FASE: Fast Asynchronous System Evaluation
A tool for performance evaluation

Federico Buti, Massimo Callisto De Donato
Flavio Corradini, Maria Rita Di Berardini
{federico.buti, massimo.callisto, flavio.corradini,
mariarita.diberardini}@unicam.it

Dipartimento di Informatica - Università di Camerino
Via Madonna delle Carceri, 9 - 62032 Camerino
http://www.cs.unicam.it

PaCo Meeting - June, 26 2009
Agenda

- FASE presentation
- Quantitative performance evaluation techniques, some hints
- A case study: three buffer implementations
- Conclusions
Software *cross-platform*, completely written in Java
Software *cross-platform*, completely written in Java
- Interprets PAFAS strings from different dialects
Software *cross-platform*, completely written in Java
Interprets PAFAS strings from different dialects
Provides an arc-labeled graph representation of the given input (RTS, rRTS...)
Software *cross-platform*, completely written in Java
- Interprets PAFAS strings from different dialects
- Provides an arc-labeled graph representation of the given input (RTS, rRTS...)
- On these representations, applies performance calculation techniques based on the PAFAS *efficiency preorder* theory.

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini
Software *cross-platform*, completely written in Java

Interprets PAFAS strings from different dialects

Provides an arc-labeled graph representation of the given input (RTS, rRTS...)

On these representations, applies performance calculation techniques based on the PAFAS *efficiency preorder* theory.

Allows to compare performance of systems that *functionally* execute the same tasks but that are implemented in different ways.
Architectural overview

Lexer

get_next_token()

Rules

usable Java source code

internal conversion

JFlex

jacc

usable Java source code

compilation

runtime code

process

graph

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
Architectural overview

PafasState

ActionPrefix  Sum  Nil
P = a.b.nil + c.d.nil

\[ P = a \mid b \mid c \mid d \]

Input & Output (rRTS)

FASE: Fast Asynchronous System Evaluation
Response performance calculation is applied to a particular kind of processes called *response processes* which have some particular characteristics:

- **finite state processes**
- **available actions are only** "in"'s and "out"'s (and "τ"'s)
- **their number have to be balanced** (and in general "out"'s cannot exceed "in"'s in number)

If a response process is correct (verification could be done in linear time), then its response performance is finite and can be calculated.
Response processes

Response performance calculation is applied to a particular kind of processes called *response processes* which have some particular characteristics:

1. finite state processes
Response processes

Response performance calculation is applied to a particular kind of processes called *response processes* which have some particular characteristics:

1. finite state processes
2. available actions are only *in’s* and *out’s* (and *τ’s*)
Response processes

Response performance calculation is applied to a particular kind of processes called *response processes* which have some particular characteristics:

1. finite state processes
2. available actions are only *in’s* and *out’s* (and \( \tau \)’s)
3. their number have to be *balanced* (and in general *out’s* cannot exceed *in’s* in number)
Response processes

- Response performance calculation is applied to a particular kind of processes called *response processes* which have some particular characteristics:
  1. finite state processes
  2. available actions are only *in*’s and *out*’s (and *τ*’s)
  3. their number have to be *balanced* (and in general *out*’s cannot exceed *in*’s in number)

- If a response process is **correct** (verification could be done in *linear time*), then its response performance is finite and can be calculated.
• If the response process is not correct its response performance is $\infty$. This means that some requests will not be satisfied from the process within any time bound, which is certainly an incorrect behaviour.

• Typically an incorrect response process contains one (or more) catastrophic cycle(s).

• When a process enters one of these cycles, it is impossible to know when (and if) it will come out.

• Verification of these cycles is so crucial for testing performances.
Catastrophic cycles - an example

\[ \tau \parallel 1 \parallel \tau \parallel 1 \parallel \tau \]

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
Catastrophic cycles - an example

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
Catastrophic cycles - an example

FASE: Fast Asynchronous System Evaluation

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini
Catastrophic cycles - an example

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
Catastrophic cycles - an example
Catastrophic cycles - an example
N-critical paths - two input example
N-critical paths - two input example

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
N-critical paths - two input example

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini
N-critical paths - two input example

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
N-critical paths - two input example

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
N-critical paths - two input example

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
N-critical paths - two input example

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
The average cost (*average performance*) of a cycle can be calculated as the ratio between the number of full time steps and the number of *in*’s that compound it.

This ratio defines the *mean* cost for an input on the cycle. The maximum mean cycle is called **bad cycle**.
Asymptotic Linearity

As stated in theory, the response performance is asymptotically linear for response processes so that:

\[
an - c \leq f(n) \leq an + c \quad \text{with} \quad n \in \mathbb{N}.
\]

While complete calculation can be taken out through n-critical path verification, the bad cycle can give a generic characterization of it.

The average performance of the bad cycle corresponds to the coefficient \(a\) of the response performance.
Bad Cycle - an example

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
Bad Cycle - an example
Bad Cycle - an example
Bad Cycle - an example

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
- verifies that a process is a response process.
- verifies that the process does not contain catastrophic cycle(s) (returning them if present).
- calculates the average performance of the bad cycle.
- calculates the maximum n-critical path, and so, the response performance of the process for a given number of input.

