9th International Erlang/OTP User Conference

Stockholm, November 18, 2003

Proceedings

EUC'2003 http://www.erlang.se/euc/03/
Ericsson AB
P.O. Box 1505
SE-125 25 Älvsjö Stockholm
Sweden

ERICSSON

ERLANG

Erlang/OTP User Conference 2003

Conference Programme

08.30 *Registration.*

Applications I

09.00 **Why we designed Erlang.**
Mike Williams, Ericsson.

09.25 **Yet Another Web Gui Framework.**
Göran Båge and Johan Blom, Mobile Arts.

09.50 **3D Video Game Development in Erlang.**
Mickaël Rémond.

10.30 *Coffee.*

Applications II

11.00 **How we utilized Erlang to Develop a Banking System and Financial Transaction Switches.**
Danie Schutte, Teba Bank.

11.30 **CyberAthletics - Open Source and the Age of Wireless.**
Ulf Wiger, Ericsson.

12.00 **Talking LDAP and Radius from Erlang.**
Torbjörn Törnkvist, Nortel.

12.30 *Lunch.*

Technology I

14.00 **Getting Erlang to Talk to C and C++** (or from ei to UBF).
Hal Snyder, Vail, and Leon Smith, Case Western Reserve University.

14.30 **Erlang/QuickCheck.**
Thomas Arts, IT-university, and John Hughes, Chalmers.

15.00 **Performance Analysis using Model Checking.**
Thomas Arts, IT-university, and Juan José Sánchez Penas, University of Corunha.

15.30 *Coffee.*

Technology II

16.00 **All you wanted to know about HiPE** (and might have been afraid to ask).
K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson and T. Lindahl, Uppsala University.

16.30 **Profile-driven Inlining for Erlang.**
Thomas Lindgren.

17.00 **What's new in R9C.**
Kenneth Lundin, Ericsson.

17.30 *Close (and pub evening).*

Demonstrations (during intermissions)

Mickaël Rémond demonstrates the **3D Video Game.**
Torbjörn Törnkvist demonstrates the **Ticket Tracker.**

ERICSSON

Why did we create Erlang?

Mike Williams
Ericsson AB
Stockholm
Sweden

mike@erix.ericsson.se

ERICSSON

Maybe it didn't happen exactly this way, but this is the way I think it should have happened

ERICSSON

Problem Domain - Highly concurrent and distributed systems

- Thousands of simultaneous transactions
  - Light weight transactions
  - Greatest CPU load is implementing concurrency and communication not computation
- Many computers
  - different types (Bigendians, Littleendians, Intel, Sparc, PowerPC etc)
  - share nothing (no shared memory, different communication mechanisms (Ethernet, ATM, Proprietary))
- Many OS's
  - Solaris, VxWorks, Windows, pSOS, Linux, etc

ERICSSON

Problem Domain - No down time

- Not allowed to have any planned or unplanned downtime
  - Acceptance criterion: five nines = 99.999% uptime or 5 minutes down time per year
- Recovery from software errors
  - Large systems will have software bugs
- Recovery from hardware failure
  - Network failure, processor failure
- Enable adding / deleting computers and other hardware at run time
- Update code in running systems

ERICSSON

Problem Domain - Ease of programming

- Highly "expressive" programming language
- Easy portability between processor architectures
- Large scale development (tens or even hundreds of programmers)
- Incremental and exploratory programming
- Debugging and tracing - even in systems running at customer sites
- Easy to fix bugs (patches) and upgrade at all phases of design even in systems running at customer sites

ERICSSON

Solution Domain - Concurrency

- No existing industry quality OS or language offers light weight enough threads / processes
- Processes must be independent
  - No shared resources
  - One process must not be able to destroy another process
  - Reduce event/state matrix by selective message reception

ERICSSON

Solution Domain - Concurrency & Distribution

- As we didn't want to modify or create and new OS, implementation of light weight processes needed to be done in "middleware", I.e. on top of the OS.
- Making processes independent requires either control of the MMU or a language without pointers (or with safe pointers)
- Reducing the event/state matrix makes the signal / state model undesirable.
  - The signal state model requires a thread only suspending at the top level, not in a function/subroutine. This makes proper RPC's impossible

ERICSSON

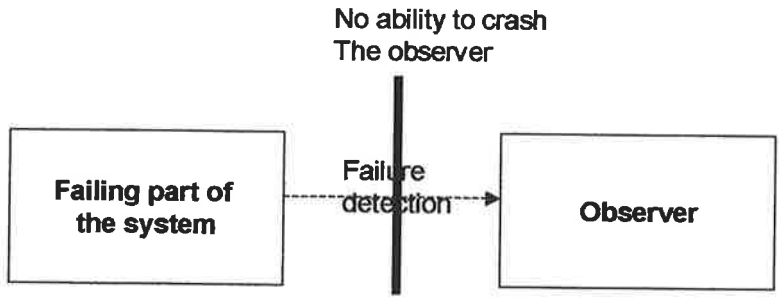
Solution Domain - Concurrency & Distribution: Design decisions

- Implement concurrency in a virtual machine on top of operating system.
- Use a language without explicit pointers.
- Use copying message passing as only interprocess communication mechanism.
- Implement selective message reception.
- Make communication between processes on different machines identical to communication between processes on same machine.
  - Type information retained at runtime enables automatic conversion of Erlang terms to an external format.

ERICSSON

Solution Domain - No down time

- Principle for error detection: *It is unsafe to allow the failing part of the system to detect and correct failures itself*



ERICSSON

Solution Domain - No down time

- A software error in one process is best detected in another process
- Failure of one processor is best detected by another processor
- Frequently we want to be able to abort all the processes in a transaction if one of them fails for some reason

ERICSSON

Solution Domain - No down time Design Decisions:

- Create a concept of a "link" between processes. If a process fails, a special message (a signal) is sent to all the processes to which it has links.
- Default action of a process receiving a signal indicating failure of a process is to "die" and send on the signal to all linked processes.
- By setting a special flag, (trap_exit) a processor can override the default behaviour and receive the signal as an ordinary message.
- Links are bi-directional - (maybe a design mistake?)

ERICSSON

Solution Domain - No down time Design Decisions:

- Two cases:
  - Server with a lot of clients. If a client fails sever needs to take corrective action
  - A lot of processes in a transaction □ if one fails, all should fail.
- Link and Signal mechanism works across processor boundaries.
  - If a processor fails, signals will be sent to all processes which have links to processes in the failing processor.
- Error handling philosophy: "Let it crash" and let other processes clear up the mess.

ERICSSON

Solution Domain - No down time

- Common design paradigm:
  - Let all active transactions be represented by groups of linked processes
  - Store inactive (steady state) transactions in replicated robust database (Mnesia)
  - Let resources needed by transactions be allocated by resource allocator processes which trap_exits and free up resources from failing transactions
  - Supervisor processes which trap_exits restart failing application on suitable processors. Data for these applications is the configuration data needed and the data for transactions in a steady state. (same mechanism used for replacing processors).

ERICSSON

Solution Domain - No down time
Design Decisions:

- Design the virtual machine so new code can be loaded and processes can migrate to the new code.
- Ability to detect processes running old code.
- Design the standard design patterns (part of OTP) so that they can convert data to a new format if needed.

- Application software needs to be aware of possible software updating and failure recovery, but with Erlang/OTP support the impact is minimised

ERICSSON

Problem Domain - Ease of programming (reminder)

- Highly "expressive" programming language
- Easy portability between processor architectures
- Large scale development (tens or even hundreds of programmers)
- Incremental and exploratory programming
- Debugging and tracing - even in systems running at customer sites
- Easy to fix bugs (patches) and upgrade at all phases of design - even in systems running at customer sites

ERICSSON

Problem Domain - Ease of programming Design Decisions:

- Use high level functional language with automatic memory handling and garbage collection
- Use execution of intermediate code by virtual machine to obtain easy portability between processor architectures
- Simple non/hierarchical module system
- Erlang shell allows testing of functions directly without any special test programs
- Virtual machine support for debugging and fault tracing
- Dynamic code replacement also very useful while developing / testing software

ERICSSON

Comments

- We have frightened some people off by using:
  - A functional language
  - A non O-O language
  - Recursion, single assignment etc
  - A virtual machine
- I.e. we have diverged a long way from industry mainstream. We are changing very many parameters at the same time.
  - Attitude changes in "mainstream" is possible (remember what people said about Garbage Collection before Java?)

ERICSSON

Comments

- The use of Erlang is accelerating, the critical mass will soon be reached!

Yet Another Web GUI Framework

Göran Båge, Johan Blom

Mobile Arts

- Provides Messaging & Presence products to Mobile Network Operators
- Offices in Stockholm and London
- References in Europe
- Founded 2001 by a team of mobile telecom experts with extensive experience from development and standardisation of GSM/UMTS/PDC/PCS platforms and applications, such as
  - HLR, MSC/VLR, SSF, SCF, WAP-GW, MLC
  - PrePaid Systems, Unified Messaging Systems
- www.mobilearts.se

Background

- GUI for an Alteon/Nortel Product
  - Beautiful & powerful
  - Never released?
  - Client required Erlang and GTK
- CLI and GUI for Mobile Arts products
  - Web based GUI

Mobile Arts

Some observations

- Using a graphics environment, e.g GTK, Java
  - \+ Powerful
  - \+ Detailed layout
  - – Client box need special software
- Web based GUI
  - \+ Everyone has a browser
  - \+ No need to install special client software
  - – Only polling
  - – Browser differences
  - – Limited layout control

Mobile Arts

Mobile Arts GUI, first implementation

- Started out pure and simple
  - Standard HTML only
  - No frames
  - CSS
- but reality rules
  - Frames
  - Javascript
- Messy result
  - Browser dependencies (support Netscape 4, 7 and IE) in Javascript and CSS
- Many pages with similar functionality and layout

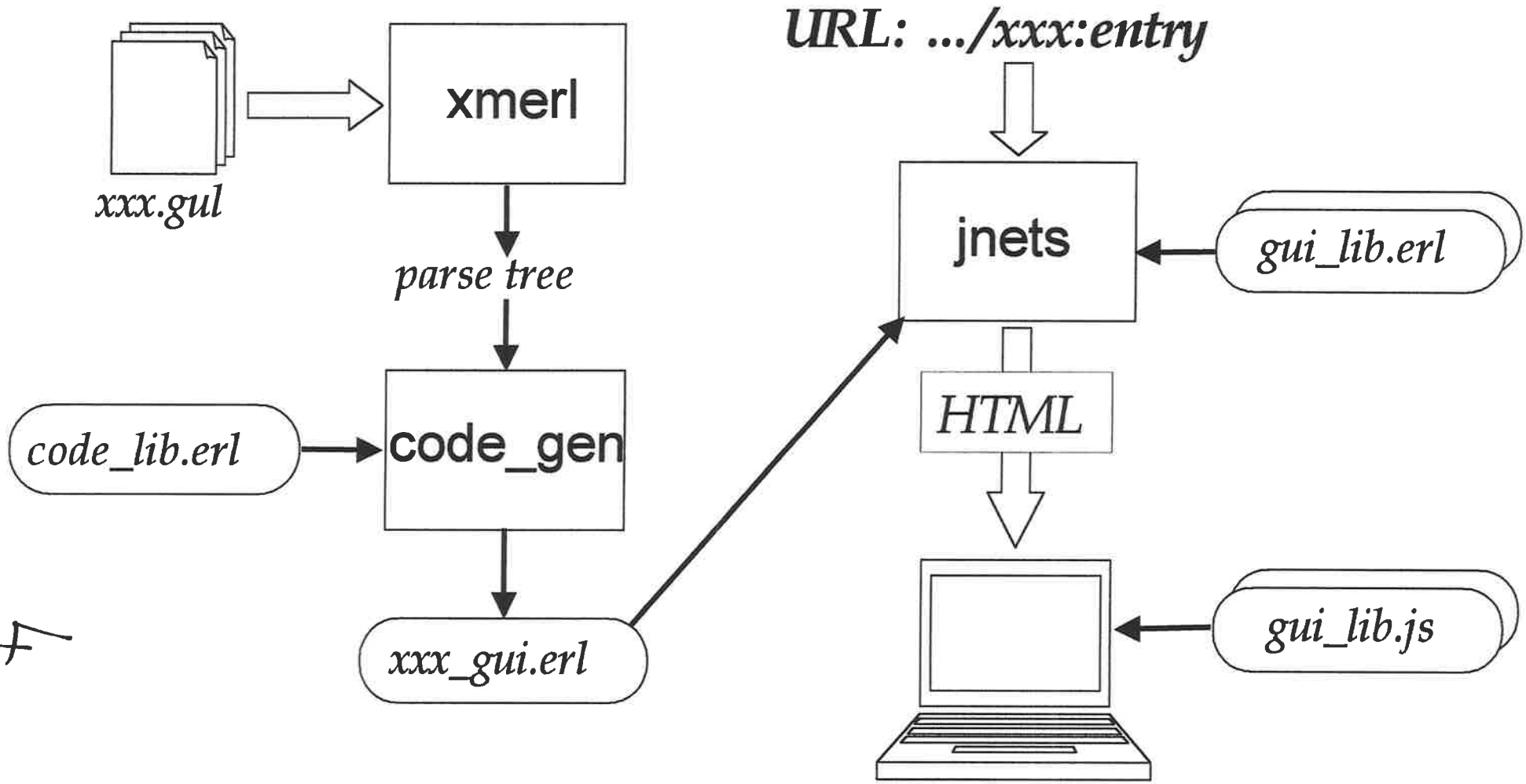
Mobile Arts

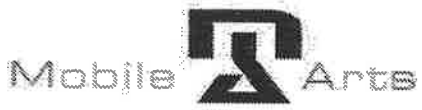
Template based GUI

- XML like template notation
- Separate different views
  - Static structure and layout
  - Dynamic content - interface to application
    - button actions
    - table content and options
    - parameter checks
    - ...
  - Help texts

Mobile Arts

How it works


xxx.gul
xmerl
parse tree
code_lib.erl
code_gen
xxx_gui.erl
URL: .../xxx:entry
jnets
gui_lib.erl
HTML
gui_lib.js



Modify operator test

Close

Password:

Name: Kalle Testare

Email: abc@def

Description:

Save

Reset input fields

Close

```
%% Template source for "modify_page_example"

%% Static/structure part
%%
<page name="operator_modify" style="command-page"
      import="user_modify_gui_lib"
      jslib="operator_client_gui_lib"
      size="500" >

  <section name="all" >
    <button name="close" title="Close" action="close" />
    <section name="data" style="command-area" layout="1" >
      <section layout="2" >
        <text> Password: </text>    <password name="pwd" size="20"
 />

        <text> Name: </text>        <input name="name" size="1x20"
 />

        <text> Email: </text>       <input name="email" size="1x20
" />

        <text> Description: </text> <input name="descr" size="3x30
" />
        <wrapper name="service" >
          <text> Service: </text>  <select name="service" />
        </wrapper>
        <wrapper name="timeout">
          <text name="timeout_text"> CLI/GUI inactivity timeout:
</text>
          <select1 name="timeout" />
        </wrapper>
      </section>
      <button name="save" title="Save" />
      <section>
        <button name="reset" title="Reset" action="reset" />
        <text> input fields </text>
      </section>

    </section>
    <ref alias="all:close" />

  </section>

</page>

%% ---------------------------------
%% Dynamic part part
%%
<code name="operator_modify"
      title="modify_title($action, $add_op:operator:name,
```

```
                              $operator_op:select:operator)"
        lock="modify_lock($action, $operator_op:select:operator)" >
   <local name="action" value="gui_lib:get_operation()" />
   <local name="js_action" value="operation_js($action)" />
   <local name="operator"
          value="get_operator($action, $add_op:operator:name,
                                $operator_op:select:operator)" />

   <section name="all" >
     <script tag="password" name="data:pwd"
             onchange="check_password(^data:pwd)" />
     <eval name="data:name" default="get_name($data:name, $operator
)" />
     <eval name="data:email" default="get_email($data:email, $opera
tor)"
/>
     <script tag="input" name="data:email"
             onchange="check_email(^data:email)" />
     <eval name="data:descr" default="get_descr($data:descr, $opera
tor)"
/>
     <eval name="data:service:" show="show_service($action)" />
     <eval name="data:service:service"

value="operator_gui_lib:service_options($data:service:service)"
           type="service_select_type()" />
     <eval name="data:timeout" show="show_timeout($action, $operato
r)"
/>
     <eval name="data:timeout:timeout"
           value="timeout_options($data:timeout:timeout)" />
     <script tag="select1" name="data:timeout:timeout"
             onclick="check_ask_value(^data:timeout:timeout,5,'posi
nt')"
/>
     <eval name="data:save"
           action="do_save_data($operator, $data:pwd, $data:name,
                                 $data:email, $data:descr,
                                 $data:service:service,
                                 $data:timeout:timeout)" />
     <script tag="button" name="data:save"
             onclick="check_user_data($js_action,^data:pwd)" />
   </section>

</code>

%% ---------------------------------
%% Help text part
%%

<help name="operator_modify">
  <info name="all:close">
```

```
    <hint> Close page </hint>
    <message> Press button to close and remove page </message>
  </info>
  <info name="all:data:save">
    <hint> Save operator data </hint>
    <message> Press button to save data and remove page </message>
  </info>
  <info name="all:data:reset">
    <hint> Reset input fields
    </hint>
    <message>
      Press button to reset all input fields to original values.
    </message>
  </info>
  <info name="all:data:pwd">
    <hint>  Enter (new) password </hint>
    <message>
      Enter (new) password here. You will be asked to verify this
password
      before the (new) operator state is saved.
    </message>
  </info>
  <info name="all:data:name">
    <hint> Enter new or edit operator name </hint>
    <message>
      Enter new or edit old operator name information. Note that t
he
name
      is not the same as the operator (login) id.
    </message>
  </info>
  <info name="all:data:email">
    <hint> Enter new or edit operator email address </hint>
    <message> Enter new or edit old operator email address. </mess
age>
  </info>
  <info name="all:data:descr">
    <hint> Enter new or edit operator description </hint>
    <message> Enter new or edit old operator description. </messag
e>
  </info>
  <info name="all:data:service:service">
    <hint>  Select service </hint>
    <message> Select service for this operator. </message>
  </info>
  <info name="all:data:timeout:timeout">
    <hint> Set CLI and GUI inactivity timeout (min) </hint>
    <message>
      Set an inactivity time limit in minutes for CLI and GUI. If
the
user is
      inactive longer that the given time the operator will be log
```

```
ged
out.
    </message>
  </info>
</help>
```

```
%% The tree parts are merged using the name (and tag) attributes

<page name="operator_modify" style="command-page" ...>

  <section name="all" >
    ...
    <section name="data" style="command-area" layout="1" >
      <section layout="2" >
        <text> Password: </text>  <password name="pwd" size="20" /
>

        <text> Name: </text>      <input name="name" size="1x20" /
>

        ...
        <wrapper name="service" >
          <text> Service: </text> <select name="service" />
        </wrapper>
        <wrapper name="timeout">
          <text name="timeout_text"> CLI/GUI inactivity timeout:
</text>
          <select1 name="timeout" />
        </wrapper>
      </section>
      <button name="save" title="Save" />
      ...
    </section>
    ...
  </section>

</page>

%% ---------------------------------
%% Dynamic part part
%%
<code name="operator_modify" ...>

  <section name="all" >
    %% Add onchange handler to password clause 'pwd' in section 'd
ata'
    %% (in 'all')
    <script tag="password" name="data:pwd"
            onchange="check_password(^data:pwd)" />
    %% Set default value of input clause 'name' in section 'data'
    <eval name="data:name" default="get_name($data:name, $operator
)" />
    ...
    %% Show wrapper combo all:data:service if condition evaluates
to
true
    %% (not visible in example page)
    <eval name="data:service:" show="show_service($action)" />
```

```
    ...
    %% Add computed options to select1 clause all:data:timeout:tim
eout
    <eval name="data:timeout:timeout"
          value="timeout_options($data:timeout:timeout)" />
    ...
    %% Add action code to button all:data:save
    <eval name="data:save"
          action="do_save_data($operator, $data:pwd, $data:name, .
..)"
/>
    %% Add onlick handler to button all:data:save
    <script tag="button" name="data:save"
            onclick="check_user_data($js_action,^data:pwd)" />
  </section>

</code>

%% ---------------------------------
%% Help text part
%%

<help name="operator_modify">
  ...
  <info name="all:data:save"> %% Button 'save' in sect 'data' in s
ect
'all'
    <hint> Save operator data </hint>
    <message>  Press button to save data and remove page </message
>
  </info>
  ...
  <info name="all:data:pwd">
    <hint> Enter (new) password </hint>
    <message>
      Enter (new) password here. You will be asked to verify this
password
      before the (new) operator state is saved.
    </message>
  </info>
  ...
</help>
```

```
%% Variables/parameters are form-input values passed from caller (
may
be
%% the calling frame/page or this page) or local variables. Variab
le
%% identifiers start with $ (value of) or ^ (reference)

%% Static/structure part
%%
<page name="operator_modify" style="command-page"
      import="user_modify_gui_lib" ....>

  <section name="all" >
    ...
    <section name="data" style="command-area" layout="1" >
      <section layout="2" >
        <text> Password: </text>    <password name="pwd" size="20"
 />
        ...
      </section>
      <button name="save" title="Save" />
      ...
    </section>
    ...
  </section>

</page>

%% ---------------------------------
%% Dynamic part part
%%
%% The title attribute is computed by calling
%%   user_modify_gui_lib:modify_title
%% with parameters
%%   $action                  value of local parameter
%%   $add_op:operator:name value of name entered in input clause
%%                            add_op:operator:name in calling frame
%%   $operator_op:select:operator  value of selected option in cla
use
%%                            operator_op:select:operator in calling
frame
<code name="operator_modify"
      title="modify_title($action, $add_op:operator:name,
                          $operator_op:select:operator)"
      lock="modify_lock($action, $operator_op:select:operator)" >
  %% The local variable action is set to the current action,  i.e.
 the
value
  %% of the calling button id ("add" or "mod")
  <local name="action" value="gui_lib:get_operation()" />
  %% The action converted to a javascript string ('add' or 'mod')
```

```
  <local name="js_action" value="operation_js($action)" />
   ...
  <section name="all" >
    ...
    %% when the button is pressed the javascript function
    %% check_user_data is called with the current action ('add' or

'mod')
    %% and a reference to the password input object corresponding
    %% to the clause  all:data:pwd
    <script tag="button" name="data:save"
            onclick="check_user_data($js_action,^data:pwd)" />
  </section>

</code>
```

ERICSSON

CyberAthletics

Open Source and the Age of Wireless

ERICSSON

Problem

- Athletics events are difficult to administer
- They are also difficult to follow

- The Swedish Athletics Federation would like some software that supports event administration, statistics reporting and continuous results service to the audience.
- ...but the Federation has no money

ERICSSON

Existing alternatives

- A few companies have their own software as part of a package deal – expensive, and only an alternative at big events.
- A few people have written their own software – often incomplete, not extensible, and with no long-term support.
- Administering a "small" event is about as difficult as administering a fairly "big" event, but there is no money in it.

ERICSSON

The Open Source alternative

- Athletics in Sweden is a non-profit Popular Movement ("folkrörelse"). So is Open Source software development
- The software can be made available for free to all clubs in the country
- Maintenance/improvements can be free/spontaneous or paid for in preparation for important events
- Commercial players can build strategic add-ons on top of the Open Source base, and charge for it (e.g. custom adaptations for televised events)

ERICSSON

The Mission

- Create an Open Source project aimed at developing software for Athletics events.
- Try to tap into the pool of IT professionals devoting their spare time to Athletics (there are lots of them)
- Try to involve universities
- Try to get commercial players to "plug in" proprietary features for high-profile events (TV adaptation, custom telemetry, custom presentation, ...)

Rev PA1 2003-09-23 5 EAB/UPD/S Ulf Wiger

ERICSSON

Not a New Idea

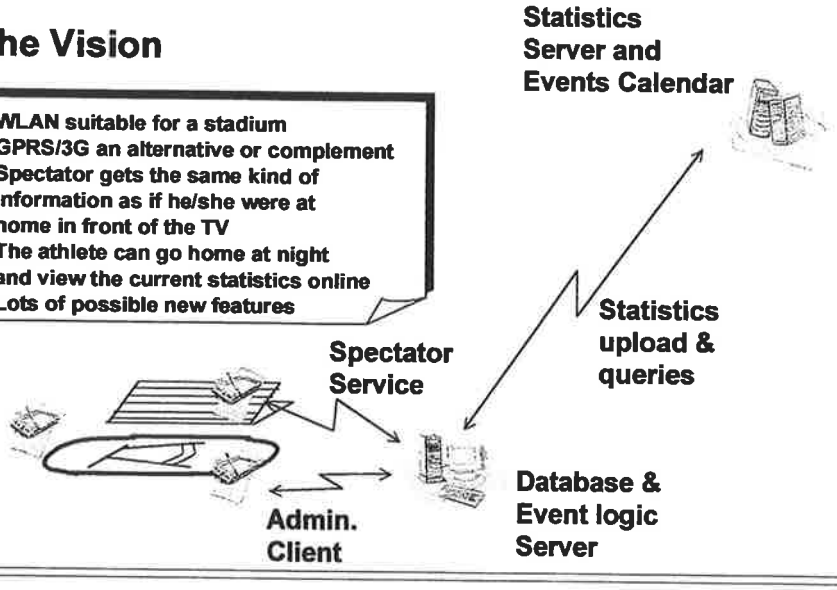
- We talked about doing this in 1985, but
  - Wireless networks didn't exist and pulling coax cable across the stadium seemed problematic
  - Computers were slow, clumsy and expensive
  - Freeware existed (GNU started in 1984), but nobody (at least not I) knew what it was
  - The World Wide Web revolution was years away
- We conducted some experiments and then shelved the idea... until now

Rev PA1 2003-09-23 6 EAB/UPD/S Ulf Wiger

ERICSSON

The Vision

- WLAN suitable for a stadium
- GPRS/3G an alternative or complement
- Spectator gets the same kind of information as if he/she were at home in front of the TV
- The athlete can go home at night and view the current statistics online
- Lots of possible new features



ERICSSON

The Prototype

- Erlang-based server (so far 3.4 KLOC) for event logic
- Java-based client (so far 6.5 KLOC) for PDAs/laptops
- Client-server protocol specified using UBF
- Two user modes: 'spectator' and 'administrator'

ERICSSON

Server components

- Ubf – for protocol specification
- Mnesia/Rdbms – for database integrity
- Builder – for boot script support
- "packages" – for name space handling (& evaluation)

ERICSSON

UBF

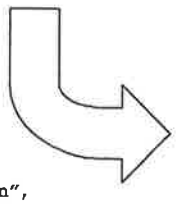
- Wonderful for client-server programming
- Quite stable
- Elegant transport format – easy to read/debug
- All type-checking code is automated – bliss!
- We had to add some stuff
  - OTP framework and packaging of the UBF code
  - Java-based UBF parser
  - Contract-to-hrl file generator

ERICSSON

UBF Example

cath_plugin.con (UBF contract)

```
athlete_id()        = constant();
entry_number()      = int();
born()              = int();
athlete()           = {athlete, athlete_id(), name(),
                       born(), gender(), club(), info()};
```



```
ubf:con2hrl(
  "cath_plugin.con",
  [{outfile,".../cath_plugin.hrl"}]).
```

cath_plugin.hrl (Erlang source)

```
-record(athlete, {athlete_id,
                  name,
                  born,
                  gender,
                  club,
                  info}).
```


ERICSSON

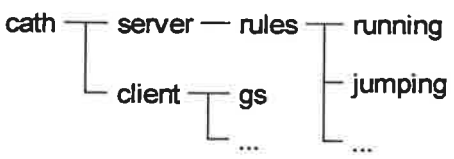
Mnesia/Rdbms

- Rdbms automates referential integrity checks for mnesia (e.g. "when deleting a club, also delete all its athletes")
- <TabName>.<Contest> naming scheme – a new set of tables for each contest (some tables have global scope)
- Prepared for physical separation of administrator and spectator access ('writer' copies and 'reader' copies) — unsure whether this will be needed even for big events.
- Making full use of Mnesia's transaction support

ERICSSON

Packages

- Someone needed to evaluate it...
- Nice to get rid of the namespace problem
- You get used to the deep directory structure
- I had to add some patches to OTP in order to get 'builder' to work
- I would use it again



ERICSSON

The future

- Goal: to run this at some major event late next year (after adding more features and running some field trials)
- Need to figure out how to best get universities to cooperate with Open Source projects
- Start tying in commercial interests
- Demonstrate to foreign Athletics federations
- Try to attract an active developer community

ERICSSON

Possible new features

- Video on demand, event camera, replays, etc.
- Audio channel, expert commentary per event
- Chat, possibly also including the athletes
- Integrate electronic timing, wind measurements, etc.
- Statistics queries on-line
  (includes the central statistics server, not yet started)
- XMLQuery filters for media
- Using e.g. ErlGuten to generate snazzy hardcopy reports

Talking LDAP and Radius from Erlang

Torbjörn Törnkvist

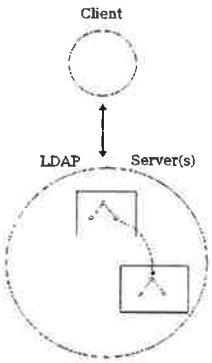
tobbe@nortelnetworks.com

NORTEL NETWORKS

Slide 1

What is LDAP ?

