Karlsruher Institut für Technologie

System Call Aggregation for a Hybrid Thread Model

Study Thesis
by

cand. inform. Marc Rittinghaus

at the Department of Informatics

Supervisor:	Prof. Dr. Frank Bellosa
Supervising Research Assistant:	Dipl.-Inform. Konrad Miller

Day of completion: 23. December 2010

I hereby declare that this thesis is my own original work which I created without illegitimate help by others, that I have not used any other sources or resources than the ones indicated and that due acknowledgment is given where reference is made to the work of others.

Málaga, December 23th 2010

Deutsche Zusammenfassung

Um die Integrität sowie Stabilität in einem Computersystem zu gewährleisten, bedienen sich moderne Betriebsysteme wie Windows oder Linux hardwaregestützter Privilegebenen, die es ermöglichen, integrale System- und Hardwarekomponenten vor unkontrolliertem Zugriff durch Anwendungen zu schützen. Zum Ausführen privilegierter Operationen (wie das Erstellen einer Datei) ist die Anwendung dabei auf definierte Systemdienste angewiesen, die über eine zentrale Betriebsystemschnittstelle zur Verfügung gestellt werden. Obwohl es gelungen ist die Effizienz diese Schnittstelle mit der Einführung eigens konzipierter Prozessorinstruktionen deutlich zu verbessern, sind die Kosten für das Aufrufen von Systemdiensten auch heute noch beträchtlich. Dies betrifft insbesondere systemnahe Anwendungen, die auf rege Interaktion mit dem Betriebsystem angewiesen sind. Die vorliegende Studienarbeit beschäftigt sich deshalb mit Verfahren zur Verringerung dieser Kosten und stellt in diesem Zuge einen neuen Mechanismus vor, der gezielt auf Anwendung mit hoher interner Parallelität (wie z.B. Server Anwendungen) ausgerichtet ist: *Cluster Calls*.

Ein bekanntes Verfahren zur Reduktion des anfragerelativen Overheads ist das sog. *System Call Batching*, bei dem mehrere Systemanfragen gesammelt und in einem einzigen Aufruf gemeinsam an das Betriebsystem abgegeben werden. Dieses Verfahren bleibt jedoch nicht ohne Nachteile für die Anwendung. So summiert sich beispielsweise die Ausführungszeit der Systemdienste und kann zu unerwünschten Latenzen führen. Dieser und weitere Umstände machen bisherige Implementierungen insbesondere für Server Anwendungen wenig attraktiv.

Cluster Calls integrieren eine fortgeschrittene Form des System Call Batchings und kombinieren diese mit einem hybriden Threadmodell, um den Nachteilen des reinen Batchings entgegenzuwirken. Dabei fügt sich das Threadmodell leicht in häufig genutzte Server Architekturen und ermöglicht dadurch ein effizientes Sammeln von Systemaufrufen. Die wichtigste

Verbesserung liegt jedoch in der parallelen und asynchronen Ausführung der Systemaufrufe durch Betriebsystemthreads, die der Anwendung ein gleichzeitiges Voranschreiten erlauben. Das Verfahren setzt dabei auf eine frühe Integration in die Zielanwendung und bietet vielfältige Möglichkeiten zur anwendungsspezifischen Konfiguration. Es erweitert in diesem Zuge die Möglichkeiten der Systemschnittestelle und ermöglicht so beispielsweise die Spezialisierung von einzelnen Prozessorkernen auf definierte Systemdienste sowie die Priorisierung bei der Systemdienstausführung. Zur bidirektionalen Kommunikation zwischen Betriebsystem und Anwendung werden geteilte Speicherbereiche benutzt.

Eine Evaluation des Verfahrens hat gezeigt, dass Cluster Calls bis zu 58% des anfragerelativen Overheads für Systemdienste einsparen können. Um darüberhinaus die Anwendbarkeit von Cluster Calls in realitätsnahen Anwendungen zu bewerten, wurde im Rahmen dieser Arbeit das Verfahren in einen Web Server integriert und mit konventionellen Server Architekturen verglichen. Dabei hat sich gezeigt, dass Cluster Calls ein konkurrenzfähiges Threadmodell darstellen und eine gute Alternative zu Modellen basierend auf synchroner I/O bieten. In Tests konnte das Model einen vergleichbaren Throughput erreichen und zeigte gute Charateristiken bzgl. Latenz und CPU Nutzung. Durch den Einsatz von Cluster calls konnten bei der Verarbeitung von Webanfragen über 99% der Systemaufrufe vermieden werden. Es wurden jedoch auch einige Schwächen in der aktuellen Implementierung aufgedeckt, die ebenso in dieser Arbeit diskutiert werden.

Contents

Chapter 1

Introduction

In this work a new approach to the traditional system call interface is presented that aims at reducing the per-system call overhead caused by transitions from user-mode to kernel-mode in modern operating systems. The proposed solution presents an advanced form of system call aggregation and combines this mechanism with a hybrid thread model to improve batching efficiency. The design is specifically targeted at applications with a high natural degree of internal parallelism such as server applications.

1.1 Problem Definition

Although the overhead for system calls was dramatically reduced with the introduction of the fast system call mechanism in modern processors, the remaining overhead is still considerably high. From an application's perspective every CPU cycle spend on the system call interface is wasted and cannot be used for application specific tasks. Especially software that heavily interacts with the underlying operating system (e.g. I/O intensive applications) suffers from system call overhead.

System call batching is an effective way to reduce this overhead and has been used in several variants in the past. Nevertheless, plain system call batching has some disadvantages like an increased total completion time that make this technique less attractive for server type applications. Furthermore, most implementations are not available for the Microsoft Windows NT series of operating systems which dominate today's desktop market and hold a considerable share in service-oriented server installations.

1.2 Objectives

The main objective of this work is to reduce the per-system call overhead by decreasing the number of kernel boundary crossings through an advanced form of system call aggregation. The goal is to address the disadvantages of current implementations and offer a solution that is customizable, flexible, easy to use and that maintains full compatibility with the existing system call interface. Furthermore, the work aims at providing an implementation for Microsoft Windows that in principle allows to use the mechanism in real world applications.

1.3 Methodology

To show that the proposed solution can enhance the performance of the system call interface several benchmarks are conducted. The evaluation examines the overhead of the new approach and compares it to that of the existing interface. Because the proposed solution offers greater flexibility than the traditional system call interface, the performance characteristics of different configuration scenarios are also tested and compared.

In addition, the work examines the applicability of the new design through the evaluation of an examplary real world server application that integrates the new approach. The benchmarks provide a detailed performance comparison with alternative conventional server architectures.

1.4 Contribution

The contribution of this work is an analysis of the performance of the Windows system call interface and the introduction of a new thread model for server type applications that can offer new extensive possiblities for customization and application-centric optimization. While enhancing the per-system call overhead of the traditional system call interface, the new approach also increases the flexibility of one of the most integral interfaces of modern operating systems that has not undergone any fundamental changes in the recent past. The new approach also for the first time allows to decouple the invocation and the execution of general system calls under the Windows operating system.

1.5 Thesis Outline

In chapter 2 the work provides background information about related technologies including threading models and Windows specific concepts and terms. In particular, it describes the current design of the system call interface in modern operating systems. Chapter 3 takes a look at the performance of the traditional system call interface and discusses the optimization potential for different types of applications. The proposed solution of this work is then presented in chapter 4, followed by a description of an implementation for the Windows Research Kernel in chapter 5. The next chapter 6 deals with a detailed evaluation of the mechanism and its implementation. The thesis concludes with chapter 7, where the most important points are summarized. This chapter also gives an overview of possible future work related to the proposed solution.

Chapter 2

Background

This chapter first provides background information about the design of the system call interface in modern operating systems. Afterwards, an introduction to common threading models is given. The chapter ends with an explaination of several concepts and terms specific to Microsoft Windows.

2.1 Traditional System Call Interface

In most modern operating systems (e.g. Microsoft Windows and Linux) hardware supported privilege levels are used to establish a security boundary between applications and integral parts of the underlying operating system. According to this principle, applications run on a restricted level called user-mode while the operating system's kernel and its device drivers operate with full privileges in kernel-mode. If an application wants to perform a privileged operation (e.g. create a file), it has to request the service from the operating system. The means to do this are offered by the operating system in the form of system calls which the application can invoke. A system call performs a controlled transition from user-mode to kernel-mode, executes the corresponding system level code and returns to user-mode to deliver the system service's result. The application can then continue its execution.

Since privilege levels are enforced by the hardware, hardware specific instructions must be used to accomplish the mode transition. Prior to the Pentium II processor the most common method on the Intel x86 architecture was to use a specific software interrupt to cause the CPU to execute a generic system service handler in kernel-mode. Under the Windows NT series of operating systems entry 0x2E (in hexadecimal notation) in the

interrupt vector table is used for that purpose.

Windows also assigns each system call a unique number which it internally maps to the according system service function. An application's thread can now execute a system service by storing the number of the system service in question as well as a pointer to the arguments in predefined CPU registers and invoke the mentioned software interrupt (using the `int 0x2E` instruction). The CPU then performs a number of privilege checks, saves part of the execution context on the thread's stack and eventually executes the system service handler. Under Windows this is the kernel's *system service dispatcher function* (`KiSystemService`). The function's main purpose is to translate the system call number (which it received via the CPU registers) into a function address by using a special mapping table, copy any arguments from the thread's user-mode stack to its kernel-mode stack and finally call the system service function.

When the function returns, the system service dispatcher stores the result in a predefined CPU register and initiates a transition from kernel-mode back to user-mode. The transition itself is performed by a special return instruction (`iret`) that switches the privilege level and restores the execution state, which was previously saved by the software interrupt instruction [28, 3, 7].

A disadvantage of this mechanism is the overhead that is introduced due to the use of software interrupts. For that reason, in the Pentium II processors Intel introduced a fast call mechanism for system procedures which comprises two new instructions: `sysenter` and `sysexit`. AMD independently developed a similar mechanism, which in essence does the same thing. The corresponding instructions introduced by AMD are `syscall` and `sysret` [1]. For simplicity reasons, this work focuses on the Intel platform.

While a software interrupt is a very generic method to do system calls, the fast system call mechanism is specifically designed to implement system calls and reduce their overhead. The decrease in overhead is achieved by forcing the processor into predefined privilege levels which reduces the number of privilege checks ordinarily required to perform a far call to another privilege level. Furthermore, the target context state is predefined in model specific registers (MSRs) and general-purpose registers which eliminates all memory accesses except when fetching the target code (the system service handler). The basic design of system call numbers and a generic system service dispatcher function needs no modification in order

to use fast system calls.

Windows XP was the first version of Windows to use the new instructions. Compared to Windows 2000, which uses the interrupt based system call interface, system calls in Windows XP are 266% faster [4, 8].

Although the overhead for system calls was dramatically reduced with the introduction of the fast system call mechanism, the remaining overhead is still very high (see chapter 3.1). In the past, several attempts have been made to further decrease the per-system call cost.

2.1.1 Multi-Calls

Multi-Calls aim at reducing the per-system call overhead with the help of system call batching. The batching itself is done by an algorithm which the authors call *system call clustering*. The algorithm is a profile-directed approach to optimize an application's system call behavior. It uses execution profiles to identify groups of related system calls that can be batched into a single system call. Furthermore, correctness preserving compiler transformations such as code motion, function inlining and loop unrolling are performed to optimize the number and size of system call batches. This way, the amount of kernel boundary crossings is reduced which in turn reduces the per-system call overhead. The execution of the batched system calls is done with the help of a new system call—the multi-call—which implements the combined functionality. On invocation a multi-call serially executes all system calls of a given batch and returns their results. Experimental evaluation showed for the *mpeg-play* video software decoder an average 25% improvement in frame rate, 20% reduction in execution time, and 15% reduction in the number of cpu cycles [27].

2.1.2 Graphics Device Interface (GDI)

The *graphics device interface (GDI)* in Windows also uses batching to reduce the number of kernel boundary crossings. With the introduction of Windows NT 4.0 major parts of the Windows subsystem (including the graphics and windowing subsystems) moved into a kernel mode driver (`win32k.sys`). One of the reasons that lead to this drastic design decision were performance considerations. Both the graphics and windowing subsystems have a very high rate of interaction with hardware through video

drivers as well as mouse and keyboard drivers.

One the other hand, (graphical) Windows applications heavily use the graphics and windowing subsystem to present their user interface. In the new design this increases the overall system call rate. Although Windows already batched GDI calls prior to Windows NT 4.0, from a system call overhead perspective, batching in the new design plays an even more important role. In that sense, graphics calls made by Win32 applications are not immediately executed by the graphics subsystem and drawn on the output device but are postponed until a GDI batching queue is filled [15].

2.1.3 Singularity

Singularity is a microkernel based experimental operating system by Microsoft Research. Most of the conventional operating systems (e.g. Windows, Linux, MacOS) are based on architectures and development tools of the late 1960's and early 1970's. Singularity started as a research project in 2003 with a focus on re-examining the design decisions and shortcomings of existing systems and software stacks.

One key architectural feature that was developed in this course are *software isolated processes (SIPs)*. SIPs are much like contemporary processes in today's operating systems, except that they use the type and memory safety of modern programming languages (i.e. C# and others) to achieve process isolation. For that reason, Singularity does not depend on many of the hardware based protection features that are in use by current operating systems. This includes the boundary between user-mode and kernel-mode. Consequently, the overhead for system calls is greatly reduced as mode transitions are completely avoided [10].

2.2 Thread Models

The simplest implementation of a threading model is known as *kernel-level threading*. In this model a 1:1 mapping between one user-level thread and one kernel-level thread is used. The only scheduling entity is offered by the operating systems kernel and it is the kernel's scheduler that is solely responsible for all scheduling decisions.

User-level threading (N:1) extends this idea with scheduling entities that are completely managed and scheduled by user-mode code. Typically, the operating system has no knowledge of these types of threads. An advantage of user-level threads is that managing and dispatching them does not involve the kernel and thereby avoids costly kernel boundary crossings. Furthermore, they allow the application to implement a custom scheduling algorithm which is optimized for the application's needs. On the other hand, user-level threads execute in the context of a kernel-level thread and therefore mostly suffer from poor system integration. Blocking system calls for example block the hosting kernel-level thread and do not allow to switch to another user-level thread instead. This can lead to unnecessary idle periods. In addition, multiple kernel-level threads are needed to exploit the parallelism of SMP machines.

