A High-Performance Operating System for Structured Concurrent Programs

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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

PLOS’07, Stevenson WA, USA, 18 Oct. 2007

State of the Art

Main problems of object-oriented languages:

- References
  - Arbitrary object interlinking => Unstructured dependencies
  - No hierarchical composition => Objects can not encapsulate (dynamic) structures of other objects
- Threads
  - Concurrency only added with hindsight to procedural programming model
  - Threads operate on arbitrary objects via method calls => error-prone

New Approaches

Trends towards improved programming models:

- First-class concurrency
  - Concurrency as primary language construct in the form of self-active objects / components
  - Message communication instead of blocking method calls
  - Examples: Active C#, Zonnon, Composita
- Pointer-free structuring
  - Hierarchical composition instead of flat object graph
  - Hierarchically controlled interface wiring
  - Examples: ArchJava, Classages, Composita

⇒ New requirements for modern runtime systems

Modern Runtime Systems

Requirements:

- Highly-scalable concurrency
  - Support of a very high number of light-weighted processes
- High-performance concurrency
  - Efficient execution of highly-interactive concurrent programs
- Efficient memory management
  - Efficient and safe memory management exploiting improved program structures
- Liberation from artifacts
  - Abandon system features that are no longer needed for a modern programming model

Example of Structured Concurrency

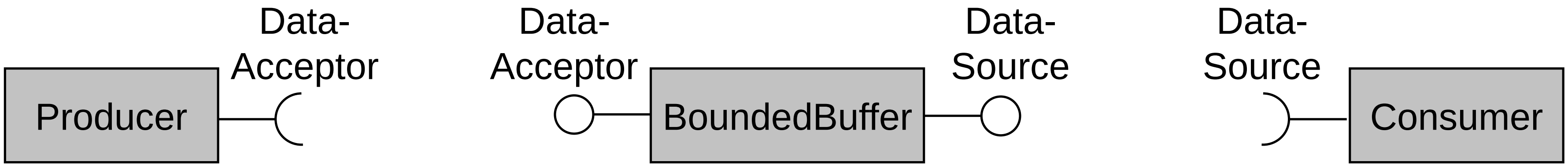
The Component Language

- Components as general abstraction units
- Strict encapsulation
  - External dependencies only via explicit interfaces allowed
- A component can offer and require interfaces
  - *offered* interfaces represent own factes of the component
  - *required* interfaces are to be offered by other components
- Multi-instantiation from component templates

```
COMPONENT Producer
  REQUIRES DataAcceptor;
END Producer;
```

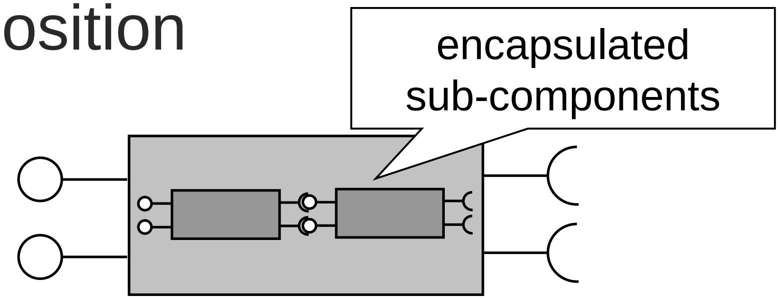
```
COMPONENT BoundedBuffer
  OFFERS DataAcceptor,
                DataSource;
END BoundedBuffer;
```

```
COMPONENT Consumer
  REQUIRES DataSource;
END Consumer;
```



