COMPOSITA: A Study in Runtime Architectures for Massively Parallel Systems

Luc Bläser[1] and Jürg Gutknecht[2]

[1] University of Applied Sciences Rapperswil
Institute for Software
lblaeser@hsr.ch
[2] ETH Zürich, Switzerland
Native Systems Group
gutknecht@inf.ethz.ch

Abstract. COMPOSITA is an experimental operating system optimized for effective multi-processing and memory management. Based on new software technology, notably including micro-stacks and software-controlled preemption, the system supports millions of concurrent light-weight processes. Thanks to hierarchical component structures, no garbage collection is used for the management of dynamic memory. Experimental evaluations have shown that, under the governance of COMPOSITA, massively concurrent programs perform and scale considerably better than under traditional operating systems.

1 Introduction

A comparison of today's most popular operating systems, notably *Windows*, *Linux*, *MacOSX* etc. shows a striking principal resemblance of their architectures. This is beyond sheer coincidence because all of these systems share the same genetic footprint: the UNIX systems of the 1960s and 1970s. While the success of these UNIX derivatives is undeniably remarkable, their efficiency factor on modern computer systems cannot possibly be close to optimal, as their genes in essence favor a totally different scenario of usage: a central system serving a large number of terminal users in time-sharing mode (or a miniaturized version hereof).

Ever more challenging requirements like parallelism across all granularities, highly concurrent data structures and seamlessly integrated real-time applications exert additional pressure on system architects towards rethinking the entire software stack of a modern computer system from the ground up. We decided to take up this challenge in the concrete form of building an super-efficient implementation of a runtime environment for a highly parallel programming language on a state-of-the-art multicore machine. CL became the language of our choice, an experimental component-oriented language that we developed in a previous project [4]. CL embodies a model of fine-granular concurrency, message communication and hierarchical composition. The result of our endeavor is an ultra-compact operating system called COMPOSITA. It is targeted at massively parallel applications and exhibits a variety of innovative features, among them

– **Fine-grained call-stacks** (micro-stacks) permitting millions of concurrent light-weight processes to coexist at any time
– **Super-fast context switches**, equally efficient as procedure calls
– **Software-instrumented checkpoints** as a replacement for expensive preemptive context switches
– **Ultra-compact process backups** exploiting context intelligence of the compiler
– **Memory management and recycling** based on hierarchical compositions instead of garbage collection

Our decision of developing a new and self-contained solution from the ground up (in contrast to fixing and extending an existing system) allowed us to concentrate on the essentials (thereby sacrificing some "bells and whistles") and to keep it compact and simple. The research was primarily meant to provide an experimental ground for exploring new models and much less to give a definitive answer on how generic operating systems should be built in the future.

In the following, we shall explain the design and the detailed implementation of the COMPOSITA operating system. Section 2 gives an overview of its architecture, including the underlying component-oriented programming language CL. Section 3 focuses on the management of processes and memory. Section 4 reports on some experimental measurements. Section 5 discusses related work and Section 6 finally draws conclusions.

2 Architecture

COMPOSITA runs on Intel PC machines, has a size of ca. 250 KB and boots up in about a second. It currently includes a limited set of device drivers, notably comprising IDE disk, keyboard and graphics card. Both the system and the applications present themselves in the form of a network of *components* interconnected via *interfaces* (Figure 1).

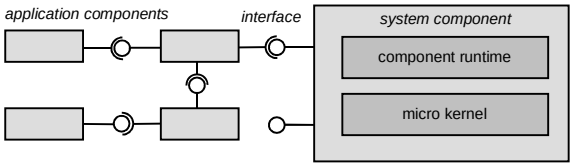

Fig. 1. System architecture

2.1 Component Structures

The component architecture is based on the following mechanisms and principles:

– **Composites.** Each component may hierarchically contain any number of sub-components.
– **Interfaces.** Interfaces specify communication ports. Each component potentially features two kinds of interfaces: offered interfaces and required interfaces.
– **Connections.** If compatible, a required interface of one component can be connected with an offered interface of another component, where compatibility means that the interfaces have the same name and reside in the same surrounding composite.
– **Delegation.** A component can delegate an offered interface of its own to an offered interface of one of its sub-components. And analogously for required interfaces. In both cases the interface names are required to match.

