MMLite: A Highly Componentized System Architecture

Johannes Helander and Alessandro Forin
Microsoft Research

Abstract

MMLite is a modular system architecture that is suitable for a wide variety of hardware and applications. The system provides a selection of object-based components that are dynamically assembled into a full application system. Amongst these components is a namespace, which supports a new programming model, where components are automatically loaded on demand. The virtual memory manager is optional and is loaded on demand. Components can be easily replaced and reimplemented. A third party independently replaced the real-time scheduler with a different implementation. Componentization reduced the development time and led to a flexible and understandable system.

MMLite is efficient, portable, and has a very small memory footprint. It runs on several microprocessors, including two VLIW processors. It is being used on processors that are embedded in a number of multimedia DirectX accelerator boards.

1 Introduction

The progressive computerization of our society involves a number of diverse computing platforms beside the general-purpose computer:

- Embedded control systems, including consumer devices, intelligent sensors and smart home controls.
- Communication-oriented devices such as digital cell phones and networking infrastructure.
- Programmable peripherals and microcontrollers.

In all these cases, the general-purpose platform approach is either not applicable, or it is prohibitively expensive. The microprocessor might be a DSP, a VLIW, or a micro-controller; the memory budget is severely restricted; there might be no MMU; the network connection might be sporadic; and real-time is essential.

Current operating systems are either inflexible, big, lack real-time support, have complex hardware requirements, or are so specialized that good development tools are unavailable and code reusability is low.

In this paper we present MMLite, a system architecture that is suitable for a wide range of applications. Our strategy is to build a system out of minimal but flexible components. Instead of mandating a fixed set of operating system services and hardware requirements, we provide a menu of well-defined components that can be chosen to compose a complete system depending on hardware capabilities, security needs, and application requirements. Components can be selected at compile time, link time, and run-time. Components can be transparently replaced while in use, via a mechanism we call *mutation*.

The componentization makes it easier to change the implementation of a component without affecting the rest of the system. Minimalism makes it possible to use the system with severely restricted hardware budgets. It also forces the system to be understandable and adaptable. Software components, when possible, are not tied to a particular layer of the system, but can be reused. For example, the same code that implements the system physical memory heap is used to provide application heaps over virtual memory.

We componentize the system more aggressively than any previous system. This includes the virtual memory system, IPC, and the scheduler in addition to filesystems, networking, drivers, and security policies.

The rest of the paper is organized as follows: Section 2 describes the system architecture. Sections 3 and 4 describe the implementation of a few major components. Results are presented in section 5, related work in section 6, and conclusions in section 7.

2 Architecture

C++ and Java provide objects at a very fine granularity level, and they are very successful with application programmers. Unfortunately, both languages confine their objects to a single address space. Object Linking and Embedding (OLE) [Brockschmidt95], CORBA [OMG98], and other similar systems extend objects across address spaces and across machine boundaries. OLE seamlessly integrates independently developed components. When editing an Excel spreadsheet inside a Word document it is in fact the Excel process that operates on objects inside of Word’s address space. Unfortunately, OLE only works for user mode applications. MMLite takes an objects everywhere approach, and extends object-orientation both across address spaces and across protection levels.

2.1 Component Object Model

MMLite components contain code and other metadata for classes of objects. When a component is loaded into an address space it is instantiated. The instantiated component creates object instances that communicate with other objects, potentially in other components. The objects expose their methods through Component Object

Eight ACM SIGOPS European Workshop, September 1998, Sintra, Portugal.

Model (COM) [Brockschmidt95] interfaces. MMLite objects can be made available to other components by registering them in a namespace. Namespaces are similar to filesystem directories but are not limited to just files. Threads execute code and synchronize through Mutexes and Condition variables. System components are typically written in C or C++ but there is no fundamental bias towards any particular language.

Every COM object has a virtual method table and at least the three methods derived from the base interface (the IUnknown): QueryInterface for agreeing on the interface protocols, and AddRef and Release for reference counting. Specific interfaces have additional methods to do the actual work. In addition, a constructor is usually provided.

The object model enables late binding, version compatibility and checking, transparency through proxies, cross language support, and is reasonably lightweight and efficient. Each object has a method table pointer and a reference count. Each call adds one indirection for fetching the actual method pointer.