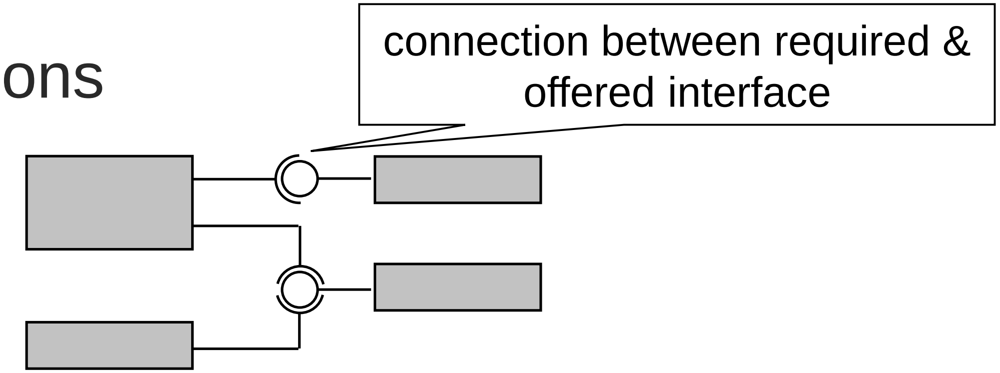
Component Relations

- Hierarchical composition



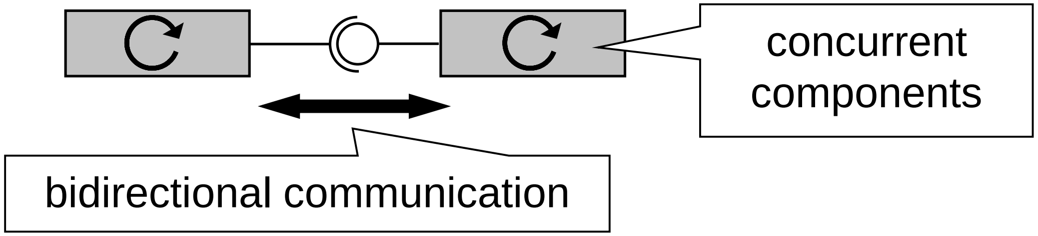
encapsulated sub-components

- Interface connections



connection between required & offered interface

- Communication-based interactions



concurrent components

bidirectional communication

Hierarchical Structuring

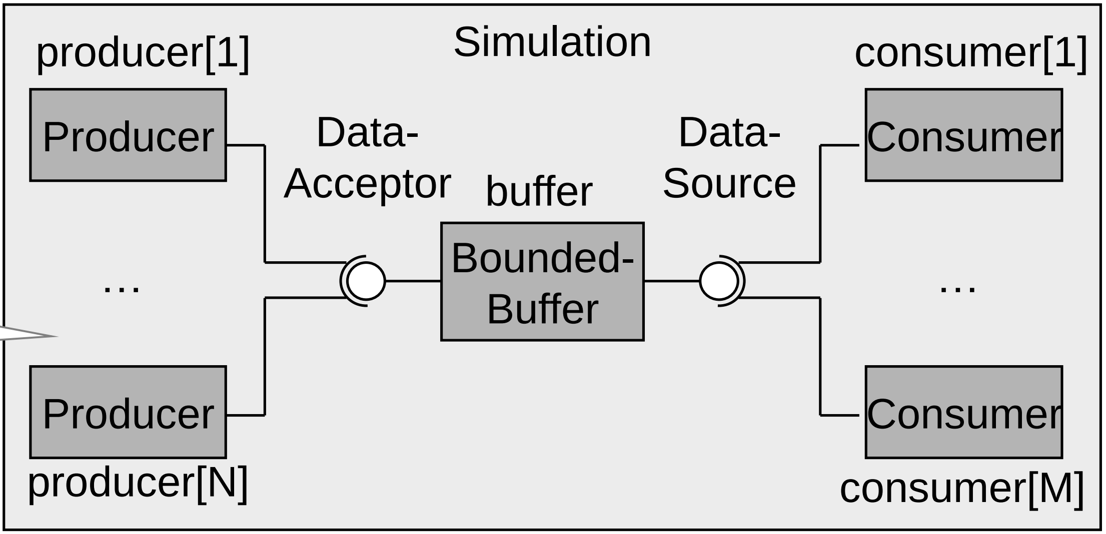
```
COMPONENT Simulation
 VARIABLE
  buffer: BoundedBuffer;
  producer[i: INTEGER]: Producer;
  consumer[k: INTEGER]: Consumer;
 BEGIN
  NEW(buffer);
  FOR i := 1 TO user input N DO
   NEW(producer[i]); CONNECT(DataAcceptor(producer[i], buffer)
  END;
  FOR k := 1 TO user input M DO
   NEW(consumer[i]); CONNECT(DataSource(consumer[i], buffer)
  END
END Simulation;
```

containers for components

dynamic collection

dynamic construction

network structure exclusively controlled by surrounding component



Simulation

producer[1]

Producer

…

Producer

producer[N]

Data-Acceptor

buffer

Bounded-Buffer

Data-Source

consumer[1]