Hybrid thread models (N:M) try to combine the efficiency of user-level threads with the functionality of kernel-level threads. In a hybrid model N user-level threads are mapped to M kernel-level threads. In practice, this offers a higher degree of flexibility. Blocking system calls for example do not necessarily imply a blocking of ready user-level threads. On the other hand, hybrid models often suffer from high complexity. Furthermore, the user-level scheduler and the kernel-level scheduler need be compatible to allow the necessary cooperation.

2.2.1 Scheduler Activations

With *scheduler activations* the operating system kernel provides each user-level threading system with its own virtual multiprocessor and uses upcalls to the user-level scheduler to coordinate scheduling decisions. Scheduler activations are very similar to traditional threads. They provide the context in which user-level threads can run. However, the kernel manages these threads in terms of processors and knows about the hosted user-level thread [2].

2.2.2 First-Class User-Level Threads

First-class user-level threads are an implementation of a hybrid-thread model that uses shared data structures between user-mode schedulers and the kernel to enable the necessary cooperation. Furthermore, virtual processors and software interrupts are employed by the kernel to inform

user-mode schedulers about important events such as a blocking system call or an elapsed time slice in preemptive user-level threading [14].

2.2.3 User-Mode Scheduling (UMS)

User-mode scheduling (UMS) is a thread model present in the 64 bit versions of Windows 7. It aims at combining the flexibility and the performance benefits of user-level threading with the functionality of kernel-level threading but without introducing the full complexity of hybrid threading. Windows natively uses a 1:1 mapping between user-level threads and kernel-level threads. UMS maintains this model but allows a user-mode scheduler to do cooperative user-level thread switching without the kernel. The kernel-level thread switch is delayed until the user-level thread does a system call. Since every user-level thread has a corresponding kernel-level thread (because of the 1:1 mapping), the kernel can identify the respective kernel-level thread and do a switch. From a system's perspective, this restores the standard thread model used in previous versions of Windows. If the thread blocks in the course of the system call, the kernel notifies the user-level scheduler and therefore allows it to execute another (user-level) thread [24].

2.2.4 Exception-Less System Calls

Exception-less system calls use a similar mechnism as proposed in this work. They use a hybrid thread model in conjunction with system call batching to reduce kernel boundary crossings. In contrast to this work exception-less system calls are targeted at the Linux operating system and use a POSIX-complaint replacement for the standard threading library to transparently change the way an application requests system services [31].

2.3 Windows Concepts and Terms

The mechanisms proposed in this work have been implemented on the Microsoft Windows operating system. Some of the concepts and terms specific to this platform are briefly described in the following sections. Later chapters will then refer to these concepts. To not go beyond the scope, only those parts of the underlying technologies are introduced that are of interest in the context of the work at hand. The reader can refer to the

corresponding references to get more information, especially on APIs and their usage. It should be noted, that most of the information presented in the next sections is taken from [29], [28] and [11]. Further references are included where appropriate.

2.3.1 Basic Architecture

All versions of Windows that are most commonly used today (XP/2003, Vista, 7) [30] are based on the system architecture which was introduced by Microsoft with Windows NT 3.1 in 1993. One of the original design goals of the Windows NT series of operating systems was to achieve compatibility across different versions of Windows and interoperability with other systems, such as UNIX, OS/2 and NetWare. The solution to this requirement was the introduction of the *Windows Native API* and so called *environment subsystems* which lie on top it. Figure 2.1 illustrates this layered design.

As within every modern operating system, the kernel and device drivers build the foundation upon which applications can run. Since the NT kernel is based on a hybrid design, most of these components reside in kernel mode to reduce performance overhead for message passing. The *Microkernel (Ke)* component is responsible for thread and interrupt dispatching, synchronization as well as trap and exception handling. The *Executive (Ex)* extends this functionality and comprises facilities for process and virtual memory management, I/O, security and many more.

The Native API sits on top of the kernel and is implemented in the *NT Layer DLL* (`ntdll.dll`). This *dynamic link library (DLL)* is mapped into every process's address space at runtime and functions as the interface between applications in user-mode and system services in kernel-mode. Most of the functions exported by ntdll.dll directly map to corresponding system services and call these via a traditional system call interface (chapter 2.1). However, in 32 bit versions of Windows the kernel dynamically chooses a system call trampoline at startup to accommodate for older hardware that relies on the interrupt (`int 0x2E`) based mechanism.

The environment subsystems build the last layer between the applications and the operating system. A subsystem consists of a subsystem process which is responsible for the subsystem's initialization and the management of its resources and one or more DLLs that form the environment specific API. The subsystem components then use the Native API and/or API

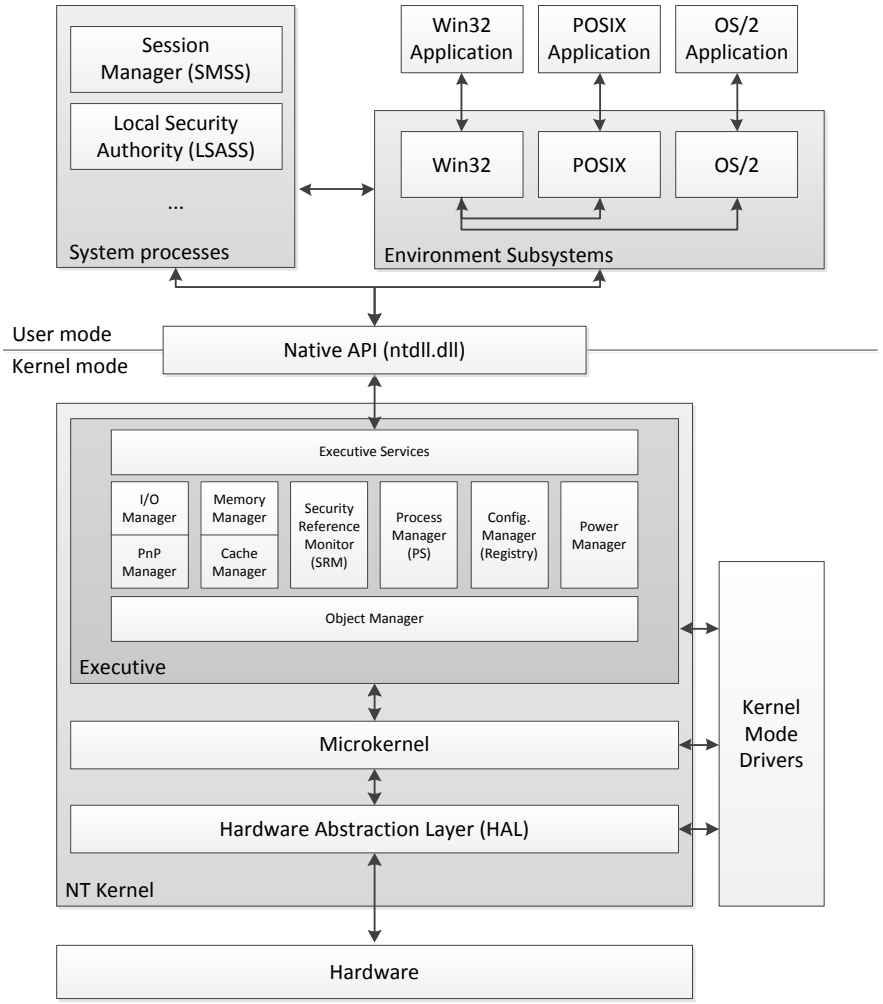

Figure 2.1: Windows NT Architecture

sets of other subsystems to implement their own functionality. As of today, Microsoft only supports the Windows subsystem and a POSIX complaint subsystem called *Interix* which is officially distributed under the name *Subsystem for UNIX-based Applications (SUA)* [32, 33].

Finally, every application that runs on Windows defines the subsystem (and thereby the corresponding API) against it was linked with the help of a special field in the *Portable Executable (PE)* header of the application's executable. Because the Native API is (although possible) not intended to be used by application developers, most of its functionality is not publicly

documented by Microsoft. Instead developers have to target one of the available subsystems and program against their APIs. In most cases this is the Windows subsystem and the corresponding Win32 API.

A typical system service request (e.g. to create a file on disk) will therefore start in the application with a call to the subsystem's API (`CreateFile`), travel through the Native API (`NtCreateFile`) and ultimately reach the corresponding system service handler in the kernel.

2.3.2 Windows Objects

One key concept of Windows is the use of objects to uniformly represent system resources. The kernel component that is responsible for the management of these objects is the *Object Manager (Ob)* which is part of the kernel's Executive.

Like in object-oriented programming, each object is an instance of a statically defined *object type*. This type defines attributes that are common to all objects of the same type. Examples are optional event handlers (open, close, delete, parse...), security mappings, quota charges, memory allocation constraints and a custom object body. At system startup kernel components register their object types with the Object Manager which from then on allows to instanciate them. Windows 7 implements a total of 42 object types. Commonly used objects include *Process*, *Thread* and *File* objects, synchronization objects like *Event*, *Mutant (Mutex)* and *Semaphore* as well as *Section* objects for shared memory and memory mapped files [23]. Since the NT kernel follows a hybrid design, some of the object types are based on simpler objects which the Microkernel implements. A thread object for example wraps the Microkernel's thread structure (the thread control block (TCB)) and extends it with further attributes (e.g a unique thread id).

Object instances are allocated in kernel memory pools and cannot be directly modified by user mode code. Instead applications reference objects with the help of *handles* and use a defined set of kernel services to do the work. The Object Manager then uses a process local *handle table* (there is also a system handle table for drivers) to translate the handles into memory pointers [18].

A process can get a handle to an object by creating a new object, inherit-

ing a handle from the parent process, duplicating a handle or by opening an object of a different process. The latter however is only possible with *named objects*. A process can choose to name an object during its creation and thereby allow other processes to identify it. Named objects are accessible through the system global *Object Manager Namespace* which is similar to a hierarchical file system with a root, directories, symbolic links and objects as files. In Windows Vista and later, applications are also able to create private object namespaces [20, 21].

To limit object access and the operations that can be performed with a corresponding handle, the Object Manager uses the Executive's *Security Reference Monitor (SRM)* to enforce a combination of token-based discretionary access control (DAC) and mandatory access control (MAC). Both are based on access control lists (ACLs) with a mixture of generic and object type specific access rights [22, 16].

If a process no longer needs access to an object, it simply closes the corresponding handle. Because the Object Manager maintains a reference count for each object, it automatically releases any resources bound to it as soon as the last reference is closed.

Another important property of most Windows objects is the fact that they are waitable. If the object body contains a *dispatcher header*, a thread can do a blocking wait on the object (provided it has the access rights to do so). Each waitable object in turn can be in two distinct states: *signaled* or *not signaled*. If an object is in the not signaled state and a thread attempts to wait on it, the dispatcher blocks the thread until the object becomes signaled and the wait can be satisfied. Although the basic mechanism of the signal state is equal for all object types, the meaning of it differs. A mutant object for example works as a simple mutex by setting the signaled state to not signaled, when a thread acquires the mutant and setting it to signaled, when the thread releases the mutant. Other threads which try to acquire the object during that time will automatically block until the mutant is released. Another example are process and thread objects that get signaled on termination and thereby allow a thread to wait for the termination of processes and threads. The current implementation allows a thread to wait on up to 64 objects at the same time. In addition, when a thread waits on multiple objects it can choose, if its wait should be satisfied as soon as any of the objects get signaled or if all objects need to be signaled [25].

2.3.3 I/O Completion Ports

The last section of this work dealt with the concept of Windows objects and mentioned a couple of object types. An *I/O completion port (IOCP)* is another object and the preferred way of receiving results from asynchronous I/O in Windows-based server type applications [19].

To do asynchronous I/O under Windows, a thread has to pass a special flag to the corresponding API when it creates or opens a file. The resulting file handle is then enabled for asynchronous I/O or *overlapped I/O* as it is called in Windows. To issue an overlapped operation on the corresponding file handle, a thread can use the standard I/O operations, but must accompany the call with an overlapped context. The context identifies the operation and enables a thread to request its status.

There are also multiple ways to receive an I/O completion notification. One way is to do a wait on the according file handle whose file object is signaled on I/O completion or to create an event object and pass it along with the overlapped context. When the I/O operation finishes, the kernel signals the event. Another way of receiving I/O completion notifications is called *Alertable I/O* which makes use of *asynchronous procedure calls (APC)* and thereby allows a thread to execute callback functions on I/O completion. A problem of all these notification mechanisms is that they are not suitable for high levels of concurrent I/O seen for example in server applications. The overhead of creating an event object (even if they can be reused for subsequent operations) or queueing an APC for every I/O operation is very high. I/O completion ports aid in reducing this overhead.

In essence, an I/O completion port is similar to a multi producer, multi consumer, unbounded blocking queue for I/O completions. The concept is to have threads issuing asynchronous I/O operations and then waiting on the port object to receive I/O completion notifications. Every thread that waits on a port object is—from an operating systems standpoint—expected to be solely in the loop of producing work (indirectly via asynchronous I/O) and consuming work (I/O completions). In that sense, producers and consumers are reflected by the same threads.

After a thread received a completion (along with the corresponding overlapped context), it can use the overlapped structure to identify the context under which the I/O operation was issued (e.g. a HTTP GET request in a web server) and make appropriate progress. This may include issuing fur-

ther I/O which will eventually lead to new completion notifications. In order to receive notifications, an application must associate any (overlapped) file handle for whose I/O operations it wants to receive notifications with the I/O completion port.

Typically, multiple worker threads (forming a thread pool) wait on the same port object. This allows one thread to issue an operation and another thread to process the completion. This way, if a thread which originally started an I/O operation is (due to other notifications) busy at the time the completion is queued, another thread can react to this event. This effectively decoupls the work from the individual threads and distributs it over the thread pool and thereby potentially over multiple CPUs as necessary.