Garbage collection is done through reference counting. When Release is called for the last reference, the implementation can finalize and deallocate the object. Even if reference counting has its limitations, it is convenient in a system environment due to its simplicity. Interaction with objects using other garbage collection models can be achieved through proxies that intercept the IUnknown methods to update their root sets.

MMLite component implementations are rarely aware of their intended system layer. The same code can be used in different address spaces or contexts and can be nested. A filesystem can be applied to a file provided by another filesystem as well as to one provided by a disk driver. A heap can be applied to any memory: physical memory, memory allocated from another heap, or memory provided by a virtual memory manager. The loader loads modules into any address space.

2.2 Communication and Namespaces

If the object that is the target of a method is in a different machine or a different address space, and thus can not be called directly, a proxy is interposed for delegation. Instead of calling the actual object, the client will call the (local) proxy object. The proxy marshals the parameters into a message and sends it where the actual object is located. There the message is received and dispatched to a stub object. The stub unmarshals the parameters and calls the actual method. On the return path the stub similarly marshals any return values and sends them in a message back to the proxy that in turn unmarshals and returns.

Aside from taking longer to execute, the remote object call through a proxy looks exactly the same as a local call directly to the actual object. Not only is the implementation of the server transparent to the client, but the location as well.

Namespaces are used to let applications gain access to objects provided by other components. A namespace is like a filesystem directory tree, except it can hold any kind of objects, not just files. Namespaces can themselves be implemented by different components, including a filesystem that exports its directories as sub-namespaces, and files as registered objects. Namespaces can be registered into other namespaces, extending the directory tree. Location transparency of all objects automatically makes namespaces distributed. Namespaces can be filtered for access control or for providing different views to different applications. There is no limit as to the number of namespaces. A component can gain access to its namespace through a call to CurrentNamespace(). In a minimal system all applications share the same boot namespace.

When an application looks up a name in the namespace, it obtains a reference to the object: a local direct reference in case the object is local, or an automatically created proxy if the object is remote. The IPC system is responsible for creating proxies, handling the delegation to remote objects, and reference counting. A namespace is also free to create objects on demand, as is the case for a filesystem. The namespace only needs to handle the IUnknown interface. It is up to the application to obtain the proper interface directly from the object, using the QueryInterface method.

We implemented a demand-loading namespace that supports the following new programming model. The *main()* entry point for an image is a constructor that returns an object. When an application tries to bind to a name that does not exist, the namespace invokes the loader, which looks for and instantiates a component with the given name. The loader then invokes the component's entry point, registers the resulting object in the namespace, and returns it to the application. When the application releases its last reference to the component the namespace can unload the component or choose to keep it cached.

2.3 Execution Model

Components have code, static data, a stack and a number of dynamic objects. A heap object allows dynamic memory allocations. The stack is pointed to by the stack pointer register; it is allocated from the heap. In a physical memory system the initial size of the stack is also the maximum size of the stack; every byte has to be paid for by real memory. Thus in an embedded application the stack size must be chosen carefully. Most compilers can generate stack checks at function entry, to guard against stack overflows. In a virtual memory system, the stack does not have to be backed by real memory, which can be allocated on demand. The stack only consumes virtual address range and can thus be allocated liberally. A real-time application might still want to pre-allocate all memory in order to avoid time fluctuations. In this case the existence of virtual memory does not affect the stack.

Memory for code and static data is also allocated from the heap. Code can be placed anywhere in memory if it is either position-independent (pc-relative) or relocatable. The Microsoft Visual C++ compiler, for instance, creates a compressed relocation table that the run-time loader uses to fix any references if the executable was placed in a different place in memory than it was linked for. All compilers for embedded use provide similar functionality, although the specific image formats and relocation schemes differ.

Unfortunately, we have found that most compilers do not support reentrancy. If the code in an image is not reentrant, it is still possible to execute multiple instances of the same image in the same address space. The code and data are simply loaded multiple times, each time relocated differently.

If the relocation information is not present, and a component virtually overlaps with another component it cannot be executed in the same address space. In this case a new address space is required, which in turn requires virtual memory.

2.4 Mutation

An object consists of an *interface*, an *instance pointer*, an *implementation*, and some *state*. The interface is a list of *methods*. The instance pointers and interfaces are exposed to other objects; the state and the implementation are not. Worker threads execute implementation code that accesses and modifies the state. Once an object instance has been created, the instance pointer, interface, and implementation are traditionally immutable, only the state can be changed by method calls.