- LDAP provides directory access, a centralized database of information about people, groups and other entities.
- Defined as a set of protocol operations against servers.
- Assumes one (or more) servers which jointly provide access to the DIT (Directory Information Tree)
- Protocol described in ASN.1

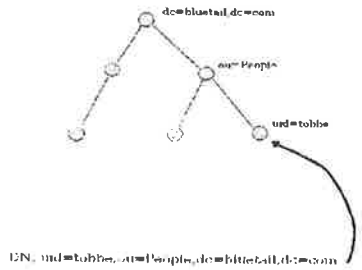


NORTEL NETWORKS

Slide 2

The Directory Information Tree (DIT)

- **The DIT is made up of entries.**
- **Entries have names consisting of one (or more) attribute values.**
- **The concatenation of the entry names form a path, the Distinguished Name (DN), which uniquely identifies an entry.**



NORTEL NETWORKS

Slide 3

(Main) Protocol Operations

- **Add/Delete/Modify entries.**
- **Search the DIT (retreiving info)**
- **Authenticate the client (the bind-operation)**

NORTEL NETWORKS

Slide 4

Example: from the Erlang shell

```
1> {_,S} = eldap:open(["192.168.128.47"], []).
{ok,<0.30.0>}
2> eldap:simple_bind(S,"cn=Torbjorn Tornkvist,cn=Users,dc=bluetail,dc=com","qwe123").
ok
3> Base = {base, "dc=bluetail,dc=com"}.
{base,"dc=bluetail,dc=com"}
4> Scope = {scope, eldap:wholeSubtree()}.
{scope,wholeSubtree}
5> Filter = {filter, eldap:equalityMatch("sAMAccountName", "tobbe")}.
{filter,{equalityMatch,{'AttributeValueAssertion',"sAMAccountName","tobbe"}}}
6> Search = [Base, Scope, Filter].
[{base,"dc=bluetail,dc=com"},
 {scope,wholeSubtree},
 {filter,{equalityMatch,{'AttributeValueAssertion',"sAMAccountName","tobbe"}}}]
7> eldap:search(S, Search).
{ok,{eldap_search_result,[{eldap_entry,
                           "CN=Torbjorn Tornkvist,CN=Users,DC=bluetail,DC=com",
                           [{"memberOf",
                             ["CN=TestGroup2,CN=Users,DC=bluetail,DC=com",
                              "CN=TestGroup,CN=Users,DC=bluetail,DC=com",
                              "CN=Pre-Windows 2000 Compatible Access,CN=Builtin,DC=bluetail,DC=com",
                              "CN=Server Operators,CN=Builtin,DC=bluetail,DC=com"]},
                            {"cn",["Torbjorn Tornkvist"]},
                            {"company",["Alteon Web Systems"]},
                            {"mail",["tobbe@bluetail.com"]},
                            {"givenName",["Torbjorn"]},
                            {"instanceType",["4"]},
                            {"lastLogoff",["0"]},
                            {"lastLogon",["127119109376267104"]},
                            {"logonCount",[...]},
                            {"msNPAllowDialin"|...},
                            {...}|...]}],
                          [["ldap://bluetail.com/CN=Configuration,DC=bluetail,DC=com"]]}}
```

NORTEL NETWORKS

Slide 5

Some *eldap* notes...

- **Build a gen_server/supervisor *harness* around the eldap library when incorporating it into your system.**
- **By using the option: {ssl, true} you will use the ssl application to setup an SSL tunnel (LDAPS). (*Make sure to also set the port to 636)***
- ***The eldap/test directory contains test code, and examples on how to setup an OpenLDAP server.***
- ***Eldap has been tested with OpenLDAP, Iplanet and ActiveDirectory LDAP-servers.***

NORTEL NETWORKS

Slide 6

RADIUS

(Remote Authentication Dial-In User Service)

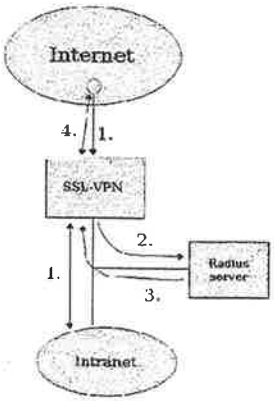
- A protocol to carry authentication, authorization, and configuration information between a Network Access Server, which desires to authenticate its links, and a shared Authentication server.
- Transactions client/server are authenticated through the use of a shared secret, which also is used to encrypt any user password sent over the network.
- Information is sent as Attribute-Length-Value 3-tuples, where new attributes (e.g vendor specific) easily can be added without disturbing existing implementations of the protocol.

NORTEL NETWORKS

Slide 7

A real example: the Nortel SSL-VPN

1. The user contact the Web-site and is presented with a login page.
2. A Radius Access-Request is sent from the SSL-VPN to the Radius server.
3. The Radius server returns an Access-Accept with authorization info.
4. The user accesses the Intranet via the SSL-VPN portal.

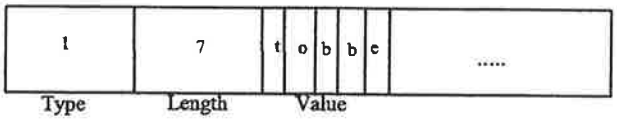


NORTEL NETWORKS

Slide 8

Attribute dictionaries (FreeRadius).

ATTRIBUTE	User-Name	1	string
ATTRIBUTE	User-Password	2	string encrypt=1
ATTRIBUTE	CHAP-Password	3	octets
ATTRIBUTE	NAS-IP-Address	4	ipaddr
ATTRIBUTE	NAS-Port	5	integer
ATTRIBUTE	Service-Type	6	integer
ATTRIBUTE	Framed-Protocol	7	integer
ATTRIBUTE	Framed-IP-Address	8	ipaddr



NORTEL NETWORKS

Slide 9

Vendor specific attribute dictionaries.

VENDOR Alteon 1872

ATTRIBUTE	Alteon-Service-Type	26	integer Alteon
ATTRIBUTE	Alteon-Xnet-Group	1	string Alteon
ATTRIBUTE	Alteon-ASA-Audit-Trail	2	string Alteon
ATTRIBUTE	Alteon-ASA-Audit-Source	3	string Alteon

VALUE	Alteon-Service-Type	Alteon-L4admin	250
VALUE	Alteon-Service-Type	Alteon-Slbadmin	251

NORTEL NETWORKS

Slide 10

Dictionaries and *eradius*

- **41 dictionaries taken from FreeRadius 0.9.1 are stored in eradius/priv/dictionaries/**
- **These dictionaries are parsed and transformed into the corresponding files containing Erlang records, as well as Erlang include files.**
- **Code that uses eradius can choose which dictionaries to load.**

NORTEL NETWORKS

Slide 11

Example: an Erlang program

①

```
go(IP, User, Passwd, Shared, NasIP) ->
  TraceFun = fun(_E,Str,Args) ->
                io:format(Str,Args),
                io:nl()
            end,
  E = #eradius{servers = [[IP, 1812, Shared]],
               user = User,
               passwd = Passwd,
               tracefun = TraceFun,
               nas_ip_address = NasIP},
  eradius:start(),
  eradius:load_tables(["dictionary",
                       "dictionary_alteon",
                       "dictionary_ascend"]),
  print_result(eradius:auth(E)).
```

②

```
print_result({accept, Attributes}) ->
  io:format("Got 'Accept' with attributes: ~p~n",[Attributes]),
  pa(Attributes);
print_result({reject, Attributes}) ->
  io:format("Got 'Reject' with attributes: ~p~n",[Attributes]),
  pa(Attributes);
print_result(Res) ->
  io:format("Got: ~p~n",[Res]).

pa([{K, V} | As]) ->
  case eradius_dict:lookup(K) of
    [A] ->
      io:format("    ~s = ~p~n",[A#attribute.name,
                                 to_list(V, A#attribute.type)]);
    _ ->
      io:format("  <not found in dictionary>: ~p~n",
    ......
```

NORTEL NETWORKS

Slide 12

Example: ...the output...

```
2> et:go({192,168,128,1}, "support", Passwd, Passwd, {192,168,128,32}).
sending RADIUS request for support to {{192,168,128,1},1812}
got RADIUS reply Accept for support with attributes: [{{529,194},
                                                        <<0,0,0,72>>},
                                                        {{1872,1},
                                                        <<115,116,97,102,102>>}]
Got 'Accept' with attributes: [{{529,194},<<0,0,0,72>>},
                               {{1872,1},<<115,116,97,102,102>>}]
   Ascend_Maximum_Time = 72
   Alteon_Xnet_Group = "staff"
true
```

① ②

NORTEL NETWORKS

Slide 13

Radius Accounting

- **Extends the use of Radius to cover delivery of accounting information.**
- **Client sends Accounting-Request containing attributes.**
- **Server replies with Accounting-Response.**

NORTEL NETWORKS

Slide 14

Types of Accounting-Requests.

- **Accounting On/Off.**
- **Start/Stop accounting info for a *user*.**
- ***Interim-Update* accounting info for a *user*.**

NORTEL NETWORKS

Slide 15

Example of use: the Nortel SSL-VPN

- **Sends info about how long time a user was logged on and what the termination cause was.**
- **Used for audit trail logging, i.e logging of operator issued CLI commands.**

NORTEL NETWORKS

Slide 16

Example: an Erlang program

```
-include("dictionary_alteon.hrl").   ②

acc() ->
    eradius:start(),
    eradius_acc:start(),
    eradius:load_tables(["dictionary",
                         "dictionary_alteon"]),
①   User = "tobbe",
    SessionId = 42,
    R = acc_start(User, SessionId),
    Login = R#rad_accreq.login_time,
    sleep(10),
②   VendAttrs = [{?Alteon, [{?Alteon_ASA_Audit_Trail,
                             "This is a test!"}]}],
    acc_update(User, SessionId, VendAttrs),
    sleep(10),
③   acc_stop(User, SessionId, Login,
             ?REASON_LOGOUT).
```

```
acc_start(User, SessId) ->
    Srvs = radacct_servers(),
    NasIP = nas_ip_address(),
    A = eradius_acc:new(),
    R = set_session_id(
            set_user(
             set_servers(
              set_nas_ip_address(
                    set_login_time(A),
                    NasIP),
              Srvs),
             User),
            SessId),
    eradius_acc:acc_start(R),
    R.
```

NORTEL NETWORKS

Slide 17

Example: the Radius accounting log

```
①   Mon Nov 10 14:14:47 2003
        Acct-Status-Type = Start
        Acct-Session-Id = "42"
        User-Name = "tobbe"
        NAS-IP-Address = 192.168.128.32
        Client-IP-Address = trana.bluetail.com
        Acct-Unique-Session-Id = "000b40c13fd3ef1a"
        Timestamp = 1068470087

    Mon Nov 10 14:14:57 2003
        Acct-Status-Type = Alive
        Acct-Session-Id = "42"
        User-Name = "tobbe"
        NAS-IP-Address = 192.168.128.32
②       Alteon-ASA-Audit-Trail = "This is a test!"
        Client-IP-Address = trana.bluetail.com
        Acct-Unique-Session-Id = "000b40c13fd3ef1a"
        Timestamp = 1068470097
```

```
    Mon Nov 10 14:15:07 2003
        Acct-Status-Type = Stop
        Acct-Session-Time = 20
        Acct-Session-Id = "42"
③       Acct-Terminate-Cause = User-Request
        User-Name = "tobbe"
        NAS-IP-Address = 192.168.128.32
        Client-IP-Address = trana.bluetail.com
        Acct-Unique-Session-Id = "000b40c13fd3ef1a"
        Timestamp = 1068470107
```

NORTEL NETWORKS

Slide 18

Available via the sourceforge jungerl cvs:

http://sourceforge.net/projects/jungerl/

Recommended References:

LDAP: RFC-2251, "*LDAP System Administration*" (O'Reilly), ,
Articles in Linux Journal July-Sep 2003,

Radius: RFC-2865,2866, "*Radius*" (O'Reilly),

NORTEL NETWORKS

Getting Erlang to talk to C and C++ from ei to UBF

Hal Snyder
Vail Systems, Inc.
hal@vailsys.com

Rick Pettit
Vail Systems, Inc.
rpettit@vailsys.com

what this talk is about

- current platform at Vail Systems
- the problem: C/C++ and Erlang
- first approach: ad hoc
- second approach: ei, C nodes
- third approach: UBF
- conclusions

1

Vail Systems

- computer telephony applications service provider
- voice over IP (SIP)
- custom IVR and VoiceXML
- 2 sites, 3 million calls / day
- OTP for distributed services: LCR, CDR, RM, stats

the problem: C/C++ and Erlang

- existing systems in separate camps: C/C++ and OTP
- connecting the two has been expensive in the past
- new apps: which way to go? unpleasant either/or
- one goal: C/C++ camp using mnesia with minimal Erlang

first approach: ad hoc

- description
- reasons
- results

ad hoc: description

- three tiers
  1. C++ app
  2. OTP request broker same host as C++ app
  3. remote OTP node
- ad hoc protocol - went through two major versions

```
request: <Id><sync|async|handoff> <M,F,A>
reply:   <Id><reply data>
```

- adds 1 thread for heartbeat
- C++, heavy use of STL, fancy OO wrappers (functors) for callbacks

ad hoc: reasons

- C++/STL/threads trusted
- TCP and ad hoc protocols trusted
- OTP ... risky

ad hoc: results

- initial use: least cost routing of outbound calls
- 5 months to write and test
- 1274 lines C++ source, 1521 line regression test
- 1 year in production
- 48 million requests
- 0 errors
- 0 downtime

second approach: ei, C nodes

- description
- reasons
- results

ei, C nodes: description

- two tiers
  1. C++ client calling Mod,Func,Arg
  2. remote OTP node
- OTP resources already in use, at known "intentional" DNS names
- software stack on C++ client
  1. C++ app
  2. C++ driver with app-specific objects
  3. our C driver, generic - heartbeat, reply matchup
  4. Ericsson's ei_rpc
- each application thread calls M,F,A, waits for response

ei, C nodes: reasons

- greater confidence in OTP
- incidental: shared lib for ad hoc method developed build problems
- didn't want to write another app-specific driver
- ei_rpc uses proven protocol - OTP transport
- Ericsson wrote a lot of code for us
  - OTP server code - erts
  - C client code - ei
- no change to existing OTP services

ei, C nodes: results (1 of 3)

- `ERL_TICK` => pthreads hell (again)
- working transactions after two days of coding
- memory management - terms, messages, queues
- still finding thread-related problems 4 weeks later
- message sequencing - feels like rewriting TCP every time we do one of these

mantra of concurrency: no shared data

ei, C nodes: results (2 of 3)

message sequencing with ei_rpc

server:

```
add1(Num) -> timer:sleep(2000), Num + 1.
```

client:

```
>./ei_rpc_cli
connected to node1@fafner.vail fd=3
0 + 1 = ? ERL_TIMEOUT
1 + 1 = ? ERL_TIMEOUT
2 + 1 = 1
3 + 1 = 2
4 + 1 = ? ERL_TIMEOUT
5 + 1 = 4
6 + 1 = 3
7 + 1 = 5
8 + 1 = ? ERL_TIMEOUT
```

ei, C nodes: results (3 of 3)

- message sequencing with ei_rpc
- OTP uses PIDs as transaction IDs
- C nodes don't have real PIDs
- work-around: in ei_connect.c, delete

  `self->num = fd;`

  from ei_reg_send() and ei_rpc_to()

ei, C nodes: results (concluded)

- initial use: C++ call control engine
  - OTP resource for call setup info
  - integration in progress
- all interface code
  - 5 weeks, so far, still debugging C
  - C shim 2698 lines, 1571 line test
  - C++ app interface 433 line, 77 lines test

third approach: UBF

- description
- reasons
- results

UBF: description

- two tiers
  1. C++ server doing telephony control
  2. OTP client coordinating work requests, resource management
- C++ server
  - lots of threads for telephony RTP streams etc.
  - one thread iterative TCP UBF server for work requests
- OTP client
  - web etc. distributed resource for command and status

UBF: reasons

- let us write apps in OTP instead of threaded C++
- better security than ei_rpc
- allow varied endpoints, e.g. Java client, C++ server
- self-documenting protocol, contract checker

UBF: results (1 of 3)

initial use: call bridging engine

- OTP gets work requests from outside, tracks resources
- C++ does outdial, patches calls together

UBF: results (2 of 3)

all work so far is on C++ server

- UBF(B) grammar - 1 hour
- Erlang simulator plugin - 3 hours
- flex grammar - 4 hours
- bison grammar - 8 hours
- hooks to threaded C++ app - weeks

UBF: results (3 of 3)

UBF experience so far

- want to separate contract checker and UBF server
- backward lists unfriendly to pipelines - [a b c] alternative?
- similarly, should semantic tag precede value it modifies?
- allow alternative radix on integers - 0x? 16#?

some payoff already

- UBF(B) grammar/contract is checkable/executable
- Erlang simulator also helps validate design

conclusions

- C++/OTP interface needed, must appeal to non-OTP-zealots
- ad hoc TCP protocol was very reliable, but great inertia to replicating the approach
- ei_rpc some implementation surprises, added to toolkit but not dernier cri
- UBF - very helpful in design and early implementation... to be continued ...

links

Joe Armstrong's UBF site:
http://www.sics.se/~joe/ubf/site/home.html

"Distribution by another means" thread on erlang-questions
http://www.erlang.org/ml-archive/erlang-questions/200006/msg00020.html

ERLANGS EXTERNAL FORMAT and distribution protocol
otp_src_R9C-0/erts/emulator/internal_doc/erl_ext_dist.txt

follow-up and code for this presentation will be at
http://www.drxyzzy.org/euc2003

Those who ignore Erlang are doomed to repeat it.