The next few slides will review three different implementations of a buffer. We will see how FASE can be profitably used for the *comparison* of their performance.
Fifo

Implements a first-in-first-out queue with capacity $N + 2$. It has no overhead and is purely sequential.
Implementation with concatenated and single-valued cells, of capacity $N + 2$. Each cell is concatenated to the previous and the following ones and can be seen as a I/O “device”; every time a value is shifted, an internal action ($\tau$) is necessarily executed.
Implementation of $N + 2$ capacity, with independent and mono-valued cells. Each cell is connected to a buffer controller that stores a value as input and one as output, the index of the oldest value saved and the number of values stored in memory. The cells are maintained as a circular queue.
### Fifo performances

<table>
<thead>
<tr>
<th>( rRTS_{nodes/edges} )</th>
<th>( U_1 )</th>
<th>( U_2 )</th>
<th>( U_3 )</th>
<th>( U_4 )</th>
<th>( U_5 )</th>
<th>( U_6 )</th>
<th>( U_7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( rp_{Fifo}(3) )</td>
<td>9/18</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>( rp_{Fifo}(4) )</td>
<td>11/23</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>( rp_{Fifo}(5) )</td>
<td>13/28</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>( rp_{Fifo}(6) )</td>
<td>15/33</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>( rp_{Fifo}(7) )</td>
<td>17/38</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

### Complexity function

\[ 2n \]
# Pipe Performances

<table>
<thead>
<tr>
<th>( rRTS_{nodes/edges} )</th>
<th>( U_1 )</th>
<th>( U_2 )</th>
<th>( U_3 )</th>
<th>( U_4 )</th>
<th>( U_5 )</th>
<th>( U_6 )</th>
<th>( U_7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( rpPipe(3) )</td>
<td>20/42</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>( rpPipe(4) )</td>
<td>48/112</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>( rpPipe(5) )</td>
<td>114/292</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>( rpPipe(6) )</td>
<td>272/759</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>( rpPipe(7) )</td>
<td>648/1958</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
</tr>
</tbody>
</table>

## Complexity function

\[ 2n + N - 1 \] (with \( N \), equal to cells number)
## Buff performances

<table>
<thead>
<tr>
<th>rRTS&lt;sub&gt;nodes/edges&lt;/sub&gt;</th>
<th>U₁</th>
<th>U₂</th>
<th>U₃</th>
<th>U₄</th>
<th>U₅</th>
<th>U₆</th>
<th>U₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>( rP_{Buff}(3) )</td>
<td>17/34</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>( rP_{Buff}(4) )</td>
<td>48/104</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>( rP_{Buff}(5) )</td>
<td>96/216</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>( rP_{Buff}(6) )</td>
<td>160/368</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>( rP_{Buff}(7) )</td>
<td>240/560</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

## Complexity function

\[ 4n \]
Recapitulatory graph (three cells)

Requests

Time

Buff  Pipe  Fifo

F. Buti, M. Callisto, F. Corradini, M.R. Di Berardini

FASE: Fast Asynchronous System Evaluation
Our results clearly state that Fifo is *the fastest* between the considered implementation. In other words, the following preorder relations holds:

\[
\text{Fifo} \sqsubseteq \text{Pipe} \sqsubseteq \text{Buff}
\]

These results are *supported* in part by the qualitative ones. Where it was possible to provide results, the formers are confirmed by the latters (see, for example Pipe and Buff or fifo and Buff).

What about the other results?
Our results clearly state that Fifo is the fastest between the considered implementation. In other words, the following preorder relations holds:

\[ \text{Fifo} \sqsubseteq \text{Pipe} \sqsubseteq \text{Buff} \]

These results are supported in part by the qualitative ones. Where it was possible to provide results, the formers are confirmed by the latters (see, for example Pipe and Buff or fifo and Buff).

What about the other results?
Our results clearly state that Fifo is *the fastest* between the considered implementation. In other words, the following preorder relations holds:

\[
\text{Fifo} \sqsubseteq \text{Pipe} \sqsubseteq \text{Buff}
\]

These results are *supported* in part by the qualitative ones. Where it was possible to provide results, the formers are confirmed by the latters (see, for example Pipe and Buff or fifo and Buff).

What about the other results?
Our results clearly state that Fifo is *the fastest* between the considered implementation. In other words, the following preorder relations holds:

\[ \text{Fifo} \sqsubseteq \text{Pipe} \sqsubseteq \text{Buff} \]

These results are *supported* in part by the qualitative ones. Where it was possible to provide results, the formers are confirmed by the latters (see, for example Pipe and Buff or fifo and Buff).

What about the other results?
For what concern Fifo and Pipe, qualitative results clearly state that they are not related or in other words it is impossible to define if one is faster than the other.

From theory we have a proposition that clearly relates the qualitative and quantitative:

Given $P$ and $Q$ testable, $P \sqsubseteq Q$ iff for all tests $O$ we have $p(P \parallel O) \leq p(Q \parallel O)$, i.e. $p_P \leq p_Q$

So...what is the catch??
Final thoughts

- For what concern Fifo and Pipe, qualitative results clearly state that they are *not related* or in other words it is impossible to define if one is faster than the other.
- From theory we have a proposition that clearly relates the qualitative and quantitative:

  Given P and Q testable, $P \equiv Q$ iff for all tests $O$ we have $p(P||O) \leq p(Q||O)$, i.e. $p_P \leq p_Q$

- So... what is the catch??
The problem lies in the tests taken into consideration. While qualitative analysis takes into account any kind of test, quantitative measuring examines only *user processes* with a well-known structure.

This means that not all the available traces in a testing scenario (qualitative one) are available also in the other (quantitative one).

It is easy to argue that, by narrowing the class of tests in the qualitative theory (and thus redefining the preorder operator), we can obtain the desired results.

Besides that, all the other results demonstrate the quality of the work done up to now.