In MMLite we allow run-time changes to the ordinarily immutable part of an object, even while the object is being used. We call *mutation* the act of atomically changing an ordinarily constant part of an object, such as a method implementation. The thread performing the mutation is called a *mutator*.

A mutator must translate the state of the object from the representation expected by the old implementation to the one expected by the new implementation. It must also coordinate with worker threads and other mutators through suitable synchronization mechanisms. Transition functions capture the translations that are applied to the object state and to the worker thread's execution state. In order to limit the amount of metadata, execution transitions only happen between corresponding *clean points* in the old and new implementations.

A number of mechanisms can be implemented using mutation. Interposition is done via replacement of the object with a filter object that points to a clone of the original object. A dynamic software upgrade would replace the incorrect implementation of a method with the corrected one. Run-time code generation might use a stub implementation as a trigger. Mutation can be used to replace generic code with a specialized version that exploits partial evaluation by treating ordinarily non-constant state as immutable. Once the optimistic conditions are no longer true, mutation allows reverting back to the generic code. Execution profiling might indicate that a different implementation would perform better, and trigger a mutation. Object mobility is realized by turning objects into proxies and vice versa.

One example where we found mutation to be useful was in device drivers. In one configuration on the x86 we used minimal floppy and disk drivers that called BIOS (ROM) functions to do the work. A loadable driver would later take over and mutate the BIOS driver with a real driver, transparently to the filesystem.

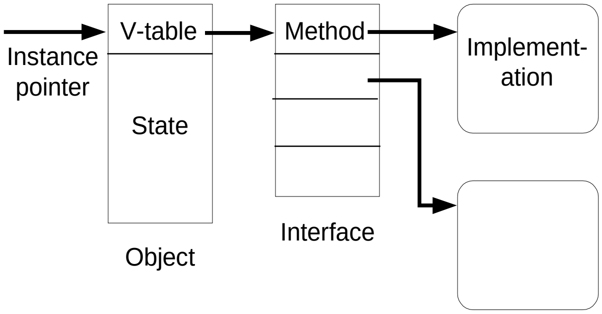

Figure 2: A run-time object representation.

The synchronization mechanisms suitable for implementing mutation can be divided into three groups:

- Mutual exclusion: Mutation cannot happen while workers are executing methods of the object to be mutated. The implementation can be a read-write lock, disabling preemption on a uniprocessor, or a holding tank [Cowan96] with reference counting. Mutual exclusion is simple in that there is no worker state associated with the object when mutation is allowed to happen.
- Transactional: Roll back the workers that are affected by mutation. Mutators and workers operate on an object transactionally and can be aborted when necessary.
- Swizzling: Modify the state of the workers to reflect mutation. Instead of waiting for workers to exit the object or forcing them out, the third mechanisms just suspends them. The mutator then modifies the state of each worker to reflect the change in the object.

2.5 Nesting

The environment provided by MMLite can be viewed as a virtual machine for C or C++ based COM programs. These programs may include other virtual machines that provide suitable environments for other languages or different environments. In this way MMLite can provide arbitrary virtual machine environments through nesting. The nested virtual machines might employ type-safe languages and software fault and trust insulation. Alternatively nested virtual machines can be implemented in terms of hardware by means of a virtual memory system.

The nested virtual machines can be loaded on demand into any other address space. Software solutions are the only means of insulation in a system without hardware protection. Even with hardware protection the software solutions might sometimes win in performance by avoiding expensive domain crossings.

Any virtual machine running in the MMLite environment can make external objects available to its nested applications via proxies, perhaps with security filtering. It can similarly export its nested objects to the outer world via stubs.

2.6 Selection of System Components

What components should be part of a deployed system depends on the applications themselves and their interface requirements, application memory requirements, security requirements, and the target hardware capabilities. Flexible loading of modules was an important design goal for our system. The loading of components can be deferred until they are actually used by an application. Device drivers and run-time services typically fall into this category. Others can be loaded just prior to running an application, such as virtual memory for untrusted applications. Most services will terminate themselves when they are no longer needed. The structure of the system might change radically during execution due to external events.

Drivers and virtual memory can not be used when the hardware to support them is not present. An application that tries to use them will look them up in the demand-loading namespace. The lookup operation fails, either because the driver is absent or it returns a NULL pointer instead of a valid IUnknown during initialization.