Erlang/QuickCheck

Thomas Arts, IT University
John Hughes, Chalmers University
Gothenburg

A little set theory…

- Recall that $X \cup Y = Y \cup X$?

A little set theory…

- Recall that $X \cup Y = Y \cup X$?
- Erlang has a sets library. Does this hold?

A little set theory…

- Recall that $X \cup Y = Y \cup X$?
- Erlang has a sets library. Does this hold?
- Property: $X \cup Y = Y \cup X$

A little set theory…

- Recall that $X \cup Y = Y \cup X$?
- Erlang has a sets library. Does this hold?
- Property: $\forall X. \forall Y. X \cup Y = Y \cup X$

A little set theory…

- Recall that $X \cup Y = Y \cup X$?
- Erlang has a sets library. Does this hold?
- Property: $\forall X{:}Set. \forall Y{:}Set. X \cup Y = Y \cup X$

A little set theory…

- Recall that $X \cup Y = Y \cup X$?
- Erlang has a sets library. Does this hold?
- Property: $\forall$ X:Set. $\forall$ Y:Set. $X \cup Y = Y \cup X$
- In Erlang/QuickCheck:

```
?FORALL(X,set(),
?FORALL(Y,set(),
sets:union(X,Y) == sets:union(Y,X)))
```

A little set theory…

- Recall that $X \cup Y = Y \cup X$?
- Erlang has a sets library. Does this hold?
- Property: $\forall$ X:Set. $\forall$ Y:Set. $X \cup Y = Y \cup X$
- In Erlang/QuickCheck:

```
prop_union_commutes() ->
   ?FORALL(X,set(),
   ?FORALL(Y,set(),
   sets:union(X,Y) == sets:union(Y,X))).
```

Verifying the property

```
12> qc:quickcheck(
        setsspec:prop_union_commutes()).
```

Verifying the property

```
12> qc:quickcheck(
        setsspec:prop_union_commutes()).
.........................................
....
Falsifiable, after 45 successful tests:
{'@',sets,from_list,[[-6,7,11,10,2]]}
{'@',sets,from_list,[[7,7,1,-4,11,-7]]}
ok
```

"function call"

These sets are a counterexample.

Fixing the Property

- Sets are not represented uniquely by the `sets` library
- `union` builds two different representations of the same set

```
equal(S1,S2) ->
  lists:sort(sets:to_list(S1)) ==
  lists:sort(sets:to_list(S2)).

prop_union_commutes() ->
  ?FORALL(X,set(),
  ?FORALL(Y,set(),
  equal(sets:union(X,Y),sets:union(Y,X)))).
```

Checking the fixed property

```
15> qc:quickcheck(
        setsspec:prop_union_commutes()).
.........................................
.........................................
...................
OK, passed 100 tests
ok
```

What is QuickCheck?

- A *language* for stating properties of programs (implemented as a library of functions and macros).
- A *tool* for testing properties in randomly generated cases.

Properties

- Boolean expressions + `?FORALL` + `?IMPLIES`.

```
prop_positive_squares() ->
  ?FORALL(X,int(),X*X>=0).

prop_larger_squares() ->
  ?FORALL(X,int(),
  ?IMPLIES(X>1,
  X*X>X)).
```

A *precondition*

What are int() and set()?

- Types?

What are int() and set()?

- Types? NO!!!
- Test data generators.
  - Define a *set* of values for test data…
  - …plus a *probability distribution* over that set.
- Test data generators are defined by the programmer.

Defining generators

- We often want to define one generator in terms of another, *e.g.* squares of ints.
- But we cannot do this by writing

```
N = int(), N*N
```

Returns a test data generator, not an integer.

Result should be a generator, not an integer.

Defining generators

- We often want to define one generator in terms of another, *e.g.* squares of ints.
- But we cannot do this by writing

```
N = int(), N*N
```

- We define a *generator language* to handle generators as an ADT.

```
?LET(N,int(),return(N*N))
```

Bind a name to the *value generated.*

Convert a value to a *constant* generator.

How can we generate sets?

- An ADT can only be generated using the ADT operations.
- Choose randomly between all ways of creating a set.

A generator for sets

```
set() -> frequency([
  {6,?LET(L,list(int()),
     return({'@',sets,from_list,[L]}))},
  {6,?LET(S,set(),?LET(E,int(),
     return({'@',sets,add_element,[E,S]})))},
  {1,?LET(P,function(bool()),?LET(S,set(),
     return({'@',sets,filter,[P,S]})))},
  ...]).
```

weights

?FORALL performs a call when it sees '@'

A problem with random generation

- How do we know we tested a reasonable range of cases, when we don't *see* them?

A problem with random generation

- How do we know we tested a reasonable range of cases, when we don't *see* them?

- **Simple approach**: collect statistics on test cases, so we see a *summary* of the test data.

- (A simple way to measure *test coverage*, which is a tangled topic in its own right).

An instrumented property

```
prop_union_commutes() ->
  ?FORALL(X,set(),
  ?FORALL(Y,set(),
  collect(sets:size(sets:union(X,Y)),
  equal(sets:union(X,Y),
        sets:union(Y,X))))).
```

Collect statistics on the *sizes* of the resulting sets.

Output: the distribution of set sizes

```
27> qc:quickcheck(
        setsspec:prop_union_commutes()).
.........................................
.........................................
...................
OK, passed 100 tests
 16% 3    7% 7    3% 16   2% 9    1% 21
 11% 4    6% 12   3% 14   2% 0    1% 18
 9% 2     5% 13   3% 11   1% 20   ok
 8% 6     4% 8    3% 5    1% 10
 8% 1     3% 17   2% 24   1% 22
```

Testing concurrent programs

A simple *resource allocator*:

- `start()` – starts the server
- `claim()` – claims the resource
- `free()` – releases the resource

in the client

These functions are called for their *effect,* not their result. How can we write QuickCheck properties for them?

Traces

- Concurrent programs generate traces of events.
- We can write properties of traces – they are lists!

Testing the resource allocator

```
client() -> claim(), free(), client().
```

`clients(N)` – spawns `N` clients.

```
system(N) -> start(), clients(N).
```

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
… property of T …))
```

The trace recorder

- What should the recorded events be?
- How should we capture them?

Random traces: a problem

- What does this print?

```
test_spawn() ->
      spawn(io,format,["a"]),
      spawn(io,format,["b"]).
```

Random traces: a problem

- What does this print?

```
test_spawn() ->
      spawn(io,format,["a"]),
      spawn(io,format,["b"]).
```

- `ab` – every time!

Random traces: a problem

- What does this print?

```
test_spawn() ->
      spawn(io,format,["a"]),
      spawn(io,format,["b"]).
```

- `ab` – every time!
- But `ba` should also be a possible trace – the Erlang scheduler is too predictable!

Solution: simulate a random scheduler

- Insert calls of `event(Event)` in code under test.
  - Sends `Event` to trace recorder
  - Waits for a reply, sent in random order
- Allows the trace recorder to simulate a random scheduler.
- Answers question: which events should be recorded?

Simple example revisited

```
do(E) -> event(spawned), event(E).

?FORALL(T,
  ?TRACE(3,begin spawn(?MODULE,do,[a]),
                 spawn(?MODULE,do,[b])
           end),
  collect(rename_pids(nowaits(T)),true)))
```

Simple example revisited

```
OK, passed 100 tests
18% [{exit,{pid,1},normal},
  {event,{pid,2},spawned},
  {event,{pid,3},spawned},
  {event,{pid,2},a},
  {exit,{pid,2},normal},
  {event,{pid,3},b},
  {exit,{pid,3},normal},
  timeout]

18% [{exit,{pid,1},normal},
  {event,{pid,2},spawned},
  {event,{pid,3},spawned},
  {event,{pid,3},b},
  {exit,{pid,3},normal},
  {event,{pid,2},a},
  {exit,{pid,2},normal},
  timeout]
```

Simple example revisited

```
OK, passed 100 tests
```

```
18% [{exit,{pid,1},normal},
  {event,{pid,2},spawned},
  {event,{pid,3},spawned},
  {event,{pid,2},a},
  {exit,{pid,2},normal},
  {event,{pid,3},b},
  {exit,{pid,3},normal},
  timeout]
```

```
18% [{exit,{pid,1},normal},
  {event,{pid,2},spawned},
  {event,{pid,3},spawned},
  {event,{pid,3},b},
  {exit,{pid,3},normal},
  {event,{pid,2},a},
  {exit,{pid,2},normal},
  timeout]
```

Pids are renamed for collecting statistics

Trace recorder times out if no events happen for a while

A surprise!

```
Pid=spawn(fun()->
   event(spawned),
   event(ok) end),
event(spawn),
exit(Pid,kill),
event(kill)
```

```
1% [{event,{pid,1},spawn},
  {event,{pid,2},spawned},
  {event,{pid,2},ok},
  {event,{pid,1},kill},
  {exit,{pid,2},killed},
  {exit,{pid,2},noproc},
  {exit,{pid,1},normal},
  timeout]
```

No doubt there is a good reason…

Trace properties

- The resource allocator guarantees exclusion
- Instrumented code:

```
client() ->
  event(request),
  claim(),
  event(claimed),
  event(freeing),
  free(),
  client().
```

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    tnot(?MATCHES({event,_,claimed}))))))))
```

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
      not(?MATCHES({event,_,claimed}))))))))
```

The trace T satisfies…

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
      not(?MATCHES({event,_,claimed}))))))))
```

…it's always true that…

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    tnot(?MATCHES({event,_,claimed}))))))))
```

...if the current event is claimed...

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    tnot(?MATCHES({event,_,claimed}))))))))
```

...then after this event...

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    tnot(?MATCHES({event   claimed}))))))))
```

...until a freeing event happens...

Trace properties

- The resource allocator guarantees exclusion

```
?FORALL(N,nat(),
?FORALL(T,?TRACE(3,system(N)),
satisfies(T,
always(timplies(?MATCHES({event,_,claimed}),
  next(until(?MATCHES({event,_,freeing}),
    tnot(?MATCHES({event,_,claimed}))))))))
```

...there will be no further claimed event.

Trace property language

- Based on *linear temporal logic*
  - Logical operations:
    `tand, tor, tnot, ?TIMPLIES.`
  - Temporal operations:
    `always, eventually, next, until.`
  - Event matching operations:
    `?MATCHES, ?AFTER, ?NOW.`

A failing property

- The resource is always eventually granted.

```
prop_eventually_granted(N) ->
  ?FORALL(T,?TRACE(3,system(2)),
  satisfies(T,
  always(?AFTER({event,Pid,request},
      eventually(N,
          tor(?NOW({event,Pid2,claimed},
                        Pid==Pid2),
              ?MATCHES(more))))))).
```

A failing property

- The resource is always eventually granted.

Failing trace of 23 steps found after 80 successful tests.

```
prop_eventually_granted(N) ->
  ?FORALL(T,?TRACE(3,system(2)),
  satisfies(T,
  always(?AFTER({event,P
      eventually(N,
          tor(?NOW({event,Pid2,claimed},
                        Pid==Pid2),
              ?MATCHES(more)))))).
```

After at most N steps

End of the recorded trace

In progress

- Testing generic leader election behaviour
- Properties
  - Eventually a leader is elected, even in the presence of failures
  - There is always at most one elected leader

Experience

- There are as many bugs in properties as in programs!
  - QuickCheck checks for *consistency* between the two, helps improve understanding
- Random testing is effective at finding errors.
- Changes our perspective on testing
  - Not "what cases should I test?"
  - But "what properties ought to hold?"

QuickCheck is Fun!

Try it out!

www.cs.chalmers.se/~rjmh/ErlangQC

References

- Erlang/QuickCheck is based on a Haskell original by Claessen and Hughes.
  - *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*, ICFP 2000.
  - *Testing Monadic Code with QuickCheck*, Haskell Workshop 2002.
  - *Specification Based Testing with QuickCheck*, in Fun of Programming, Palgrave, 2003.
  - *Testing and Tracing Functional Programs*, in Advanced Functional Programming Summer School, Springer-Verlag LNCS, 2002.

Questions?

Answers

(The remaining slides may be used to answer specific questions).

Random functions *are* pure functions!

```
1> F = qc:gen(qc:function(qc:nat()),10).
#Fun<qc.46.146913[illegible]>
2> F(1).
8
3> F(2).
9
4> F(3).
3
5> F(1).
8
```

Invokes a generator

Random results

But consistent ones

Controlling sizes

- Test cases are regenerated w.r.t. a *size parameter*, which increases during testing.