As already mentioned, the concept expects a worker thread to only produce and consume work in the context of a single port object. For that reason, Windows internally attaches the thread to the port object, which allows the kernel to throttle the number of concurrently running threads on the same port. In practice, the default maximum concurrency is set to the number of CPUs in the system to avoid CPU overloading and to maximize throughput. If a thread which is attached to a port does a blocking system call, another thread waiting on the same port is readied. Although, this can lead to temporary oversubscription when the wait of the blocked thread is satisfied, the concept is self regulatory. This is because, the thread will block on the port as soon as it tries to receive new work and the number of concurrently running threads exceeds the defined port limit.

2.3.4 Fibers

A *fiber* is a very lightweight and purely user-level managed unit of execution [17]. Microsoft added fibers to the operating system to make it easier to port existing UNIX applications to Windows, which often use threading libraries to schedule their own threads.

A fiber shares the address space with others fibers and threads in the same process. Like a thread, each fiber has its own stack which allows it to execute code. Since fibers are neither managed nor scheduled by the Windows kernel, all related data structures reside in user mode and are managed by the application with the help of the Win32 API. This way context switching between fibers is especially fast, because the overhead of kernel entrance and exit is avoided. Per-fiber data structures (about

200 bytes) comprise the execution context (various CPU registers, including stack pointer and instruction pointer), the top and bottom memory addresses of the fiber's stack, the head of a structured exception-handling chain and a user-defined start routine and context. By default, fibers do not preserve the state of the floating point unit (FPU) and therefore do not allow the use of floating point arithmetic. If an application needs floating point support, it must specify a corresponding flag on fiber creation.

A fiber always runs in the context of the Windows thread, which scheduled it. A thread can execute multiple fibers, but only one fiber at a time. Therefore, if an application wants to run several fibers at the same time (e.g. to exploit the parallelism of SMP machines), it must create multiple Windows threads to execute them. The Win32 API does not include any form of fiber scheduler. Instead, fibers use co-operative multitasking and manually yield to other fibers. For that reason, Windows does not offer any fiber-aware synchronization mechanisms.

From an operating systems standpoint, a fiber always assumes the identity of the thread that scheduled it. In practice, this can lead to several problems. Blocking system calls always block the executing thread and do not allow to switch to another fiber instead. If a fiber exits, the whole thread and all of its fibers exit. Problems can also arise, if thread-owned kernel objects (like mutants) are used. Furthermore, fibers do not possess a private thread environment block (TEB). Since the Win32 subsystem extensively uses the TEB to store thread specific state information, applications must be particularly carefull when using fibers in conjunction with the Win32 API. The *last error number* and the *thread local storage (TLS)* are two examples of such thread specific data structures which are stored in the TEB [9].

Because of the mentioned difficulties which are related to fibers, Microsoft recommands against their usage. Instead, in Windows 7 Microsoft introduced *user-mode scheduling (UMS)* as an alternative (see chapter 2.2.3 for more details) [24].

Chapter 3

Analysis

In the following chapter the costs of the traditional system call interface are evaluated and the optimization potential for different types of applications is discussed. Subsequently, system call batching is presented as one way to achieve better performance by reducing per-system call overhead. In that course, several advantages and disadvantages of this technique are illustrated.

3.1 System Call Cost

Although the cost of system calls was greatly reduced with the introduction of the fast system call mechanism, a system call is still noticeably slower than a normal procedure call. Two small experiments were made to verify this statement.

The first experiment measured the number of CPU cycles needed to do a system call on a fully patched 32 bit Windows Server 2003 Enterprise Edition (Service Pack 2) running the Windows Research Kernel (Build 3800). The operating system was running on an Intel Core i7 920. For more information about the testing platform see section 6.2. The benchmark uses the `rdtsc` (Read Time Stamp Counter) instruction to retrieve timing information [5]. This instruction returns the value of a 64 bit hardware cycle counter that is incremented with each clock tick. Before each read the benchmark also executes the `cpuid` instruction to flush the CPU's intruction pipeline and thereby guarantee that all prior instructions have finished.

To get a better overview of the costs, the benchmark measures the CPU cycles needed to enter kernel-mode on the one hand and the number

of cycles needed to switch back to user-mode on the other hand. In this course a new system call (`NtReadTimeStampCounter`) was implemented. The system call reads the time stamp counter and returns the low 32 bit of the result with the help of the system call's return/status value. The built-in mechansim via `NtQueryPerformanceCounter` was not used in the experiment, because the function is internally more complex (and therefore unprecise in this test) to compensate for changing CPU cores and frequencies between measurements. Instead the benchmark uses the thread's affinity mask and BIOS settings to guarantee the same contraints. Furthermore, the benchmark executes with realtime priority (31) to minimize the risk of preemption. The benchmark itself works by reading the time stamp counter in user-mode immediately before and after the invocation of `NtReadTimeStampCounter` (as well as within the system call as mentioned) to compute the cycle counts in question via subtraction.

The second experiment measured the number of CPU cycles needed to do a regular procedure call and a respective return with the help of the commonly used `call/ret` instruction pair. The methodology is the same as in the first experiment, except that a normal procedure in user-mode was used instead of a system call.

The benchmark results are presented in table 3.1. The results of both experiments were stabilized by taking the average out of 1000 runs. Furthermore, the incurring measurement overhead (98 cycles) due to extra instructions (e.g. `cpuid`) was subtracted.

	Entry	Exit	Total
System Call	152	140	292
Procedure Call	2	3	5

Table 3.1: CPU cycles needed for system- and procedure calls. Procedure calls are about 58x faster than system calls.

The benchmarks revealed that on the test system a normal procedure call is about 58x faster than a system call. Although, the system call entry is slightly slower than the exit, both operations are costly (especially when compared to their procedure call counterparts).

Besides the direct cost of system calls, a significant indirect cost from the application's perspective is caused by the pollution of processor structures during the execution of the system service itself. These structures include

various data and instruction caches, translation look-aside buffers, branch prediction tables and others. The user-mode instructions per cycle (IPC) for Xalan (a benchmark from the SPEC CPU 2006 benchmark suite that is known to invoke few system calls) for example degrades by up to 65% when executing a `pwrite` (a Linux system call) every 1.000 to 2.000 instructions. Although the experiment was conducted under Linux, similar results can be expected for Windows, because of the hardware-centric nature of the experiment and the architectural similarities between the two platforms[31].

The frequency in which system calls are executed by an application determines its responsivness to optimizations of the system call interface. An interactive application like a simple text editing software (e.g. Microsoft Notepad) will from an end-user experience standpoint most probably not show any noticable performance improvements, because its system call frequency is low. Most activity in these applications is triggered by the direct interaction of the user and is therefore limited in its frequency. An application that heavily interacts with the system and not with the user has a better chance to show performance improvements, since the system call frequency is naturally higher. System monitoring software (e.g. a virus scanner), backup solutions and even simple programs like the file copy tool `copy` and its derivatives (`xcopy`, `robocopy`) are examples for such applications. A second group of applications that has a high chance of showing performance improvements due to optimizations of the system call interface are server applications (e.g. web servers). These types of applicaitons invoke I/O system calls at a high frequency to gather data from disk on the one hand (e.g. a web site and related content) and to send the data over a network to the client on the other hand. For each of these I/O operations a system call is needed. Even the very basic evaluation web server which was developed in the course of this work (see section 5.5 for more details) does at minimum 7 system calls to serve a single request. Production web servers like Apache however can sustain more than 20.000 parallel connections [13]. This indicates that there is a potential for performance improvements.

3.2 System Call Batching

System call batching is one way to reduce the per-system call cost by batching multiple system calls and executing them together. This has two positive effects. One the one hand, the number of kernel boundary cross-

ings is reduced and on the other hand the increased locality (due to longer execution times in user-mode and kernel-mode respectively) potentially increases cache and TLB hit rates.

However, simple system call batching alone and especially the batched execution have some disadvantages, too:

1. In order to do system call batching in the first place, an application must be specifically designed with this mechanism in mind. This in turn increases the application's development complexity. Another possibility to perform batching is to use compiler techniques to optimize the binary code for system call batching like it is done with system call clustering in conjunction with multi-calls 2.1.1. This takes aways the burden from the application developer but is however most likely suboptimal compared to a specifically designed application.

2. System calls in the same batch need to be independent. They either have to be completely unrelated to each other or the output of each connected system call does not need any further processing through the application before it can be passed as input for subsequent system calls.

3. System calls are executed serially. Therefore, the completion times for all system calls add up. This can be a problem with batching mechanisms that are intransparent to the application like it is the case with multi-calls. The batch for example may already include several system calls that will cause long blocking periods (e.g due to synchronous I/O). When the application then does a system call that is under normal conditions expected to return quickly (e.g. `NtClose` to close an object handle), the batching might cause bad performance or even break functionality (timeouts in communications).

4. The latency between the system call invocation and the corresponding result delivery increases. This is another consequence of the serial execution. As it is the case with the total system call completion time, this can be especially problematic with intransparent batching mechanisms. The `NtQueryPerformanceCounter` system call which reads the current value of the high performance counter is an example. A greatly delayed execution is most probably not favourable in that case. Another example is the request processing in server type applications (e.g. a file server). If two requests from which one can be served by the file system cache (request A) and the other one

can not (request B) reach the server at the same time and the system calls to read the respective file data are batched together and eventually executed serially, the reply to request A is delayed by the I/O time for request B.

The presented disadvantages are a direct result of the serial aggregation in sequential code or they emerge from the serial execution of the system calls in the kernel. To address these problems, the way system calls are aggregated as well as the way they are executed need to be changed. An alternative to the serial aggregation in sequential code is to batch only those system calls that are invoked by different threads. This has the advantage of a naturally given independence of system calls due to the parallel nature of threads. Unfortunately, this approach does not solve the other problems such as the serial execution in the kernel.

3.3 Summary

In this chapter the cost of system calls in terms of CPU cycles for kernel entry and exit were presented and compared to regular procedure calls. The experiments showed that even with the fast system call mechanism the per-system call costs are still 58 times higher than for a normal procedure call. In addition, significant indirect costs in terms of processor data structure pollution exist. While system call batching is a viable solution to reduce these costs, the serial aggregation and execution of system calls in existing solutions have disadvantages such as increased latency and completion time.

Chapter 4

Design

This chapter deals with the introduction of a new approach to the traditional system call interface. In the first section several design goals are discussed that derive from the analysis of the current system call interface. Afterwards, a new mechanism is presented that satisfies these design goals through a combination of system call aggregation and the use of a hybrid thread model.

4.1 Design goals

The last chapter showed that the traditional system call interface still suffers from high overhead. However, reducing the direct costs is very difficult, because the interface code itself is in most cases already highly optimized. The Windows system call dispatcher for example is even written in assembler to reduce the overhead at the instruction level. In addition, this code is specifically optimized for each target platform and makes use of respective hardware features (e.g. greater register set available in x64 systems compared to x86 systems). It is unlikely, that such code offers further potential for noticeable overhead reduction and performance improvements. The incurring costs are design inherent and the system call interface needs fundamental changes. Drastic modifications to the operating system design and its major interfaces like they are done in Singularity (see section 2.1.3) are not an option for conventional general purpose operating systems. They introduce a great risk of breaking compatibility which is a high price for improved performance. Windows still has to run a broad set of legacy applications and is therefore highly dependent on long term compatibility. Furthermore, due to the proprietary characteristics of the operating system and its ecosystem, applications on Windows are commonly

distributed in binary form. This further increases the importance of compatibility as changes to an application or even its recompilation can often only be done by the manufacturer itself (with the use of human and/or financial resources). A new approach to the system call interface should therefore maintain **full compatibility** with the existing design.

A reduction of the direct *system call interface overhead* (e.g. through code optimization) is as explained not particularly promising. System call batching (see section 3.2) or *system call aggregation* as it is refered to from this point on, however can be a feasable approach to reduce the *per-system call overhead* by collecting system calls and executing them together. Furthermore, the improved locality it offers has the potential to decrease processor structure pollution. Since system call aggregation is compatible to the existing system call interface design, it can also satisfy the compatibility design goal. The second design goal is therefore to use **system call aggregation** to reduce per-system call overhead.

In section 3.2 the disadvantages of the batching mechanism were mentioned. A new approach should address these issues. This work holds the thesis, that the size of system call batches can be optimized by an application design, that has system call aggregation in mind. A new approach should therefore naturally fit in the application at **development time** but do not (greatly) increase **development complexity**.

The batch size also heavily depends on the number of independent system calls (as explained) that can be collected. A new approach should therefore instrument the application design to maximize the number of **independent system calls**.

A major drawback that was mentioned in section 3.2 is the serial execution of batched system calls. This can lead to unwanted latency and increased completion time. A solution to this problems would be an **asynchronous** and **parallel** execution of system calls (also but not limited to within the same batch).

Finally, a new approach should take advantage of the fact, that it is integrated at development time and allow an application developer in some degree to **customize** and/or tune the mechanism to the application's needs.

To sum up, a new approach should satisfy the following criteria:

- Maintain **full compatibility** with the existing system call interface
- Use **system call aggregation** to reduce per-system call overhead
- Integrate at **development time**
- Do not (greatly) increase **development complexity**
- Maximize the number of **independent system calls**
- Allow **parallel** and **asynchronous** execution of system calls
- Allow application specific **customizations**

4.2 Cluster Calls

Cluster calls are a new approach to the traditional system call interface that meets the design goals presented in the last section. The mechanism is specifically targeted at server applications that internally use synchronous I/O and therefore must create hundreds of threads to achieve parallelism. Cluster calls exploit this parallelism. Furthermore, the high system call frequency of these types of applications make them ideal candidates for optimization.

Server applications that use kernel-level threading and synchronous I/O typically need a dedicated thread for every parallel request. To avoid massive context switching or even thread trashing (where the CPU spends more time switching thousands of threads than actually doing work) these types of servers often create a thread pool with a limited number of threads and use them to process requests. If all threads in the pool are busy, incoming requests are then stored in a queue for later processing. The fundamental concept behind cluster calls is to transition from this kernel-level threading scheme to a **hybrid thread model**. Instead of using threads to actually process requests, the hybrid thread model instruments fibers (see section 2.3.4) to do the work. The kernel-level threads then only function as an execution context for the fibers. From this point on, these threads are referred to as *cluster call clients*.