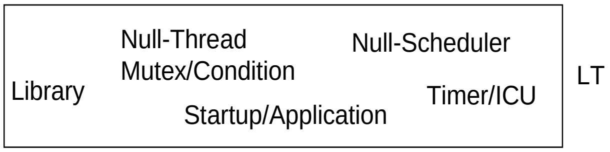

Figure 1: A minimal system configuration. Components are loaded at link time (LT).

Figure 1 shows a minimal system configuration that can be used in a watch. Figure 2 shows a larger system configuration that can be used in a cell phone. Applications can run within the physical address space, within a separate address space, and within a virtual machine. The IPC system uses the network and virtual memory mappings as its transports.

2.7 Virtual Memory

Unlike most existing operating systems, in MMLite the support for virtual memory is not an integral part of the system. The system can function with or without it, and it executes the same binaries. The virtual memory manager is a component like any other, and is loaded dynamically on demand.

Loading or unloading of the virtual memory system does not interfere with applications already running, or started later on in the physical memory space. Once the virtual memory system has been started, new components can be loaded into any address space. Component code may be shared between different address spaces, as is the case with shared libraries. For instance, any code loaded into the physical memory space is made visible to applications regardless of their address space. There is nothing secret in the systems code, so there is no security problem.

Virtual memory might be required for:

- Security reasons, when a program is not trusted. The virtual memory system implements firewalls between applications.
- Covering for common programming errors such as NULL pointer references and memory leaks.
- Creating a sparse address space. Often leads to better memory utilization with fragmented heaps.
- Paging: provides more memory than available, working set adaptation, and mapped files.
- Safe and flexible memory sharing: Copy-on-write for libraries, shared memory windows.
- Running non-relocatable executables as described above.
- Implementing garbage collection and other protection tricks.

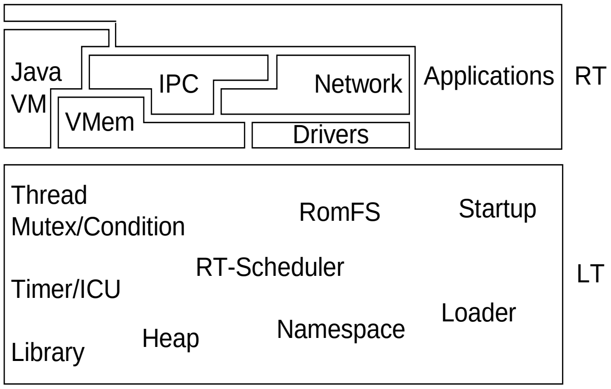

Figure 2: A sample system configuration. Link time (LT), and run time (RT) loadable components.

2.8 Interprocess Communication

An IPC system is needed if applications are to be run in separate address spaces. Otherwise the applications can not talk to each other or to system services. An IPC system allows:

- Communication between address spaces within a machine.
- Communication between applications in different machines in a distributed environment.
- Graceful termination and cleanup of applications even within one address space.

Cleanup involves releasing the memory held by an application. It also involves closing all references into and out of the application’s objects. A level of indirection is needed for bookkeeping and for providing a cutoff point. This level of indirection is what an IPC system provides.

Our IPC system implements the COM model. It is possible, however, to replace it with another communication model for applications that expect a different model. Components implementing various communication paradigms can be loaded into the system as needed.

3 System Components

This section describes the base menu of MMLite system components. Other components are described in section 4. All of them have been implemented and tested. For sizes, see Table 1 in section 5.

- **Heap:** Implements (physical) memory management, allowing dynamic memory allocations with specifiable alignments. The constructor allows creating nested heaps or heaps over virtual memory. Two different implementations are provided.
- **Loader:** Used to load additional components into a running system. Most embedded systems do not provide a loader, and it can be eliminated at link time from our system as well. Multiple image formats are supported. The loader loads images into the same address space, or given a flag and a virtual memory manager, it creates a new address space and loads the image in there.

We make no particular distinction between executables and DLLs (shared libraries). An executable is simply a DLL that exports no other entry points besides *main()*.

- **Support Library, Machine Initialization:** A shared support library includes common base utilities like memcpy, int64, AtomicAdd, CurrentThread, etc. It is used by many system components and is available to all components.

Basic machine initialization code is used at startup and system reset. Most of the machine dependent code of MMLite goes here.

