Embracing diversity in the Barrelfish manycore operating system

Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe
Systems Group, Department of Computer Science, ETH Zürich

Paul Barham, Tim Harris, Rebecca Isaacs
Microsoft Research, Cambridge

ABSTRACT

We discuss diversity and heterogeneity in manycore computer systems, and identify three distinct types of diversity, all of which present challenges to operating system designers and application writers alike. We observe that most current research work has concentrated on a narrow form of one of these (non-uniform memory access) to the exclusion of the others, and show with measurement why this makes sense in the short term.

However, we claim that this is not viable in the long term given current processor and system roadmaps, and present our approach to dealing with both heterogeneous hardware within a single system, and the increasing diversity of complete system configurations: we directly represent detailed system information in an expressive "system knowledge base" accessible to applications and OS subsystems alike, and use this to control tasks such as scheduling and resource allocation.

1. INTRODUCTION

This paper argues that diversity of hardware resources will be as much a problem as raw scalability as increasingly complex manycore processors become the norm.

In the manycore era, processor performance grows by increasing the number of cores rather than individual clock rates. Consequently, if an application's performance is to improve over time, then it must be designed to work over a wide range of degrees of hardware parallelism. However, while there is a great deal of work on programming abstractions for expressing parallelism, there is very little work on the operating systems and services that will be needed to support them. This is a problem because parallelism on tomorrow's manycore systems will look very different from parallelism on today's shared-memory multiprocessors. There are three main differences:

First, mass-market deployment means that it will not be practical to manually tune an application when deploying it on particular hardware. "Sensible" things must happen by default. Performance should meet users expectations – hopefully increasing, but certainly not decreasing, with the addition of resources. We do not want contention in the memory system to cause a program to slow down when it is allocated more cores; it is possible to even imagine some situations where the additional cores should be left idle.

Second, the hardware will be shared between multiple applications rather than a whole machine or static machine-partition being dedicated to one. The system will have to manage workloads with multiple CPU-hungry applications, deciding how to allocate the different kinds of cores available, and how to adjust applications' allocations over time. Often, the number of cores will vastly dominate the number of applications.

Finally, the range of execution environments that applications, and more importantly an operating system, have to confront is increasing over time. Writing system software that can adapt to this heterogeneity has received comparatively little attention in the literature.

1.1 Types of diversity

Diversity of hardware execution environments has at least 3 distinct aspects, which for the purposes of this paper we term *non-uniformity*, *core diversity*, and *system diversity*.

Non-uniformity includes the trend towards non-uniform memory access architectures for scalable multiprocessing, but we argue that the concept should be broader. Multiple levels of cache sharing are now the norm on multicore processors, rather than the single-level NUMA models embodied in many current OS kernels. Furthermore, interconnect technologies like HyperTransport and QuickPath mean that the inside of a general-purpose computer will increasingly resemble a network, and cost of messaging between cores may depend significantly on hop length and routing.

Core diversity refers to the expected heterogeneity of cores within a single system. Most existing NUMA machines still have homogeneous cores, but the trend for x86 processors is towards cores specialised for particular tasks with differing performance and power tradeoffs, and instruction set extensions [10]. The IBM Cell processor has radically heterogeneous cores. The practice of general-purpose processing using GPUs [20], and the ability to put FPGAs (many of which have their own ARM cores, for example) into processor sockets introduce further diversity in available processing resources.

In contrast, we use the term *system diversity* to refer to the fact that the resources available on completely separate systems are increasingly diverse, to the extent that it is impractical to write code optimised for any hardware design for any purpose other than bespoke scientific applications. At the same time, of course, efficiently using a given hardware design is a software problem. Those of us outside the scientific computing community, including those writing and maintaining operating systems, will have to write code that can run efficiently on a very wide range of hardware.

1.2 Aims

In the Barrelfish project we are tackling the question of how the OS and language runtime system should be architected for heterogeneous manycore systems; how the two should interact and how the interfaces between them should be expressed. We are doing this in the context of a new operating system specifically designed for a diverse and unpredictable range of future heterogeneous manycore architectures.

Our goals include scalable performance, but we acknowledge that we are unlikely to match the performance of an operating system backed by the resources of Microsoft or the Linux community. We choose to write an OS from scratch because of the clarity that comes from considering design issues unburdened with compatibility worries and legacy code [29].

However, an equally important goal for us is to reduce the code complexity involved in dealing with diversity. In particular, *mechanism* code should be as simple as possible for reasons of reliability and security. By producing better internal representations of hardware heterogeneity, we hope to reduce the complexity of the code on which correct and secure system operation depends.

As we discuss Section 2, conventional operating systems present fixed abstractions and a simplified view of hardware to applications and, indeed, to higher level parts of the OS. We show in Section 3 why this approach has been sufficient so far, but argue in Section 4 that this it will no longer be tenable in future manycore systems, and that now is the time to investigate alternatives. We are exploring the idea of exposing rich representations of the hardware in a manner amenable to automated reason and optimisation by system services and application libraries.

