satellite kernels.

3.7.3 Turning Policies into Actions

Helios processes the affinity values within a manifest to choose a satellite kernel where a process will start. Affinity values are prioritized first by platform affinities, then by other positive affinities, and finally by negative affinities. CPU utilization acts as a tie breaker if more than one kernel meets the criteria expressed in the manifest.

Helios uses a three-pass iterative algorithm, shown in Figure 3, when making a placement decision. Helios begins by processing affinity values to kernel control channels, which each represent a particular platform. We assume these channels will be standardized by convention and will therefore be easily identifiable. If preferences for platforms exist, then Helios starts with the platform with the highest affinity value and searches for satellite kernels with matching platforms advertised in the namespace. If none exist, than it moves to the next preferred platform. If only one kernel is available for a particular platform, then the process is complete and the process is started. If platforms are preferred and no kernels are available, then Helios returns an error. However, if no platform is preferred, then all satellite kernels are kept. If multiple satellite kernels for the preferred platform are available, then only those kernels are used in the second step of the algorithm.

In the second step, a tally of the total positive affinity is kept on behalf of each remaining satellite kernel. The total positive affinity for each satellite kernel is calculated by summing the affinity values for each service, specified by the manifest, which is executing on that satellite kernel. Helios then selects the satellite kernel(s) with the maximum affinity sum. If a single satellite kernel remains after this step, then the kernel is returned. If multiple satellite kernels remain, either because there were multiple kernels with the same maximum sum or because positive affinity was not specified, then negative affinity values are processed. The same algorithm is applied to negative affinity values as was used with positive affinity values. If after processing negative affinity values there are multiple satellite kernels available, then Helios selects the satellite kernel with the lowest total CPU load.

As can be seen, the algorithm prioritizes positive affinity over negative affinity by processing positive affinity values first. Alternatively, positive and negative affinities could be summed. We chose to use a priority approach because we found it easier to reason about where a process would run given different topologies. For example, we knew that a platform preference would always be honored no matter what other affinity values were present.

Affinity provides a simple mechanism for placing processes on satellite kernels. One could imagine more complex mechanisms for determining where a process should execute. For example, in addition to affinities, application manifests could specify CPU resource requirements, so that the placement mechanism could try to optimally allocate CPU time among applications. However, determining resource requirements ahead of time is more difficult for application developers than specifying affinities, and if the requirements are known exactly, the placement problem is a bin-packing problem, which is NP-hard. Alternatively, the system could dynamically measure CPU and channel usage, and use the measurements to influence placement policy. While such dynamic placement is common on homogeneous systems, it is much more difficult to implement on heterogeneous systems where processes cannot easily migrate between processors. For heterogeneous systems, affinity strikes a practical balance between simplicity and optimality.

3.7.4 A Multi-Process Example

As an example of a multi-process application that takes advantage of affinity, consider the Helios mail server application that is composed of an SMTP front-end application, an antivirus service, and a storage service. The SMTP server has message-passing channels to the TCP service, the antivirus service, and the storage service. The storage service has additional channels to an instance of

the FAT32 file system. All of the components of the application are multithreaded, and each process also has message-passing channels to stdin and stdout. The SMTP server processes incoming SMTP commands; when a mail message is sent through the server, it is sent to the storage service. The storage service sends the email to the antivirus service to be scanned. If the antivirus service finds the email is clean, than the storage service writes the email into an appropriate mailbox. The antivirus process is CPU and memory bound, since it checks messages against an in-memory database of virus signatures. The SMTP server is I/O-bound to the network, and the storage service is I/O-bound to disk. Therefore, the antivirus service expresses a negative affinity with the mailstore, so that it has a maximum number of CPU cycles available to it, and is not interrupted by hardware interrupts either from the disk or the network. On the other hand, the mail service expresses a positive affinity with the file system, and the SMTP service expresses no preference with regards to the networking stack. Thus, while many components are involved in the mail server application, there are few affinity values that must be expressed to maximize performance. The mail server benchmark results are covered in detail in Section 4.5.2.

4. EVALUATION

Our evaluation sets out to answer the following questions:

- Does Helios make it easier to use programmable devices?
- Can offloading with satellite kernels improve performance?
- Does kernel isolation improve performance on NUMA architectures?
- Does Helios benefit applications executing on NUMA architectures?

