A High-Performance Operating System for Structured Concurrent Programs

Luc Bläser
Computer Systems Institute, ETH Zurich, Switzerland
blaeser@inf.ethz.ch

ABSTRACT

With the advent of multi-processor machines, the time has definitively come to use new programming models that offer an improved support of concurrency. While various interesting new models have been recently presented for concurrent and structured programming, no appropriate runtime systems currently exists. Therefore, we have developed our own new operating system which has been particularly optimized for high-performance execution of such programs.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.1 [**Operating Systems**]: Process Management—*concurrency, scheduling, synchronization* [concurrent programming structures]

General Terms

Performance, Languages

Keywords

concurrency, memory management, hierarchical composition, message communication

1. MOTIVATION

While current operating systems have been particularly optimized for the execution of classical sequential programs, they typically fail in providing efficient runtime support for modern concurrent programming languages. More specifically, in recent years, new languages have been proposed which use a substantially different programming model and thus present new requirements on a runtime system:

1. *Concurrency.* Modern and advanced programming languages [3, 12, 10, 7, 17, 11, 6] usually offer concurrency as a primary programming concept, which is directly associated with objects or components. Instead of primarily programming with procedures, such languages rather encourage using a large number of very fine-grained processes which can directly interact by means of message communication.

2. *Structuring.* While traditional programming languages allow arbitrary referencing of objects (with some type limitations) without ability of guaranteed encapsulation, new programming languages increasingly focus on more expressive program relations and general hierarchical encapsulation [3, 14, 8, 1]. One approach is introducing structured components to replace classical references by more expressive program relations, such as hierarchical composition and dual interface wiring.

An example of a programming language which supports the abovementioned features in an uncompromising way is our recently presented Component Language [3]. However, for such programming languages, no efficient operating system currently exists. On the one hand, this is because today's operating systems only offer concurrency with limited scalability in the number of processes and poor efficiency for short-running and frequently synchronized processes. On the other hand, modern runtime systems often prescribe a specific memory management model with institutionalized garbage collection [9], even if the new language permits more efficient and accurate memory management.

For this purpose, we have developed a new small operating system which enables the high-performance execution of concurrent and structured programs. The highlights of the new system are:

1. The system supports *millions* of parallel processes on conventional computer machines.

2. The execution speed of concurrent programs is faster by a factor of 3 than in conventional systems.

3. The system offers a high predictability of the runtimes for our programs.

The system is primarily designed to support our Component Language very efficiently. However, it is deliberately kept generic and extendible to also support other concurrent languages with similar requirements. In this paper, we present the new operating system and explain its design and implementation. By means of experimental measurements, the scalability and performance of the system is compared to existing systems.

The remainder of this paper is structured as follows: Section 2 gives a short overview of what programming model

PLOS '07, October 18, 2007, Stevenson, Washington, USA.

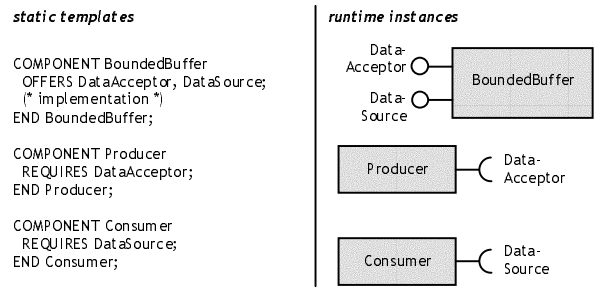

Figure 1: Component templates and instances

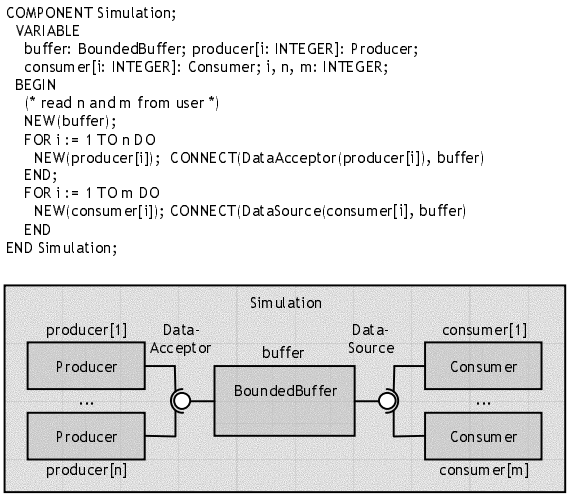

Figure 2: Hierarchical composition

is supported by the system. Section 3 then describes the design and implementation of the operating system. In Section 4, we present the results of the experimental measurements with our system, by comparing it to existing solutions. Section 5 finally reports on related work, before we draw a conclusion in Section 6.