Storing and querying a representation of a heterogeneous machine's hardware and characteristics is in itself a complex task, and one that we envisage will be provided by a new OS service. Section 5 describes the use of the *system knowledge base* that performs this task in Barrelfish, and describes how it might be used to direct OS policy.

Section 6 discusses some of the outstanding challenges to building such a system, and our initial approaches. We conclude in Section 7.

2. BACKGROUND AND RELATED WORK

The trend in PC-class hardware is towards increasing diversity. But, paradoxically, most (though not all) of the complexity of modern hardware is ignored by current operating systems like Vista and Linux.

A modern OS can get away with this because, at the basic level of functionality, PC hardware itself does a good job of abstracting hardware differences: if written conservatively, the same piece of code will correctly execute on pretty much any piece of PC hardware. The exception is very low-level differences such as page-table formats on new processors, and the presence or absence of certain architectural features like virtualisation support or SIMD instructions. While strict correctness is preserved by such hardware uniformity, optimal performance is not, as we will see in Section 3.

IBM's Cell processor [11] represents a substantially more heterogeneous architecture than the x86-based systems that predominate today. Projects such as CellVM [19] (a Java Virtual Machine for Cell) show how difficult it is to transparently support such heterogeneous systems.

Most of the existing work to date on dealing with hardware diversity in commodity operating systems has been concerned with performance issues over NUMA architectures. We present a brief survey here centred on Linux, but similar techniques exist in other commodity operating systems, including Solaris [26] and Windows [24].

2.1 NUMA optimisations

The mainline Linux 2.6 kernel supports discontiguous memory and *memory allocation policies* [3]. Efficient support for discontiguous memory is implemented by adding another layer of address space, called *linear physical addresses* that are finally mapped to physical addresses. This way, Linux makes regular page tables appear to map to contiguous physical memory – a measure to make a heterogeneous machine look like an SMP. Processes may then currently either define a memory allocation policy for their whole virtual address space, or different policies for restricted areas of their virtual address space (currently, no memory mapped files and only specific cases of memory sharing are supported). The default system-wide policy is hard-coded into the kernel and tries to allocate memory that is local to the CPU a process is running on.

At present, common practice is for the OS to abstract away non-uniformity and application requirements, and subsequently have the kernel scheduler and memory allocators infer applications' needs through limited monitoring [3]. There is, however, a recognition that more information about the underlying hardware architecture, and more control over the allocation of resources, should be available to applications for better scalability. In the light of this, some information about the particular NUMA configuration present can be discovered using new kernel APIs under development, e.g. [12].

There is also work on the current Linux scheduler implementation, increasing awareness of NUMA architectures by allowing for processes to have affinity with sets of CPUs and their associated cache and memory, called *scheduling domains* [3]. Processes are load-balanced on these domains, as well as migrated between domains according to defined policies in order to abstract this information away from user-space and higher-level kernel layers. However, this process does not take instruction set and I/O heterogeneity into account and has been identified to perform badly on single-board heterogeneous architectures like AMD's Opteron when scheduling domains contain only a single CPU [3].

With regard to scaling the kernel itself, the Linux kernel documentation acknowledges that replication of kernel code and data would help scaling on heterogeneous architectures. This replication is currently implemented for the boot memory allocator and page allocation data structures by encapsulating all of their variables into special replication data structures that export CPU-specific allocation functions. On homogeneous architectures, there is one statically allocated version of each of these data structures to make kernel code uniform. Kernel code replication is currently only supported through special patches on a limited number of architectures that involve hardware tricks to map a copy of the kernel at its well-known base address and does not extend to kernel modules, as Linux expects certain shared data structures at fixed addresses in virtual memory.

The SGI Altix NUMA research effort identifies contention on locks and cache-lines as scalability-preventing factors on such heterogeneous architectures [4]. They further divide cache-line contention into *false cache-line sharing* and *cache-line ping-ponging*. False sharing occurs when an application's or the kernel's data is not structured with scalability in mind, resulting in at least two otherwise not related variables to end up in the same CPU cache-line. In this case, when at least one variable is written to, all variables in the cache-line lose cache locality. Ping-ponging occurs when different CPUs write to shared variables on the same cache-line, resulting in frequent change of exclusive ownership of the cache-line. This occurs frequently with busy multiple-writer locks, as the lock count is updated by different CPUs. They argue for CPU-local code and data to improve the situation.

Similarly to Linux scheduling domains, IBM AIX employs *resource sets* [17] to support processor and memory affinity. Resource sets utilise processor, memory and multi-chip module topology information, supplied directly by the hardware. Applications can be bound to processors either by the system administrator or through a system call interface. Once bound to a resource set, a thread will only be scheduled on the processors within that set and allocate memory near to these processors.This allocation is static unless changed manually.

Within Linux, there is also work on memory/cache locality [15], multi-queue scheduling [8,13], asynchronous I/O and NUMA-aware locking.