Figure 2 depicts an exemplary composite called `FileConverter`, where the offered/ required interfaces are represented as lollipops/ forks respectively. The wiring comprises connecting the required interface `Reader` of the `Transformer` with the offered interface `Reader` of the `Parser` and delegating the offered interface `ConversionControl` to the offered interface of the `Transformer`.

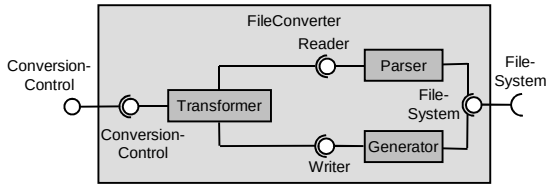

Fig. 2. Exemplary component structure

Components are entirely managed by their container (surrounding component), i.e. only an outer component can create, connect and delete inner components. All other kinds of interactions necessarily involve communication (via interfaces).

2.2 Concurrency Model

Concurrency in CL interoperates with the compositional structure in the following way:

– **Processes and internal synchronization.** A component typically runs a number of concurrent processes inside of its scope, where each process is allowed to access the component's (shared) memory. For the purpose of synchronization both exclusive locks and shared locks are provided, as well as a generic wait (Boolean condition) operation [11].

– **Communication.** Interfaces allow bidirectional communication. The message format as well as the protocol supported by a certain interface is specified in terms of a formal grammar bound to the respective interface. Messages may include numbers, strings, etc. but also components (transported from sender to receiver). Communications between each pair of connected components run separately and in parallel.

3 Operating System

The COMPOSITA system supports dynamic loading of components. It presents itself as a pre-existing component loaded and started after system boot up. It makes available system services and device drivers to application components via interfaces and the communication mechanism discussed earlier. In the following sections we explain the implementation of the framework behind the scenes of these concepts in relevant detail.

3.1 Process Management

As our system is aimed at accommodating a very large number of concurrent processes, the (footprint) size of individual processes is an important factor. Therefore, we employ special care to minimize a) the stack size and b) the backup space needed (after context switch) for each individual process.

Process Stacks Unlike in many other operating systems, process stacks are not implemented as contiguous pre-allocated memory blocks of a certain (often generously guessed) maximum size in some virtual address domain in COMPOSITA but as a linear list of heap blocks in real memory, with the immediate benefit that stacks can dynamically grow or shrink, see Figure 3. At each procedure call a runtime check is instrumented to determine whether sufficient stack space is still present. If not, a new stack block of a precalculated size (by the compiler) is allocated, and the stack grows correspondingly. When the procedure later returns, the extra stack block is deallocated and returned to the heap. The size of a process' initial stack block is also determined by the compiler.

It is worth noting that CL does not require frequent dynamic stack extensions as it uses communication-based component interactions in place of method calls in all but local cases. As an additional precaution interrupts run in privileged mode, using a separate, preallocated kernel stack.

Context Switches The efficiency of *context switches* is a decisive measure for the performance of any highly parallel system. A *synchronous* context switch occurs whenever a command of type *wait/suspend* is to be executed. As our kernel does not run in a separate protection ring, no (expensive) software interrupt is needed to handle this case and an ordinary procedural kernel call (prolog and epilog operating on different stacks though) does the job. Only three registers

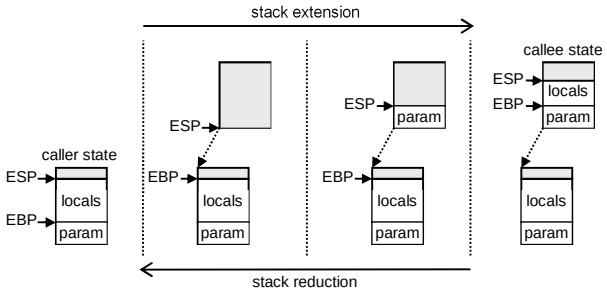

Fig. 3. Stack extension and reduction

must be saved and restored namely the *program counter*, the *stack pointer* and the *frame pointer*.