- **Timer and Interrupt Drivers:** A driver for the timer chip is used by the scheduler to keep track of time and for thread pre-emption. A driver for the Interrupt Control Unit (ICU) dispatches interrupts and keeps a registry of interrupt routines, which can be installed and removed by other components. The system has no particular notion of a "device driver" per se. It does enforce strict limits as to what an interrupt routine can do: wakeup a thread.
- **Scheduler:** A policy module that determines which thread should run at any given time. Low-level management of blocking and switching between threads is handled by the thread and synchronization component.

The timer interrupt and thread and synchronization modules call into the scheduler, possibly passing callback functions as arguments.

Four schedulers have been implemented: the null scheduler, a simple round robin scheduler, a constraint based real-time scheduler, and another independently implemented real-time scheduler. The null scheduler is for systems that use only one thread. Constraint scheduling is for consumer real-time applications and is described in [Jones97].

- **Threads and Synchronization:** Basic thread support and synchronization primitives. A thread is created in the address space of the component in which it is started. If there is no virtual memory, the address space is always the physical address space. Threads can block on mutexes and conditions. They can inform the scheduler of their time constraints, but these calls will fail if the scheduler is not a constraint scheduler. The constraint scheduler performs priority inheritance when threads block on mutexes.
- **Namespaces:** A simple boot namespace where applications register objects. A namespace that cooperates with the loader in demand-loading and caching of components. A namespace, used for displaying the status (e.g. running threads) and performance parameters (e.g. execution times) of a system during development. Filesystems are also loadable namespaces.
- **Filesystem:** Used to load additional components during run-time. We implemented *RomFS* for read-only in-memory images (arbitrary files and the system can be merged into one image) and *FatFS* for reading/writing disks. *NetFile* is a simple network filesystem client built on top of sockets.
- **Network:** The complete BSD4.4Lite network protocol code, with minor adaptations. The interface is a COM interface that provides sockets. The protocols operate the network drivers through another interface.
- **Startup Program:** Started once the system has been initialized. It can be a simple command interpreter that configures the system and launches applications, or the (only) application itself.
- **A small Win32 compatibility library** for making it easier to use WindowsNT code in some of the drivers and applications.
- **Atomic queues and a DMA manager** are useful to device driver writers.
- **Virtual Memory Implementation:** Can be viewed as a driver for MMU hardware. It creates virtual memory mappings using physical memory and MMU hardware. Loading and starting the virtual memory manager executable does not interfere with applications already running. Unloading can be done once all references to objects provided by the manager are released. A new one can be started if needed.

A virtual memory space looks like the physical memory space, except it can be larger, doesn't have to be contiguous, can be paged, protected, replicated, and can be (recursively) mapped to other objects.

Our virtual memory manager exports a number of control interfaces that are used to create new address spaces (VSpace), to map address spaces or files to address spaces (VMap), to map address spaces to threads (VView), and to control state and protections (VSpace).

Realistically, any MMU driver will need exclusive control of the MMU hardware. However, other objects implementing the virtual memory interfaces can be interposed between an application and the MMU driver. This way logical virtual memory systems can be arbitrarily composed, for instance stacked like in the Exokernel [Engler95].

The rest of the system is unaware of the virtual memory component, with a few exceptions. A thread must hold a pointer to its VView so that page faults can be resolved within the correct context, and the context switch path must check for a VView change. If the context switch path detects an address space change, it calls a VView method to synchronize the MMU hardware with the change. The virtual memory server registers its trap handler with the ICU driver. The Heap may choose to modify its behavior when running over virtual memory. The loader can only create new address spaces when a virtual memory system is present. The IPC system may utilize virtual memory mappings for data transfers.

4 Applications

Equator Technologies has ported MMLite to its proprietary VLIW media processor, and is using it to run multiple simultaneous multimedia applications, including MPEG2 Decode, AC3, 3D Graphics (D3D) and tele-communications.

The MMLite system has been used at Microsoft in a prototype Talisman [Torborg96] 3D graphics and multimedia card. This involved supporting DirectDraw and Direct3D, DirectSound and a wavetable based software MIDI synthesizer implementing the DLS level 1 specification. An interactive game (Doom) can also run directly on the board, along with a number of other demo programs and regression tests.

We will now look into some of these applications in more detail.