It is clear from the size of this section that much effort has been directed at optimising operating systems for NUMA, however we believe that future manycore processors will offer further challenges in this area, not only because they will include complex NUMA hardware topologies, but also because they will increasingly be used in general-purpose desktop and server systems, rather than the more specialised high-performance environments in which NUMA systems have traditionally appeared. In general-purpose computing, we argue that there is a need for more efficient automatic resource allocation by system software, and less opportunity for static optimisation and fine-tuning of a workload to a specific system.

2.2 Scalability in research systems

Intel's manycore runtime McRT [25] addresses the scalability of an application runtime, including some notion of inter- and intra-machine heterogeneity at the application level. McRT illustrates how important it is to make use of knowledge about the hardware, and provides a valuable tool for application writers. However, as an application runtime McRT addresses neither sharing a heterogeneous manycore machine between competing applications, nor dealing with many hardware issues that are not visible outside the OS kernel.

The Tornado [9] and K42 [1, 14] multiprocessor operating systems investigated scalability on large parallel NUMA systems with a clean-slate, to which Linux compatibility was later added. K42's NUMA support has been highly influential on our work, as has its use of online profiling mechanisms [5, 30] to drive OS policy, a topic we touch on later.

Uhlig [28] has investigated scaling the L4 microkernel by applying a wide variety of optimisations, some of which include choosing appropriate kernel mechanisms at runtime based on information such as application-provided hints. Our approach in Barrelfish builds on this by generalising the knowledge needed to choose among these alternatives, and exporting the hardware information to user-space.

2.3 Core and system diversity

While there has been a plethora of work on scaling on NUMA architectures, we have found relatively little work dealing with core– or system diversity.

Infokernel [2] stresses the importance of providing detailed information to user space. An Infokernel exports general abstractions describing internal kernel state to user-space applications to allow them to build more sophisticated policies on-top of kernel policies to direct those kernel policies in various ways. In contrast to our work in Barrelfish, Infokernel uses this technique in the context of an existing OS design, rather than looking at a new OS built around the concept.

In the Exokernel [7], all fixed, high-level abstractions are avoided and all information about the underlying hardware (such as page numbers, free lists, and cached TLB entries) is exposed directly to user-space. An Exokernel thus sacrifices the portability of applications in favour of more information and therefore more room for policy specialisation. The complexity this might imply is managed using standard interfaces supplied with library operating systems. As we will discuss further in Section 4, we are using similar principles in Barrelfish to provide more information to applications about the underlying architecture, albeit with the different goal of managing manycore heterogeneity. However, we are also exploring how to utilise hardware information, resource discovery and online measurement in such a system.

The Resource Kernel [21] is a loadable kernel module which interacts with the host kernel and allows the applications to reserve system resources which are then guaranteed. The module runs completely in kernel mode and is designed to run together with the host kernel. The main goal of this work is to satisfy the reservations made by applications on system resources. Neither user nor kernel space has a global view of the system's structure or resources, and thus they cannot apply global optimisations for resource allocation.

The Q-RAM [22] project is designed to satisfy minimum resource constraints and furthermore to optimise a utility function to allocate more resources to applications than minimally required, if available. If an application gets more resources than the minimum specified, it adapts itself to provide better QoS. Searching for optimal solutions is a hard problem [23] in practice, and quickly becomes infeasible with many applications. Furthermore, the utility function must be statically specified by the programmer, something we would prefer not to require for all Barrelfish applications.

Finally, there are a few examples in commodity operating systems of rich, high-level descriptions of heterogeneous hardware resources. In particular, the ACPI and EFI standards have an explicit representation of many board-level resources, and the CIM standard [6] defines a schema for a description of higher-level resources. It is easy and convenient for us to inject such representations into our own, but our intention is to build a system-wide knowledge base that is broader in scope.

3. HARDWARE DIVERSITY

As we have seen, commodity operating systems so far focused on multicore support efforts solely on NUMA abstractions. To date, and in the short term, there are good reasons for this, as we demonstrate in this section. As an early stage of our design for Barrelfish, we benchmarked the latency of various low-level operations on two different recent machines of the x86-64 architecture.

The first (see Figure 1) consists of an Intel s5000XVN workstation board with two Intel Xeon X5355 quad-core processors running at 2660MHz. They are connected to a single memory controller by the system's *front-side bus*. The memory and single PCI Express bus are accessed through the memory hub controller. This is a traditional non-NUMA system.

The second system is quite different (see Figure 2), consisting of a Tyan Thunder n6650W board with two dual-core AMD Opteron 2220 processors running at 2800MHz. They are connected by *HyperTransport* point-to-point links, and each processor socket has two directly-connected memory banks plus its own PCI Express root complex. This is a NUMA system.