Asynchronous context switches are indispensable with runtime strategies employing *preemptive process scheduling*, such as for example preemptive priority scheduling or processor-sharing based on the allocation of time-slices. As asynchronous context switches are uncooperative, they tend to be highly expensive both in terms of time and space (request for saving the entirety of registers including FPU, MMX, SSE2). Therefore, we were looking for an alternative and more efficient solution that allows COMPOSITA to mimic preemption in a slightly weaker form. Rather than letting hardware interrupts govern the context switch, we are using an instrumented inline approach. More concretely, the CL compiler injects checks for pending preemption requests directly into the machine code at suitable locations, where suitable means that the runtime can guarantee at least one check within each time-interval of a desired length. Furthermore, the compiler can quite easily optimize the locations in a way to minimize the number of registers to be saved when the context switch actually occurs, which substantially contributes to compact sizes of process backups.

According to our heuristics, the compiler injects a check for preemption requests (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. Table 1 quantifies the performance costs of preemption checks for sample programs. For typical concurrent scenarios, no or very low overheads can be observed. Of course, costs become more significant if programs involve hot loops or frequent procedure calls: The artificial spinning loop example shows approximately 20% overheads for components running long counting loops.

Process Scheduling All runnable (including preempted) processes are queued up in a single FIFO list (ready list) and assigned to processors by a simple round-robin scheduler. Components are implemented as monitor-protected shared resources. Each component features two waiting lists, one for processes waiting for an exclusive lock or shared lock and one for processes waiting on a Boolean

Program	With checks	No checks	Overheads
ProducerConsumer (1 producer, 1 consumer, capacity 1)	1011	1048	-3.5%
Eratosthenes (10000 upper limit)	234	230	1.9%
TokenRing (1000 nodes)	266	273	-2.7%
SpinningProcesses (16 instances)	181	151	19.6%

Runtime in milliseconds, rounded to millisecond, average of 3 subsequent executions, Intel 2 Core i7 3520M, 2.9GHz, 8GB main memory.

Table 1. Runtime costs of preemption checkpoints

condition. The prioritization among processes follows a so-called eggshell model [11] that grants priority handling to waiting processes whose condition has be established over newly entering processes. This is implemented by checking the waiting lists for established conditions whenever a process releases the monitor lock (signal and exit strategy). If an established condition has been found, the lock is immediately passed to the corresponding process. In order to avoid potential starvation among reader or writer processes, a first-come first-served strategy is applied to the processes entering a lock-controlled region.

Communication Channels The formal specification of communication protocols in CL empowers the runtime system to validate the implementation of protocols at both ends, thereby upgrading the level of intercomponent communication in CL in terms of consistency checking to the level of method calls in strongly typed languages. A state machine derived from the protocol specification is used for this purpose. In principle, static consistency checking of protocol implementations would also be possible up to a certain degree, albeit a complete static analysis is impossible because the problem is equally hard as the halting problem that is undecidable. Implementation-wise, communication channels in CL are based on small bounded buffers (maximum of 4 messages).

3.2 Memory Management

Perhaps the biggest pay-out of our strictly hierarchical component system is a drastically simplified memory management. Traditional systems maintain a non-hierarchical object graph. Lacking any natural ownership relation, a sophisticated system-wide process is put into action with the mandate of continuously identifying garbage (objects to be recycled) via *reachability*. In COMPOSITA, *containment* defines a natural ownership relation that authorizes containers (the owners) to finalize, deallocate and recycle their content components explicitly. In detail, deallocation of component C involves the following steps (see Figure 4 for an illustration):

1. Recursively delete (inner) components of C

2. Wait for all communications involving C to terminate
3. Disconnecting interfaces with C

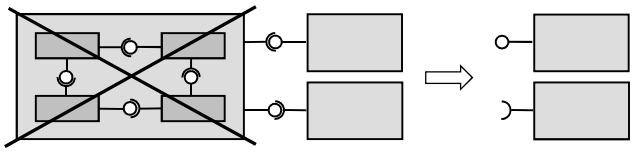

Fig. 4. Component deletion

It is worth noting that our approach does not suffer from the traditional problems of delayed finalization or from phenomena like resurrection etc., nor is the consistency of the memory compromised at any time. Dangling pointers do not exist, as external accesses to components never involve pointers but communication instead and as any message traffic is suspended after the interface has been disconnected. Memory leaks are impossible as well because components remain accessible (by their container) until they are deallocated explicitly. No component ever becomes undeletable.