This approach is accompanied by a greatly decreased number of threads. Since the parallelism is achieved through fibers, there is typically no need to create more cluster call clients than there are CPU cores in the hosting machine. As explained later, it is a viable option to use even less cluster

call clients. On the other side, the hybrid thread model allows to increase the number of requests that are concurrently processed (not in terms of real parallelism, but in terms of requests, that are not waiting in a queue for later processing), because of the lightweight nature of fibers. An application can therefore create more fibers than threads (although this is not a problem with 64 bit address spaces). Every cluster call client has its own independent set of fibers and a user-mode scheduler for dispatching.

The integration of the hybrid thread model and is accompanying technologies is done at development time. This besides others allows the application developer to adapt the mechansim to the application's needs. For example, the user-mode scheduler can be replaced by a custom facility that takes special request characteristics into account. There are various more possibilities for customization which will be mentioned in the course of this section. This satisfies two of the design goals mentioned in the last section (*development time* and *customizations*).

The second key concept of cluster calls is to use **system call aggregation** to reduce per-system call overhead and maintain full compatibility with the traditional system call interface. This satisfies two more of the design goals (*system call aggregation* and *full compatibility*). In contrast to multi-calls, cluster calls limit the system call aggregation to a defined set of threads, the cluster call clients. Every other thread in the same application does not take part in the aggregation process. This enables the application developer to use system call aggregation in a more transparent as well as focused way and helps avoiding unwanted side effects of this technique (see section 3.2) in critical code paths (e.g. performance monitoring code). The aggregation of system calls is done within the fibers of a cluster call client. When a fiber makes a system call, the hosting cluster call client does not immediately execute it but instead stores it (system call number and corresponding arguments) in a special buffer and yields to another ready fiber. If no ready fiber within the same cluster call client exists, the client delivers the whole set of aggregated system calls to the kernel for execution. The process of doing so is called a *cluster call* and uses a new system call (`NtClusterCall`) (similar to multi-calls). The system calls, which take part in the aggregation are chosen at development time. This way, performance critical system calls like `NtQueryPerformanceCounter` for example can be excluded, allowing a more granular adjustment to the application.

Since the fibers represent the server's worker threads in the original de-

sign, the cluster call mechanism can exploit their independency to maximize the number of independent system calls. This satisfies another design goal (*independent system calls*).

A major difference to simple system call aggregation is the way, system calls are executed in the kernel. Multi-calls for example execute the system calls synchronously and serially which can lead to the mentioned disadvantages of increased total completion time and latency. Cluster calls instead use dedicated threads to process the system calls. These threads are kernel-mode only threads (or *system threads* as they are called in Windows) and run within the same address space as the cluster call clients. From this point on, these threads are referred to as *cluster call workers* or simply *workers*. By using dedicated workers, asynchronous and parallel execution of system calls becomes possible and effectively decouples the invocation of a system call and its execution. This also satisfies the corresponding design goal.

At first sight, using dedicated threads for system call execution might seem costly, but in fact system threads under Windows are cheap. They neither possess a TEB or a user-mode stack (which is 1 MB by default) nor does the *Client Server Runtime Subsystem (CRSS)* of the Windows subsystem allocate per-thread information (which it does for normal Windows application threads). Instead, system threads only have (besides various small management structures) a kernel-mode stack that occupies by default 16 KB (12 KB + 4 KB guard page) of pageable system virtual address space. Furthermore, if a system thread is blocked for a couple of seconds, the kernel stack's memory pages get swapped out (not directly to disk) by the memory manager.

The cluster call mechanism groups multiple workers together into a single *(cluster call) worker pool*. This allows to control the workers as a whole. If no system calls need to be processed, all workers in the pool are blocked. When a cluster call client then makes a cluster call, the workers wake up and execute the given system calls. However, the number of concurrently running workers is limited by the worker pool's *concurrency* similar in the way I/O completion ports limit the number of running threads. When a worker has finished processing a system call, it stores the result in a per-client shared memory region (called *communication buffer*) that is accessible by the client from user-mode. This allows the user-mode scheduler to ready and eventually execute fibers whose system calls have finished. The design and implementation of the communication buffer is explained

in detail in section 5.3.2.

The client's behaviour after the delivery of the system calls depends on the worker pool's *wait policy*. The policy determines if the client immediately returns to user-mode or if it waits in kernel-mode until a set of conditions specified by the wait policy are met (e.g. at least one system call must be finished). More details on the wait policy can be found in section 5.3.3. When the conditions specified by the wait policy are satisfied, the cluster call client returns to user-mode, checks the communication buffer for results and executes one of the ready fibers. If there are still outstanding system calls, the workers simultaneously continue to process them and to report the result to the client.

The concept of cluster calls is summarized in Figure 4.1. The lights show the current state of each scheduling entity: red means blocked, yellow means ready, green means running and grey means waiting for work. The number beneath each fiber's light indicates the times a fiber has been scheduled.

Step 1 illustrates a possible basic layout. In user-mode one cluster call client exists that hosts four fibers while executing one of them. The other fibers are ready and the client's communication buffer is empty. The worker-pool resides in kernel-mode and comprises five workers, that are waiting for system calls.

Step 2 depicts what happens if the executing fiber invokes a system call. The user-mode scheduler interrupts the system call and blocks the running fiber. The call is then stored in the client's communication buffer and ultimately one of the ready fibers is scheduled.

In the next step the last ready fiber tries to make a system call. This leads as previously to its blocking and the storing of the system call in the communication buffer. But in contrast to step 2, the user-mode scheduler cannot schedule another fiber as there are no ready fibers left. Instead it makes a cluster call to deliver the aggregated system calls to the worker pool. The wait policy of the worker pool ensures that the client waits in kernel-mode until at least one system call has finished.

Since the concurrency level of the worker pool is set to 1, only a single worker wakes up and starts executing the system calls. When the worker finishes a system call, it stores the result in the communication buffer of

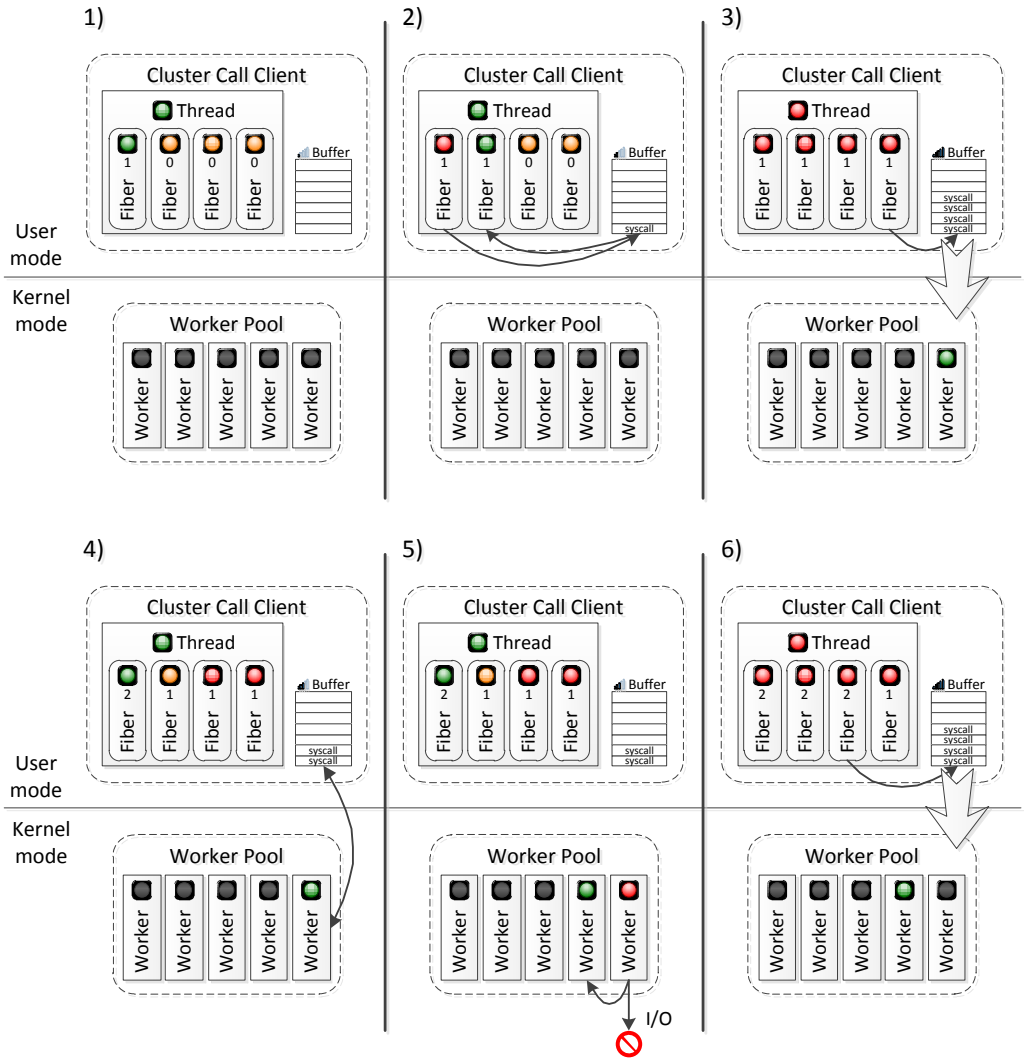

Figure 4.1: System calls made by fibers are aggregated in user-mode and passed to the kernel using a cluster call. Kernel-mode workers asynchronously process the calls and return the results to user-mode. The number of active workers is limited by the pool's concurrency. Blocking of workers is detected.

the client, that submitted the call. If there are more system calls pending, the worker continues with the execution of the next call.

Step 4 illustrates the situation, after two system calls have been finished. Since the client's wait is satisfied as soon as the first result appears, the client has already returned to user-mode and executes one of the ready

fibers. While the fiber corresponding to the second finished system call is also ready, the last two fibers are blocked. Their system calls are still pending. This property of the cluster call mechanism clearly seperates it from a pure user-level threading model, were such simultanious execution of (blocking) system calls in kernel-mode and fibers in user-mode is not possible.

Step 5 illustrates the concurrency limiting mechanism. Because the system call of the third fiber is a synchronous I/O operation, the worker blocks during the execution. This reduces the number of active workers in the pool to 0. This allows another worker to wake up and process the system call of the fourth fiber.

When the first worker wakes up (i.e. I/O finished), it will block as soon as it tries to get new work. This is because the pool's concurrency limit would otherwise be permanently exceeded. The last step illustrates this behaviour. It also shows that the client can make further cluster calls (e.g. because there are no ready fibers left), while the workers are still processing the system call from previous cluster calls.

The scenario depicted in figure 4.1 is very simple and only exemplary. In practice, more complex designs are possible that even offer further possibilities especially on multi-core machines. Besides the concurrency limit already mentioned, a worker pool has a cpu affinity and a priority assigned to it. Furthermore, apart from having multiple cluster call clients, it is also possible to create multiple worker pools. For each system call in a cluster call the client can then specify the target worker pool. This enables several interesting scenarios. An application can for example specialize one or more CPU cores for system call execution. It is even possible to specialize CPU cores for specific system calls (e.g. I/O related). Another possibility is the prioritization of specific system calls by directing them to a worker pool with an according priority setting. In addition, each worker pool's level of parallelism can be controlled by limiting the available CPU cores with the affinity mask and/or setting the worker pool's concurrency accordingly.

The presented options allow developers a high degree of customization. On the other hand, the new approach should not noticebly increase the *development complexity*, which is the last open design goal. To achieve this goal cluster calls are targeted to applications that can naturally make use of the presented hybrid model. This is for example the case for thread

pool based server applications. In these types of applications only moderate modifications are necessary to make use of cluster calls. Furthermore, most of the mechanism's complexity is not exposed to the developer but contained in a loadable DLL and the approriate kernel interface. The main task for the developer is therefore to first choose a layout (i.e. the number of cluster call clients, worker pools and workers) and then make appropriate settings (i.e. affinities, concurrencies, priorities etc.). However, all settings have default values that are sufficient for a basic configuration. The priority settings for example can be left untouched, if the application has no need for system call prioritization and the default affinities need no modification, if core specialization is not used.

4.3 Summary

In this chapter several design goals were defined and a new approach to the traditional system call interface named cluster calls was presented that satisfies these goals. The chapter explained how cluster calls combine a fiber-based hybrid thread model with system call aggregation to reduce the per-system call overhead and simultaniously improve simple system call batching through asynchronous and parallel execution of system calls. Furthermore, emerging possibilities in application design like core specialization and system call prioritization were mentioned which the traditional system call interface cannot offer.

Chapter 5

Implementation

In the following chapter an implementation of the cluster call mechanism introduced in 4.2 is presented. The cluster call mechanism was implemented into Microsoft Windows Server 2003 Enterprise Edition (Service Pack 2) using the Windows Research Kernel (Build 3800). This kernel is a slightly revised version of the standard NT kernel (Build 3790) delivered with all versions of Windows Server 2003 and the 64 bit versions of Windows XP. Differences between the kernels mainly focus on cleanup and removal of server support code, such as architectural code related to the Itanium (IA64) platform [26]. Because some parts of the mechanism's implementation rely on architecture specific assembler code, only a 32 bit version of the cluster call mechanism was implemented. Nevertheless, the concept is not confined to 32 bit and only minor modifications are needed to port the code to 64 bit. All architecture independent software elements developed for this work are written in C (no C++ involved). Some components however use Microsoft specific language extensions such as nameless unions and structs. The implemented user-mode support libraries are programmed for the Windows subsystem and the corresponding Win32 API. In fact, the concept of fibers used in this work is special to this subsystem.

The chapter begins with an overview of the different components involved and a coarse describtion of the basic control flow. Afterwards, the modifications to the NT kernel and the necessary components in user-mode are presented in detail. The chapter ends by giving a brief overview of the implementation of the evaluation web server, which was developed in the course of this work to give an example of how cluster calls can be used in practice.