4.1 Methodology

We evaluated Helios on two different platforms. The first platform is an Intel Core2 Duo processor running at 2.66 GHz with 2 GB of RAM. The programmable device is a hardware RAID development board based on Intel XScale IOP348 processor running at 1.2GHz with 256 MB of RAM, and featuring an onboard Intel Gb Ethernet adapter. The development board is attached to the host by an 8-lane PCI Express interface. The second platform is a dual dual-core (2 chips, 2 cores per chip) motherboard where each core is a 2 GHz Opteron processor. Each chip runs within its own NUMA domain, and each NUMA domain has 1 GB of RAM, for a total of 2 GB. The machine runs with an Intel Gb PCI Ethernet card and a 200 GB hard drive. When we run networking experiments, machines are connected with a gigabit Ethernet switch. Performance was measured by using the total number of CPU cycles that elapsed during an experiment; the cycle-count was then converted into standard units of time.

When we run experiments evaluating the benefits of satellite kernels on NUMA architectures we run in two configurations. First, a configuration that uses a non-NUMA aware version of Helios with BIOS interleaved-memory turned on, which interleaves memory on 4 KB page boundaries. This version runs a single satellite kernel across all processors. Second, a NUMA-aware version of Helios (no interleaved memory) that runs the coordinator kernel in NUMA domain 0, and a satellite kernel in NUMA domain 1. Each kernel executes on the two processors native to its NUMA domain. We measured the raw memory latency of each type of memory and found that L2 cache misses to remote memory were 38% slower than local memory, and accesses to interleaved memory were on average 19% slower than accesses to local memory.

Name	LOC	LOC changed	LOM changed
Networking stack	9600	0	1
FAT32 file system	14200	0	1
TCP test harness	300	5	1
Disk indexing tool	900	0	1
Network driver	1700	0	0
Mail benchmark	2700	0	3
Web Server	1850	0	1

This table shows several example applications and services that were run both on an x86 host and an XScale (ARM) programmable device. All applications had lines within their manifests (LOM) modified to add affinity settings to channels. All applications were written originally for Singularity, without programmable devices in mind. Most worked without modification using satellite kernels.

Figure 4: Changes required to offload applications and services

4.2 Using Affinity to set Placement Policies

To see how well satellite kernels, remote channels, affinity and our 2-stage compilation strategy worked, we took a number of applications originally written for Singularity (i.e., with no thought towards offloading to another architecture) and added affinity values to each application's manifest. Once we arranged for our build system to compile to both ISAs, all of our examples ran without modification except for one: our TCP testing harness used floating point code to calculate throughput numbers. Our programmable device does not have a floating point unit, and we had not yet added floating point emulation code to the kernel. As a stop-gap measure we changed the way floating-point code was calculated. Later, we added floating-point emulation code to the satellite kernel to ensure other applications could run unmodified.

We had the same networking chip set on both the programmable device and on a non-programmable networking card. The networking driver for this chip, which was written for Singularity, worked without modification once compiled to ARM byte-code. Further, our networking stack, which supports TCP, IP, UDP, DHCP, and ARP protocols, worked without modification on the programmable device by adding a positive affinity value between it and the networking driver.

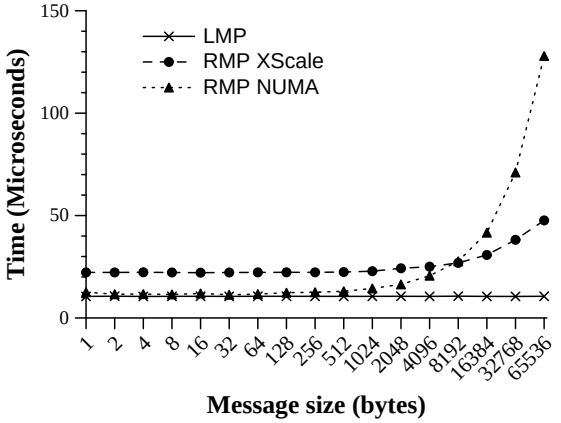
4.3 Message-passing Microbenchmark

Our next benchmark measured the cost of remote (RMP) and local (LMP) message passing on the two platforms that support satellite kernels. We ran SingBench, which was originally designed to benchmark the performance of Singularity message passing, using local message passing and remote message passing both to a satellite kernel on a NUMA domain and a satellite kernel on the XScale I/O card. The benchmark forks a child process and then it measures the time to send and receive a message of a certain size.