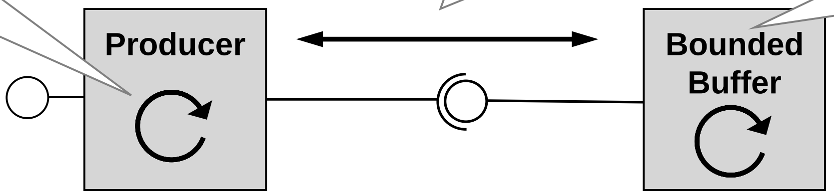
Consumer

…

Consumer

consumer[M]

Message Communication



intrinsic process

communication

maintains separate communication with each client

communication protocol in EBNF

```
INTERFACE DataAcceptor;
  { IN Element(x: INTEGER) }
  IN Finished
END DataAcceptor;
```

separate service process per client

```
COMPONENT Producer
  REQUIRES DataAcceptor;
BEGIN
  FOR i := 1 TO N DO
   DataAcceptor!Element(i)
  END;
  DataAcceptor!Finished
END Producer;
```

monitor synchronization inside component

```
COMPONENT BoundedBuffer
  OFFERS DataAcceptor;
  IMPLEMENTATION DataAcceptor;
  BEGIN {EXCLUSIVE}
    WHILE ?Element DO
     AWAIT(empty);
     ?Element(x); empty := FALSE
    END;
    ?Finished
  END DataAcceptor;
END BoundedBuffer;
```

Component Operating System

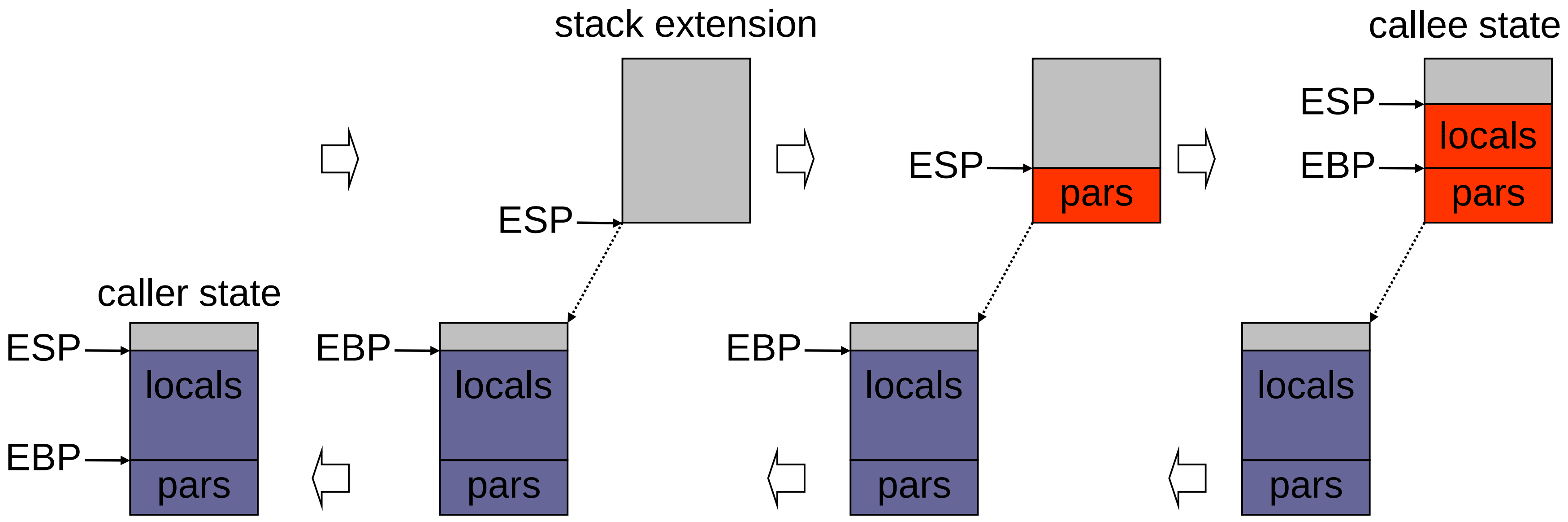
High-performance runtime system for structured concurrent programs of the component language

Highlights:

- Light-weighted processes
  - Micro stacks with size that can dynamically grow and shrink
  - Enables very high number of processes
- Fast context switches
  - Direct synchronous context switches
  - Low-cost and efficient preemption based on code-instrumentation
- Safe and efficient memory management
  - Garbage collection no longer needed due to hierarchical structures
  - Virtual memory management not needed