```
prop_union_commutes() ->
    ?SIZED(N,resize(5*N,…))
```

Bind N to the size parameter

Reset the size parameter

- Set sizes now range up to 135 elements.

Performance Analysis with Model Checking*

Extracting performance information from Erlang source code

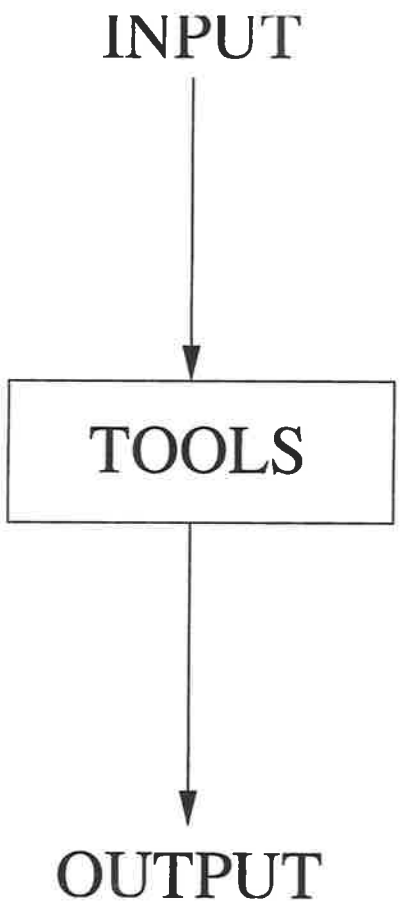
Juan José Sánchez Penas
LFCIA, University of Corunha, Spain
juanjo@lfcia.org

Thomas Arts
IT-university, Göteborg, Sweden
thomas.arts@ituniv.se

EUC 2003. Stockholm, 18th November

*Work partially supported by MCyT, Spain, Project TIC 2002-02859

General Approach Overview



- Given a real **distributed system**:

  Functional requirements
  Design and implementation of the system
  Performance requirements

- Using techniques from **formal methods**:

  Erlang to process algebra compiler
  Process algebra tools
  Model checking and graph analysis algorithms

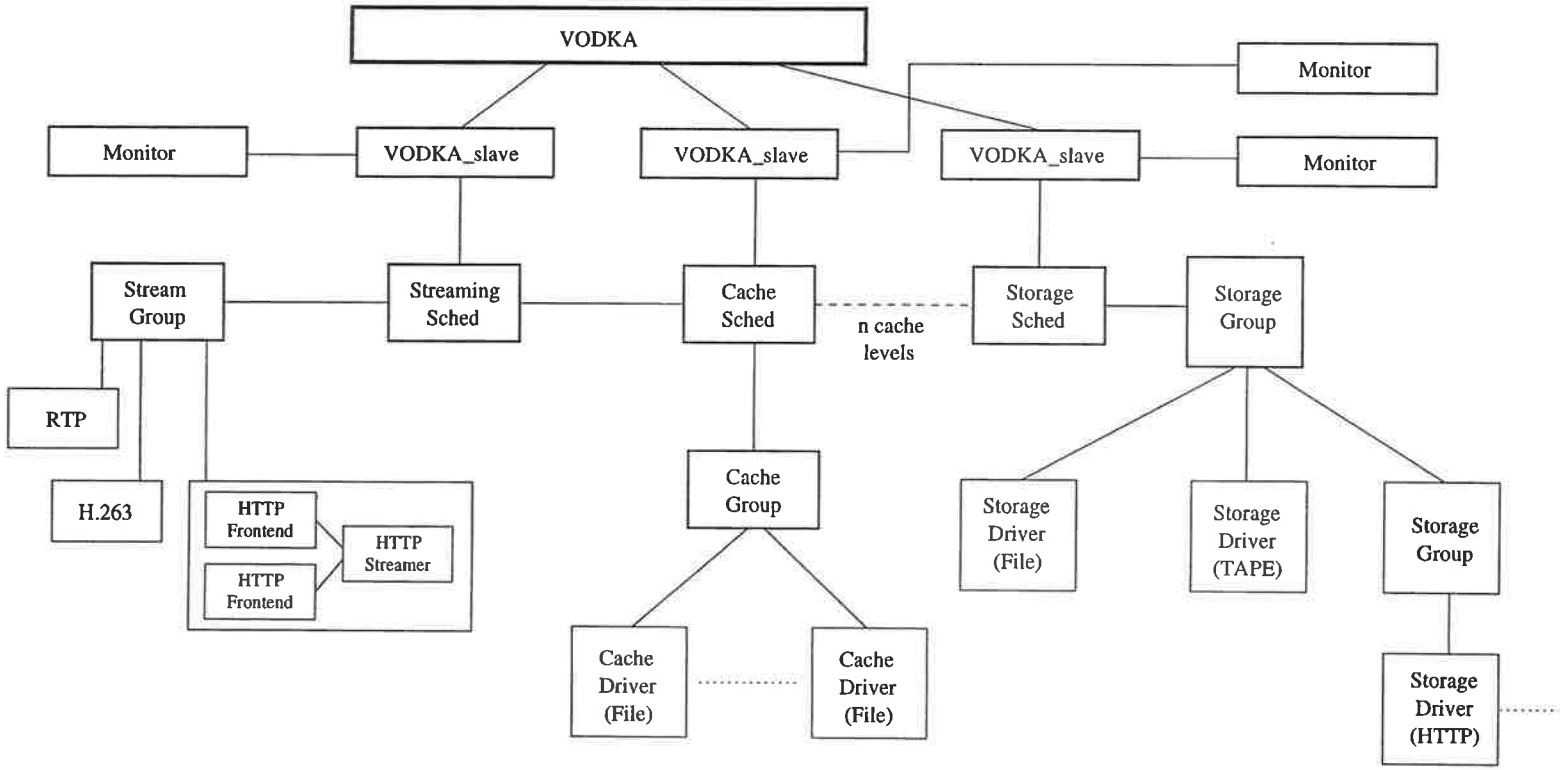
- We want to find and fix:

  Functional problems
  Performance problems
  Design problems (maintainability, flexibility)

Case Study: VoDKA Server - The project

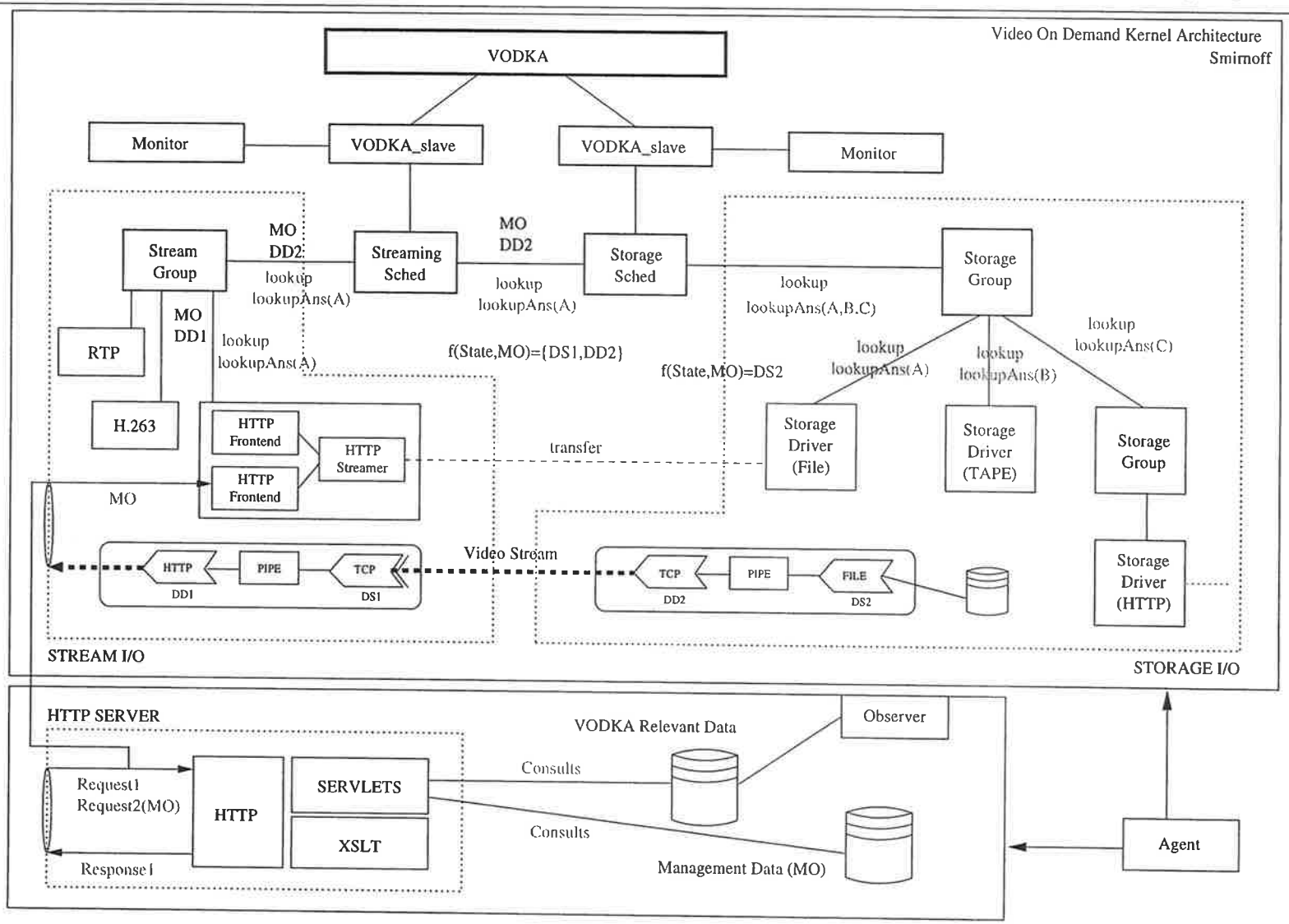
- Hierarchical distributed multimedia server (LFCIA, last 3 years)
- Funded by an European Project and *R*, a Cable Telecommunications Company
- Classical VoD server requirements: Huge storage capacity, high bandwidth, predictable (low) response time, support for a great amount of concurrent users and fault tolerance
- Special requirements for the VoDKA project: Scalability (upwards and downwards), adaptability and low cost
- Hardware: Adaptation of Beowulf Cluster architecture
- Software: Flexible distributed architecture based in Erlang/OTP platform

Case Study: Hierarchical Flexible Architecture



- Flexible architecture based on a hierarchy of specialized **levels**
- Each level is composed by distributed Erlang processes
- Extensive use of *generic server* and *supervision tree* Erlang behaviours

Case Study: Hierarchical Flexible Architecture (II)



Case Study: Distributed Scheduling

- Completely distributed scheduling subsystem:

  no global state, no global decision

- Each process in the scheduling subsystem can implement:

  Restrictions (number of connections, maximum bandwidth)
  Scheduling function (filtering, cache algorithms, admission policy)
  Cost (state of the component and resources still available)

- We want to analyse the system:

  Information for the 'user' of the system (R) - capacity of the system
  Information for the designer of the system - how to improve it (bottlenecks)

The Goal: What do we mean by performance analysis?

User point of view

- **Black-box evaluation** (requirements oriented)
  - System capacity
  - Component capacity
  - Scenario checking

Developer point of view

- **Architecture and protocol analysis** (internal design oriented)
  - Finding/checking bottlenecks
  - Bottleneck summary from the program graph
  - Extracting/checking message protocol and architecture
  - Finding required capacity for a new component

The Goal: Problem Explanation by Example

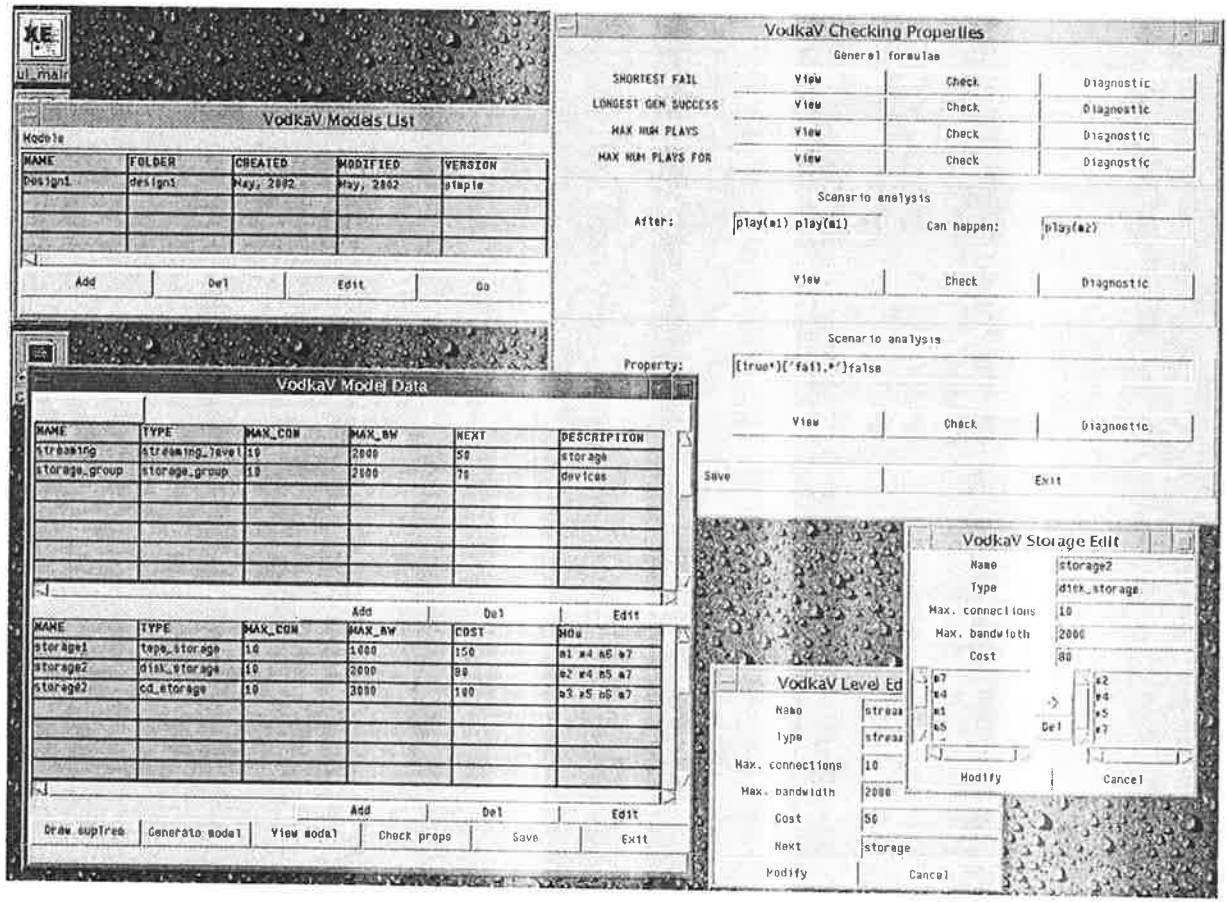
- Goal: Given a *configuration* for the server (the processes, the storage devices, all the restrictions, scheduling functions, and costs):

How can we extract performance information from the source code of the system?

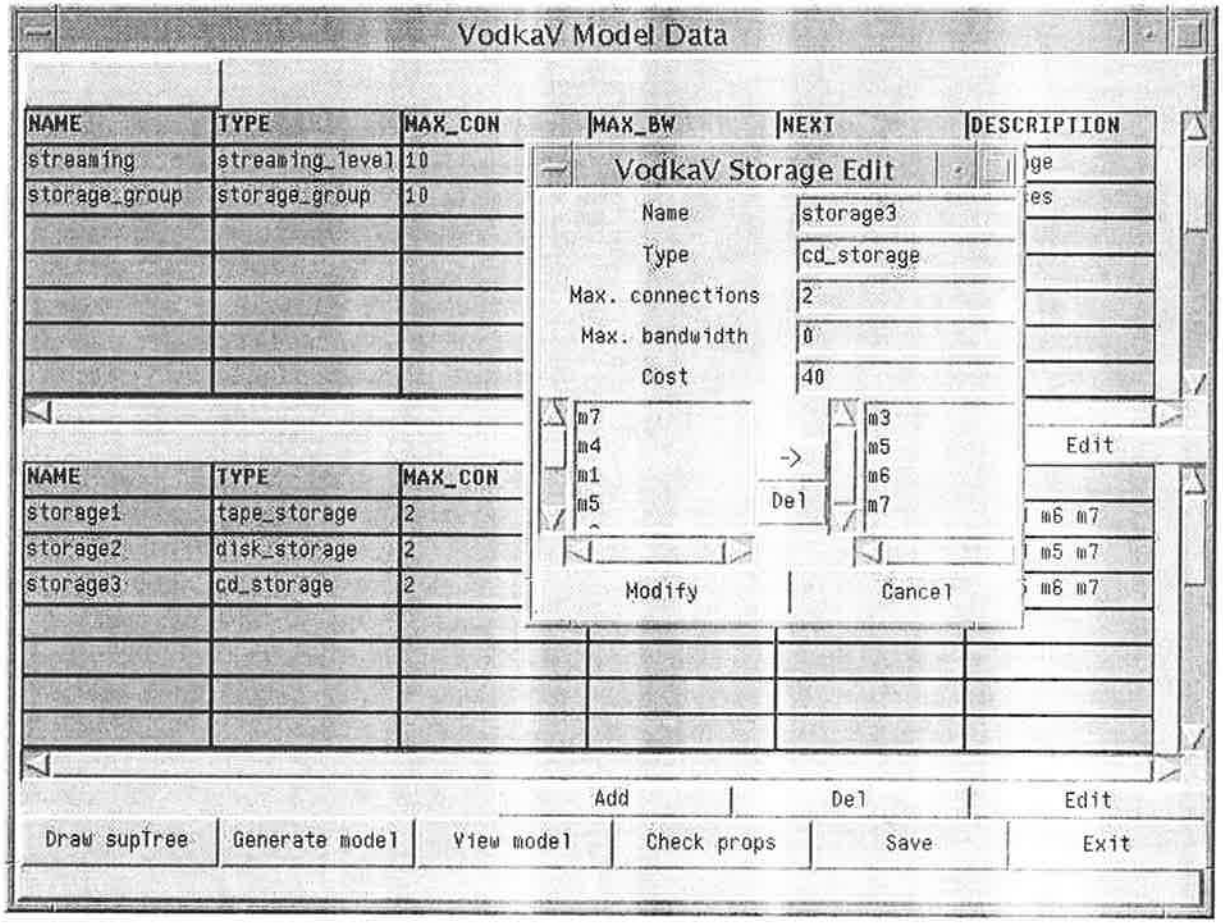
- We want to be able to answer questions like:

  What is the maximum number of users in the system?
  What is the minimum number of users such that serving any MO is not possible?
  What is the minimum number of users such that serving MO1 is not possible?
  How many people can watch MO at the same time? (best case)
  How many people can watch MO1 such that the system can still serve MO2?
  How many people can watch MO1 such that serving MO2 is guaranteed?
  Would it be better to move MO from storage1 to storage2?
  Where should we move MO1 for being able to serve it to N users?
  Why (bottleneck) MO cannot be served to N users at the same time
  What are the minimum requirements for a new component

The Proposed Tool: A Prototype (I)

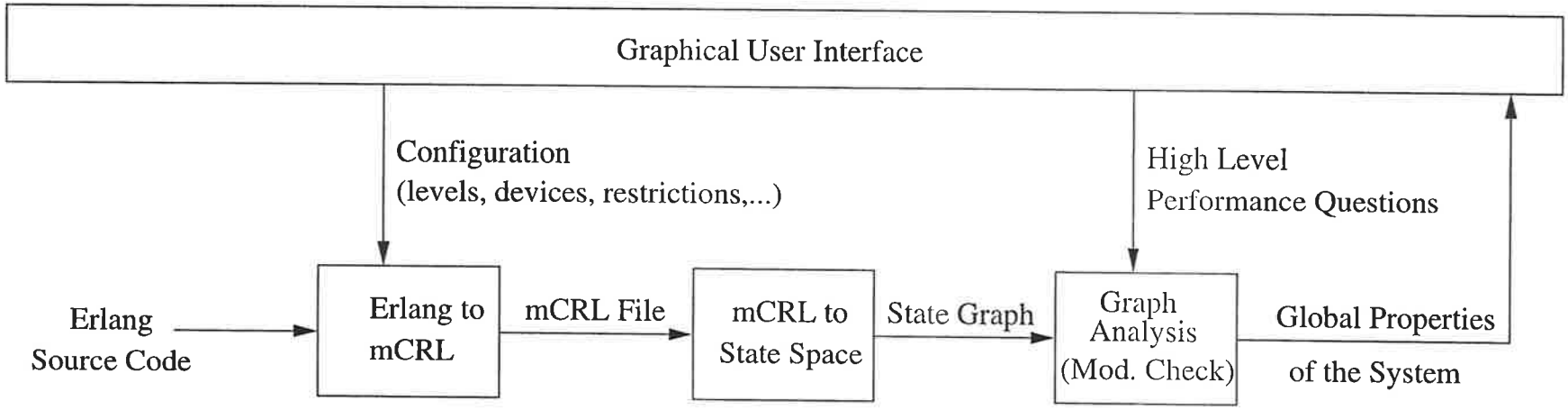


The Proposed Tool: A Prototype (II)



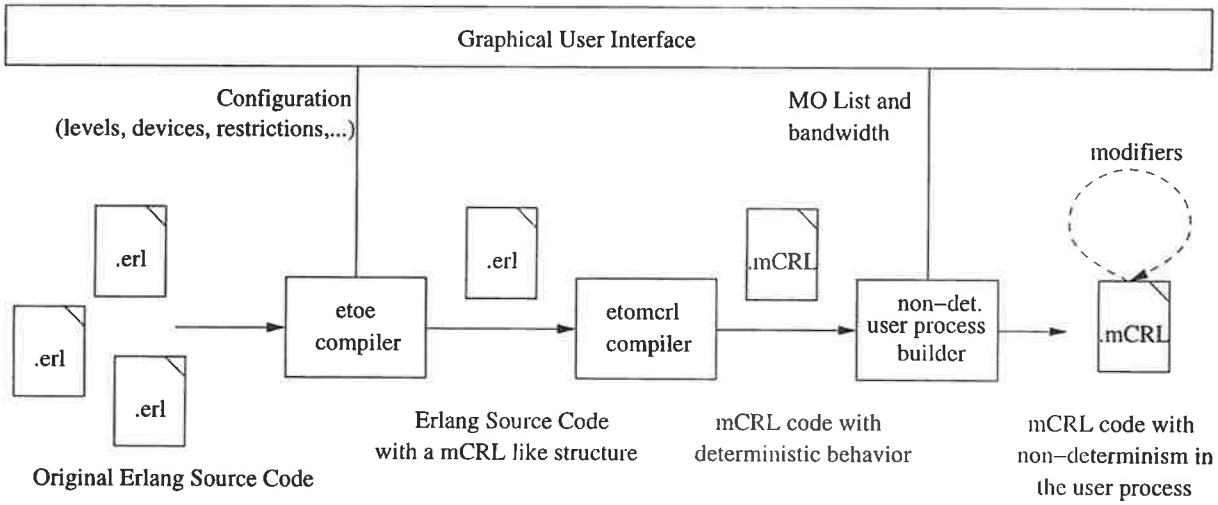
10

The Proposed Methodology



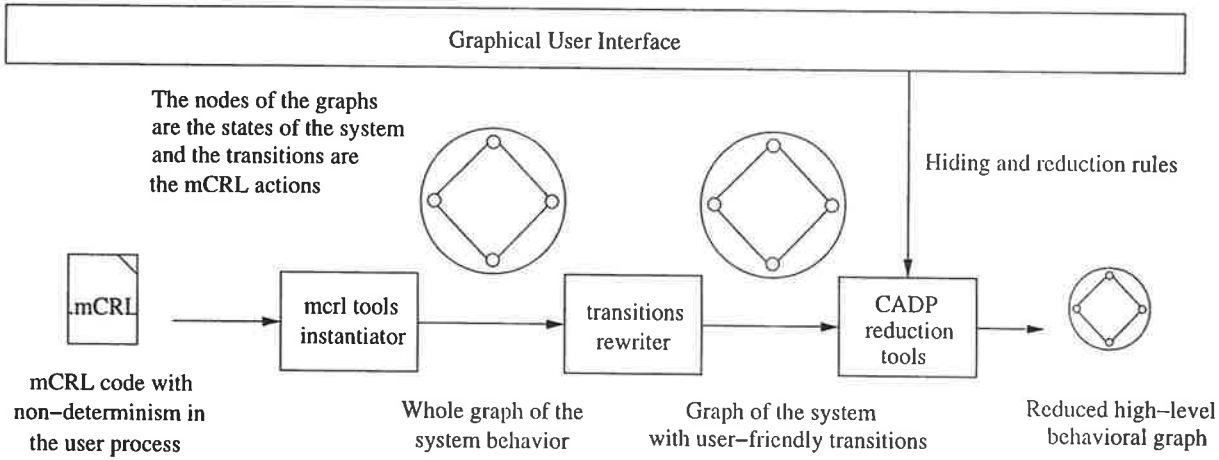
- Generating the full state space of the system **from its configuration**
- Starting directly **from the Erlang source code** of the system (easier with design patterns)
- The source code is already an abstraction of the real one (only the scheduling subsystem and no resources are released)
- μCRL as intermediate step (efficient tools for generating state space). Semantics similar to Erlang
- A high level GUI separates the theoretical details from the users of the methodology

Step One: Erlang to μCRL



- Compiler developed before (Thomas Arts and Clara Benac, STTT2003)
- Symbolically evaluates the supervision tree and generic servers for a given set of arguments
- Differences between Erlang and μCRL:
  - μCRL has no asynchronous communication mechanism: buffers
  - μCRL has strict separation of communication and computation (without any side effect): call_stack
  - μCRL has no higher-order functions, records, list-comprehension: translate to simpler code
- Clients are modeled as non-deterministic *users process* for avoiding state explosion
- μCRL has tools like a confluence analyzer that can be used to reduce the final state space (reduction of 10-20%)

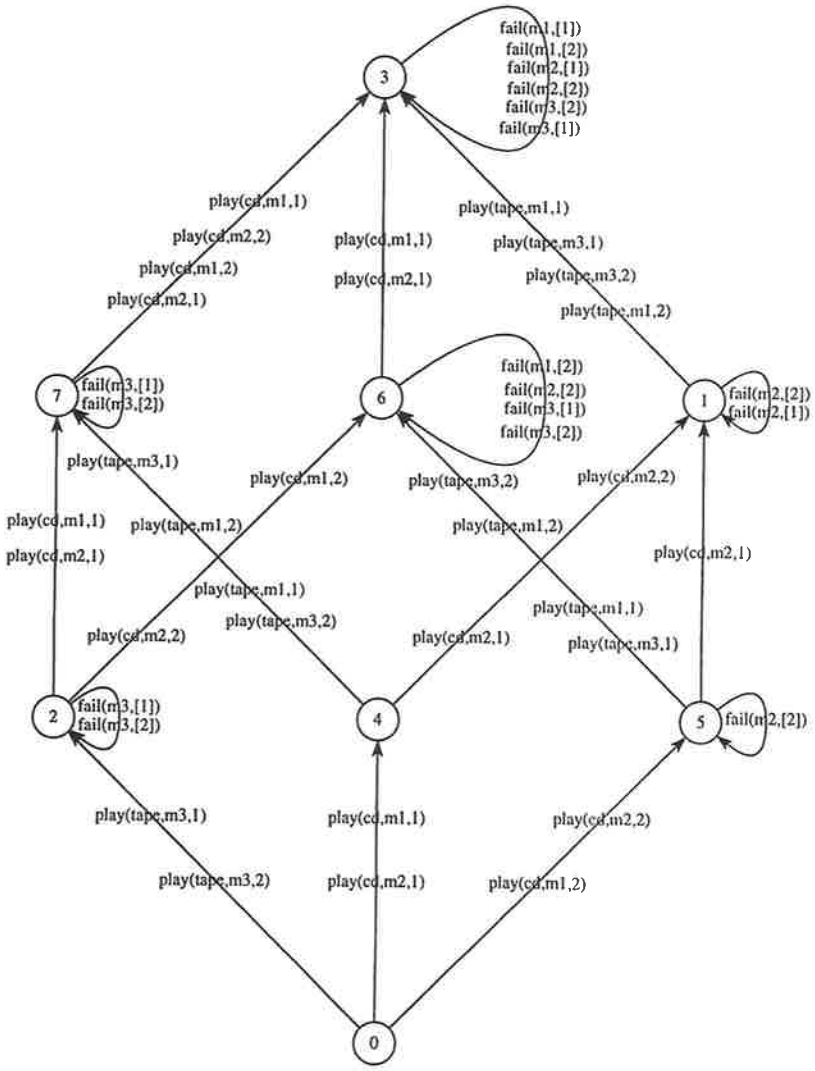
Step Two: Generating State Space from μCRL



13

- Standard tools for μCRL (*mcrl tools* developed at CWI) are used to generate the state space
- Cæsar/Aldébaran tool set is used for hiding and renaming labels, and for reducing the graph
- (For some properties) we hide the internal details of the system, because they are not going to be used in the next step
- Example: the state space of a two level configuration, without cache, with four devices in the storage level and all the possible combination of MOs distributed over the devices in two different qualities, contains up to a few million states. Its generation takes some hours and it is reduced to about one thousand states

Step Two: Example of a Simple Reduced Graph



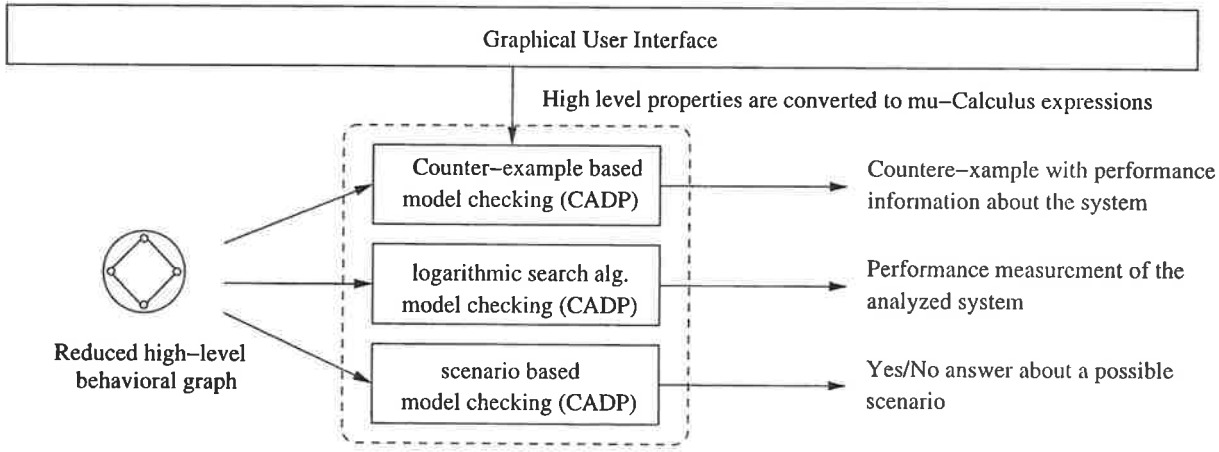
- Two linear levels: streaming level and massive storage level
- Two storage devices:
  - Tape with 20MBit/s, no simultaneous access
  - CD with 30MBit/s, 2 simultaneous access
- No extra restrictions than the trivial cost functions
- Abstract approach for the MOs (m1 in both, m2 and m3 in one of them)
- Two possible qualities: 10/20 MBits/s
- Original state space of 2547 states and 2747 transitions, the reduction results in the 8 states and 48 transitions

14

Step Three: Extracting Performance Information

- Verifying global properties with a black-box approach
- Extracting the architecture from the messages
- Extracting bottleneck information
- Calculating resources for a new component

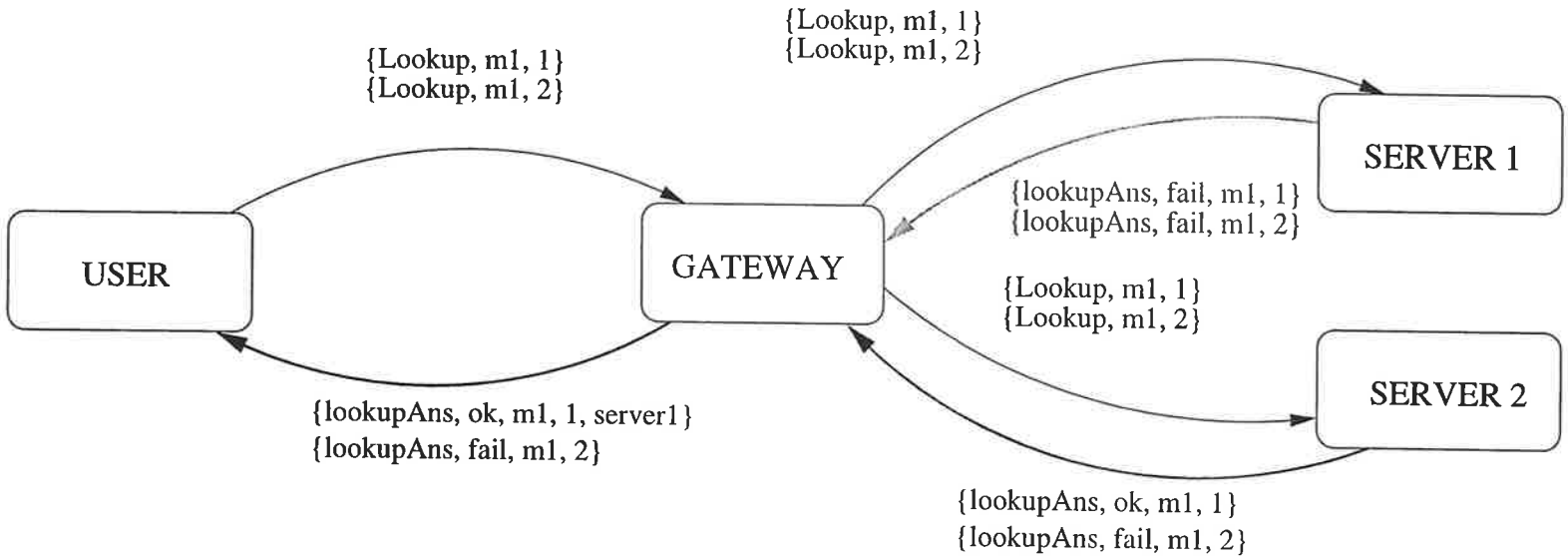
Step Three: Verifying Global Properties (black-box)



- Counter-example based:
  - 'the worst case scenario in which the system reaches its maximum load': `[true*]<not 'fail.*'>true`
  - 'the maximum number of simultaneous users after which a next user always can be served': `[true*](<'fail.*'>true\/<true>['fail.*']false)`
- Existential and eventually existential
  `<'play(.*,m1,.*)'.'play(.*, m1,.*)','play(.*,m3,.*)'>true`
- Feedback information
- Importance of the user interface

Step Three: Architecture from the Messages

- Can we extract the process/component architecture and the protocol of the messages they exchange, from the analysis of the source code?
- In the generic servers: source process, destination process, and message are easy to extract from the analysis of the code
- We can build this kind of graphs:



Step Three: Bottleneck information

- **Internal independent bottleneck**: first place were we can see a fail in the system, in any of the possible execution paths.

  Stopping the graph generation when a fail occurs