2. PROGRAMMING MODEL

Our operating system already supports programs written in our concurrent and structured language. Instead of featuring classical objects or modules, this language is entirely based on a concept of components. A *component* constitutes a closed program unit at runtime which encapsulates state (data values and sub-components) and behavior (interactions and functionality). Strict encapsulation is enforced, i.e. a component can only be accessed from outside via explicitly defined *interfaces*. A component may both *offer* and *require* interfaces. An offered interface thereby represents an external facet of the component itself, enabling interactions between the component and its outer environment. Conversely, a required interface specifies an interface that is to be offered by another external component. A component is statically defined in the program code by a *template*, which allows creating multiple instances of components at runtime (see Figure 1).

2.1 Hierarchical Composition

A component is able to contain an arbitrary number of components within its implementation scope. This is enabled by means of *variables*, which represent separate containers in which components can be stored. A variable can either store a single component (e.g. `buffer` in Figure 2) or denotes a collection (e.g. `producer[i: INTEGER]` in Figure 2), in which a dynamic number of components can be allocated. In the first case, the component is directly identified by the variable name, whereas in the latter case, a component in the collection is identified by the variable name and an index value. For each variable, a type is associated which describes either the template of the component or (for the sake of polymorphism), only postulates a set of offered and required interfaces (see [3]).

At runtime, components can be created and installed within the variables by use of the NEW-statement. As a result, the components inside the variables are fully encapsulated and constitute sub-components of the surrounding instance. No explicit pointers or references are involved here, as a component is only accessible via its variable identifier (and index value). Of course, the components need to be appropriately organized in the memory by using memory indirections. This however happens automatically in the runtime system and is not visible at the level of the programming language.

Components within the same scope may be also connected to networks. For this purpose, each required interface of a component can be connected to an interface that is offered by another component. In order to be connectable, the required and offered interfaces must have the same name (see Figure 2).

2.2 Concurrency and Communication

In our programming language, every component runs its own inner processes and only interacts by message communication. Two components, which are connected, can directly exchange messages via their connected interfaces. The component which offers the interface may be regarded as the *server*, while the other component acts as the *client* with regard to this interface. A message transmission involves one side sending the message and the other one explicitly receiving it. As a particular feature, the server component maintains a separate communication for each client individually. This means that each pair of a client and server has a separate communication channel via the interface, such that the server may communicate with multiple clients in parallel (see Figure 3).

However, two components cannot send and receive messages in an arbitrary way but must follow a communication protocol. This protocol is defined in EBNF [18] (right-hand side of Figure 3) within the interface and specifies all message transmissions that are allowed between a client and a server of that interface. A message transmission is therein prefixed by `IN`, if it is sent from the client to the server, and `OUT`, if sent in the opposite direction. A sequence of messages is denoted by subsequent expressions, while curly braces represent arbitrary repetition (including zero times) of a sub-protocol.

As for the implementation of the components, the server component automatically starts a separate service process for each client. This service process runs inside the server component and only performs the server-side communication with the corresponding client. As the interactions be-

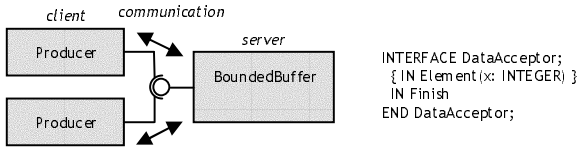

Figure 3: Message communication

tween different components are inherently synchronized by the communication, only the processes *within* the same component instance need to be explicitly synchronized by monitor protection. For a more detailed explanation, the reader is however referred to [3].

3. RUNTIME SYSTEM

Our system allows direct loading and execution of the machine code that has been generated by the compiler of our component language. For a coherent design, the system represents itself as a component, which already preexists as a first component instance when the system is started. The system component runs together with application components and offers necessary system interfaces, such as for system services or device drivers. The system component itself has not been implemented in the Component Language but in a reduced version of Oberon [19] (with explicit memory management). This is because we need a more machine-close programming language, in which the concepts directly correspond to the underlying machine model. The possibility of implementing terminal components in a different language is a particular feature of our component model, where interfaces and implementation are clearly separated. In the following sections, we describe the most important features of the operating system.