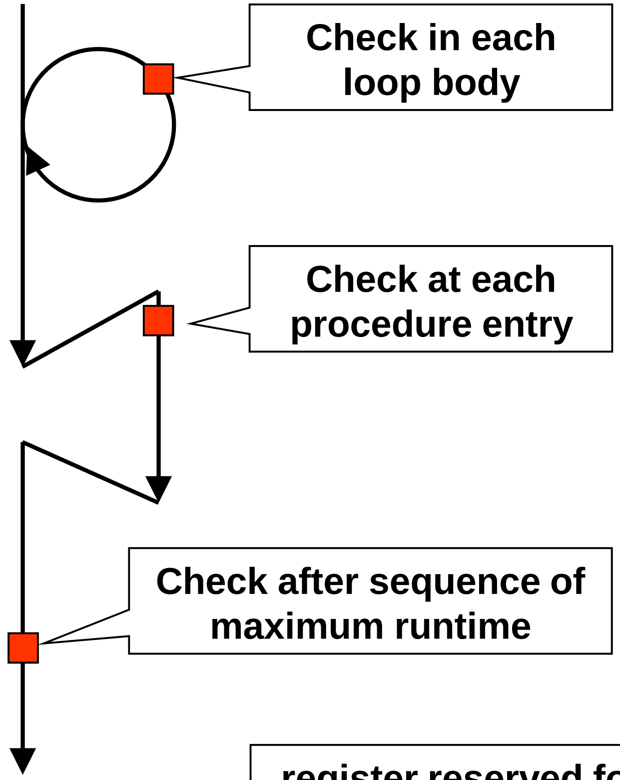
Stack Management

- Arbitrarily small stack sizes (not fixed to pages)
- Uniformly represented as heap blocks
- No method calls
  - Stacks only grow due to component-internal procedures
- System calls run on processor-associated system stacks
- Dynamic growing and shrinking
  - Compiler-inserted checks at procedure entry and exit



Process Management

- Direct context switches within monitors and communication
- Software-controlled preemption
  - Compiler automatically inserts checks in the machine code
  - Checks are executed in guaranteed small time intervals
  - Checks initiate preemption after a defined time interval
  - No cooperative multi-task programming
- Save only the necessary registers on preemption
  - Mostly no temporary registers used between statements
  - Economize extensive register backup space for each process
  - Runtime overhead of instrumented checks about 0.5%



Check in each loop body

Check at each procedure entry

Check after sequence of maximum runtime

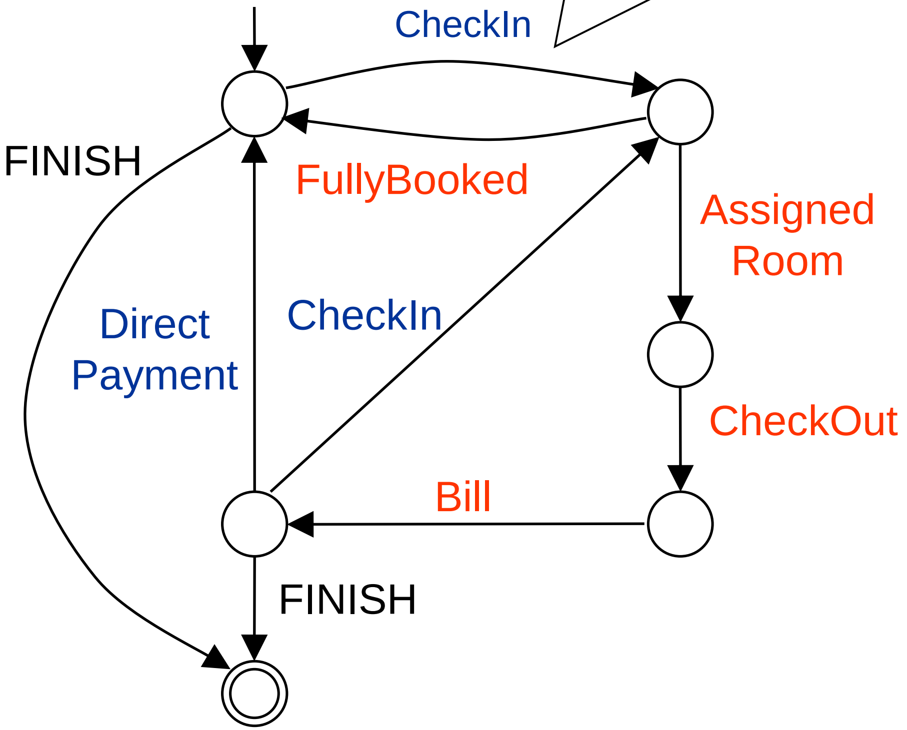
register reserved for preemption signal

```
  CMP EDI, 0
  JL continue
  save used registers
  CALL Preempt
  restore used registers
continue:
```