5.1 Overview

The implementation is illustrated in figure 5.1. The upper half of the schematic shows the user-mode components and the lower half the kernel-mode components (major components are marked in bold). Most of the implemented elements reside in kernel-mode. These comprise components for resource and layout management (*cluster call group*), client management (kernel-mode part of cluster call clients), system call execution (worker pools and integrated logics) as well as a component for bidirectional communication between user-mode and kernel-mode (*communication buffer*). In user-mode most components are encapsulated in the user-mode part of the cluster call clients. The most important elements are the *fiber scheduler* and the corresponding thread which hosts and executes the fibers. The scheduler includes the logics for fiber scheduling and dispatching. These make up the user-mode threading part of the hybrid thread model. The scheduler also implements the system call aggregation mechanism. Another user-mode component is the *Nt Cluster API*, which is the cluster calls counterpart to the Windows Native API. It exposes the cluster call specific kernel extensions to user-mode applications. Each of these components is described in detail in the upcoming sections.

5.2 Control Flow

In the following, an overview of the cluster call mechanism from the implementation's control flow standpoint is described. The basic flow in a cluster call enabled application is depicted in figure 5.1. It is coarsely seperated into three different types:

1. *Orange*: Application specifc control flow (including not aggregated system calls) as well as flow dedicated to fiber scheduling and system call aggregation. The execution of this type is mainly situated in user-mode.

2. *Blue*: The cluster call itself (i.e. the delivery of batched system calls to the kernel).

3. *Green*: Control flow dedicated to the execution of system calls and result delivery. This type of control flow never leaves kernel-mode.

In the top left corner of the schematic several fibers and their hosting thread are shown. Since the fibers are user-mode scheduling entities,

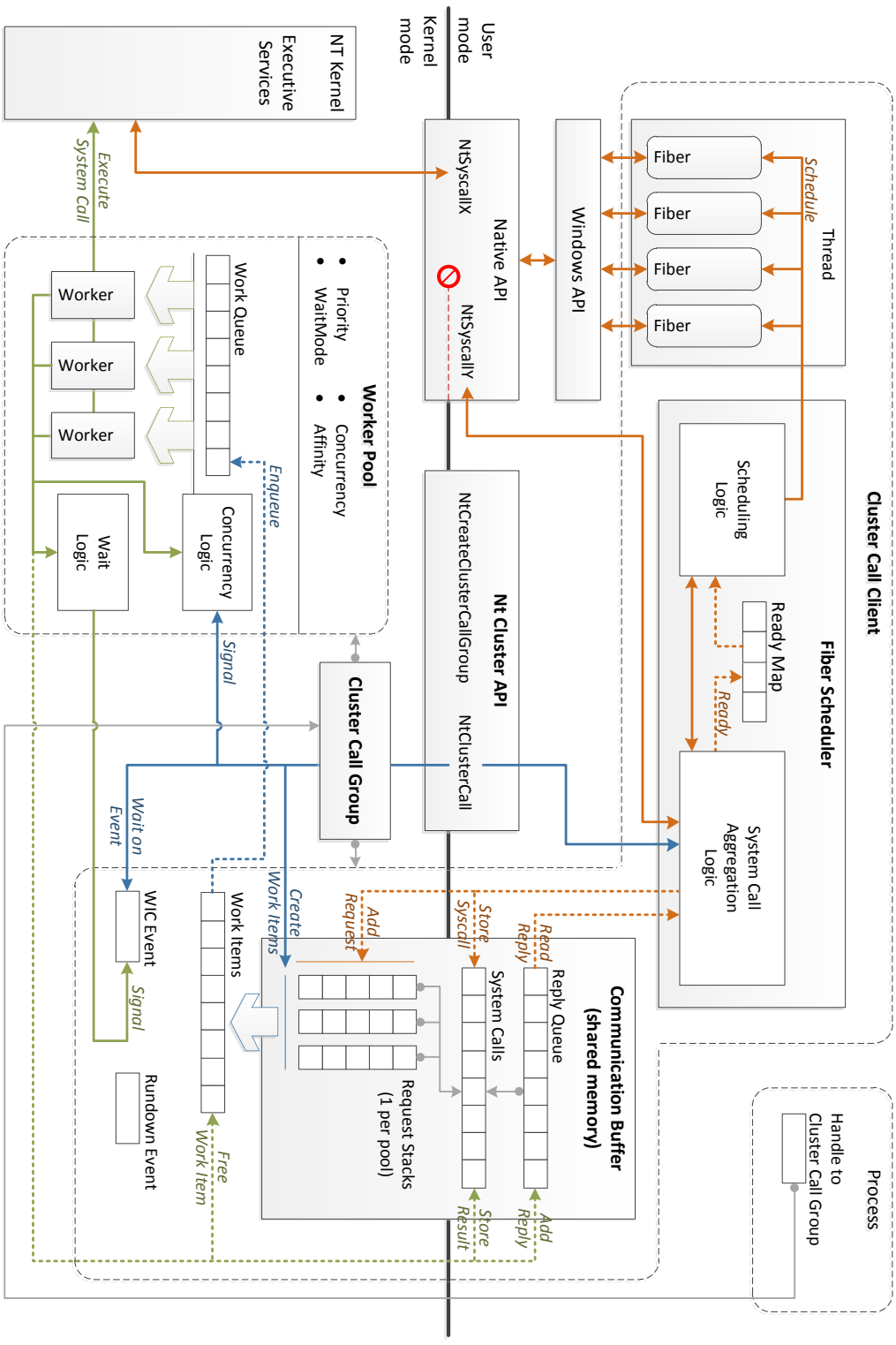

Figure 5.1: Cluster Call Implementation Overview

only one fiber per thread can run at the same time. The fibers execute application specific code and thereby interact with the Win32 API (orange control flow). As described in chapter 2.3.1, the subsystem layer sits on top of the Native API and uses it to make system calls. Since the cluster call mechanism allows to selectively aggregate system calls, only those specified by the application are redirected. All other calls directly reach the kernel's executive services as it is the case for unmodified applications. Redirected calls in turn are processed by the *system call aggregation logic (SCAL)* which stores the system call (or *system service request*) in the communication buffer for later access through kernel-mode components. A cluster call client can however only save requests for itself. Because system call execution is delayed, the fiber which originally invoked the call can no longer continue its execution. Instead, the aggregation logic leads the scheduler to yield execution to another ready fiber. In that course, it also updates the ready map (a list of ready fibers) by reading system call completion information from the communication buffer. If the ready map is empty and the scheduler cannot further dispatch fibers, the aggregation logic makes a cluster call and transitions to kernel-mode.

The cluster call (blue control flow) informs the kernel about the new requests and prepares their execution. The first step involves the creation of work items from the requests stored in the client's communication buffer. Each work item is statically allocated from a pool in the client's kernel-mode data structures. It enables a worker pool to identify the system call and the requesting client. It also indicates where to store the system call result (see 5.3.3). Each system call is bound (through application settings) to a specified destination worker pool. The next step in a cluster call is therefore to enqueue the work items for each pool to the correnspondingwork queues. Afterwards, the *concurrency logic* of each pool is signaled. This ulimately leads to the processing of the work items through the kernel-mode worker threads. The cluster call client in turn, waits on its *work item completion event (WIC Event)* object before it returns to user-mode.

Signaling the concurrency logic of a worker pool starts the execution of the aggregated system calls (green control flow). In this process, the concurrency logic limits the number of concurrently running worker threads to avoid CPU overloading and to adhere to the pool's concurrency setting (see 5.3.3). After a worker executed a system call, it writes the result to the client's communication buffer. Furthermore, it invokes the *wait logic* (see 5.3.3) to check if the client's wait can be satisfied. In that case, the client's wic event is signaled. Finally, the worker calls into the concurrency

logic which potentially blocks its execution. Otherwise, it tries to dequeue the next work item.

When its wic event is signaled, a cluster call client wakes up and immediately returns to user-mode. The aggregation logic then updates the ready map by reading system call results from the communication buffer. It then invokes the scheduler which in turn continues the execution of a previously blocked fiber.

5.3 NT Kernel Extensions

The following sections give a more detailed insight into the cluster call kernel-mode components. These include the cluster call group object, the kernel-mode part of the cluster call client, the communication buffer and the worker pools. All explanations focus on important data structures and their behaviour. Low-level implementation details are omitted. Instead, the reader is encouraged to take a look at the source code to get more information. The last section covers new system calls related to cluster calls and gives a short description for each of them.

5.3.1 Cluster Call Group Object

The cluster call group is a new object type registered with the Object Manager at boot time. Every application that wants to make use of cluster calls must at least create one cluster call group object. It then represents the central connection between multiple cluster call clients and worker pools and is used to manage necessary system resources. An application can create a cluster call group by calling one of the new system calls (`NtCreateClusterCallGroup`). The system call invokes the Object Manager to instantiate a new cluster call group object. After successfull initialization the object is inserted into the process's handle table and the handle is returned to the caller. This allows the application to reference the object in other system calls.

`NtCreateClusterCallGroup` also expects information about the number and configuration of worker pools. These pools (eight at maximum) including the worker threads are created in the course of the object initialization. Internally, the group object and the worker threads (which are also objects,

see 2.3.2) mutually reference each other to ensure proper object lifetimes.

Exposing the cluster call mechanism to applications with the help of a new object type has several advantages:

- Consistency with the Windows object model (fits into the uniform way of resource access and management)
- System-controlled resource access, management and accounting through the Object Manager
- Secure sharing and controlling of state information across process boundaries (a cluster call group object is nameable and securable).

5.3.2 Cluster Call Clients

If a thread wants to make cluster calls and thereby enqueue work to a cluster call group's worker pools, it must be attached to the group object. A thread can establish this connection by calling `NtAttachToClusterCallGroup` (one of the new system calls) and passing a handle to the group object. Attaching to the cluster call group increases its reference count and leads to the allocation of the kernel-mode cluster call client data structures including the client's private communication buffer. This transforms a thread into a cluster call client. A thread can attach to a single cluster call group only and the current implementation does not allow it to detach (only on thread termination). Furthermore, a thread can only attach to a group, which is owned by the same process as the thread. This is because the group's worker threads execute in the address space of the process which created them and therefore cannot access system call arguments for threads that are running in a different address space. The current implementation limits the number of clients that share the same group to eight.

Communication Buffer

The *communication buffer* is the primary communication channel between the fiber scheduler in user-mode and the worker pools in kernel-mode. The buffer is mapped into the virtual address space of the client's process and a second time into the system virtual address space. This twofold mapping scheme is necessary to ensure that a malicious client cannot crash

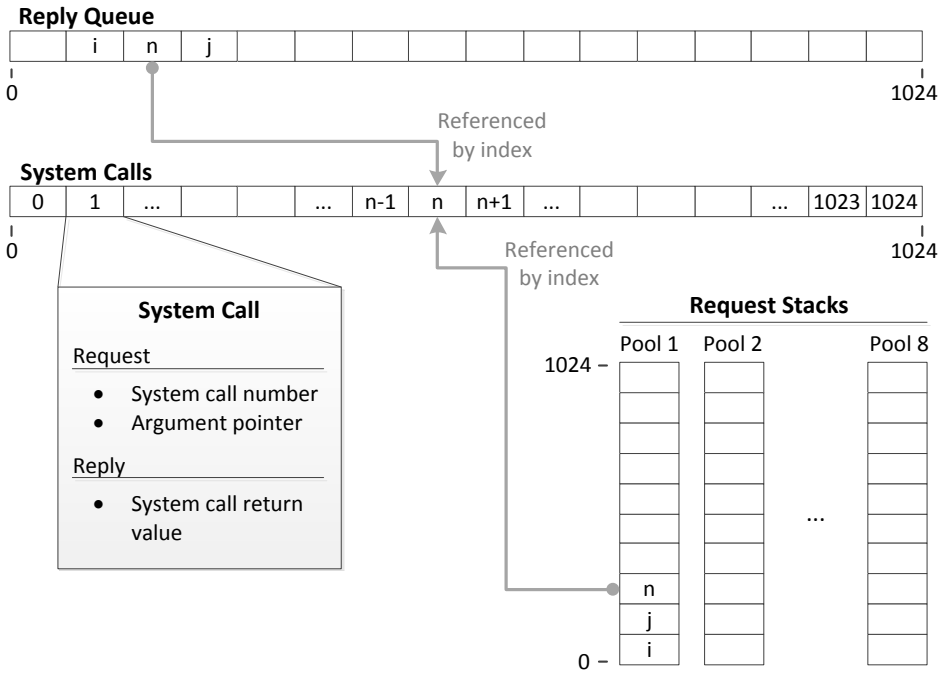

Figure 5.2: Communication Buffer: System calls and their results are stored in the system call array. Per-pool request stacks and a single reply queue point to elements in the array to hold new requests or replies respectively.

the system by unmapping the buffer or changing page protections (user-mode code can modify user-mode mappings but not system mappings). The buffer comprises three different data structures: the *system call array*, per-pool *request stacks* and a single *reply queue*.

The system call array is used to store the requests (system call number and argument pointer) and the replies (system call result). The current implementation allows a maximum of 1024 pending system calls (which also limits the number of fibers per client). Each entry in the element is 8 Bytes long (request and reply use the same memory location). The request stacks and the reply queue only reference entries in the system call array. Both data structures are each 1024 elements wide to potentially hold a reference to every system call. When the aggregation logic stores a system call, it takes the entry from the system call array with the same index as the fiber's id and updates it appropriately. Afterwards, it pushes

a reference to this entry onto the request stack, which corresponds to the destination worker pool of the system call. A worker thread in turn writes the result into the same entry in the system call array and adds a reference to it in the reply queue. This way the aggregation logic can determine if new system calls have finished by simply checking the reply queue.

5.3.3 Kernel Worker Pools

A worker pool encapsulates all mechanisms that are necessary for the delayed execution of system calls. The main components comprise the worker threads, a work queue and the concurrency and wait logics.