The results are presented in Figure 5. The x-axis shows the amount of data sent one-way during the test, and the y-axis shows the time in microseconds to send and receive a single message. Since no copies are made during LMP, the time is constant as the size of the message grows. RMP on NUMA grows with the size of the message, while RMP on XScale is bound mainly by the time to program the ADMA controller. Therefore, once messages are greater than 32 KB in size, message passing over the PCI-E bus is more efficient than message passing between two NUMA domains.

4.4 Benefits of XScale Offloading

We next examined the benefits of offloading different components of Helios onto the XScale I/O card.



This graph shows the difference in time to communicate over a local message passing channel and a remote message passing channel to the XScale I/O card and a NUMA domain. Results are the mean of 20 runs. Note the x-axis is a log scale.

Figure 5: Message passing microbenchmark

4.4.1 Netstack Subsystem Offload

We were interested to see whether we could improve the performance of an application by offloading a dependent operating system component onto the XScale I/O card. We took an existing Singularity service that decompresses PNG images and made it available via a TCP/IP connection. We then ran two versions of it, one where everything ran on a single x86 CPU, and one where the NIC driver and networking stack (ARP, DHCP, IP, TCP) were offloaded onto the XScale programmable device.

We then ran the PNG decompression service and connected 8 clients (chosen by the number required to saturate the x86 CPU) over the network and then sent PNGs of varying sizes as fast as possible to be decompressed. The results are shown in Figure 6. The first column shows the size of the PNG sent over the network. Larger PNGs took longer to decompress, and therefore slowed down overall performance. The second and third columns show the average uploads per second processed by the PNG decompression service when the netstack and NIC driver ran on the x86 CPU, and when they were offloaded to the XScale I/O card. In general, the benefits of offloading tracked the amount of CPU freed up by offloading the networking stack and NIC driver, which can be seen in the average speedup in the fourth column. Finally, the fifth column shows the reduction in interrupts to the x86 processor when the netstack and NIC driver were offloaded. The reduction occurred because the XScale CPU absorbed all of the interrupts generated by the NIC. Since the x86 processor took fewer interrupts, the PNG decompression service operated more efficiently and therefore its performance improved.

4.4.2 Storage Subsystem Offload

Since the XScale I/O card had a PCI-X slot, we decided to emulate a programmable disk by inserting a PCI SATA card to test the benefits of offloading portions of the Helios storage subsystem. Unlike the netstack offloading benchmark, where the partitioning of work was straightforward, we were unsure which portions of the storage subsystem should be offloaded to improve performance.

We therefore used the PostMark benchmark [17], which emulates the small file workload of a mail/news server. We ported PostMark to Singularity and enhanced it to run with a configurable number of driver threads to increase the load on the system. We also added code to synchronize the filesystem at configurable intervals, since this functionality would be typically used in a mail/news server. We synchronized the file system once per PostMark transaction to exercise the file system and IDE driver continuously.

PNG Size	x86	+ XScale	Speedup	% cswi
28 KB	161	171	106%	54%
92 KB	55	61	112%	68%
150 KB	35	38	110%	65%
290 KB	19	21	110%	53%

This benchmark shows the performance, in uploads per second, the average speedup, and the reduction in the number of context switches (cswi) by offloading the networking stack to a programmable device when running a networked PNG decompression service. All results were the mean of 6 trials and 95% confidence intervals were within 1% of the mean.

Figure 6: Netstack offload benchmark

PostMark	FatFS	IdeDriver	C1 (S)	C2 (S)
x86	x86	x86	26.2	4.1
x86	x86	XScale	15.5	8.0
x86	XScale	XScale	47.4	41.7
XScale	XScale	XScale	34.4	29.4

This table shows the time, in Seconds, to execute the PostMark file system benchmark. Each row depicts a different distribution of processes between the x86 and XScale CPU. The two times (C1 and C2) represent two different configurations of the IDE PCI card.

Figure 7: PostMark offload benchmark

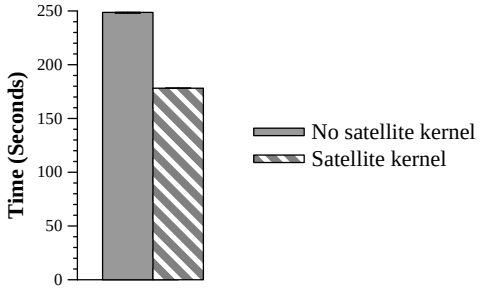
We used affinity to quickly change and test four different policies for offloading portions of the storage subsystem. We incrementally offloaded the IDE driver, then the file system, then the PostMark driver process. We used the same IDE controller (Silicon Image 3114) attached to the XScale I/O card and the PC host, and the same physical disk drive in all experiments. The results are shown in Figure 7.