Communication Channel

- Bounded FIFO message buffer for each communication
- Automatic communication start and termination
  - Start by the first message command
  - Termination on the last protocol transition or on component finalization
- Dynamic protocol monitoring

```
INTERFACE Hotel
 {
  IN CheckIn
  (
   OUT AssignedRoom
   IN CheckOut OUT Bill
   [ IN DirectPayment ]
  |
   OUT FullyBooked
  )
 }
END Hotel
```

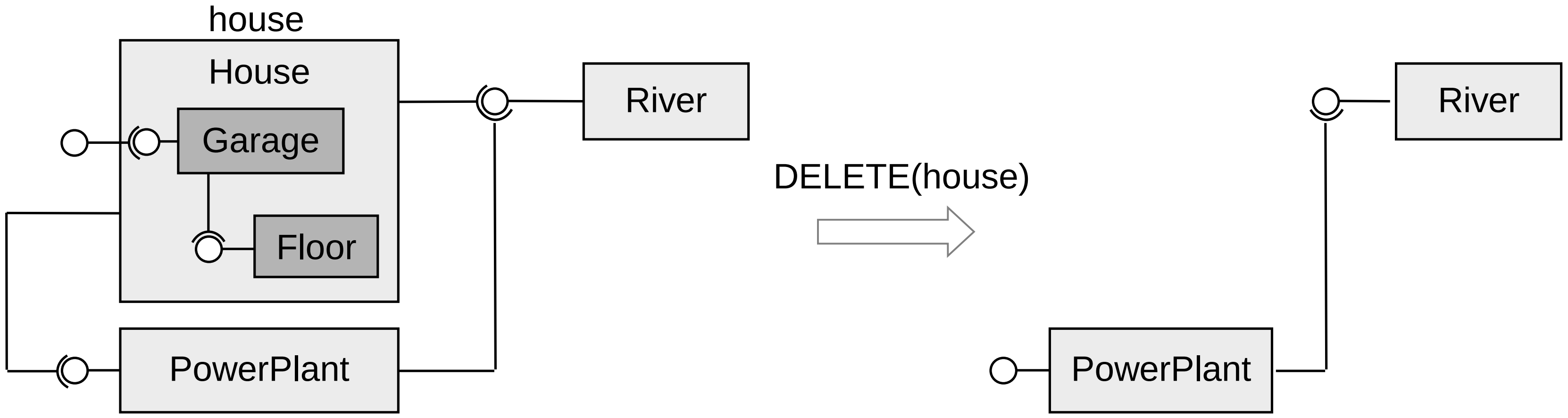


Compiler-generated state machine

Memory Management

- Hierarchy of component networks
  - Arbitrary *n*-to-*m* component networks within each component possible
  - Interface connections solely set by the surrounding component
- Hierarchical lifetime dependencies
  - Deletion of a component => automatic deletion of the sub-components
  - Explicit deletion of a component => outer interfaces of the component are safely disconnected

=> Safe memory management without garbage collection



Scalability und Performance

- Maximum number of processes (threads)