System Call Execution

To execute a system call, a worker thread performs the following tasks:

- Dequeue a work item from the work queue
- Read the associated entry in the corresponding client's system call array
- Capture and validate the system call number and the argument pointer
- Fake the worker's *previous mode* to user-mode (see [28] for more information regarding a thread's previous mode)
- Copy the system call arguments onto the worker's stack
- Call the appropriate executive service (by looking up the function pointer in the service descriptor table)
- Store the result in the client's communication buffer

Work Queue

A pool's work queue stores new work (system calls) to be processed by the workers and is populated through cluster calls. The queue is optimized for performance to reduce overhead introduced by the cluster call mechanism. The queue is essentially a statically allocated bounded array. In the current implementation the queue has 8192 elements so it can face the maximum possible load (at most 8 clients with 1024 system calls each). Adding work to the queue is synchronized by a queue specific lock that

must be exclusively acquired by the client prior to adding work items. Dequeuing work on the other side is more performance critical, because the number of threads (i.e. workers) potentially accessing the queue at the same time is higher. Furthermore, while adding work items can be done as a batched operation (i.e. acquire lock once and add any number of work items), dequeuing work items in batches makes no sense. This is because the complex execution behaviour of the workers (determined by unknown system call behaviour and the concurrency model) makes it difficult to predict the optimal batch size. Consequently, the queue is optimized for fast and lock-free (single work item) dequeuing by using an atomic compare-and-exchange operation. Each element in the work queue is a pointer to one of the work items in the work item pool of the client that enqueued the item. The work item itself is a 4 Byte value, which references an item in the system call array of the client's communication buffer.

Concurrency Logic

The concurrency logic is responsible for regulating the number of concurrently running worker threads within a single pool. The logic is implemented as a new synchronization object in the Microkernel: the *concurrency barrier*. Before a worker tries to dequeue work from the pool's work queue, it has to cross the concurrency barrier. If the limit for currently running worker threads is exceeded or if the pool's work queue is empty, the worker cannot cross the barrier and is blocked. If new work is available and the concurrency limit allows further workers to execute or if a running worker is blocked (e.g. through executing a blocking system call), one of the workers waiting on the barrier is woken up.

Internally, the concurrency barrier uses a data layout that is compatible with the *kernel queue* object which is the basis for the I/O completion port object. This allows the barrier to reuse most of the concurrency limiting functionality already built into the kernel. In contrast to the kernel queue, the barrier object does not store elements. It therefore cannot decide, if the pool's work queue is empty or not. Instead, it combines the concurrency limiting functionality with that of an event object which can be set and reset. This way, a cluster call signals the concurrency barrier that new work is available and the worker that is the first to run out of work, resets the barrier (which blocks all workers regardless of the number of running threads).

Wait Logic

Based on the worker pool's wait mode, the wait logic determines when the wic event of the cluster call client is signaled. Up to this point, the client waits in kernel-mode (directly after enqueuing the work items of the cluster call). Since a client can enqueue work to multiple worker pools (each with its own wait mode), the pool that is first to signal the wic event satisfies the client's wait. If all worker pools that process work of the client have a wait mode of *NoWait*, the client returns to user-mode immediately after delivering the system calls to the kernel. The logic implements four possible wait modes:

- *NoWait*: Do not wait
- *WaitAny*: Wait until at least one system call has finished
- *WaitSome*: Wait until at least a specified number of system calls has finished or a given timeout has expired. The timeout calculation however is based on the moment the first system call finishes (and not from the moment, the client starts to wait).
- *WaitAll*: Wait until all system calls have finished

5.3.4 Kernel Interface Extensions

System Call Name	Description
`NtCreateClusterCallGroup`	Creates a new cluster call group and returns a handle to it.
`NtOpenClusterCallGroup`	Allows (another process) to retrieve a handle to a named group object
`NtQueryInformationClusterCallGroup`	Returns various information about a group.
`NtSetInformationClusterCallGroup`	Changes the configuration of a group (currently only a stub).
`NtAttachToClusterCallGroup`	Attaches a thread to a group.
`NtWaitForSystemCallCompletion`	Blocks a client until its wic event is signaled.
`NtClusterCall`	Performs a cluster call.

Table 5.1: New cluster call related system calls

Table 5.1 gives an overview of all new cluster call related system calls. All of this system calls can be accessed by user-mode code through the NT Cluster API implemented in `ntclusterapi.dll`. The library is therefore similar in its function to `ntdll.dll`. To make the actual system call, the library also uses the system call trampoline.

5.4 User-Mode Library

The user mode library (`clusterapi.dll`) implements all user-mode components of the cluster call mechanism. This includes the fiber scheduler with its scheduling and system call aggregation logics as well as fiber-aware synchronization mechanisms (e.g. a fiber-aware critical section). An application can make us of cluster calls by linking to this library. This enables the application to create cluster call groups and allows an application's thread to *enter fiber mode*. In this process, a thread-private fiber scheduler is created and the thread is attached to a specified cluster call group. Afterwards, an initial fiber is created and its execution starts in a specified fiber main routine (similar to a thread main routine).

5.4.1 System Call Redirection

System call redirection is done using a modified version of *EasyHook*[6]. EasyHook allows to redirect a function to a user-specified one by replacing the function's head with a unconditional jump to the new function. The cluster call mechanism uses this feature to redirect chosen system calls in the Native API to a routine in the system call aggregation logic. The current implementation does not allow the application to specify which system calls are redirected, but instead uses a hard-coded set. Future versions, could expose an approriate interface that could even allow to redirect whole sets of system calls (e.g. I/O related) to ease application development. The modifications applied to the library primarily focus on optimizations (possible due to project constraints) and a partial rewrite to make the library fiber-aware.

5.4.2 Scheduling Logic

The scheduling logic in the fiber scheduler uses a round robin scheme to make reasonable fair scheduling decisions. In order to achieve that, the scheduler remembers the id of the last dispatched fiber and runs the next

ready fiber with a higher id. The search for this fiber is done with the help of the ready map which is implemented as a bitmap/bit array. This allows to use hardware accelerated bit scanning. On the other side, this makes the fiber scheduler a O(n) scheduler. The current implementation does not allow to replace the scheduler. An improved version of the cluster API could enable the application to replace the scheduler to adapt the scheduling to the application's needs.

5.5 Evaluation Web Server (net)

In the course of this work, a small web server was developed that makes it easy to directly compare the cluster call mechanism with other server thread models. The web server implements a total of four different models. Each of these receive incoming requests through an I/O completion port that is used in conjunction with an asynchronous socket accept (`AcceptEx`). The server installs a dedicated thread that is responsible to monitor the number of pending accept operations and to issue further ones as necessary. The models implemented are:

1. *Dedicated thread per request*: In this model, a thread waiting on the I/O completion port creates a dedicated worker thread for every incoming HTTP request. The worker then uses synchronous I/O (I/O port not involved) to process the request and terminates after sending a reply.

2. *Thread pool*: This model employs a static pool of multiple persistent worker threads. It is described at the beginning of section 4.2. Each worker processes a single request at a time. After completion, it does not terminate, but instead waits for the next request. Again, synchronous I/O is used.

3. *I/O completion port*: The third model makes use of asynchronous I/O and uses the I/O completion port to do all network and disk related I/O. As described in section 2.3.3, this model uses a thread pool, too. According to Microsoft, this is the preferred method for Windows-based server applications, as it offers the best scalability (see [12]).

4. *Cluster calls*: The last model uses the cluster call mechanism (with synchronous I/O) as presented in this work.

To allow a performance comparison of the models most of their code is shared. This includes the whole network and HTTP related components. Furthermore, model specifc code is designed to be as common as possible (especially between the synchronous I/O models).

5.6 Summary

In this chapter the most important implementation details were presented. The chapter started with an implementation-centric overview of how the cluster call mechanim works and continued by describing internals of the kernel-mode and user-mode components involved. The chapter ended with a brief introduction to the small web server used in the evaluation.

Chapter 6

Evaluation

This chapter evaluates the cluster call mechanism presented in the work at hand. It examines how the proposed solution compares to the traditional system call interface and how cluster calls perform in real world applications.

In particular, the evaluation will show that:

- In conjunction with a sufficiently high system call aggregation rate, cluster calls are capable of reducing the per-system call overhead.
- Cluster calls offer a viable thread model for server applications based on synchronous I/O.

The chapter begins with an introduction to the methodology and a description of the evaluation platform. Afterwards, the results are presented for each of the benchmarks conducted. The chapter ends with a discussion of the results.

6.1 Methodology

To evaluate the performance of cluster calls two sets of benchmarks were conducted. The first set examines the per-system call overhead of cluster calls and compares it to that of the traditional system call interface. More specifically, the benchmark measures the time needed for the processing of one or more invocations of a special no-operation system call and thereby allows to make conclusions about the computation overhead that is induced by the respective solution. The results are further differentiated through the use of varying worker pool configurations. These futher allow

to measure the costs for remote-core system call execution. The benchmark does not address potential benefits from reduced cache-pollution through increased temporal locality of user-mode and kernel-mode code and data. This is because of the no-operation system call's small cache footprint.

The second set of benchmarks uses the web server (net) presented in section 5.5 as an examplary real world application to evaluate how the cluster call mechanism compares to some conventional server thread architectures. The benchmark uses the ApacheBench HTTP client to generate traffic in the form of HTTP requests that are processed by the server application. The evaluation focuses on the achievable throughput, total request time and cpu usage of the different server models. In addition, it examines how many system calls can be saved through the cluster call's system call aggregation.

6.2 Evaluation Platform

The following system was used for the evaluation:

Component	Model/Specification
CPU	Intel Core i7 920
Cores	4
Frequency	2.67 Ghz
Private L1 i-cache	32 KB, 3 cycles latency
Private L1 d-cache	32 KB, 4 cycles latency
Private L2 cache	256 KB, 11 cycles latency
Shared L3 cache	8 MB, 35-40 cycles latency
Memory	6 GB (DDR3-1066) (only 3 GB accessable)
Operating System	Windows Server 2003 Enterprise Edition
Kernel	Windows Research Kernel (Build 3800)
Architecture	32 Bit (x86)
Service Pack	2 (fully patched)

Table 6.1: Evaluation Platform

To simplify the evaluation, the hyper-threading feature of the CPU was disabled so that only physical CPU cores were used by the operating system. Furthermore, the CPU clock speed was fixed to avoid measurement errors due to changing core frequencies.

6.3 Benchmarks

The following sections briefly describe how the benchmarks were conducted and finally present results for each of the respective benchmarks. The first section describes the methodology and results for the overhead benchmark and the second section compares the performance of cluster calls in the evaluation web server with conventional thread models.

6.3.1 Overhead

The overhead benchmark measures the time needed to process one or more system calls with the traditional system call interface and the cluster call mechanism respectively. As previously (section 3.1), the benchmark uses the `rdtsc` (Read Time Stamp Counter) instruction to retrieve timing information. In addition, a special no-operation system call (`NtNoOperation`) was implemented that only returns a constant status value (indicating success). Since the system call itself translates to only two machine instructions (a `mov` and a `ret`), its cost is negligible. For that reason, the benchmark can be considered to only measure the overhead for the *invocation* of a system call. To reduce the risk of preemption the measuring thread and all workers of the cluster call mechanism run with highest priority. Furthermore, the measuring thread is always bound to the first CPU core.

To examine the traditional system call interface the benchmark executes the no-operation system call a specified number of times and measures the time needed through a pair of `rdtsc` instructions which embrace the execution code. Afterwards, the total (overhead) time is divided by the number of executed system calls to calculate the per-system call overhead (as execution time).

The cluster call mechanism is basically evaluated in the same way, except that a cluster call is used to execute the system calls. For that purpose, the benchmark creates a cluster call group and attaches the thread that is used to measure the execution time as a cluster call client. However, the benchmark does not use the user-mode support library to conduct the tests (which would include the fiber scheduler), but directly interacts with the Native Cluster API. This allows to measure the overhead of the kernel-mode components only. Prior to the cluster call, the thread populates the request stacks with an appropriate number of no-operation system calls. Afterwards, the per-system call overhead is determined like it is for the tra-

ditional system call interface (measure total execution time for the cluster call and divide it by the number of batched system calls). Although, the benchmark uses varying worker pool configurations to examine the cluster call performance in different scenarios (e.g. remote-core execution), all configurations use a wait mode of WaitAll. This is necessary to ensure that the client (which measures the time) waits in the kernel until all batched system calls have finished.

The first scenario in the benchmark examines the per-system call overhead for increasing amounts of system calls under the constraint that system call invocation and execution is bound to the same (single) CPU core. This is always the case for the traditional system call interface since it does not support a decoupled execution. For the cluster call mechanism, this scenario is configured by creating a worker pool that is bound to the same core as the client (core 1). Since the no-operation system call used in the benchmark is non-blocking, the worker pool contains only a single worker that serially processes the system call batch. Figure 6.1 depicts this scenario:

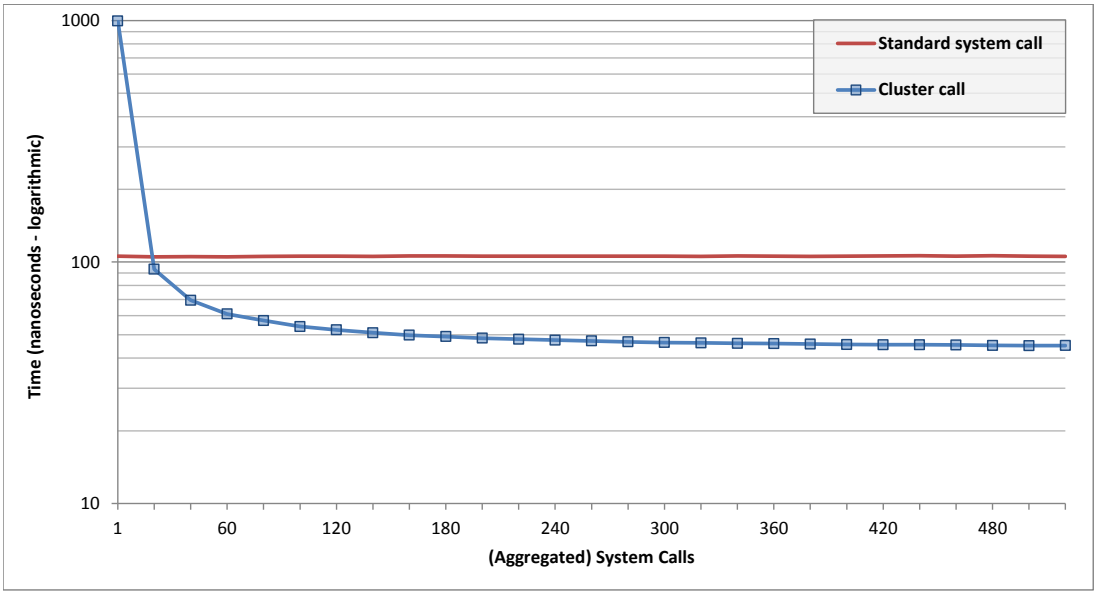

Figure 6.1: The traditional interface has an overhead of 105ns. For small batch sizes cluster calls are almost 10x slower. After 17 aggregated calls, cluster calls become more efficient than the traditional interface and finally reach up to 58% overhead reduction.