Our first set of runs, shown under the column C1, demonstrated that offloading the IDE driver to the XScale I/O card improved performance by 70%. Upon further investigation, we discovered that the PCI card overloaded a configuration register (initially used to define the cacheline size of the CPU) to define a low-water mark that determined when the card should fill its hardware queue with operations destined for the disk. Since the cacheline size was used for the water mark, which differed between each architecture, the low-water mark was set too high and was not set uniformly. After reconfiguring the card, the performance of executing all processes on the x86 was roughly 2x faster than offloading the IDE driver to the XScale I/O card.

Our experience with configuring the PCI card was not unusual; in heterogeneous systems, small configuration changes often lead to large performance changes, and these performance changes may require changing the placement of processes. Using affinity simplifies performance tuning by allowing us to quickly cycle through different placement policies.

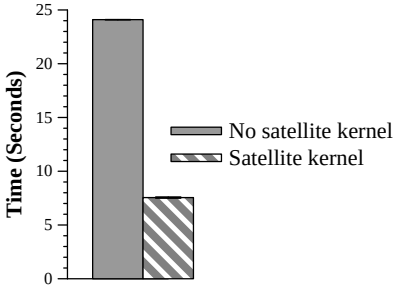
4.4.3 Indexing Application Offload

Our next experiment measures the benefits of offloading general work to the XScale I/O card. One of our colleagues had written an indexing tool for Singularity. The indexer builds a word-based inverted index on a set of documents by using a sort-based inversion algorithm with n-way external merge sort. The indexer is computationally intensive, but also interacts with the file system to read files and output its results. We decided to model a common problem: an



This figure shows the time to run a SAT solver while running a disk indexer. The left hand bar shows the time when both programs are run on the same CPU. The right hand bar shows the time when the indexer is offloaded automatically to a satellite kernel running on the XScale programmable device. The results are the mean of 5 tests. Error bars represent 90% confidence intervals.

Figure 8: Indexing offloading benchmark



This figure shows a scheduler microbenchmark that measures the time to spawn 16 threads hat each call `threadyield` 1 million times. The left hand bar shows a single-kernel version of Helios across two NUMA domains. The right hand bar shows Helios with two kernels, one in each domain. The results are the mean of 5 tests. Error bars represent 90% confidence intervals.

Figure 9: Scheduling NUMA benchmark

OS indexer running in the background and impacting computation of foreground processes. We ran the indexer at the same time we ran a SAT solver on a reasonably sized problem.

We ran the indexer with two different affinity values with the message-passing channel to the file system. A value of 0, which expressed no preference with respect to the file system, and a value of -1, which hinted that it might benefit from isolation. By changing the affinity value, Helios offloaded the indexer to the XScale I/O card. Figure 8 shows the results. By automatically offloading the indexer, the SAT solver runs 28% faster than when sharing the CPU with the indexing tool. The benefits of Helios were apparent when running the experiments, no recompilation or code editing was required. The author of the indexing tool did not need to be involved to create an offloaded version, since it only required adding an affinity value to the application's manifest.

4.5 Benefits of Performance Isolation

4.5.1 Scheduling Microbenchmark

Our next benchmark tested the utility of kernel isolation on our two-node NUMA machine. Many operating systems have had much work put into them to eliminate locking bottlenecks as the number of cores increases. We hypothesized that executing satellite kernels would be a simple way to scale around locking serialization problems. We ran a Singularity scheduling stress-test, which measures the time it takes 16 threads to call `threadyield` one million times. Helios uses the Singularity MinScheduler, which is a round-robin scheduler without priorities. The MinScheduler favors threads that have recently become unblocked and tries to avoid or minimize reading the clock, resetting the timer, and holding the dispatcher lock. The scheduler also does simple load-balancing by assigning work to idle processors through an IPI interrupt. The scheduler has known scaling problems, mainly because a single lock protects the dispatching mechanism.

The results of the scheduling benchmark can be seen in Figure 9. The version labeled "No satellite kernel" ran all 16 threads on a version of Helios that is not NUMA-aware. The right hand bar graph shows a NUMA-aware version of Helios that runs two kernels. The version with two kernels scales around the serialization bottleneck and must run only 8 threads per kernel; it is 68% faster than the single-kernel version of Helios. The greater than 2x improvement in performance over a single kernel is caused by the load-balancing algorithm, which issues expensive IPI interrupts and harms performance as the number of cores grows.