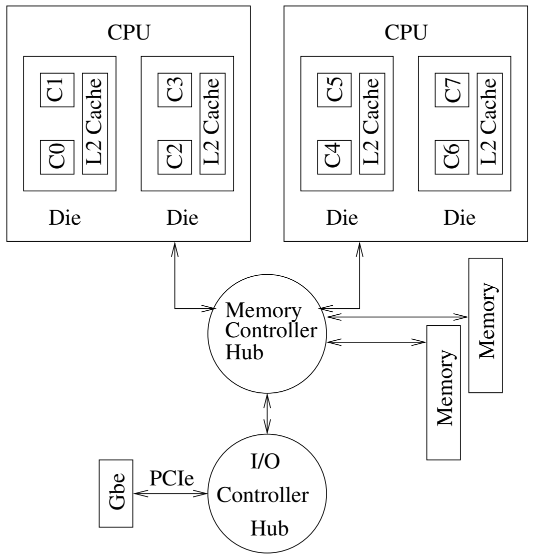

Figure 1. Structure of the Intel system

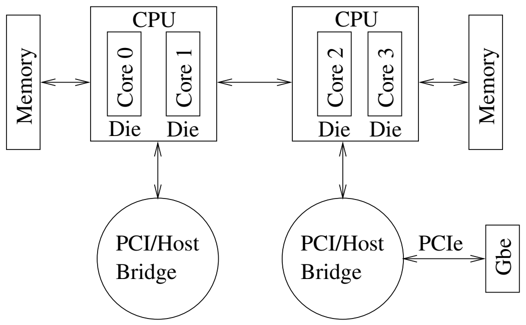

Figure 2. Structure of the AMD system

To run our benchmarks, we booted the hardware using our bare Barrelfish kernel. No interrupts, other than the interprocessor interrupt when required, were enabled and no tasks other than the benchmark were running. Every benchmark was repeated 1,000,000 times, the aggregate measured by the processor's cycle counter, and the average taken.

3.1 IPI latency

To learn more about the communication latencies within a modern PC, we measured the interprocessor interrupt (IPI) latency between cores in our test systems. IPI is one example of direct communication between cores, and can be important for OS messaging and synchronisation operations.

IPI roundtrip latency was measured using IPI ping-pong. Included in the total number of ticks is the code overhead needed to send the IPI and to acknowledge the last interrupt in the APIC. For our measurements, this overhead is not relevant, because we are interested in the differences rather than absolute latencies.

We measured the various IPI latencies on our two systems; the results are shown in Tables 1 and 2. As expected, sending an IPI between two cores on the same socket is faster than sending to a different socket, and sending an IPI to a core on the same die (in the Intel case) is the fastest operation. The differences are of the order of 10–15%. These may be significant, but it seems plausible that a simple OS abstraction on this hardware that treats all cores the same will not suffer severe performance loss over one that is aware of the interconnect topology.

	Roundtrip Latency	
	Ticks	μ sec
Same Die	1096	0.41
Same Socket	1160	0.43
Different Socket	1265	0.47

Table 1. IPI latencies on the Intel system

	Roundtrip Latency	
	Ticks	μ sec
Same Socket	794	0.28
Different Socket	879	0.31

Table 2. IPI latencies on the AMD system

3.2 Memory hierarchy

Modern multicore systems often have CPU-local memory, to reduce memory contention and shared bus load. In such NUMA systems, it is possible to access non-local memory, and these accesses are cache-coherent, but they require significantly more time than accesses to local memory.

We measured the differences in memory access time from the four cores on our AMD-based system. Each socket in this system is connected to two banks of local memory while the other two banks are accessed over the HyperTransport bus between the two sockets. Our system has 8 gigabytes of memory installed evenly across the four available memory banks. The benchmark accesses memory within two gigabyte regions to measure its the latency. The memory regions were accessed through uncached mappings, and were touched before starting to prime the TLB. This benchmark was executed on all four cores.

Table 3 shows the results as average latencies per core and memory region. As can be seen, the differences are significant. We also ran the same benchmark on the Intel-based SMP system. As expected, the latencies were the same (299 cycles) for every core.

Memory access is one case where current hardware shows substantial diversity, and not surprisingly is therefore where most of the current scalability work on commodity operating systems has focused.

3.3 Device access

In systems (such as our AMD machine) with more of a network-like interconnect, the time to access devices depending on core. Modern systems, such as our AMD machine, have more than one PCI root complex; cores near the root complex have faster access to

Memory region	Core 0	Core 1	Core 2	Core 3
0–2GB	192	192	319	323
2–4GB	192	192	319	323
4–6GB	323	323	191	192
6–8GB	323	323	191	192

Table 3. Memory access latencies (in cycles) on the AMD system

Core	Ticks	μ sec
0	567	0.20
1	567	0.20
2	544	0.19
3	544	0.19

Table 4. Device access latencies on the AMD system

the devices on that PCI bus than to devices on remote buses. Device access within the same system is therefore heterogeneous.

We measured this behaviour on our AMD-based system. To measure the access latency to a device we measured the cycles needed to read a PCI device's memory-mapped register. The device used was an Intel e1000 Gigabit Ethernet card connected to the root complex at the second socket (refer to Figure 2).

Table 4 shows that cores 0 and 1 need slightly longer to access the device register than cores 2 and 3. However, the difference is less than 5%.

3.4 Discussion

We have shown that heterogeneity present in current multicore systems is fairly limited; beyond the differences in memory access latency, which modern "NUMA-aware" operating systems such as Linux already abstract and handle, other non-uniformity is limited to minor differences in IPI latencies and device access times.