4 Experimental Results

The COMPOSITA system was uncompromisingly designed with the single goal in mind of running an extremely large number of concurrent processes that is clearly beyond the capabilities of existing systems. Correspondingly, our first benchmark focuses on comparative measurements of the maximum number of (light-weight) processes that can be accommodated concurrently (on top of the given hardware) by COMPOSITA and other systems respectively. For this purpose, the benchmark creates and runs fake processes that merely execute a simple waiting activity. Table 2 summarizes the results. As can be seen, COMPOSITA is able to accommodate millions of processes, while the classical systems reach their limits at a lower order of magnitude. This can quite easily be explained by COMPOSITA's scaling characteristics that are by design inversely linear with regard to the stack size and linear with regard to the size of the physical memory.

For measuring the runtime performance we assembled a small set of programs representing typical concurrency patterns, all of them rich in terms of context switches: Producers/ Consumers (N producers and M consumers interacting via a bounded buffer of capacity C), Sieve of Eratosthenes (a pipeline of filtering processes determining the prime numbers between 2 and an upper limit N) and TokenRing (a ring of N processes circularly ordered and pushing a token around the ring 1000 times). In COMPOSITA, component interactions are modeled as message communications with an extra service process per channel. In all other languages, conventional method calls are used for this purpose. All measurements

were performed on an Intel 2 Core i7 3520M 2.9GHz platform, with 8GB main memory. Table 3 clearly demonstrates that our architecture outperforms classical thread-based systems, mostly due to COMPOSITA's highly optimized context switches.

COMPOSITA	.NET (Win8)	Java (Win8)
4,367,000	100,000	100,000

Number of light-weight processes under COMPOSITA, number of threads under .NET x64 and Java 64 bit VM, 8GB main memory, rounded on 3 figures, .NET Framework 4.5 under Windows 8 (x64), Java 7 (1.7.0.21, 64 bit server VM) under Windows 8 (x64).

Table 2. Maximum number of threads

Program	COMPOSITA	C#	Java
ProducerConsumer (N=M=C=1)	**1011**	5427	6020
ProducerConsumer (N=1, M=10, C=1)	**1327**	22324	26255
ProducerConsumer (N=10, M=10, C=10)	**10141**	40158	30513
Eratosthenes (N=1000)	**5**	31	31
Eratosthenes (N=10'000)	**235**	877	640
Eratosthenes (N=100'000)	**14594**	49216	38023
TokenRing (N=1000)	**266**	4500	4596
TokenRing (N=10'000)	**2822**	49945	53582
TokenRing (N=100'000)	**30106**	518956	1,163,088

Runtime in milliseconds, rounded to millisecond, average of 3 subsequent executions, Intel 2 Core i7 3520M, 2.9GHz, 8GB main memory. C# on .NET Framework 4.5, x64, with optimization compiler option, on Windows 8, Java 7 version 1.7.0.21, 64 bit server under Windows 8.

Table 3. Runtime of concurrent programs

5 Related Work

Stack architectures. Several systems share with COMPOSITA the representation of call-stacks as dynamic lists of memory blocks with the goal of to reducing the memory footprint of thread representations [1, 9]. However, unlike COMPOSITA's micro-stacks, the stack size in these systems still remains roughly at

page granularity [9, 14]. Even lighter-weight threads are usually less useful to these systems because context switches such as the ones needed for preemptive multitasking [14, 12] are no longer possible. Such restrictions apply by purpose for coroutines and fibers. Other implementations apply stack hijacking that is they require the system to back up the stack contents at each context switch, so to allow the new thread to continue on the same stack [3]. Lazy thread creation [8] optimizes certain concurrent executions by sequentializing them. Light-weight thread APIs and thread pools are standard facilities provided by most parallel system implementations. However, the formers usually suffer from the need of mapping user threads to rather inefficient kernel threads (for truly parallel execution) [14, 3], while the latters come at a cost of a de facto restricted applicability to threads with no mutual dependencies.