We are certain that the scheduler could be optimized to remove these serialization bottlenecks and improve single-kernel performance. Yet there would undoubtedly be other bottlenecks as the number of cores grows. Helios provides a simple mechanism around scaling problems: by growing the number of kernels, lock contention is decreased.

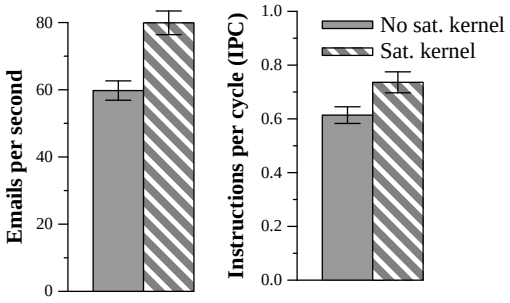
4.5.2 Mail Server Macrobenchmark

We were interested to see how Helios could improve performance of a more involved benchmark on a NUMA machine. We therefore ran a mail server benchmark and measured application throughput. The application, which is describe in Section 3.7.4, is composed of an SMTP server, an antivirus scanner, and a storage service. Since Singularity had no antivirus agent, we wrote our own and used the virus definitions for the popular ClamAV antivirus application. We attempted to make the antivirus agent as efficient as possible, but it is the bottleneck in the system since the SMTP server waits for emails to be scanned and written to disk before confirming an email's receipt. We hypothesized that the antivirus scanner, which is memory and CPU bound, would benefit from kernel isolation. Its negative affinity with the storage service causes it to run in isolation when two satellite kernels are present.

We randomly selected 1,000 emails from the Enron email corpus [19] and had a single client send the emails as fast as possible. The results are in Figure 10. The left hand graph shows the emails-per-second processed by the email server (which includes a blocking scan to the virus scanner). Using satellite kernels improves throughput by 39%. Since satellite kernels isolate remote vs local memory, every application component benefits from faster memory accesses on L2 cache misses. Further, satellite kernels ensure that processes always use local memory when accessing kernel code and data structures.

The right hand graph of Figure 10 shows the instructions-per-cycle (IPC) of the anti-virus scanner. We captured this metric using the performance counters available on AMD processors. The graph shows that the antivirus scanner improves its IPC by 20% when running on a satellite kernel. By running on its own kernel, other applications never have the opportunity to pollute the antivirus scanner's L1 or L2 cache, improving its cache-hit rate and therefore improving its IPC

We were interested to see how performance changed when multiple applications were executing at the same time. We therefore created a second configuration of the benchmark that ran both the email server application and the indexer. In addition, we changed



These graphs show the performance of an email server application that processes 1,000 emails from a single connected client. Each email is first scanned by an antivirus agent before being written to disk. The results are the mean of 10 trials, the error bars are 90% confidence intervals.

Figure 10: Mail server NUMA benchmark

the affinity values in the email server so that the antivirus scanner used self-reference affinity. Figure 11 shows the time to process 1,000 emails sent from two clients when running either one or two copies of the antivirus scanner. Each entry below an application lists the NUMA domain chosen by Helios to launch each process. When the antivirus scanner first launches, the affinity algorithm falls back on CPU utilization. Since the indexer creates CPU contention in domain 0, Helios launches the first antivirus scanner in domain 1. When the second antivirus scanner is launched, self-reference affinity causes Helios to choose domain 0. Since domain 0 is already executing many other processes (including the networking stack and the indexer), performance is improved by only 10% by adding an additional antivirus scanner. However, affinity made it simple to run multiple copies of the antivirus scanner in different satellite kernels without any knowledge of the topology of the machine.

5. RELATED WORK

To the best of our knowledge, Helios is the first operating system to provide a seamless, single image operating system abstraction across heterogeneous programmable devices.

Helios treats programmable devices as part of a "distributed system in the small," and is inspired by distributed operating systems such as LOCUS [33], Emerald [16, 31], and Quicksilver [27].

Helios is derived from the Singularity operating system [13] and extends Singularity's message-passing interface [6] to work transparently between different satellite kernels. A pre-cursor to this approach is found in the Mach microkernel [1], which provided an IPC abstraction that worked identically for processes within a single system, and for processes on remote systems over a local or wide area network.