However, because there is already substantial work on NUMA systems, and because we believe that future manycore systems will feature more diversity in hardware diversity than NUMA memory, to the extent even of diversity between features available on different cores, we are focusing on a much broader problem in Barrelfish: exploiting the heterogeneity present in all aspects of a manycore system. The following section describes our approach in more detail.

4. HETEROGENEITY IN BARRELFISH

As we have seen, today's systems hide heterogeneity by abstracting the underlying hardware. System components and applications for the most part see a homogeneous SMP system where every core appears the same. By hiding the hardware heterogeneity, the system has no chance to optimise execution on the appropriate hardware components.

Stepping back somewhat, we observe that the same kernel data structures are typically used for two purposes:

1. *Policy*. These data structures represent the hardware's complexity to the operating system, and from them the OS derives policies and tradeoffs to guide its future execution. The requirements for this usage are ease of extensibility (in anticipation of future hardware developments), expressibility (to capture the richness and diversity of the underlying hardware), and flexibility (by covering a wide range of machine properties, an OS may optimise globally over a number of different tradeoffs).

2. *Mechanism*. The same kernel data structures are traversed at runtime to perform IPC, service page faults, send network packets, etc. The principal requirement for this second usage is performance. Under this requirement, these structures should be specialised for scalability, low latency of traversal, and high throughput.

These two sets of requirements are, of course, contradictory. Where high performance is the absolute priority (exemplified by the early design principles of L4 [16]), the kernel's representation of its hardware environment is highly specialised to the particular configuration the kernel has been ported to, and incapable of performing on another configuration without considerable code modification. Where functionality on diverse hardware is a requirement (as in most mainstream operating systems), the emphasis is more on flexibility.

In this paper we argue for an alternative approach. First, we are employing two different representations of the hardware in Barrelfish, one to drive policy and the other for efficient implementation of mechanism. Second, we want to expose as much information as possible to the OS components using these datastructures, and ultimately, runtime systems and applications. This enables the system and also the applications to make better use of the hardware capabilities and to deal with heterogeneity as best as possible.

4.1 The system knowledge base

In Barrelfish we envisage an operating system service, termed the *system knowledge base* (SKB), that contains a representation of the machine's hardware and current state. The SKB is populated with a mix of statically and dynamically determined data about the system as a whole, such as individual devices and the machine's interconnect topology. The information in the SKB can be derived in three ways:

1. It can result from *resource discovery*, such as PCIe bus enumeration. We expect such resource discovery and monitoring to be an ongoing process, as hotplugging of components (including processors and memory) becomes more commonplace.

2. It can be derived from *online measurement and profiling*, whether of devices, architectural facilities, interconnect links, or application performance and behaviour.

3. It can be *asserted* in the form of *a priori* knowledge about particular pieces of hardware, derived from data sheets, for example.

The SKB represents this knowledge in a form that is flexible, expressive, and easy to use in a sophisticated way by clients (both applications and other parts of the operating system). One possible approach (which we are currently developing) is to use constraint logic programming techniques – such an approach is more limited in expressive power than the description logics favoured by the semantic web community, but has the advantages of much lower evaluation complexity, and the ease of posing constrained optimisation queries. We anticipate the latter being useful in deriving OS policies.

Note that, as we discussed in the previous section, the SKB is used to drive system resource policy, but is not directly involved in the underlying mechanisms provided by the system. This means that access to the SKB is never on the system's critical fast path, and that therefore the performance of SKB access is not critical for the overall system performance, enabling the use of techniques such as constraint programming for queries, which run as background activity. Furthermore, we intend to build Barrelfish in such a way that even when the SKB is temporarily unavailable, the system continues to operate, albeit with a perhaps non-optimal resource allocation.

Clearly in the case of online measurement, there is a trade-off between the frequency at which this information is updated, the timescale over which it applies, and the potential performance improvements that can be gained. We are exploring this trade-off within Barrelfish, where a prime client of the SKB service is expected to be the task scheduler.

4.2 The role of the application

Information about system state should go through to the application; furthermore, the application, being a part of the system, should be involved in giving more information about its state and demands to the operating system.

Here is where the interface between the operating system and the application plays an important role. Two approaches are feasible:

1. The OS interface provides primitives to the application, so the application can direct the operating system on what devices to run which parts of its code.

2. The application is able to specify demands in terms of bandwidth and latency, as well as other factors, to the operating system, while also being able to convey information, including for example scalability figures of its run-time on different combinations of hardware.

The first approach is sub-optimal when dealing with a mix of applications. Local decisions on what to run where are made by applications that the scheduler tries to fulfil by best effort. The second approach conveys more information on the goals of an application to the scheduler, enabling it to find a better compromise. Again, in the second case, whenever there is not enough information conveyed by the application, the scheduler can infer some of it by trying solutions and determine their effectiveness by monitoring the system. This would not be possible within the first approach without violating the demands of the application.

Within both cases it is at least beneficial for an application to be able to explore the topology of the hardware on which it is running. While being absolutely necessary for the first approach, the application would be able to better balance its demands for the second. For example, if it knew that there was only a limited amount of bandwidth available between CPUs but ample amounts of processing power, it could choose an algorithm that favours local processing over communication.