Component OS	C#	Java	Oberon AOS
5’010’000	1’890	10’000	15’700

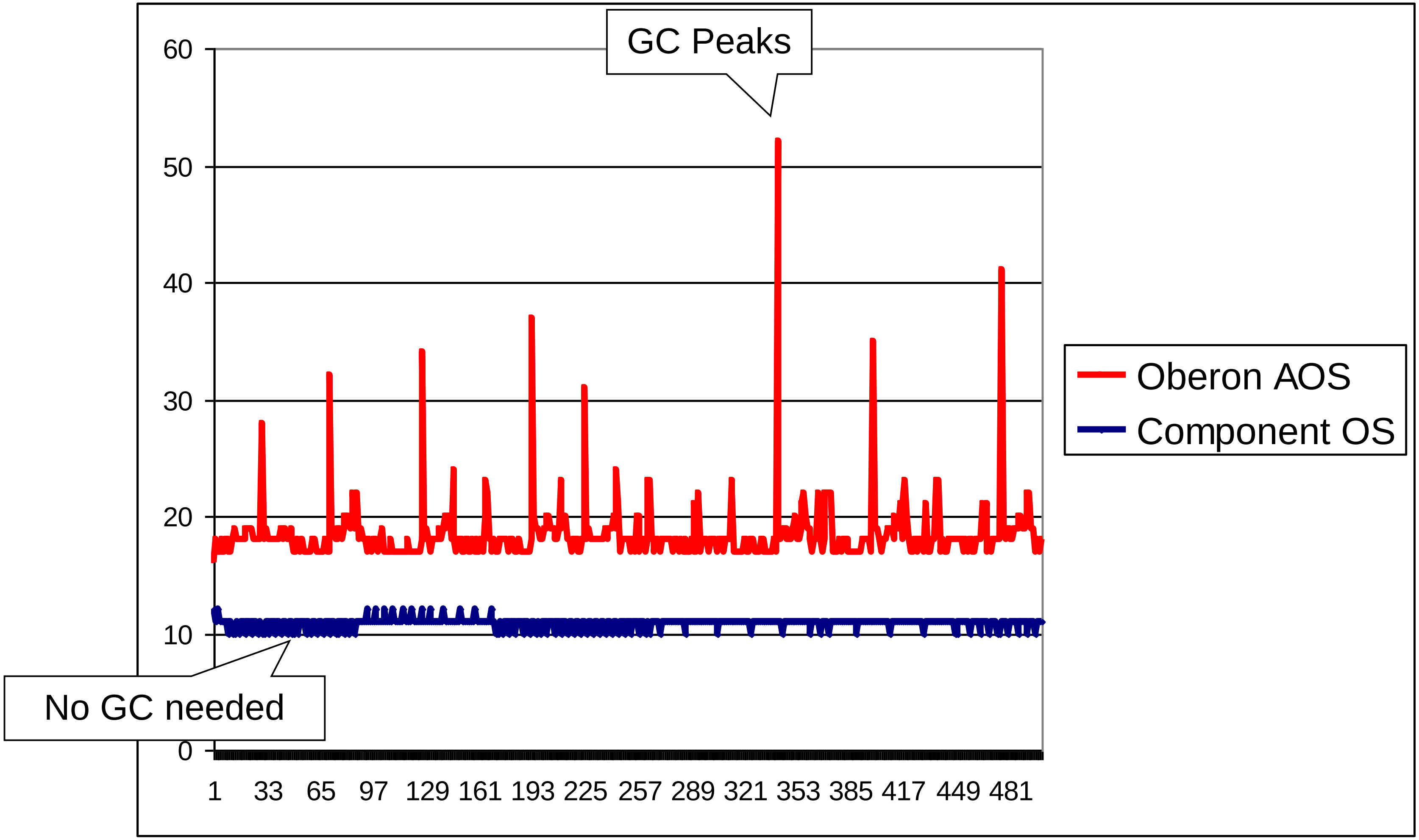
4GB main memory, City Benchmark

- Execution performance

Program (in sec)	Component OS	C#	Java	Oberon AOS
City	**0.24**	0.66	440	4.1
ProducerCons.	**16**	19	130	60
Eratosthenes	**1.8**	6.8	4.6	5.8
News	**2.0**	3.5	3.9	4.6
TokenRing	**2.1**	22	22	18
Mandelbrot	0.88	0.43	**0.39**	0.6
TrafficSimulation	**0.05**	33	-	-

6 CPUs Intel Xeon 700MHz, C# & Java Windows Server Enterprise Edition
C#, Java, AOS: analogous programs with methods instead of communication

Predictability without Garbage Collection



GC Peaks

No GC needed

500 subsequent executions of the Small City program (in ms)

Conclusions

A new efficient runtime system for a modern structured concurrent programming language

- Technical benefits
  - High scalability in the number of processes
  - Fast execution of concurrent programs
  - No garbage collection needed
  - Customized and optimized for structured concurrency
- Conceptual benefits
  - Hierarchical controlled structured and guaranteed encapsulation
  - Inherent concurrency with communication-based interactions
- Practical application in traffic simulation (TU Berlin)
- Project Website: http://www.jg.inf.ethz.ch/components