Programmable devices and runtimes. Programmable devices have a long history in computer systems, and many prior operating systems have found ways to offload work to them. The IBM 709 [14] used *channel processors* to remove the burden of I/O from the main CPU and to allow I/O transfers to complete asynchronously. Subsequent systems such as the IBM System/360 and System/370 [15] and Control Data CDC6600 [32] continued this trend with increased numbers of channel processors. However, the range of operations supported by channel processors appears to have been limited to basic data transfer and signaling. The Bus Master DMA devices found on commodity PCs today are the logical descendants of channel processors.

SMTP Server	Indexer	AV1	AV2	Time (S)
D0	D0	D1	N/A	20.7
D0	D0	D1	D0	17.7

This benchmark shows a multi-program benchmark which measures the time to process 1,000 emails as fast as possible with 1 or 2 antivirus scanners while a file indexer is run concurrently. The entries in the second and third row describe which NUMA domain Helios chose for each process. Standard deviations were within 2% of the mean.

Figure 11: Self-reference affinity benchmark

Auspex Systems [12], which designed network file servers, ran a UNIX kernel on one CPU and dedicated other CPUs to the network driver, the storage system, and the file system. The offloaded processes were linked against a compact runtime that supported memory allocation, message passing, timers, and interrupt servicing. The process offloading avoided the costs associated with UNIX for time critical components.

Keeton et al. [18] proposed *intelligent disks*, adding relatively powerful embedded processors to disks together with additional memory to improve the performance of database applications. The Active Disk project [26] has validated this approach empirically, and the Smart Disk project [5] has validated this approach theoretically for a range of database workloads.

Network adapters have also been a focus of offloading. SPINE was a custom runtime for programmable network adapters built by Fiuczynscki et al. [9, 8]. The SPINE runtime provided a type safe programming environment for code on the network adapter and included abortable per packet computation to guarantee forward progress and to prevent interference with real-time packet processing tasks, such as network video playback.

More recently, the AsyMOS [23] and Piglet [24] systems dedicated a processor in an SMP system to act as a programmable channel processor. The channel processor ran a lightweight device kernel with a virtual clock packet scheduler to provide quality-of-service guarantees for network applications. McAuley and Neugebauer [20] leverage virtual machine processor features to create *virtual channel processors*. The virtual channel processors are used to sandbox I/O system faults and they run I/O tasks on either the CPU, or an I/O processor, depending on which performs better.

Helios benefits from this large body of prior work that demonstrates the benefits of using programmable devices to offload work. Helios builds on this work by focusing on a generalized approach for developing, deploying and tuning applications for heterogeneous systems.

Helios uses satellite kernels to export general OS abstractions to programmable devices. An alternative approach is to instead create specialized runtime systems. The Hydra framework developed by Weinsberg et al. [34] provides a programming model and deployment algorithm for offloading components onto programmable peripherals, including network adapters. Hydra assumes components are able to communicate through a common mechanism, and provides a modular runtime. Unlike Helios and SPINE, the runtime environment does not provide safety guarantees.

Heterogeneous architectures. The Hera-JVM [21] manages heterogeneity by hiding it behind a virtual machine. This additional layer of abstraction allows developers to exploit heterogeneous resources through code annotation or runtime monitoring, and it allows threads to migrate between general purpose processors and powerful floating point units that are part of the Cell architecture. This approach is closely related to Emerald [31], which allows objects and threads to migrate between machines of different architec-

tures over a local area network. Helios takes a different approach by making the differences in heterogeneous architectures explicit and by compiling multiple versions of a process for each available architecture.

Multi-kernel systems. Other operating systems have been built around the premise of running multiple kernels within a single machine, but prior efforts have focused on a homogeneous CPU architecture. Hive [4] exported a single instance of the IRIX operating system while dividing work and resource management among different NUMA domains to improve fault tolerance. Each NUMA domain ran its own copy of the IRIX kernel. Hive could migrate threads between NUMA domains, and would share memory between domains to give threads the illusion of executing on a single kernel. Hive shared I/O devices by running distributed applications, such as NFS, in different NUMA domains. Alternatively, Cellular Disco [11] ran multiple kernels within a single machine by running multiple virtual machines across NUMA domains to create a virtual cluster. Chakraborty [3] explored dividing cores between OS and applications functionality, and reserving certain cores for specific system calls. Finally, the Corey [2] operating system optimized multi-core execution by exposing APIs to applications to avoid unnecessary sharing of kernel state.

In general, there has been much research into improving the performance of NUMA architectures. For example, the Tornado operating system [10] and K42 [30] decompose OS services into objects to try and improve resource locality. In contrast, Helios uses satellite kernels to ensure that resource requests are always local to an application.