There are two classes of application that we consider in Barrelfish:

1. applications that know their resource demands exactly and specify them to the system,

2. applications that don't know exactly what they require and can benefit from automatic profiling.

The first class of application is able to specify its resource demands in terms of CPU cycles, memory consumption, bandwith and other factors, and thus can directly interact with the SKB. A classic example of this class of application may include a database system, which performs its own optimisation and would prefer that the operating system give it certain available resources for it to manage.

Other applications (or their programmers) may not know or care about optimal resource requirements. Such applications can still benefit from the SKB, due to the system's use of online profiling and self-monitoring. Through this permanent background monitoring, the scheduler is able to adapt to the situation as new applications are started by the user or already running applications change their demands. The balance between what has to be decided immediately, possibly based on incomplete information, and what can be determined over longer periods of time by the operating system is an important area for further research. However, we note that previous work has shown that predictive resource management based on prior behaviour is sufficient to model demand for many applications [18].

5. USING SYSTEM KNOWLEDGE

In Section 4 we introduced the system knowledge base and explained what information can be gathered and added to the knowledge base using device classes containing properties of the devices. We also pointed out that the scheduler is one of the main clients of the knowledge base. In this section we show how the knowledge base can be used to make decisions, giving more concrete examples.

5.1 Managing core diversity

Dealing with diversity between cores will be a challenge for future operating systems; for example, some cores may not include SIMD or floating point instructions. The classic approach to managing this diversity would be to extend the CPU abstraction with a set of the features supported by each core, and use this information to schedule applications only on cores that contain the appropriate features.

However, in Barrelfish we export such information to the system knowledge base, which in addition to the simple strategy described above, allows applications to adapt to the available hardware features. For example, an application may query the SKB to find out what features are available; if all the cores with SIMD extensions are in use, and a (slower) non-SIMD version of an algorithm is available, it may be more efficient to use the non-SIMD variant on one of the free cores rather than forcing the applications to share the already-loaded cores, and also incurring additional context-switching overheads.

Even on current hardware with homogeneous cores, in some circumstances we may benefit by scheduling applications that use specific features (such as SIMD or FPU) on different cores, thereby avoiding extra context-switching overheads for saving and restoring the associated register state.

5.2 Interconnect-aware device access

As a concrete example of using the SKB to optimise runtime behaviour, consider the hardware configuration shown in Figure 3 which is typical of a contemporary multicore machine such as the AMD system described in Section 3. In this example, three applications are processing high-bandwidth incoming TCP streams. The NIC is a PCI bus-master device which accesses main memory via a bridge attached Node1. The NIC's device driver must enqueue receive buffer descriptors telling it where to DMA incoming packets, and these buffers may reside in either Memory1 or Memory2.

Some newer NICs provide hardware support for TCP offload and "zero-copy receive" and such NICs can potentially choose a DMA buffer which resides on the local-node of the destination process. Without TCP offload all three data streams will be written by DMA to the same buffer. With the applications App-A running on Node1, and App-B and App-C on Node2, information in the knowledge

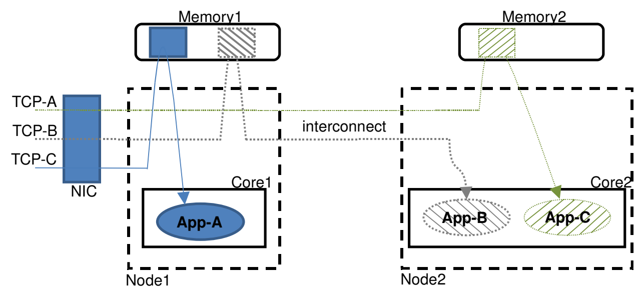

Figure 3. Example of a scenario in which information from the knowledge base can be used to guide placement of DMA buffers

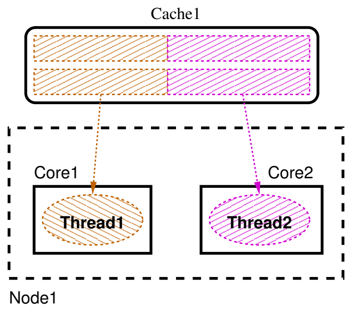

Figure 4. Example of a scenario in which two threads on different cores but on the same die share a cache line

base can be used to infer where best to locate the DMA buffer(s) according to the capabilities of the hardware and the observed behaviour of the applications.

Given that App-B must fetch its data across the system interconnect (for example, HyperTransport), its performance will depend on what proportion of the data it actually touches. If all bytes are touched, it may be better to DMA directly into memory on Node1, or to migrate the process to Node1. If only a few bytes are touched then transferring the entire packet to Node2 would be an unnecessary waste of system interconnect bandwidth. To make an informed decision about these tradeoffs, the operating system must estimate the cost of non-local memory traffic due to App-B and weigh this against the increased interconnect utilisation caused by NIC DMA to a remote node. This information can be derived from changes in the hardware performance counters whilst the process is running and stored in the knowledge base.