3.1 Memory Management

The system is based on a uniform memory model, where all program structures, such as for components, processes, stacks, collections, communications, are managed in the heap. Stacks are maintained in the heap because we need to implement processes with very small stacks that may dynamically grow or shrink. The memory block of a component contains a slot for each offered and required interface. This is used to store the necessary connection that can either refer to another interface slot or to the program code of a service process, if it forms an implemented offered interface. Process blocks only need very small memory space for context switch backup. In the case of a service process, the heap block also includes the space for the communication buffer (a circular FIFO-list) that is used between the service process and its individual client. By analyzing the largest message declaration in the communication protocol, the maximum size of the message entry in buffer can be statically determined. Therefore, it is possible to pre-allocate a buffer with a defined buffer capacity, where the buffer capacity may vary for each communication. During a communication, the EBNF-protocol is also automatically monitored by the system. For this purpose, the compiler generates for each interface specification a finite state machine, which encodes the feasible message transmissions during a communication.

For each process, the compiler statically determines an initial stack size which is inlined in the process block. Thereby, the size is chosen with a small reserve (up to 512 bytes), to support the most frequent system calls on the initial stack. More complex system calls, as well as interrupts, are executed on an extra processor-associated kernel stack. The stack may be also dynamically extended or reduced for the execution of component-internal procedures. At the entrance of a procedure, a small compiler-inserted check determines whether a new stack block has to be allocated. On stack extension, the local data of the current procedure activation frame remains accessible on the old stack, as the caller always keeps sufficient stack reserve for the local variables of its directly invoked procedures. However, as the programming model uses communication instead of method calls, procedure calls and dynamic stack extensions are rather seldom. A component may only feature procedures within its implementation scope that cannot be called from outside.

As for the implementation of the language-inbuilt dynamic collections, they are generally implemented as B+-trees. While not occupying more than double the effectively needed memory, B+-trees offer access times that are asymptotically logarithmic to the number of contained elements. The system integrates particular optimizations for collections with integer indexes. At runtime, it is determined whether a simple array or direct hash structure is faster and the data structure is adapted correspondingly at runtime.

Thanks to the concept of hierarchical composition, all memory structures have an exact deallocation time. The deletion of a component leads to the finalization and deallocation of all inner components and collections. A service process can only be deleted when its client process has finished the communication. Moreover, the explicit deletion of a single component involves disconnecting its interfaces. Hence, the system does not require an automatic garbage collector to guarantee memory safety.

3.2 Process Management

To offer efficient and fair scheduling of processes, all runnable (including preempted) processes are queued in a single FIFO list (ready queue) and assigned to processors by a simple but efficient round-robin scheduler. As processes can directly operate on their enclosing component, the components are implemented as monitor-protected shared resources, implemented with three local waiting queues, one for processes waiting for exclusive lock, one for those waiting on shared locks and one for all processes that are waiting on a Boolean condition. The prioritisation among processes follows a three-stage shell model. In order of priority, processes with a fulfilled await-condition gain access to the resource, then processes waiting for the entrance with an exclusive lock, and finally those waiting for a shared lock. This is implemented by checking the waiting queue, when a process should release the exclusive monitor lock. If the condition is fulfilled for a waiting process, the corresponding process directly obtains the corresponding lock.

Synchronous context switches involve very low costs, as only three registers (program counter, stack pointer and frame pointer) have to be saved and restored. In our system, preemptive scheduling of processes is not implemented by conventional hardware interrupts but with a technique of code instrumentation. The compiler automatically inserts checks in the machine code, initiating preemption of a process if a certain execution time has passed. Notably, the system guarantees time-sliced processor sharing without any help or knowledge of the programmer, i.e. no coop-

main memory	Component OS	Windows C#	Windows Java	AOS
256 MB	321,000	1,950	7,230	15,700
1024 MB	1,300,000	1,960	7,130	15,700
4096 MB	5,010,000	1,900	10,000	15,700

Table 1: Maximum number of processes

erative multi-task has to be programmed. With this technique, the system only saves the necessary program state before a software-based preemption, while interrupt-based preemption needs to take a snapshot of all registers. In fact, the checks are mostly inserted between language statements, where typically no temporary registers are in use. Therefore, processes do not need to have large pre-allocated memory space for register backups. To guarantee that the limits of a time slice are fulfilled, the preemption checks need to be continuously executed in small time steps. Therefore, the checks are inserted (1) in each loop body, (2) in each procedure entry, and (3) after a statement sequence of a maximum (worst-case) runtime. The current implementation of a check only requires a few simple instructions and we measured that the total cost of preemption checks is on average less than 0.5% of the total program runtime.

It is well known that process synchronization is quite expensive on today's computer machines due to the required synchronization of the processor caches. In fact, one can only gain from parallelism if the system uses long-running nearly independent processes, which need no or very occasional mutual synchronization. However, our programming language encourages a model of fine-granular interacting processes. Therefore, we equipped our system with a smart scheduler that only schedules processes in parallel, if they indeed run faster on multiple processors. All other processes are scheduled in a serial but time-sliced way on the same processor. To determine the appropriate scheduling strategy, the system measures the synchronization rate for each process. More specifically, a process mostly runs independently of other ones, if it has been recently preempted and has not been recently waiting on a monitor lock or condition. Such a process is then elected to run in parallel with other processes.