More recently, the Barrelfish [28] operating system strives to improve performance by using a mix of online monitoring and statically defined application resources requirements to make good placement decisions for applications. Helios and Barrelfish are complimentary efforts at managing heterogeneity. Barrelfish focuses on gaining a fine-grained understanding of application requirements when running applications, while the focus of Helios is to export a single-kernel image across heterogenous coprocessors to make it easy for applications to take advantage of new hardware platforms.

6. CONCLUSION AND FUTURE WORK

Helios is an operating system designed for heterogeneous programming environments. Helios uses satellite kernels to simplify program development, and it provides affinity as a way to better reason about deploying and performance tuning applications for unknown hardware topologies. We demonstrated that applications are easily offloaded to programmable devices, and we demonstrated that affinity helps to quickly tune application performance.

In the future, we see three main areas of focus. First, although we have found satellite kernels and affinity to be useful tools, their deployment has been limited to only one programmable device. Further, Helios has not yet been ported to the most promising programmable device: a GPU. We currently lack a compiler to create DirectX textures from CIL, and graphics cards do not provide timers or interrupt controllers, which are required to run a satellite kernel. In the future, we want to port Helios to the Intel Larrabee graphics card and measure the benefits provided by satellite kernels to applications targeted at GPUs.

Second, while we found workloads that benefit from the strong kernel isolation provided by Helios, the inability of processes to span NUMA domains limits the scalability of large, multi-threaded processes. In the future, we want to allow processes to span NUMA domains (should they require it) by moving CPUs and memory between satellite kernels.

Third, Helios is an experimental operating system with a limited number of applications. In the future, we want to determine how to create satellite kernels from a commodity operating system such as Windows, which supports a much larger API. A commodity operating system provides a much larger set of applications with which we can experiment.

Ultimately, the hardware on programmable devices has outpaced the development of software frameworks to manage them. Helios provides a general framework for developing, deploying, and tuning applications destined for programmable devices of varying architectures and performance characteristics.

7. ACKNOWLEDGEMENTS

We thank our shepherd, Andrea Arpaci-Dusseau, the anonymous reviewers, Jeremy Condit, John Douceur, Tim Harris, Jon Howell, Jay Lorch, James Mickens, and Don Porter for comments that improved the quality of this paper. We also thank Jeremy Condit and Derrick Coetzee for their work in writing applications for Helios. Aaron Stern's development efforts, which added ARM instruction sets to the Bartok compiler, made it possible to target Singularity code on the XScale I/O board. Manuel Fähndrich provided guidance on the Singularity channel architecture implemented by the Sing# compiler, and Songtao Xia made changes to the compiler that facilitate the addition of remote channels to Helios.

We thank Intel Corporation for providing the IOP348 development boards. Particular thanks goes to Curt Bruns, Paul Luse, Doug Foster, Bradley Corrion, and William Haberkorn for their assistance as we developed the XScale satellite kernel.

8. REFERENCES

[1] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A new kernel foundation for UNIX development. In *In Proceedings of the Usenix Summer '86 Conference* (Atlanta, GA, June 1986), pp. 93–112.

[2] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)* (San Diego, CA, December 2008), pp. 43–58.

[3] CHAKRABORTY, K., WELLS, P. M., AND SOHI, G. S. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM, pp. 283–292.

[4] CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., AND GUPTA, A. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (1995), pp. 12–25.

[5] CHIU, S. C., LIAO, W.-K., CHOUDHARY, A. N., AND KANDEMIR, M. T. Processor-embedded distributed smart disks for I/O-intensive workloads: architectures, performance models and evaluation. *J. Parallel Distrib. Comput. 64*, 3 (2004), 427–446.

[6] FÄHNDRICH, M., AIKEN, M., HAWBLITZEL, C., HODSON, O., HUNT, G. C., LARUS, J. R., , AND LEVI, S. Language support for fast and reliable message-based

communication in Singularity OS. In *Proceedings of the 1st Annual European Conference on Computer Systems (Eurosys '06)* (April 2006), pp. 177–190.

[7] FITZGERALD, R., KNOBLOCK, T. B., KNOBLOCK, T. B., RUF, E., STEENSGAARD, B., STEENSGAARD, B., TARDITI, D., AND TARDITI, D. Marmot: an optimizing compiler for Java. Tech. rep., 1999.

[8] FIUCZYNSKI, M. E., BERSHAD, B. N., MARTIN, R. P., AND CULLER, D. E. Spine: An operating system for intelligent network adapters. Tech. Rep. UW-CSE-98-08-01, University of Washington, August 1998.