The x-axis shows the number of executed system calls and the y-axis shows the per-system call overhead in nanoseconds of execution time. The total overhead for the traditional system call interface grows linearly with the number of system calls. Consequently, the per-system call overhead stays practically constant at about 105ns. In contrast, the overhead for the cluster call mechanism changes with the amount of aggregated system calls. If only a few system calls are batched, the overhead is noticebly higher (almost 10x for a batch size of 1) but decreases rapidly with increased batch sizes. If 17 system calls are aggregated, the overhead is the same for both solutions. After this point, the overhead decreases down to approximately 50ns for 140 calls and finally reaches 44ns for the maximum of 1024 aggregated calls. The benchmark shows that compared to the traditional interface, cluster calls can reduce the overhead by up to 58%.

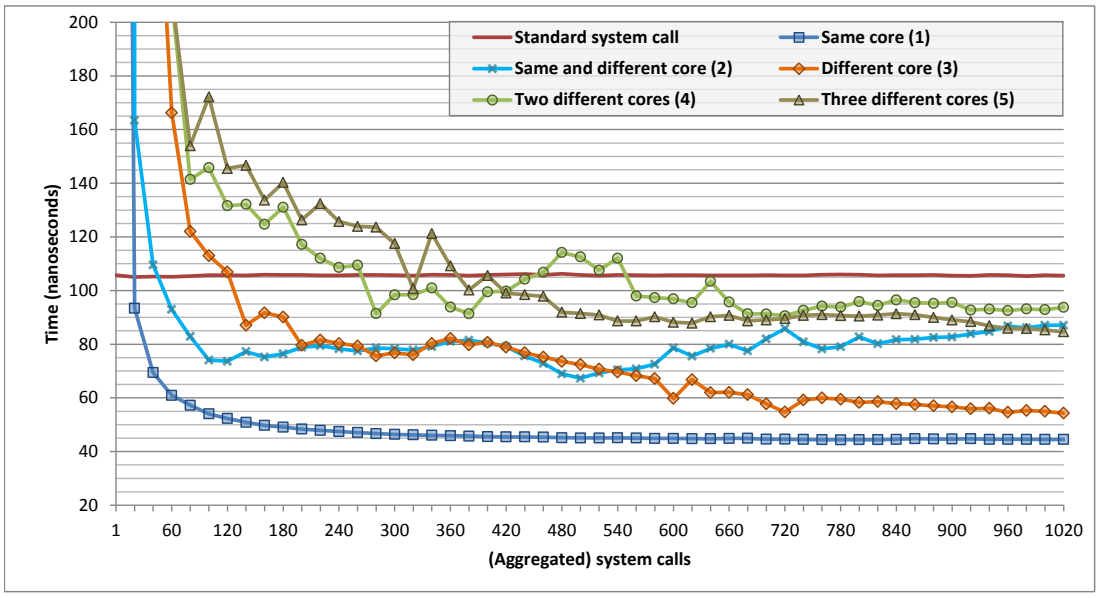

Figure 6.2: Remote-core execution is more expensive and particularly benefits from large batch sizes. Serial remote-core execution can reach up to 49% reduction. Parallel execution achieves only an average of 16% for a maximum batch size.

Cluster calls allow more complex scenarios that include remote-core execution of system calls. These scenarios show different characteristics in overhead reduction. The benchmark implements a test for mixed system call execution between the client's core and a single different (remote) core

(scenario two) as well as tests for the execution on up to three different cores (scenario three - five). All scenarios are realised through a change in the worker pool's configuration (number of workers, concurrency and affinity). Creating two worker threads with a concurrency of two and setting the affinity of the worker pool to the cores two and three, reflects for example the fourth scenario. Figure 6.2 illustrates the corresponding measurements. For better comparability, the overhead measurements of the traditional system call interface and those of the single core scenario are also included.

The overhead for small batch sizes is much higher if a system call is executed on a different core. For a single different core the overhead is more than 13x higher than for the first scenario. For three different cores the cost is still 10x higher (for a better overview, these high marks are not shown in the figure). The number of system calls that need to be aggregated to benefit from cluster calls also increases. Execution on a single different core needs a batch size of at least 120 to be as efficient as the traditional interface. What is most noticeable is that for scenarios that include parallel execution (two, four and five) the overall progress in overhead reduction is far more unpredictable and convoluted than in the first or second scenario. Nevertheless, for parallel execution cluster calls still reach an average overhead reduction of 16% with an average execution time of 88ns. In contrast, serial execution on a remote core comes (for large batch sizes, i.e. greater 600) close to the reduction achieved with the execution on the same core (42% - 49% overhead reduction).

Table 6.2 summarizes the most important results. The last column indicates the average reduction that could be achieved in the given scenario. However, this value does not include results where no overhead reduction was observed.

	Scenario	Overhead (1 call)	BEP* (calls)	Overhead (1024 calls)	Average reduction
Serial	Traditional	105ns	-	105ns	-
	Same core	997ns	17	44ns (58%)	55%
	Different core	13710ns	120	54ns (49%)	35%
Parallel	Same and different core	11864ns	40	87ns (17%)	25%
	Two different cores	10914ns	240	93ns (11%)	10%
	Three different cores	10643ns	320	84ns (20%)	14%

Table 6.2: Performance Overview (* Break-even point)

6.3.2 Server Benchmark

The server benchmark examines how cluster calls perform in a real world application and how they compare to conventional server thread architectures. As already described in section 5.5, the evaluation web server used for the benchmark implements four distinct thread models: The *dedicated thread (1)* model, which creates a dedicated thread for every request, the *thread pool (2)* model that uses persistent threads in a fixed thread pool, the *I/O completion (3)* model, which is architectured around an I/O completion port and the *cluster call (4)* model, which implements the cluster call mechanism presented in this work. Except for the third model, which employs asynchronous disk and network I/O, all models use synchronous operations for request processing.

Two characteristics measured by the benchmark are the achievable throughput (HTTP requests per second) and the average total request time (latency + processing time) of each model at a given number of parallel requests. The ApacheBench (2.3) tool is used to obtain these values. For that purpose, the tool generates traffic on the server by sending an appropriate number of concurrent HTTP GET requests. The requested page is purely static and has a size of 4.729 bytes. The tool does not request any elements (such as images) embedded in the page. To reduce interferences with the server application, ApacheBench is executed on a dedicated core (core 4). The server is bound to the three remaining cores.

The configuration of the thread models is summarized in table 6.3. Except for the cluster call model, all models use the full set of available cores (1,2 and 3) for its threads. Consequently, the concurrency of the I/O completion port which is used to receive requests is set to 3 in these models. The number of threads used has been determined experimentally to offer a good average performance.

Model	# Threads / Clients	# Fibers	Affinity (Cores)	Receive port concurrency
Dedicated thread	1 per req.	-	1,2,3	3
Thread pool	200	-	1,2,3	3
I/O completion	20	-	1,2,3	3
Cluster call	1	1024	1	1

Table 6.3: General Model Configuration

The cluster call model creates a single cluster call client which hosts a pool of 1024 fibers. The client is bound to core 1. The two remaining cores (2 and 3) each run a worker pool dedicated to a specific set of system calls. Table 6.4 gives an overview of the pool's configuration.

Pool	# Worker	Affinity (Cores)	Concurrency	Wait mode
1	10	2	1	WaitSome
2	40	3	1	WaitSome

Table 6.4: Worker Pool Configuration

The first worker pool is used to receive new requests from the I/O port by exclusively processing `NtRemoveIoCompletion` system calls. Since the pool's affinity restricts its workers to a single core, the concurrency of the receive port as well as the pool's concurrency are set to 1. The second worker pool runs on core 3 and processes the following system calls: `NtCreateFile` (creates or opens a file on disk), `NtReadFile` (reads data from a file), `NtDeviceIoControlFile` (in this context used for network related operations like receiving or sending data) and `NtClose` (closes handles). These are the only system calls directly used by the fibers. All other system calls are not redirected or aggregated by the cluster call client. The affinity for all components is chosen to minimize the interference between the workers and to increase the parallelism between the workers and the client.

The web server uses two pools because `NtRemoveIoCompletion` blocks the calling worker until a new request is received (which may never happen). In a single pool layout all workers could therefore block for an indefinite amount of time whichs leads to the starvation of other pending calls (like a file read). Consequently, the HTTP request whose processing originally caused such a call would timeout. The web server avoids this situation by seperating `NtRemoveIoCompletion` calls from other aggregated system calls. Another solution is to use as many workers as there are fibers (i.e. pending system calls). But this in turn is accompanied by a higher resource footprint.

The wait mode of both worker pools is set to WaitSome (see section 5.3.3). Because the system call limit and the timeout setting belonging to this wait mode greatly determines the behaviour of the web server, two examplary configurations are benchmarked: The 'normal' setting uses a system call

limit of 1000 and a timeout of 14ms, whereas the 'small' setting uses a limit of 400 system calls and a timeout of 4ms. These settings define how long the cluster call client waits in the kernel after delivering aggregated system calls.

The first test conducted examined the throughtput of each model for a given number of parallel requests. The results are illustrated in figure 6.3.

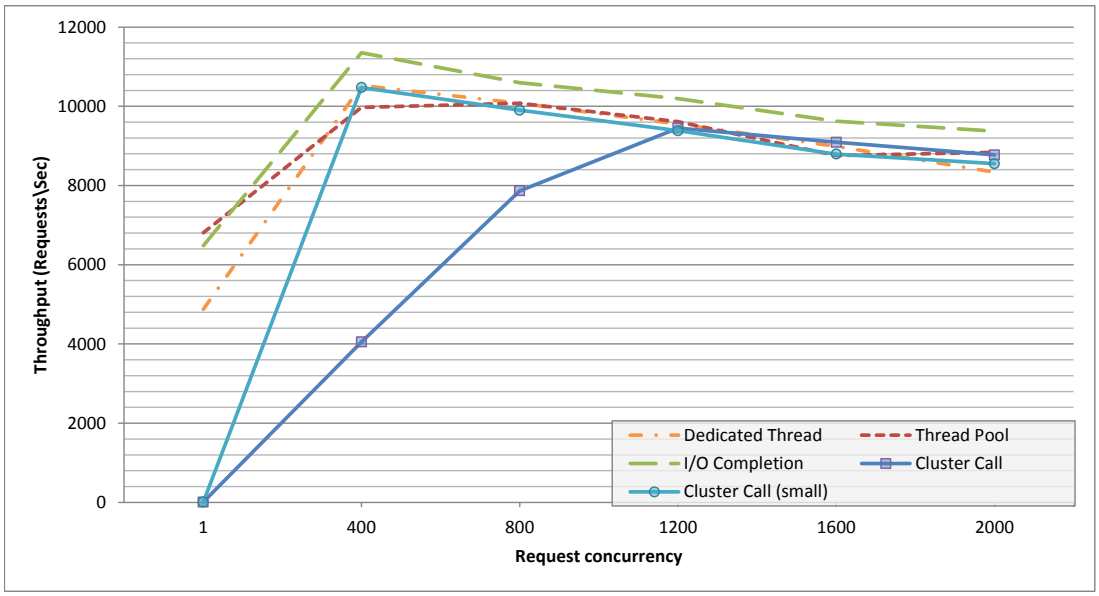

Figure 6.3: Cluster calls perform similar to other synchronous I/O models, but highly depend on proper wait mode configuration. The asynchronous model outperforms all other models.

Although the thread pool model offers slightly better throughput (up to 5%) if only a few requests are served concurrently, the asynchronous model offers the highest throughput in the overall picture. On average it reaches about 6% more throughtput for every reading than the best of the other models. The dedicated thread model shows the worst performance of the conventional thread models for a low request concurrency (about 28% less throughput than the thread pool model) but catches up with the thread pool model for higher values of concurrency.

The overall trend for the conventional thread models is very similar. The throughtput increases up to a request concurrency of 400, reaching 11350 for the I/O completion model and 9970 for the thread pool model, and then slowly declines down to 9372 and 8335 requests per second respectively

at 2000 concurrent requests. While the dedicated thread pool model has a peak-throughput (at 400) that is about 450 requests per second higher than that of the thread pool model, it serves about 1000 requests less at a concurrency of 2000.

As the benchmark shows cluster calls perform similar to conventional synchronous I/O models, although they employ much less kernel-managed threads. The most noticeable difference however is the dramatically reduced throughput for small concurrency values. This reduction is a direct effect of the wait mode configuration. If not enough requests arrive at the server, the wait mode's timeout determines how long the client waits in the kernel (for finishing `NtRemoveIoCompletion` system calls, i.e. new requests). This of course, has a tremendous impact on the number of requests that can be served within a defined period of time. In the worst case, every single system call made by a fiber needs to timeout before request processing in user-mode can continue. On the other hand, as soon as the system call limit defined in the wait mode can be satisfied through a sufficient high load on the server, cluster calls can compete with conventional synchronous models. Consequently, the normal cluster call model needs a higher server load (>1000 parallel requests) than the small cluster call model (>400) to catch up. But then the peak-throughput for the small cluster call model for example is practically the same as for the dedicated thread model (10475) and thereby outperforms the thread pool model in this regard. An important observation is that the normal cluster call model outperforms the small cluster call model by an average of about 200 requests per second for concurrency values higher than its wait mode limit.

The total processing times and latencies for each model are summarized in table 6.5. The total processing time is not necessarly the time that is needed by the server to build the reply, but only measures the time between the client's connection and the receiving of the last reply byte. Therefore, the total processing time also includes the latency.