5.3 Improved cache sharing

Another example where improved operating system knowledge benefits the performance of applications is cache locality: on symmetric multi-threading hardware, cores on the same die share an L2 cache, as shown in Figure 4. Shared caches lead to different performance tradeoffs than private caches. If cores sharing a cache access the same memory regions, they can benefit from each other; in such a scenario, a cache miss by one core will cause a cache line to be fetched, and the other cores sharing the cache therefore won't miss on the same access. On the other hand, if the cores sharing a cache have many conflicting memory accesses (including reads), a lot of extra cache misses can occur due to capacity limits.

Suppose an application does matrix multiplication in parallel and its threads exhibit many interleaving memory accesses. In this case it is beneficial when the application's threads are scheduled on cores of the same die, so they can benefit from the increased cache locality, as they access memory on the same cache line.

When thread 1 accesses data on a cache line that is shortly thereafter also accessed by thread 2, as in the figure, thread 2 would benefit from the improved latency of the warm cache.

The operating system needs to know the hardware topology to be able to know how to optimally place the threads on the cores. This information can only be inferred from the knowledge base. Also, it needs to know that the threads are interleaving their memory access. This can be inferred by run-time observations of the application's behaviour through an on-line monitoring module.

Linux is able to achieve this goal by means of *scheduler domains* [3]. Within this example, one would define a scheduler domain for the two cores on the same die and the Linux scheduler would keep threads affine to that domain. The knowledge base in this case is hardcoded to the kernel: One writes an architecture-specific domain initialisation function.

Again, scheduler domains represent an abstracted view from the real hardware topology. They do not state the real interconnect between CPUs and they ignore all other hardware interconnects. This can become a problem when application behaviour changes. If, within the example, thread 2 suddenly starts accessing memory on another cache line, both threads start competing for cache lines and are best scheduled on separate dies, having their own L2 cache. It is not clear whether it is possible, but it is at least difficult to define such a dynamic policy using scheduler domains.

6. CHALLENGES

In the preceding sections, we have argued that exploiting heterogeneous hardware can improve performance, and that representing knowledge about the available hardware is important to do this. In Barrelfish, the hardware knowledge base provides such information to the system and applications.

There are several challenges in building such a knowledge base and having different clients use it. One of the major challenges is the design of the SKB itself, including designing the data structures to store the hardware knowledge. There are three key requirements for the SKB. First, it must have a sufficiently expressive representation to capture the full variety and complexity of hardware. Second, it must enable clients of the service to make use of it without prior knowledge of the range of hardware that might be available. Finally, there is a tension between the complexity of the information available through the service and the runtime performance cost of accessing that information. Therefore a "query fast-path", with potential degradation of accuracy or loss of detail, is also a necessary component.

The challenge of the first two requirements has been explicitly acknowledged in distributed systems for some time, where the approach has been to represent knowledge about services and networks in a machine-readable format. In the early days of name servers and service trading this was typically a set of name-value pairs. More recently, the matchmaking approach used in Condor [27] enables a system to match requests with available resource offers. In the modern world of web services, resource description is accomplished using various subsets of first-order logic such as the resource description framework and the OWL-DL web ontology language from the W3C.

Another challenge comes in populating the SKB with all the relevant information. Some information will come from static knowledge or once-off inspection of the hardware, however we will also require online profiling and information gathering services which measure the hardware's behaviour as well as the utilisation of devices. There is a trade-off here between the frequency and detail level of updates, the resources consumed to perform these updates, and the real performance gains enabled by having more recent knowledge in the SKB. The system needs criteria for deciding when more frequent updates will improve performance, and at which point the overall system starts to slow down, and thus less frequent updates would be better. The impact of performance monitoring can also be minimised by exploiting concurrency where possible, running analysis or monitoring tasks on a spare core or in idle processor cycles.

Performing online measurements and driving policies from the acquired knowledge should not conflict with the fast path of the system, because there should be no knowledge-base queries on the fast path. The final challenge is therefore to structure the performance-critical parts of the system that allows knowledge base operations to be moved out of band and operate concurrently with the rest of the system.

7. CONCLUSION

This paper has argued that core diversity and system diversity, while not a short-term barrier to exploiting multicore processors in general-purpose operating systems, will become so in the future as a consequence of scale and heterogeneity. As such, it is an important area of long-term systems software research, and complements current NUMA performance work.

We have described our approach to dealing with this challenge, in the context of Barrelfish, a from-scratch research operating system. We introduce the system knowledge base, a logical representation of hardware complexity, that serves as a queryable repository of system information derived from a combination of resource discovery, online measurement, and *a priori* knowledge, and outlined some of its uses.

We are now in the process of implementing Barrelfish, including the system knowledge base. The system currently boots on multiple cores, gathers hardware performance data, and populates an initial version of the knowledge base with information obtained from PCIe enumeration.

References

[1] Appavoo, J., Auslander, M., Da Silva, D., Krieger, O., Ostrowski, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Stumm, M., Gamsa, B., Azimi, R., Fingas, R., Tam, A., and Tam, D. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. IBM Research Report RC22863, Jul 2003.