[9] FIUCZYNSKI, M. E., MARTIN, R. P., OWA, T., AND BERSHAD, B. N. SPINE - a safe programmable and integrated network environment. In *Proceedings of the 8th ACM SIGOPS European Workshop* (1998), pp. 7–12.

[10] GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (1999).

[11] GOVIL, K., TEODOSIU, D., HUANG, Y., AND ROSENBLUM, M. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (1999).

[12] HITZ, D., HARRIS, G., LAU, J. K., AND SCHWARTZ, A. M. Using UNIX as One Component of a Lightweight Distributed Kernel for Microprocessor File Servers. In *Proceedings of the Winter 1990 USENIX Conference* (1990), pp. 285–296.

[13] HUNT, G., AIKEN, M., FÄHNDRICH, M., HAWBLITZEL, C., HODSON, O., LARUS, J., LEVI, S., STEENSGAARD, B., TARDITI, D., AND WOBBER, T. Sealing OS processes to improve dependability and safety. In *Proceedings of the 2nd Annual European Conference on Computer Systems (EuroSys '07)* (March 2007), pp. 341–354.

[14] INTERNATIONAL BUSINESS MACHINES. 709 data processing system. http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP709.html, 1957.

[15] INTERNATIONAL BUSINESS MACHINES. *IBM System/370 Principles of Operation*, 1974. Reference number GA22-8000-4, S/370-01.

[16] JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst. 6*, 1 (1988), 109–133.

[17] KATCHER, J. Postmark: a new filesystem benchmark. Tech. Rep. 3022, Network Appliance, October 1997.

[18] KEETON, K., PATTERSON, D. A., AND HELLERSTEIN, J. M. A case for intelligent disks (idisks). *SIGMOD Rec. 27*, 3 (1998), 42–52.

[19] KLIMT, B., AND YANG, Y. Introducing the Enron corpus. In *First Conference on Email and Anti-Spam (CEAS)* (July 2004).

[20] MCAULEY, D., AND NEUGEBAUER, R. A case for virtual channel processors. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence* (New York, NY, USA, 2003), ACM, pp. 237–242.

[21] MCILROY, R., AND SVENTEK, J. Hera-JVM: Abstracting processor heterogeneity behind a virtual machine. In *The 12th Workshop on Hot Topics in Operating Systems (HotOS 2009)* (May 2009).

[22] MICROSOFT CORPORATION. Singularity RDK. http://www.codeplex.com/singularity, 2008.

[23] MUIR, S., AND SMITH, J. AsyMOS - an asymmetric multiprocessor operating system. *Open Architectures and Network Programming, 1998 IEEE* (Apr 1998), 25–34.

[24] MUIR, S., AND SMITH, J. Functional divisions in the Piglet multiprocessor operating system. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications* (New York, NY, USA, 1998), ACM, pp. 255–260.

[25] PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. The use of name spaces in Plan 9. *Operating Systems Review 27*, 2 (1993), 72–76.

[26] RIEDEL, E., FALOUTSOS, C., GIBSON, G. A., AND NAGLE, D. Active disks for large-scale data processing. *Computer 34*, 6 (2001), 68–74.

[27] SCHMUCK, F., AND WYLIE, J. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 239–53.

[28] SCHÜPBACH, A., PETER, S., BAUMANN, A., ROSCOE, T., BARHAM, P., HARRIS, T., AND ISAACS, R. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems* (June 2008).

[29] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 3 (2008), 1–15.

[30] SILVA, D. D., KRIEGER, O., WISNIEWSKI, R. W., WATERLAND, A., TAM, D., AND BAUMANN, A. K42: an infrastructure for operating system research. *SIGOPS Oper. Syst. Rev. 40*, 2 (2006), 34–42.

[31] STEENSGAARD, B., AND JUL, E. Object and native code thread mobility among heterogeneous computers. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1995), ACM, pp. 68–77.

[32] THORNTON, J. *Design of a Computer – The Control Data 6600*. Scott, Foreman, and Company, 1970.

[33] WALKER, B., POPEK, G., ENGLISH, R., KLINE, C., AND THIEL, G. The locus distributed operating system. In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles* (New York, NY, USA, 1983), ACM, pp. 49–70.

[34] WEINSBERG, Y., DOLEV, D., ANKER, T., BEN-YEHUDA, M., AND WYCKOFF, P. Tapping into the fountain of cpus: on operating system support for programmable devices. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2008), ACM, pp. 179–188.