4.1 DirectX

DirectDraw is an interface for a Windows platform that lets applications efficiently create 2D graphics without unnecessary operating system overhead. Direct3D is the equivalent interface for 3D graphics. They both use a combination of direct memory access and in-kernel driver support to do their job. DirectSound is the interface for audio I/O. Collectively, they are known as DirectX.

The Talisman prototype implementation utilizes the multimedia card (Figure 4) to do the bulk of the work. The card contains a processor and local memory, controlled by MMLite. On the card MMLite provides the framework and basic services for the graphics and audio engines.

Two components handle the communication over the PCI bus: MMH on the NT side and MMD on the MMLite side. They send packets to each other. The user application on the workstation interfaces with the DirectX library, which logically makes method calls to objects that reside on the card. The MMH and MMD act as proxies and stubs, and perform the necessary argument marshaling and unmarshaling. A graphics component does the rendering and writes the results into a frame buffer. An audio component uses the on-board A/D converters.

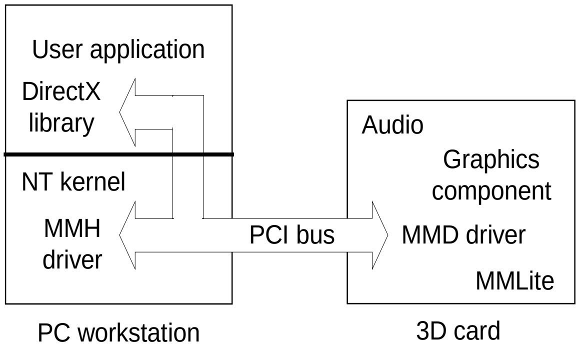

Figure 4: DirectX communication.

MMD uses the demand-loading namespace, and exports this functionality to the MMH side as well. The filesystem delegates any file requests back to MMH that fetches them from the filesystem on the workstation. Constraints are used to schedule computations and device driver level work. Feedback to the graphics component lets it adapt the rendering precision to time availability.

In this setup communication latency is important for the performance of the application using DirectX. On a 90MHz PC we measured in excess of 7,800 RPC/second between a user mode WindowsNT process and a MMLite component, with time predominantly spent in NT.

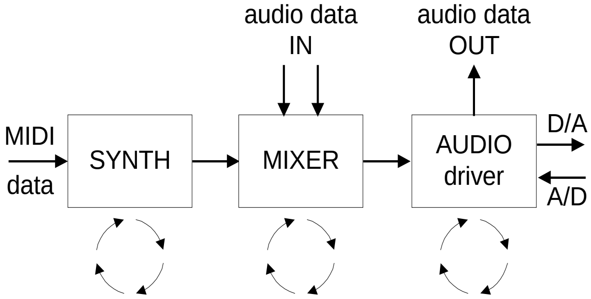

Figure 5: MIDI synthesizer components.

4.2 MIDI Synthesizer

Figure 5 depicts the structure of the MIDI synthesizer components. The SYNTH thread runs periodically and when new MIDI data arrives. It produces an output buffer of 44KHz, 16bit, stereo audio samples. The MIXER thread also runs periodically, and expects to find a new buffer available on each of its input channels. If an input buffer is not present, a buffer of silence data is used instead. Buffers contain data in any number of formats; the mixer adapts frequencies, stereo/mono, and number of bits per sample. The result of the mixing is given to the AUDIO driver. This thread awaits signals from the interrupt routines, and switches buffers. Completed buffers are returned to the mixer.

All of the audio components use time constraints and can without interference run together with DirectDraw and other applications. The overall CPU load when running these components under MMLite on an 80MHz TriMedia processor is 7% for the SYNTH component, and 3% for the remaining audio components.

5 Results

The system currently runs on the x86, ARM, Philips TriMedia, and Equator Technologies' VLIW processors. The Talisman Software Development Kit [SDK97] contains the MMLite kit, and was first shipped in July 1996 to Talisman licensees.

Heap1	2635	Boot NS	1265
Heap2	3420	Dload NS	512
PE loader	4661	RomFS	1417
Library	3799	FatFS	8229
Machdep	2086	NetFile	6944
Timer	1205	Startup	118
ICU	1005	Network	84832
Null-sched	316	Win32Event	634
RR-sched	599	AtomicQueue	415
RT-sched	1228	MMD	1852
Thread	426	VM	17712
Synchro	1090	Doom	285696

Table 1: X86 components and their binary sizes in bytes.