4. EXPERIMENTAL RESULTS

Our operating system was primarily designed to support a particularly high number of processes that is clearly beyond the capabilities of existing systems. Therefore, we have measured the maximum support number of processes on our system and on other systems. For this purpose, we have used a concurrent program that scales well with the number of parallel processes. We have once implemented the programs in our language and then, in object-oriented languages by using threads. Table 1 summarizes the maximum number of processes for different sizes of main memory size. As can be seen, our system (Component OS) indeed permits millions of processes, where the number of processes scales linearly with the available size of physical main memory. The evaluation also shows that the other systems only allow a very small number of processes, which is not more than about 16,000 (such as in AOS [15]). This deficiency can be traced back to the heavy-weighted stack design in those systems, as well as in their high memory demands for preemption backups.

test program	Component OS	C#	Java	AOS	speedup[2]
City	**0.26**	0.66	440	4.1	2.5
ProducerConsumer	**18**	19	130	60	1.1
Eratosthenes	**1.6**	6.8	4.6	5.8	2.9
News	**0.82**	3.5	3.9	3.7	4.3
Library	0.78	0.74	1.5	**0.59**	0.76
TokenRing	**2.1**	22	22	18	8.6
Mandelbrot	0.89	0.43	**0.39**	0.6	0.44
TrafficSimulation	**3**	2000	-[1]	-[1]	-

runtimes in seconds rounded on 2 figures, Intel Xeon, 6 CPU with 700MHz, 4GB main memory, C# and Java ran on Windows Server RC2 Enterprise Edition
[1] not implemented; [2] Component OS compared to the fastest other system

Table 2: Performance comparison

To measure the runtime performance, we assembled a set of concurrent programs, ranging from very simple programs (ProducerConsumer) to extensive simulation packages (TrafficSimulation). Again, we implemented programs in our component language and in classical object-oriented languages. Naturally, the versions for the different languages are modeled as similarly as possible, using threads in place of the intrinsic component processes. To stick to the right paradigm, we implemented the object-oriented programs by using classical methods for object interactions. Therefore, the test programs of our language have a higher number of threads as each communication involves an additional service process. All measurements were performed on a machine with 6 Intel Xeon processors with 700MHz each. As depicted in Table 2, the new system outperforms C#, Java and AOS [15] with an average speedup of factor 3, if we compare the result of each test case with the corresponding best other system. Naturally, the performance advantage of our system is mainly due to the underlying concurrency model, which offers low-cost context switches and fast software-controlled preemption. Only for some simple programs with a lot of array accesses is our implementation of dynamic collections slower. Naturally, our operating system is also capable of offering linear speedup with multiple processors. For instance, the Mandelbrot program has a runtime of 5.1 seconds with one processor and 0.89 seconds with six processors on the same machine. All test programs are available at [4].

Without a garbage collector, our runtime system also prohibits unexpected system disruptions. To illustrate the difference, we have performed a measurement series of 500 subsequent executions of a small instance of the TokenRing program (Figure 4). While AOS [15] suffers from continuous peaks as a result of the garbage collector disruptions, our runtime system exhibits a nearly constant execution time among all iterations. The only time fluctuations that occur in the component system are due to the non-predicable process synchronization and caching effects.

5. RELATED WORK

As for our programming model, related work has already been discussed in [3]. Therefore, we focus here on the operating system. The design of the kernel of our system is influenced by AOS [15], although the memory and process management is substantially different. The idea of representing a stack as a dynamic list of memory blocks is already known from existing systems [16, 13]. In some of these systems, the stack size can however not be arbitrarily small