While the asynchronous model offers the best overall characteristics, the dedicated thread model performs best in the group of conventional synchronous models when looking at the latency. The total processing time however, is very similar to that of the thread pool model. The thread pool model generally suffers from a high latency that mostly determines its processing time. The cluster call models again show worse characteristics for concurrency values below the limit configured in their wait mode. Apart from that, each configuration performs in its optimal range (small: 400 -

	Request concurrency					
Model	1	400	800	1200	1600	2000
Dedicated thread	0	38/22	79/49	125/66	177/100	239/148
Thread pool	0	40/29	79/69	125/113	182/170	225/215
I/O completion	0	35/18	75/38	113/58	165/89	212/104
Cluster call	98/31	99/33	101/38	126/93	175/131	225/165
Cluster call (small)	98/31	38/22	80/57	127/93	180/122	232/150

Table 6.5: Total processing time/latency (ms): Processing time similar for all models. Cluster calls suffer from wait mode configuration at the beginning. Latency is best in asynchronous model. Cluster calls generally offer better latency than the thread pool model.

1000 / normal: 1000 - 2000) similar or even better than the conventional synchronous models. Typically, the latency lies between the dedicated thread model and the thread pool model, while the processing time tends to the better one. The higher throughput (and lower processing times) of the normal cluster call model for high concurrency values compared to the small configuration however come at the expense of a slightly higher latency.

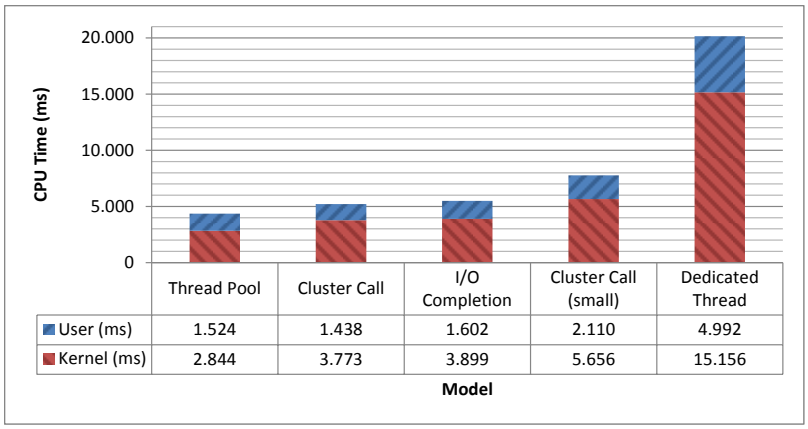

Figure 6.4: CPU time (ms): The dedicated thread model consumes dramatically more CPU time than the other models. Cluster calls however are competitive.

The benchmark also examined the CPU time needed for each model to process 200000 requests at a concurrency of 2000. The CPU usage is an important factor in regard to scalability. The results are illustrated in figure

6.4. The measurements show that the dedicated thread model needs noticeable more CPU time to achieve its throughput and latency compared to the other models. This is caused by the overhead of continuous thread creation and termination. This clearly limits the scalability of this model. In addition, the high number of needed threads is also a limiting factor, considering the amount of process and system resources consumed. In fact, the dedicated thread model cannot sustain higher concurrencies than measured by this benchmark because of the address space used by the thread stacks (default stack size in 32 bit address space).

Cluster calls show similar CPU usage than the other more efficient thread models. The test however underlines the previous measurements in that the small configuration performs worse for high concurrency values. This is a direct result of the smaller batch sizes, which lead to less per-system call overhead reduction.

A last aspect examined by the benchmark is the amount of system calls that can be saved through the use of cluster calls. To process 200000 requests the fibers make a total of 1401025 system calls (about 7 calls per request). In its optimal range (400 - 1000) the small configuration can reduce this number to an average of only 3572 real system calls. This results in an cluster call length of 391 aggregated system calls, which approximately corresponds to the wait mode setting (400). The normal configuration on the other hand can even further reduce the amount of real system calls to an average of about 1589 in its optimal range (1000 - 2000), which leads to a batch size of 883 aggregated system calls. This number is slightly smaller than the wait mode setting (1000) and is caused by the timeout of 14ms. Nevertheless, the higher system call limit allows to benefit from fluctuations in the system call processing speed. To sum up, both configuration allow to save more than 99% of kernel boundary crossings (normal: 99,89%; small: 99,75%).

6.4 Discussion

The overhead benchmarks show that after the high initial overhead, cluster calls are able to reduce the per-system call overhead by up to 58%. However, the worker pool configuration plays an important role and highly determines the efficiency. The measurements reveal that the current implementation performs poorly if parallel execution of system calls is done through the use of multiple CPU cores. Although cluster calls are still able

to reduce the per-system call overhead, the average reduction for these scenarios is noticeably worse. This leads to the conclusion that the current implementation shares too many data structures between CPUs and that even the lock-free atomic operations used to manipulate those data structures are too costly in this context (every system call needs up to three atomic operations, depending on the worker pool configuration and batch size). The fluctuations in the measurements for the parallel scenarios are most probably also caused by this intertwined nature of operation. This can become a problem in regard to scalability to future many-core architectures, where it is feasible to spread a worker pool over potentially hundreds of CPU cores.

The overhead benchmark also shows that ff the asynchronous execution of system calls is needed and the expected batch size is high, it is a good idea to shift the system call processing to a dedicated core. This configuration performs good in regard to the overall overhead reduction and enables the concurrent execution of user-mode code through one or more cluster call clients.

The server benchmarks reveal that cluster calls are a competitive thread model for server type applications that use synchronous I/O. They offer similar throughput like the dedicated thread model or the thread pool model and generally tend to the better one for different request concurrencies. The latency in addition lies between the one offered by the conventional synchronous models. Cluster calls also show good scalability in terms of CPU usage and consume less resources (especially threads) than its synchronous competitors. The evalutation however also shows that the performance is highly dependend on a proper worker pool configuration. This is especially the case for the wait mode settings. Depending on the requirements of the server application wrong settings can lead to performance problems if the request concurrency falls below the specified limit. In the current implementation cluster calls therefore need careful planning and work best if the server load is constant and predictable.

6.5 Summary

In this chapter the performance of cluster calls was examined. The chapter started with a brief outline of the evaluation platform and then presented results of the overhead and server benchmarks. The examinations revealed

that cluster calls can reduce the per-system call overhead by up to 58% and save over 99% of kernel boundary crossing in a server application. In addition, the evaluation showed that the hybrid thread model can compete with conventional synchronous thread models in terms of throughput, latency and CPU usage. Nevertheless, more work is needed to optimize the current implementation in regard to multi-core performance. The strong dependency on proper configuration for good performance can also lead to problems in real world scenarios.

Chapter 7

Conclusion

The main objective of this work was to reduce the per-system call overhead of the traditional system call interface through the use of system call aggregation. For that purpose, a new mechanism named cluster calls was presented. Cluster calls take the advantages of system call batching and exploit the parallelism of server applications by leveraging a hybrid thread model to overcome the disadvantages (like serial execution) of previous solutions. Cluster calls also expose various new features for application optimization like core specialization and system call prioritization.

Although the current implementation needs further improvement, the evaluation showed that the main objective could be achieved. Cluster calls are capable of reducing the per-system call overhead by up to 58%. Furthermore, the combination of system call aggregation and hybrid thread model can avoid over 99% of all kernel boundary crossings in a typical real world server application. This allows cluster calls to compete with conventional server thread architectures based on synchronous I/O in terms of throughput, latency and CPU usage.

7.1 Future Work

Future systems tend to have far more cores than computers available today (e.g. than the evaluation platform). Windows Server 2008 R2 already scales up to 256 cores and it can be expected that this number will rise in the future. A software's scalability to many-core architectures therefore plays an increasing role especially in the area of server applications. The evaluation however showed that the current implementation of cluster calls needs improvement in this regard.

Another interesting area for future work lies in the enhancement of the wait logic. The current implementation relies to much on proper configuration and thereby limits the use of cluster calls in practice. A possible solution might be to dynamically adapt the wait mode settings to the current server load by measuring the rate of system call completions. Future research is needed to evaluate such mechanisms.

Cluster calls expose user-level threading to Windows applications and thereby allow developers in principle to customize the threading mechanisms to fit the application. Unfortunately, the current version of the cluster call user-mode library does not offer a possiblity to for example replace the integrated round robin scheduler. This functionality could be integrated into a future version of the library. Furthermore, an interface for the configuration of the system call redirection logic is needed.

Another improvement of the user-mode library would be to replace fibers as scheduling entities with more advanced user-level threads (see section 2.3.4) and to expose a more complete user-mode threading package that includes further functionality (e.g. more synchronization mechanisms).

Bibliography

[1] Advanced Micro Devices Inc. (AMD). *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. Nov. 2009.

[2] Thomas E. Anderson et al. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*. In ACM Transactions on Computer Systems (TOCS) Volume 10 Issue 1. Feb. 1991.

[3] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*. Sept. 2010.

[4] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3: System Programming Guide*. Sept. 2010.

[5] Intel Corporation. *Using the RDTSC Instruction for Performance Monitoring*. 1997. URL: www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf.

[6] *Easy Hook - The reinvention of Windows API Hooking*. (Used version: 2.6). URL: http://easyhook.codeplex.com/.

[7] John Gulbrandsen. *How Do Windows NT System Calls REALLY Work?* (Accessed on: Nov. 2010). Aug. 2004. URL: http://www.codeguru.com/cpp/w-p/system/devicedriverdevelopment/article.php/c8035.

[8] John Gulbrandsen. *System Call Optimization with the SYSENTER Instruction*. (Accessed on: Nov. 2010). Oct. 2004. URL: http://www.codeguru.com/cpp/w-p/system/devicedriverdevelopment/article.php/c8223.

[9] Ken Henderson. *The Perils of Fiber Mode*. (Accessed on: Nov. 2010). Feb. 2005. URL: http://msdn.microsoft.com/en-us/library/aa175385(SQL.80).aspx.

[10] Galen C. Hunt and James R. Larus. *Singularity: Rethinking the Software Stack*. In ACM SIGOPS Operating Systems Review, vol. 41, no. 2, pp. 37-49, Association for Computing Machinery, Inc. Apr. 2007.

[11] Christophe Nasarre Jeffrey Richter. *Windows via C/C++, Fifth Edition*. Microsoft Press, 2007.

[12] Anthony Jones and Jim Ohlund. *Network Programming for Microsoft Windows, Second Edition*. Microsoft Press, Feb. 2002.

[13] Colm MacCárthaigh. *Scaling Apache 2.x beyond 20,000 concurrent downloads*. July 2005.

[14] Brian D. Marsh et al. *First-Class User-Level Threads*. In Proceedings of the thirteenth ACM symposium on Operating systems principles. 1991.

[15] Microsoft. *MS Windows NT Kernel-mode User and GDI White Paper*. (Accessed on: Nov. 2010). URL: http://technet.microsoft.com/en-us/library/cc750820.aspx.

[16] Microsoft. *MSDN Library: Access Control*. (Accessed on: Nov. 2010). Nov. 2010. URL: http://msdn.microsoft.com/en-us/library/aa374860(v=VS.85).aspx.

[17] Microsoft. *MSDN Library: Fibers*. (Accessed on: Nov. 2010). Nov. 2010. URL: http://msdn.microsoft.com/en-us/library/ms682661(VS.85).aspx.

[18] Microsoft. *MSDN Library: Handles and Objects*. (Accessed on: Nov. 2010). Nov. 2010. URL: http://msdn.microsoft.com/en-us/library/ms724457(v=VS.85).aspx.

[19] Microsoft. *MSDN Library: I/O Completion Ports*. (Accessed on: Nov. 2010). Nov. 2010. URL: http://msdn.microsoft.com/en-us/library/aa365198(VS.85).aspx.

[20] Microsoft. *MSDN Library: Object Names*. (Accessed on: Nov. 2010). Nov. 2010. URL: http://msdn.microsoft.com/en-us/library/ms684292(v=VS.85).aspx.

[21] Microsoft. *MSDN Library: Object Namespaces*. (Accessed on: Nov. 2010). Nov. 2010. URL: http://msdn.microsoft.com/en-us/library/ms684295(v=VS.85).aspx.

[22] Microsoft. *MSDN Library: Securable Objects*. (Accessed on: Nov. 2010). Nov. 2010. URL: http://msdn.microsoft.com/en-us/library/aa379557(v=VS.85).aspx.

[23] Microsoft. *MSDN Library: Synchronization Objects*. (Accessed on: Nov. 2010). Nov. 2010. URL: http://msdn.microsoft.com/en-us/library/ms686364(VS.85).aspx.

[24] Microsoft. *MSDN Library: User-Mode Scheduling*. (Accessed on: Nov. 2010). Nov. 2010. URL: http://msdn.microsoft.com/en-us/library/dd627187(VS.85).aspx.

[25] Microsoft. *MSDN Library: Wait Functions*. (Accessed on: Nov. 2010). Nov. 2010. URL: http://msdn.microsoft.com/en-us/library/ms687069(v=VS.85).aspx.

[26] Microsoft. *Windows Research Kernel*. (Accessed on: Dec. 2010). URL: http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.mspx.

[27] Mohan Rajagopalan et al. *System Call Clustering: A Profile-Directed Optimization Technique*. 2003.

[28] Mark E. Russinovich and David A. Solomon. *Windows Internals, Fifth Edition - Covering Windows Server 2008 and Windows Vista*. Microsoft Press, June 2009.

[29] Mark E. Russinovich and David A. Solomon. *Windows Internals, Fourth Edition - Microsoft Windows Server 2003, Windows XP and Windows 2000*. Microsoft Press, Jan. 2005.

[30] W3 Schools. *OS Platform Statistics*. (Accessed on: Nov. 2010). URL: http://www.w3schools.com/browsers/browsers_os.asp.

[31] Livio Soares and Michael Stumm. *FlexSC: Flexible System Call Scheduling with Exception-Less System Calls*. Oct. 2010.

[32] Wikipedia. *Interix*. (Accessed on: Nov. 2010). URL: http://en.wikipedia.org/wiki/Interix.

[33] Wikipedia. *Windows Services for UNIX*. (Accessed on: Nov. 2010). URL: http://en.wikipedia.org/wiki/Windows_Services_for_UNIX.