The size of the minimal system in Figure 1 is 10KB on x86, excluding a boot stack. The size of the base system (the LT box in Figure 2) is 26KB on x86, 20KB on ARM. Table 1 lists the sizes of the components of Section 3.

5.1 Development Experiences

In our experience at Microsoft, componentization reduces development time and makes the system flexible and understandable. Because not all components have to be functional at once, porting and development can be incremental. The system also makes little assumptions of the hardware. The Philips TriMedia port was functional in one week, of which five days were spent on learning how to use the development tools. The VLIW architecture of this microprocessor is quite new and different from the ones that MMLite supported at the time.

Equator Technologies has ported the system to their new microprocessor and platform. They told us their experience also was favorable and that the system was reasonably bug free. They used the demand-loading namespace and programming model, and the MMH/MMD pair. They found the separation of machine dependencies was successful. They modified one of the existing loader modules for their image format and implemented a new real-time scheduling component.

6 Related Work

[Ford97] shows how a base set of system components can be composed in different ways to build an operating system kernel. The granularity is fairly coarse, and the techniques are limited to static linking. Components that should be of interest to OS researchers (VM, IPC, scheduling, etc.) cannot be replaced or removed, neither statically nor dynamically. The decomposition is otherwise limited to the "OS" component; it is not meant as a whole-system approach. This work bears similarity to MMLite, but does not go as far in the componentization. It provides a few convenient components, such as bootstrap loader and filesystems, but is mostly concerned in reusing existing device drivers and Unix code. MMLite, on the other hand, componentizes the core services and extends the paradigm to applications.

Chorus [Rozier88] is the only system we know of that can be configured to use either a page-based or a segment-based VM system. MMLite is the first one that can run with or without VM, and dynamically load and unload it — unless, of course, we look at MS-DOS in a very twisted way.

Synthetix [Cowan96] employs a limited form of object mutation for specialization. We have generalized mutation and made it usable in a number of new contexts. The synchronization mechanism presented in [Cowan96] is a holding tank that keeps workers from entering an object that is being mutated. The holding tank can be viewed as an asymmetric read-write lock, but it does not solve the problem of workers that have *already* entered the object. We solved this problem, including cases where the worker blocks.

Rialto [Jones96, Draves97] shows how the COM model can be implemented in the presence of VM, and argues for a unified programming model that is independent of the privilege issue (no user versus kernel distinction). We show how those same principles are beneficial in scaling down a system to cope with resource poor domains.

CORBA [OMG98] forces all calls to go through the object request broker thus penalizing the local case. Real-time support in CORBA is still at the research stage [Yang98].

Componentization and location independence has also been studied in the context of filesystems and network protocols [Maeda93] and in a number of existing embedded systems, such as pSOS [ISI95]. In a typical embedded system there is no loader, components can only be chosen at static link time when the load image is built. Services are extremely limited, sometimes just to the scheduling component. The number and priority of threads might have to be specified statically as well.

Spin [Bershad95] addresses the issue of expensive address space crossings by letting user code compiled by a trusted compiler run inside the kernel. MMLite enables equal functionality through loadable nested virtual machines.

Modularity has always been an important paradigm in software design. By breaking a complex system into pieces, the complexity becomes more manageable. Address spaces provide security by installing firewalls between applications. These two issues are orthogonal, but the distinction has been lost in systems research that has been concentrating on so-called microkernels [Black92, Cheriton94, Engler95, Hildebrand92, Julin91, Young89].

[Liedtke95] argues that microkernels have failed exclusively on performance grounds, and that poor performance is their only cause for inflexibility. Our argument is the opposite: inflexibility is inherent in the design, and leads to unavoidable inefficiencies that can only be mitigated by good implementations, never eliminated. Where L3's [Liedtke95] granularity is at the address space level, ours is at the single object level thanks to the COM's model. We also prove by construction that a modular system does not require VM, contradicting the axioms in L3's theoretical model.

7 Conclusions

Building an operating system for emerging computing platforms out of components pays off in terms of flexibility, minimalism, and adaptability. It naturally leads to good software design, rapid implementation, and good portability. COM is a good way of adding transparency both to location and privilege level, without too much overhead.

The MMLite system is efficient, portable, and has a very small memory footprint that makes it suitable for embedded use. It has successfully been used in several multimedia devices.

Acknowledgements

Thanks to Andy Raffman, Jerry Van Aken, and the rest of the Talisman team for their invaluable help.