Garbage collection. Garbage collection in real-time systems has become a pain in the neck. Extremely complicated and subtle algorithms have been conceived with the goal of eliminating or at least reducing the non-deterministic disruptions caused by garbage collection [7, 1, 13], without having yet achieved a truly satisfactory solution. Other approaches employ multiple isolated object spaces, managed by separate garbage collectors (see for example Singularity OS). COMPOSITAs memory management rests on the belief that eliminating the problem is its best solution.

Composita. An earlier version of the system has been outlined in [5, 6].

6 Conclusion

We have invested a substantial research effort into the question of how a modern runtime system architecture, optimally supporting massive parallelism and free of legacy should or could look like. Our strategy was radical: develop a prototype from the ground up as a proof of concept. The result is COMPOSITA, a kind of a mockup of a fully grown system and uncompromisingly targeted at running an extremely large number of concurrent processes. We consider our experiment a success. Thanks to a number of innovative solutions, including hierarchical memory structures, micro-stacks, low-cost context switches and code-instrumented preemption, we are able to convincingly demonstrate that both the performance and the scalability of massive parallel hardware systems can be substantially improved if orchestrated by cleverly designed software.

Project Website

The COMPOSITA system and all test programs used for measurements are available at: http://concurrency.ch/Projects/Composita

Acknowlegments

We gratefully acknowledge the collaboration with Kai Nagel. Thanks to his excellent support and advice we were not only able to realize a realistic traffic

simulation on top of COMPOSITA but in addition to practically demonstrate its performance advantage over traditional simulation systems [6]. During this project, Felix Friedrich and Svend Knudsen continuously and patiently provided most helpful input and valuable feedback on both the design and the implementation of our system.

References

1. D. F. Bacon, P. Cheng, and V. T. Rajan. *A Real-Time Garbage Collector with Low Overhead and Consistent Utilization.* Proceedings of the Symposium on Principles of Programming Languages (POPL), January 2003.
2. R. von Behren, J. Condit, F. Zhou et al. *Capriccio: Scalable Threads for Internet Services.* Proceedings of the 19th ACM symposium on Operating systems principles (SOSP), October 2003.
3. A. Begel, J. MacDonald, and M. Shilman. *PicoThreads: Lightweight Threads in Java.* Technical Report, UC Berkeley, 2000.
4. L. Bläser. *A Component Language for Structured Parallel Programming.* Proceedings of the Joint Modular Languages Conference (JMLC), Oxford, UK, September 2006.
5. L. Bläser. *A High-Performance Operating System for Structured Concurrent Programs.* Proceedings of the Workshop on Programming Languages and Operating Systems (PLOS), October 2007.
6. L. Bläser. *A Component Language for Pointer-Free Concurrent Programming and its Application to Simulation.* PhD Thesis, Diss. ETH No. 17480, ETH Zurich, November 2007.
7. P. Cheng and G. E. Blelloch. *A Parallel, Real-Time Garbage Collector.* Proceedings of the Conference on Programming Language Design and Implementation (PLDI), June 2001.
8. S. C. Goldstein, K. E. Schauser, and D. E. Culler. *Lazy Threads: Implementing a Fast Parallel Call.* Journal of Parallel and Distributed Computing, 37: 5-20, 1996.
9. G. C. Hunt and J. R. Larus. *Singularity: Rethinking the Software Stack.* ACM SIGOPS Operating Systems Review, 41(2): 37-49, April 2007.
10. G. Hunt, J. Larus, M. Abadi, et al. *An Overview of the Singularity Project.* Technical Report MSR-TR-2005-135, Microsoft Research, October 2005.
11. P. J. Muller. *The Active Object System: Design and Multi-processor Implementation.* PhD Thesis, Diss. ETH No. 14755, ETH Zurich, 2002.
12. E. D. Polychronopoulos, X. Martorelli, D. S. Nikolopoulos et al. *Kernel-Level Scheduling for the Nano-Threads Programming Model.* Proceedings of the 12th International Conference on Supercomputing (ICS), Melbourne, Australia, July 1998.
13. Y. Sun and W. Zhang. *Overview of Real-Time Java Computing.* Journal of Computing Science and Engineering 7.2 2013: 89-98, 2013.
14. K. B. Wheeler, R. C. Murphy and D. Thain. *Qthreads: An API for Programming with Millions of Lightweight Threads.* Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS), April 2008.