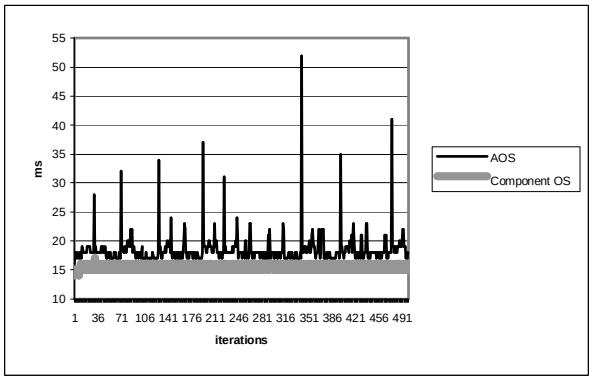

Figure 4: Variations in the runtimes

but has to be at least of a page (4KB) [13]. In our language and system, the stack sizes of a process can be adequately determined by the compiler, such that dynamic stack extensions only occur for relatively seldom system calls. Similar to our runtime system, the Singularity OS [13] incorporates a communication-oriented programming model [7]. The kernel, drivers and applications are designed as separate object spaces that have to be isolated and can only interact by message exchange or via shared objects in a special exchange heap. In our approach, all normal application components ought to be programmed in a structured programming model that is similar to ours, such that encapsulation can be inherently guaranteed. To reduce the disruption times of garbage collection, rather complicated and expensive real-time collectors [5, 2] exist. The Singularity OS [13] thereby takes a particularly interesting approach. As a result of the concept of isolated object-spaces, each space can be managed by its individual runtime systems. Therefore, garbage collection becomes customizable at the granularity of the object spaces.

6. CONCLUSION

This paper has presented a new operating system which enables highly scalable and efficient execution of concurrent and structured programs, clearly surpassing conventional runtime systems. With a rigorous liberation from a classical system design, we have developed a system that employs innovative techniques, such as fine-granular process, low-cost context switches as well as a memory model that can be safely managed without use of a garbage collector. The source code and the binaries of the operating system, as well as the language report and the test programs, are available at [4].

Acknowledgments

I would like to thank Prof. Dr. Jürg Gutknecht for his support and supervision in this project. My thanks also go to Dr. Felix Friedrich, Dr. Svend Knudsen, and other colleagues who commented on this language and system.

7. REFERENCES

[1] J. Aldrich, C. Chambers, D. Notkin. *ArchJava. Connecting Software Architecture to Implementation.* Intl. Conference on Software Engineering (ICSE), May 2002.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. *A Real-Time Garbage Collector with Low Overhead and Consistent Utilization.* Symp. on Principles of Programming Languages (POPL), Jan. 2003.

[3] L. Bläser. *A Component Language for Structured Parallel Programming.* Joint Modular Language Conference (JMLC), Sept. 2006, LNCS Vol. 4228, Springer Verlag, 2006.

[4] L. Bläser. *The Component Language and System.* http://www.jg.inf.ethz.ch/components.

[5] P. Cheng and G. E. Blelloch. *A Parallel, Real-Time Garbage Collector.* Conf. on Programming Language Design and Implementation (PLDI), June 2001.

[6] O. - J. Dahl and K. Nygaard. *SIMULA — An ALOGL-based Simulation Language.* Communications of the ACM, 9(9):671-678, 1966.

[7] M. Fähndrich, M. Aiken, C. Hawblitzel, et al. *Language Support for Fast and Reliable Message-Based Communication in Singularity OS.* EuroSys 2006, April 2006.

[8] D. Gay, P. Levis, R. von Behren, et al. *The nesC Language: A Holistic Approach to Networked Embedded Systems.* Conf. on Programming Language Design and Implementation (PLDI), June 2003.

[9] J. Gough. *Compiling for the .NET Common Language Runtime (CLR).* Prentice Hall, 2002.

[10] R. Güntensperger and J. Gutknecht. *Active C#.* Intl. Workshop on .NET Technologies, May 2004.

[11] J. Gutknecht. *Do the Fish Really Need Remote Control? A Proposal for Self-Active Objects in Oberon.* Joint Modular Language Conference (JMLC), March 1997. LNCS Vol. 1204, Springer Verlag, 1997.

[12] J. Gutknecht and E. Zueff. *Zonnon Language Report.* http://www.zonnon.ethz.ch.

[13] G. Hunt, J. Larus, M. Abadi et al. *An Overview of the Singularity Project.* Technical Report MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[14] Y. D. Liu and S. F. Smith. *Interaction-Based Programming with Classages.* Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Oct. 2005.

[15] P. J. Muller. *The Active Object System Ű Design and Multiprocessor Implementation.* PhD Thesis, Diss. ETH No. 14755, ETH Zurich, 2002.

[16] R. von Behren, J. Condit, F. Zhou, et al. *Capriccio: Scalable Threads for Internet Services.* Symp. on Arch. Support for Programming Languages and Operating System Principles (SOSP), Oct. 2003.

[17] P. H. Welch. *The JCSP Home Page.* http://www.cs.ukc.ac.uk/projects/ofa/jcsp.

[18] N. Wirth. *What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?* Communications of the ACM, 20(11):822-823, Nov. 1977.

[19] N. Wirth. *The Programming Language Oberon.* Software – Practice and Experience, 18(7):671-690, July 1988.