[2] Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Burnett, N. C., Denehy, T. E., Engle, T. J., Gunawi, H. S., Nugent, J. A., and Popovici, F. I. Transforming policies into mechanisms with Infokernel. In *19th SOSP* (Oct 2003), pp. 90–105.

[3] Bligh, M. J., Dobson, M., Hart, D., and Huizenga, G. Linux on NUMA systems. In *Ottawa Linux Symp.* (Ottawa, Canada, Jul 2004), pp. 89–101.

[4] Bryant, R., and Hawkes, J. Linux scalability for large NUMA systems. In *Ottawa Linux Symp.* (Ottawa, Canada, Jul 2003), pp. 83–95.

[5] Caşcaval, C., Duesterwald, E., Sweeney, P. F., and Wisniewski, R. Performance and environment monitoring for continuous program optimization. *IBM J. for Research & Development 50*, 2/3 (Mar 2006), 239–248.

[6] Distributed Management Task Force, Inc. *Common Information Model (CIM) Standards*. Portland, OR, USA, Apr 2008. http://www.dmtf.org/standards/cim/.

[7] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: An operating system architecture for application-level resource management. In *15th SOSP* (Copper Mountain, CO, USA, Dec 1995), pp. 251–266.

[8] Franke, H., Nagar, S., Kravetz, M., and Ravindran, R. PMQS: scalable Linux scheduling for high end servers. In *ALS'01* (Nov 2001), pp. 71–85.

[9] Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *3rd OSDI* (New Orleans, LA, USA, Feb 1999), USENIX, pp. 87–100.

[10] Held, J., Bautista, J., and Koehl, S. From a few cores to many: A tera-scale computing research overview. White paper, Intel, Sep 2006. ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf.

[11] IBM. *Cell Broadband Engine Progamming Handbook*, 1.0 ed., Apr 2006.

[12] Kleen, A. A NUMA API for Linux. Technical Whitepaper 462-001437-001, Novell, Apr 2005. http://www.novell.com/collateral/4621437/4621437.pdf.

[13] Kravetz, M., Franke, H., Nagar, S., and Ravindran, R. Enhancing Linux scheduler scalability. In *Ottawa Linux Symp.* (Ottawa, Canada, Jul 2001).

[14] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V. K42: Building a complete operating system. In *EuroSys Conf.* (Leuven, Belgium, Apr 2006), pp. 133–145.

[15] Lameter, C. Local and remote memory: Memory in a Linux/NUMA system. ftp://ftp.kernel.org/pub/linux/kernel/people/christoph/pmig/numamemory.pdf, Apr 2006.

[16] Liedtke, J. Improving IPC by kernel design. In *14th SOSP* (Asheville, NC, USA, Dec 1993), pp. 175–188.

[17] Mall, M. *AIX Support for Memory Affinity*. IBM Corporation, Armonk, NY, USA, Jun 2002.

[18] Narayanan, D., and Satyanarayanan, M. Predictive resource management for wearable computing. In *MobiSys'03* (May 2003), pp. 113–128.

[19] Noll, A., Gal, A., and Franz, M. CellVM: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. Tech. Rep. 06-17, Nov 2006.

[20] Ohshima, S., Kise, K., Katagiri, T., and Yuba, T. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In *VECPAR* (2006), pp. 305–318.

[21] Oikawa, S., and Rajkumar, R. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *5th RTAS* (1999), pp. 111–120.

[22] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D. A resource allocation model for QoS management. In *18th RTSS* (Dec 1997), pp. 298–307.

[23] Rajkumar, R., Lee, C., Lehoczky, J. P., and Siewiorek, D. P. Practical solutions for QoS-based resource allocation problems. In *19th RTSS* (Dec 1998), pp. 296–306.

[24] Russinovich, M. Inside Windows Server 2008 kernel changes. *Microsoft TechNet Magazine* (Mar 2008).

[25] Saha, B., Adl-Tabatabai, A.-R., Ghuloum, A., Rajagopalan, M., Hudson, R. L., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., and Fang, J. Enabling scalability and performance in a large scale CMP environment. In *EuroSys Conf.* (Lisbon, Portugal, 2007), pp. 73–86.

[26] Sun Microsystems. Solaris memory placement optimization and SunFire servers. Technical white paper, Mar 2003. http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf.

[27] Thain, D., Tannenbaum, T., and Livny, M. Distributed computing in practice: the Condor experience. *Concurrency: Pract. & Exp. 17*, 2–4 (2005), 323–356.

[28] Uhlig, V. *Scalability of Microkernel-Based Systems*. PhD thesis, Computer Science Department, University of Karlsruhe, Germany, Jun 2005.

[29] Wisniewski, R. W., da Silva, D., Auslander, M., Krieger, O., Ostrowski, M., and Rosenburg, B. K42: lessons for the OS community. *Operat. Syst. Rev. 42*, 1 (2008), 5–12.

[30] Wisniewski, R. W., and Rosenburg, B. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *17th Int. Conf. Supercomp.* (Phoenix, AZ, USA, Nov 2003).