References

[Bershad95] Brian Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, C. Chambers. *Extensibility, safety and performance in the Spin operating system.* In 15th ACM Symposium on Operating System Principles, pages 267-284, Copper Mountain Resort, Colorado, December 1995.

[Black92] David Black, David Golub, Daniel Julin, Richard Rashid, Richard Draves, Randall Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, David Bohman. *Microkernel Operating System Architecture and Mach.* In 1st USENIX Workshop on Micro-kernels and Other Kernel Architectures, pages 11-30, Seattle, April 1992.

[Brockschmidt95] K. Brockshmidt. *Inside OLE, Second ed.* Microsoft Press, Redmond WA, 1995.

[Cheriton94] David Cheriton, Kenneth Duda. *A Caching Model of Operating System Kernel Functionality.* In 1st Symposium on Operating Systems Design and Implementation, Seattle, 1994.

[OMG98] *CORBA/IIOP 2.2 Specification.* Available from http://www.omg.org/corba/corbiiop.htm.

[Cowan96] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. *Fast Concurrent Dynamic Linking for an Adaptive Operating System.* In the proceedings of the International Conference on Configurable Distributed Systems (ICCDS'96), Annapolis MD, 1996.

[Draves97] Richard Draves, Scott Cutshall. *Unifying the User and Kernel Environments.* Microsoft Research Technical Report MSR-TR-97-10, 16 pages, March 1997. Available from ftp://ftp.research.microsoft.com/pub/tr/tr-97-10.ps.

[Engler95] D. R. Engler, M. F. Kaashoek, J. O'Toole Jr. *Exokernel: an operating system architecture for application-specific resource management*. In 15th ACM Symposium on Operating System Principles, pages 251-266, Copper Mountain Resort, Colorado, December 1995.

[Ford97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, Olin Shivers. *The Flux OSKit: A Substrate for Kernel and Language Research*. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, pages 38-51. ACM SIGOPS, Saint-Malo, France, October 1997.

[Hildebrand92] D. Hildebrand. *An architectural overview of QNX.* In 1st USENIX Workshop on Micro-kernels and Other Kernel Architectures, pages 113-126, Seattle, April 1992.

[ISI95] Integrated Systems Inc. *pSOSystem System Concepts.* Part No. COL0011, May 1995, ISI, Sunnyvale CA.

[Jones96] Michael B. Jones, Joseph S. Barrera, III, Richard P. Draves, Alessandro Forin, Paul J. Leach, Gilad Odinak. *An Overview of the Rialto Real Time Architecture.* In Proceedings of the 7th ACM SIGOPS European Workshop, pages 249-256, September 1996.

[Jones97] Michael B. Jones, Daniela Roşu, Marcel-Cătălin Roşu. *CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities.* In Proceedings of the 16th ACM Symposium on Operating Systems Principles, pages 198-211. ACM SIGOPS, Saint-Malo, France, October 1997.

[Julin91] Daniel Julin, Jonathan Chew, Mark Stevenson, Paulo Guedes, Paul Neves, Paul Roy. *Generalized Emulation Services for Mach 3.0: Overview, Experiences and Current Status.* In Proceedings of the Usenix Mach Symposium, 1991.

[Liedtke95] Jochen Liedtke. *On ų-kernel construction*. In 15th ACM Symposium on Operating System Principles, pages 237-250, Copper Mountain Resort, Colorado, December 1995.

[Maeda93] Chris Maeda and Brian Bershad. *Protocol Service Decomposition for High-Performance Networking*. In 14th ACM Symposium on Operating System Principles, pages 244-255, 1993.

[Rozier88] M. Rozier, A. Abrassimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, W. Neuhauser. *CHORUS distributed operating system*. In Computing Systems, pages 305-370, Vol. 1-4, 1988.

[SDK97] mailto:tmantech@microsoft.com.

[Torborg96] Jay Torborg and Jim Kajiya. *Talisman: Commodity Real Time 3d Graphics for the PC*. In Proceedings of SIGGRAPH 96, August 1996.

[Yang98] Zhonghua Yang and Chengzheng Sun. *CORBA for Hard Real Time Applications: Some Critical Issues.* In Operating Systems Review, pages 64-71, Vol. 32-3, 1998.

[Young89] Michael Wayne Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.* Ph.D. Thesis CMU-CS-89-202, Carnegie Mellon University, November 1989.