- **External/user independent bottleneck**: the point in the architecture that makes the first fails to be answered to a user request.

  Graph analysis of the fails in the top level

- **Internal relative bottleneck**: the part of the system where a fail in a component is too far away from a fail in a different component.

  Model checking with formulae talking about the distance between fails

For all of them, using graph analysis tools, we can extract the table summary with statistic information about the bottlenecks in the system.

Step Three: Calculating resources for a new component

- We want to know the resources needed in order to avoid a new component added to a system to be the bottleneck point in the architecture
- We can use the same methodology, adding to the system architecture the new component without any resource restriction
- We compute the system capacity in the execution graph, and we extract by graph analysis the information about the maximum number of times that the new component is asked
- The new component can be designed in order to be able to serve all the possible requests that is going to receive, thus avoiding it to be the bottleneck

Conclusions

- **We can get performance information from the source code**

  Case study: VoDKA, a distributed functional VoD server
  We use formal methods techniques for extracting information

- **We use the fact that the systems are built on top of OTP** modules and design principles in order to be able to handle complex systems with model checking

- **The methodology can be used in other distributed systems**

- **Some advantages** against testing, tracing and simulation

All you wanted to know about the HiPE compiler (but might have been afraid to ask)

K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson, T. Lindahl
Information Technology Department, Uppsala University, Sweden
hipe@csd.uu.se

ABSTRACT

We present a user-oriented description of features and characteristics of the High Performance ERLANG (HiPE) native code compiler, which nowadays is part of Erlang/OTP. In particular, we describe components and recent additions to the compiler that improve its performance and extend its functionality. In addition, we attempt to give some recommendations on how users can get the best out of HiPE's performance.

1. INTRODUCTION

During the last few years, we have been developing HiPE, a high-performance native code compiler for ERLANG. HiPE offers flexible, fine-grained integration between interpreted and native code, and efficiently supports features crucial for ERLANG's application domain such as light-weight concurrency. HiPE exists as a new component (currently about 80,000 lines of ERLANG code and 15,000 lines of C and assembly code) which nowadays is fully integrated with Ericsson's Erlang/OTP implementation; in fact, HiPE is available by default in the open-source version of R9. The HiPE compiler currently has back-ends for UltraSPARC machines running Solaris and Intel x86 machines running Linux or Solaris.

The architecture and design decisions of HiPE's SPARC and x86 back-ends have been previously described in [5] and [11] respectively. A brief history of HiPE's development appears in [6]. As performance evaluations in these reports show, HiPE considerably improves the performance characteristics of ERLANG programs, and on small sequential programs makes Erlang/OTP competitive in speed to implementations of other 'similar' functional languages such as Bigloo Scheme [13] or CML (Concurrent SML/NJ [12]).

Performance evaluation aside, all the above mentioned reports address quite technical compiler and runtime system implementation issues which most probably are not so informative for ERLANG programmers who are simply interested in using HiPE for their everyday application development.

Erlang Workshop '03 29/08/2003, Uppsala, Sweden

To ameliorate this situation, the current paper is targeted towards HiPE users. Its aims are to:

1. describe features – and sometimes secrets – of the HiPE compiler that are of interest to its users;
2. introduce recent and planned additions to the HiPE compiler in a way that focuses on how these new features affect users (i.e., without obfuscating their presentation by getting deep into technical details); and
3. give recommendations on how users can get the best out of HiPE's performance.

To make the paper relatively self-contained and provide sufficient context for the rest of its contents, Section 2 begins by overviewing HiPE's current architecture, then describes basic usage, compiler options and recent improvements, and finally presents some extensions to HiPE's functionality which are currently underway and will most probably be included in release R9C. Section 3 offers advise on HiPE's use, followed by Section 4 which reveals and documents limitations and the few incompatibilities that currently exist between the BEAM and the HiPE compiler. Finally, Section 5 briefly wraps up.

We warn the reader that the nature of certain items described in this paper is volatile. Some of them are destined to change; hopefully for the better. HiPE's homepage[1] might contain a more up-to-date version of this document.

2. HIPE COMPILER: A USER-ORIENTED OVERVIEW

2.1 HiPE's architecture

The overall structure of the HiPE system is shown in Fig. 1. The Erlang/OTP compiler first performs macro preprocessing, parsing, and some de-sugaring (e.g., expanding uses of the record syntax) of the ERLANG source code. After that, the code is rewritten into Core Erlang [2, 1]. Various optimizations such as constant folding, and (optional) function inlining, are performed on the Core Erlang level. After this, the code is again rewritten into BEAM virtual machine code, and some further optimizations are done. (The BEAM is the de facto standard virtual machine for ERLANG, developed by Erlang/OTP. It is a very efficiently implemented register machine, vaguely reminiscent of the WAM [14].)

The HiPE compiler has traditionally started from the BEAM virtual machine code generated by the Erlang/OTP

[1] http://www.csd.uu.se/projects/hipe/

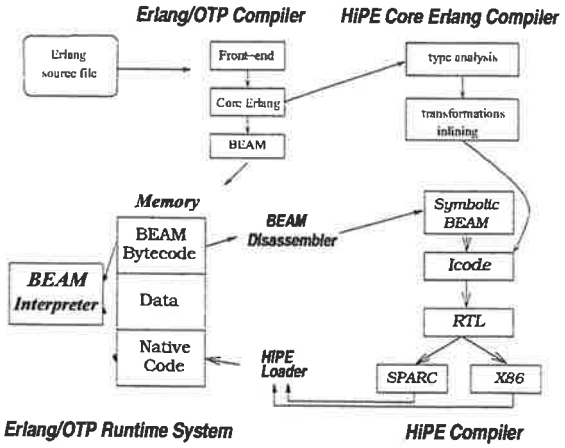


Figure 1: Structure of a HiPE-enabled Erlang/OTP system.

compiler. The BEAM code for a single function is first translated to ICode, an assembly-like language with a high-level functional semantics. After optimizations, the ICode is translated to RTL ("register-transfer language"), a low-level RISC-like assembly language. It is during this translation that most Erlang operations are translated to machine-level operations. After optimizations, the RTL code is translated by the backend to actual machine code. It is during this translation that the many temporary variables used in the RTL code are mapped to the hardware registers and the runtime stack. Finally, the code is loaded into the runtime system.

The Erlang/OTP runtime system has been extended to associate native code with functions and closures. At any given point, a process is executing either in BEAM code or in native code: we call this the *mode* of the process. A *mode switch* occurs whenever control transfers from code in one mode to code in the other mode, for instance when a BEAM-code function calls a native-code function, or when the native-code function returns to its BEAM-code caller. The runtime system handles this transparently, so it is not visible to users, except that the native code generally executes faster.

A new feature, described further below, is that the HiPE compiler can compile directly from Core Erlang. When used in this way, the compiler compiles a whole module at a time, and performs global analyses and optimizations which are significantly more difficult to perform (and thus not available) in the traditional mode.

2.2 Basic usage

The normal way of using the HiPE native code compiler is via the ordinary ERLANG compiler interface, by adding the single compilation option `native`. From the ERLANG shell, using the `c` shell function, this looks as follows:

```
1> c(my_module, [native]).
```

This will compile the file `my_module.erl` to native code and load the code into memory, following the normal module versioning semantics of ERLANG.

Calling the standard compiler function `compile:file/2` (which by default does not load the resulting code) will produce a `.beam` file that contains both the native code *and* the normal BEAM code for the compiled module; e.g.:

```
compile:file(my_module, [native])
```

produces a file `my_module.beam` which can be loaded later. When a `.beam` file is loaded, the loader will first attempt to load native code, if the file contains native code that is suitable for the local system, and only if this fails is the BEAM code loaded. In other words, the `.beam` files may be "fat", containing code for any number of different target machines.

The compiler can also be called from the external program `erlc` (which indirectly calls the `compile:file/2` function). E.g., from a UNIX command line shell or make-file:

```
erlc +native my_module.erl
```

producing a file `my_module.beam`.

Additional compiler options may be given between the `erlc` command and the file name by prefixing them with `+`. Quoting may be necessary to avoid expansion by the shell, as for example in:

```
erlc +native +'{hipe,[verbose]}' my_module.erl
```

Generating native code and loading it on-the-fly into the system is possible even in cases when the ERLANG source code is not available but the `.beam` file (containing BEAM bytecode) exists. This can be done for whole modules using:

```
hipe:c(my_module)
```

or even for individual functions `{M,F,A}` using:

```
hipe:c({M,F,A}).
```

The function `hipe:c/2` can also be used, which takes the list of the HiPE compiler options as its second argument.

Finally, should you forget everything else, you can always type the following from the ERLANG shell:

```
2> hipe:help().
```

which will display a short user's guide to the HiPE compiler.

2.3 HiPE compiler options

For the average user, it should not be necessary to give any extra information to the compiler than described in the previous section. However, in some cases it may be useful or even necessary to control the behavior of the native code compilation. To pass options to the HiPE compiler via the normal ERLANG compiler interface, these must be wrapped in a term `{hipe, ...}`. For example:

```
3> c(my_module, [native, {hipe, [verbose, o3]}]).
```

will pass the flags `verbose` and `o3` to the HiPE compiler. Note that if only a single option is given, it does not have to be wrapped in a list, as in e.g.:

```
c(my_module, [native, {hipe, verbose}]).
```

The main useful options are the following:

`o0, o1, o2, o3` Selects the optimization level, `o0` being the lowest. The default is `o2`. Upper case versions of these options also exist, i.e., `O2` is an alias for `o2`, etc.

verbose Enables HiPE compiler verbosity. Useful if you want to see what is going on, identify functions whose native code compilation is possibly a bottleneck, or just check that the native code compiler is running.

If a module takes too long time to compile, try using a lower optimization level such as `o1`. You can also try keeping the current optimization level, but specifically select the faster but less precise *linear scan* algorithm for register allocation [7]. (Register allocation is one of the major bottlenecks in the optimizing native code compilers.) This is done by adding the option `{regalloc,linear_scan}`, as in:

```
c(my_module, [{hipe, [{regalloc,linear_scan}]}]).
```

If you wish to always use certain HiPE compiler options for some particular module, you can place them in a `compile` directive in the source file, as in the following line:

```
-compile({hipe, [o1]}).
```

Note: options early in the list (i.e., further to the left) take precedence over later options. Thus, if you specify e.g.

```
{hipe, [o3, {regalloc,linear_scan}]}
```

the `o3` option will override the `regalloc` option with the more advanced (and more demanding compilation-time wise) `o3`-level iterated coalescing register allocator. The correct way would be:

```
{hipe, [{regalloc,linear_scan}, o3]}
```

which specifies `o3`-level optimizations but with fast register allocator.

More information on the options that the HiPE compiler accepts can be obtained by:

```
hipe:help_options().
```

2.4 Recent improvements

2.4.1 *Local type propagator*

ERLANG, being a dynamically typed language, often provides the developer with freedom to experiment with data structures whose handling is possibly still incomplete, and rapidly prototype applications. However, this also means that a lot of run time is spent in performing type tests (that usually succeed) to ensure that the operations performed are meaningful, e.g., that a program does not accidentally succeed in dividing a float by a list or taking the fifth element of a process identifier.

One of the recent additions to the HiPE compiler is a *local type propagator* which tries to discover as much of the available (per-function) type information as possible at compile time. This information is then propagated throughout the code of the function to eliminate redundant type tests and to transform polymorphic primitive operations that operate on general types into faster operations that are specialized to the type of operands actually being used.

Since the type propagator is a recent addition that is still under development and further extensions of its functionality are underway, we have not yet conducted a proper evaluation of the time performance improvements that one can expect from it in practice. However, preliminary numbers indicate that the size of the native code is noticeably reduced, something which in turn has positive effects on the later optimization passes, often resulting in compile times even shorter than those of the HiPE compiler in R9B.

The type propagator is enabled by default at the normal optimization level `o2` (or higher).

2.4.2 *Handling of floats*

In the runtime system, atomic ERLANG values are represented as tagged 32-bit words; see [10]. Whenever a tagged value is too big to fit into one machine word the value is *boxed*, i.e., put on the heap with a header word preceeding it which is pointed to by the tagged value. Floating point numbers have 64-bit precision and are therefore typically boxed. This means that whenever they need to be used as operands to a floating point operation, they need to be unboxed, and after the operation is performed the result must then be boxed and stored back on the heap.

To avoid this overhead, starting from R9B, the BEAM has been enhanced with special floating point instructions that operate directly on untagged values. This has sped up the handling of floats considerably since the number of boxing/unboxing operations are reduced. However, since the BEAM code is interpreted, floating point arithmetic is still not taking advantage of features available at the floating point unit (FPU) of the target architecture, such as machine registers. More specifically, the operands are put into the FPU, the operation is performed, and then the result is taken out and stored in memory.

In the HiPE compiler, floating point values are mapped to the FPU and are kept there for as long as possible, eliminating even more overhead from floating point calculations. In [8] we have described in detail the two back-end specific schemes used in the mapping. Our performance comparison shows that HiPE-compiled floating point intensive code can be considerably faster than floating-point aware BEAM bytecode. Table 1 gives an idea of the performance improvements that can be expected across a range of programs manipulating floats.

To maximize the gain of the floating point instruction the user is encouraged to use appropriate `is_float/1` guards that currently communicate to the BEAM compiler the floating point type information[2] and to try to keep floating point arithmetic instructions together in blocks, i.e., not split them up by inserting other instructions that can just as well be performed before or after the calculations.

The more efficient, target-specific compilation of floating point arithmetic is enabled by default starting at optimization level `o1`.

2.4.3 *Handling of binaries*

Proper support for the bit syntax [9] was introduced into Erlang/OTP in R8B. Initially, the HiPE compiler used a rather naïve compilation scheme: binary matching instructions of the BEAM were translated into calls to C functions which were part of the interpreter's supporting routines. As a result, the HiPE-compiled code was actually slightly slower than the BEAM code because of costs in switching between native and interpreted code (cf. also Section 3). To remedy this, we proposed and implemented a scheme that relies on a *partial translation* of binary matching operations. This scheme identifies special "common" cases of binary match-

[2]Explicitly writing such guards will become unnecessary when the global type analysis gets fully integrated in HiPE; see Section 2.5.2.

Table 1: Performance of BEAM and HiPE in R9B on programs manipulating floats (times in ms).

Benchmark	BEAM	HiPE	speedup
float_bm	14800	4040	3.66
barnes-hut	10250	4280	2.39
fft	16740	8890	1.88
wings	8310	7370	1.12
raytracer	9110	8500	1.07
pseudoknot	3110	1440	2.16

(a) Performance on SPARC.

Benchmark	BEAM	HiPE	speedup
float_bm	1930	750	2.57
barnes-hut	1510	600	2.51
fft	2830	1450	1.95
wings	1160	850	1.36
raytracer	1200	1070	1.12
pseudoknot	380	140	2.71

(b) Performance on x86.

ings and translates these completely into native code, while the remaining "uncommon" cases still call C functions in order to avoid extensive code bloat. The implementation of this compilation scheme is described in [4] and is included in the HiPE compiler as of the R9B release of Erlang/OTP.

The performance of this scheme on several different benchmarks involving binaries is shown in Table 2. The first three benchmarks test the speed of binary matching: the **bsextract** benchmark takes a binary containing a GTP_C message as input, extracts the information from the message header, and returns it. The **bsdecode** benchmark is similar but rather than simply extracting a binary, it also translates the entire message into a record. The **ber_decode** benchmark, generated by the ASN.1 compiler, parses a binary. The last two benchmarks, **bsencode** and **ber_encode**, test the speed of binary creation rather than matching.

As expected, speedups are obtained when there is information available at compile time to identify cases which can be compiled fully to native code. Such, for example, is the case when the binary segment sizes are constant or when it is possible to determine statically that a segment of a binary starts at a byte boundary. In other words, to achieve best time performance, it might be advisable to use some extra space to guarantee that each element starts at a byte boundary. For example, if one wants to use binaries to denote 16 integers where each integer needs 7 bits it is possible to pack them so that they only take up 14 bytes. If each integer is put a byte boundary, the binary will take up more space (16 bytes), but the binary matching operations will be performed faster.

The HiPE compiler option `inline_bs` enables native code compilation of the bit syntax. This option is selected by default at optimization level `o1` or higher, and the only reasons for a user to disable it is either to test its performance effects or if code size is a serious concern.

2.5 Planned extensions for the near future

2.5.1 Better translation of binaries

The compilation scheme introduced in R9B made binary matching faster, but more work has since been done to make it even faster. In the upcoming R9C release, a new scheme for compiling binary matching will be included. Rather than relying on a partial translation and having BEAM be in control, the entire binary matching operation will fall in the hands of the native code compiler. This has made it possible to avoid several unnecessary mode switches. With this scheme most binary matching code will make no calls to the C functions which are used strictly as a last resort when a single operation becomes very complex.

In addition to this a new scheme to compile binary creation has been developed. It is developed in a similar fashion to the binary matching scheme by changing calls to C functions into specialized versions of these functions that are then translated into native code.

As seen in Table 3, the performance of HiPE compiled code has been improved substantially. The speedup for the matching benchmarks ranges from 1.5-4 times compared to BEAM. The speedup for benchmarks that create binaries is more than 2 times on x86 and more than 1.5 times on SPARC.

In connection with the effort to compile directly from Core Erlang to native code a project has started to further improve the compilation of binary matching as new possibilities open up when the structure of the matching becomes visible to the compiler. The result of this project will likely be available in the R10 release.

2.5.2 Global type analysis

As described in Section 2.4.1, the HiPE compiler now includes a local type propagator which works on individual functions. However, if no assumptions can be made about the arguments to the functions, only the most basic type information can be found. We have therefore implemented a *global type analyzer* which processes a whole module at a time. It can generally find much more detailed type information, but the precision depends to a large extent on the programming style of the analyzed code. Since an exported function can potentially be called from anywhere outside the module and with any inputs, it is not possible to make any assumptions about the types of the arguments to exported functions. The best precision is achieved when only the necessary interface functions are exported, and the code does all or most of its work within the same module. When module boundaries are crossed, type information is lost. For most built-in functions, however, we can know what types of data they accept as input and what types they return.

We are currently working on how to take advantage of the gathered type information (in combination with the local type propagator). First of all, we are often able to remove unnecessary type checks from the code. Second, it is sometimes possible to avoid repeatedly tagging and untagging values (cf. Section 2.4.2). Third, global type analysis makes it possible to avoid creating tuples for returning multiple values from a function, when the result of the function is always immediately unpacked – instead, multiple values can be passed directly in registers or on the stack.

Note that the global type analysis is *not a type checker*

Table 2: Performance of BEAM and HiPE in R9B on programs manipulating binaries (times in ms).

Benchmark	BEAM	HiPE	speedup
bsextract	15540	8450	1.84
bsdecode	27070	26860	1.01
ber_decode	14350	9130	1.57
bsencode	14210	15870	0.90
ber_encode	18720	16280	1.15

(a) Performance on SPARC.

Benchmark	BEAM	HiPE	speedup
bsextract	14500	7350	1.97
bsdecode	13490	12970	1.04
ber_decode	16500	9200	1.79
bsencode	17540	16030	1.09
ber_encode	21870	17420	1.26

(b) Performance on x86.

Table 3: Performance of BEAM and HiPE in a pre-release of R9C on programs manipulating binaries.

Benchmark	BEAM	HiPE	speedup
bsextract	13380	4060	3.30
bsdecode	26060	21110	1.23
ber_decode	13980	5720	2.44
bsencode	16960	11070	1.53
ber_encode	18150	9510	1.91

(a) Performance on SPARC.

Benchmark	BEAM	HiPE	speedup
bsextract	14700	3560	4.13
bsdecode	13670	7780	1.76
ber_decode	15610	6070	2.57
bsencode	16790	7290	2.30
ber_encode	22560	10860	2.08

(b) Performance on x86.

or type inference system, i.e., the user is not able to specify types (because the user cannot be completely trusted), and furthermore, the fact that an input parameter is always *used* as e.g. an integer does not mean that the passed value will always *be* an integer at runtime. Indeed, the current implementation does not even give a warning to the user if it detects a type error in the program, but just generates code to produce a runtime type error. This might change in the future, to make the type analyzer useful also as a programming tool.

2.5.3 Compilation from Core Erlang

A new feature of the HiPE compiler is the ability to compile to native code directly from the ERLANG source code, (i.e., instead of starting with the BEAM virtual machine code, which was previously the only way). This is done by generating HiPE's intermediate ICode representation directly from the Core Erlang code which is produced by the Erlang/OTP compiler. No BEAM code needs to have been previously generated. The advantages of this are better control over the generated code, and greater ability to make use of metadata about the program gathered on the source level, such as global type analysis information.

Currently, the way to do this is to add an extra option `core` when compiling to native code:

```
c(my_module, [native, core]).
```

However, this method of compiling is not yet fully functional in the coming R9C release, in that some programming constructs are not yet handled properly. We intend to have compilation from source code completely implemented in release R10.

We expect that in the future, compilation from source code will be the default method of the HiPE compiler. The compilation from BEAM will however still be available, for those cases when the source code is not available, or it is for other reasons not possible to recompile from the sources.

3. RECOMMENDATIONS ON HIPE'S USE

3.1 Improving performance from native code

- If your application spends most of its time in known parts of *your* code, and the size of those parts is not too large, then compiling those parts to native code will maximize performance.
- Largish self-contained modules with narrow external interfaces allow the compiler to perform useful module-global type analysis and function inlining, when compiling via Core Erlang.
- While very deep recursions are not recommended, they are much more efficient in native code than in BEAM code. This is because the HiPE runtime system includes specific optimizations (*generational stack scanning* [3, 11]) for this case.
- Monomorphic functions, functions that are known to operate on a single type of data, are more likely to be translated to good code than polymorphic functions. This can be achieved by having guards in the function heads, or by avoiding to export the functions and always calling them with parameters of a single type, known through guards or other type tests.
- When using floating point arithmetic, collect the arithmetic operations in a block and try to avoid breaking it up with other operations; in particular try to avoid calling other functions. Help the analysis by using the guard `is_float/1`. You might still benefit from this even if you do not manage to keep the operations in a block; the risk of losing performance is minimal.
- Order function and case clauses so that the cases that are more frequent at runtime precede those that are less frequent. This can help reduce the number of type tests at runtime.

3.2 Avoiding performance losses in native code

- If the most frequently executed code in your application is too large, then compiling to native code may give only a small or even negative speedup. This is because native code is larger than BEAM code, and in this case may suffer from excessive *cache misses* due to the small caches most processors have.

- Avoid calling BEAM-code functions from native-code functions. Doing so causes at least two mode switches (one at the call, and one at the return point), and these are relatively expensive. You should native-compile *all* code in the most frequently executed parts, including Erlang libraries you call, otherwise excessive mode switching may cancel the performance improvements in the native-compiled parts.

- Do *not* use the `-compile(export_all)` directive. This reduces the likelihood of functions being inlined, and makes useful type analysis impossible.

- Avoid crossing module boundaries too often (making remote calls), since the compiler cannot make any assumptions about the functions being called. Creative use of the pre-processor's `-include` and `-define` directives may allow you to combine performance and modular code.

- Avoid using 32-bit floats when using the bit syntax, since they always require a mode switch. It is also costly to match out a binary that does not start at a byte boundary, mainly because this requires that all the data of the binary is copied to a new location. If on the other hand a binary starting at a byte boundary is matched, a sub-binary which only contains a pointer to the data is created. When variable segment lengths are used it is beneficial to have a unit that is divisible by 8, because this means that byte boundary alignment information can be propagated.

3.3 Cases when native code does not help

- Be aware that almost all BIF calls end up as calls to C functions, even in native code. If your application spends most of its time in BIFs, for instance accessing ETS tables, then native-compiling your code will have little impact on overall performance.

- Similarly, code that simply sends and receives messages without performing significant amounts of computation does not benefit from compilation to native code; again, this is because the time is mostly spent in the runtime system.

4. THE TRUTH, THE WHOLE TRUTH, AND NOTHING BUT THE TRUTH

Significant work has been put into making HiPE a robust, "commercial-quality" compiler.[3] As a matter of fact, we have mostly tried to follow BEAM's decisions in order to preserve the observable behavior of ERLANG programs, even if that occasionally meant possibly reduced speedups in performance. Still, a couple of small differences with code produced by BEAM exist, and the user should be aware of some limitations. We document them below.

[3] At the time of this writing, July 2003, we are not aware of any outstanding bugs.

4.1 Incompatibilities with the BEAM compiler

- Detailed stack backtraces are currently not generated from exceptions in native code; however, where possible, the stack trace contains at least the function where the error occurred. Performing pattern matching on stack backtraces is not recommended in general, regardless of the compiler being used.

- The old-fashioned syntax `Fun = {M,F}, Fun(...)` for higher-order calls is not supported in HiPE. In our opinion, it causes too many complications, including code bloat. Proper `funs` should be used instead, or explicit calls `M:F(...)`.

- On the x86, floating-point computations may give different (more precise) results in native code than in BEAM. This is because the x86 floating-point unit internally uses higher precision than the 64-bit IEEE format used for boxed floats, and HiPE often keeps floats in the floating-point unit when BEAM would store them in memory; see [8].

4.2 Current limitations

- Once inserted into the runtime system, native code is never freed. Even if a newer version of the code is loaded, the old code is also kept around.

- The HiPE compiler recognizes literals (constant terms) and places them in a special *literals area*. Due to architectural limitations of the current Erlang/OTP runtime system, this is a single area of a fixed size determined when the runtime system is compiled. Loading a lot of native code that has many constant terms will eventually cause the literals area to fill up, at which point the runtime system is terminated. A short-term fix is to edit `hipe_bif0.c` and explicitly make the literals area larger.

5. CONCLUDING REMARKS

We have presented a user-oriented description of features and characteristics of the HiPE native code compiler, which nowadays is integrated in Erlang/OTP and easily usable by ERLANG application developers and aficionados. We hold that HiPE has a lot to offer to its users. Some of its benefits are described in this paper. Others, perhaps more exciting ones, await their discovery.

One final word of advice: HiPE, like any compiler, can of course be treated as a "black-box", but we generally recommend to creatively explore its options and flexibility and add color to your life!

6. ACKNOWLEDGMENTS

The HiPE compiler would not have been possible without the prior involvement of Erik "Happi" Stenman (formerly Johansson) in the project. Its integration in Erlang/OTP

would still be a dream without close collaboration with members of the Erlang/OTP group at Ericsson (Björn Gustavsson, Kenneth Lundin, and Patrik Nyblom), and the active encouragement of Bjarne Däcker. HiPE's development has been supported in part by the ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson Utvecklings AB.

7. REFERENCES

[1] R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, Sept. 2001.

[2] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 030, Information Technology Department, Uppsala University, Nov. 2000.

[3] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'98*, pages 162–173. ACM Press, 1998.

[4] P. Gustafsson and K. Sagonas. Native code compilation of Erlang's bit syntax. In *Proceedings of ACM SIGPLAN Erlang Workshop*, pages 6–15. ACM Press, Nov. 2002.

[5] E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, Sept. 2000.

[6] E. Johansson, M. Pettersson, K. Sagonas, and T. Lindgren. The development of the HiPE system: Design and experience report. *Springer International Journal of Software Tools for Technology Transfer*, 2003. To appear.

[7] E. Johansson and K. Sagonas. Linear scan register allocation in a high performance Erlang compiler. In *Practical Applications of Declarative Languages: Proceedings of the PADL'2002 Symposium*, number 2257 in LNCS, pages 299–317. Springer, Jan. 2002.

[8] T. Lindahl and K. Sagonas. Unboxed compilation of floating point arithmetic in a dynamically typed language environment. In R. Peña and T. Arts, editors, *Implementation of Functional Languages: Proceedings of the 14th International Workshop*, number 2670 in LNCS, pages 134–149. Springer, Sept. 2002.

[9] P. Nyblom. The bit syntax - the released version. In *Proceedings of the Sixth International Erlang/OTP User Conference*, Oct. 2000. Available at http://www.erlang.se/euc/00/.

[10] M. Pettersson. A staged tag scheme for Erlang. Technical Report 029, Information Technology Department, Uppsala University, Nov. 2000.

[11] M. Pettersson, K. Sagonas, and E. Johansson. The HiPE/x86 Erlang compiler: System description and performance evaluation. In Z. Hu and M. Rodríguez-Artalejo, editors, *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, number 2441 in LNCS, pages 228–244. Springer, Sept. 2002.

[12] J. H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305. ACM Press, June 1991.

[13] M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In A. Mycroft, editor, *Proceedings of the Second Static Analysis Symposium*, number 983 in LNCS, pages 366–381. Springer, Sept. 1995.

[14] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.

All you wanted to know about the HiPE compiler (but might have been afraid to ask)

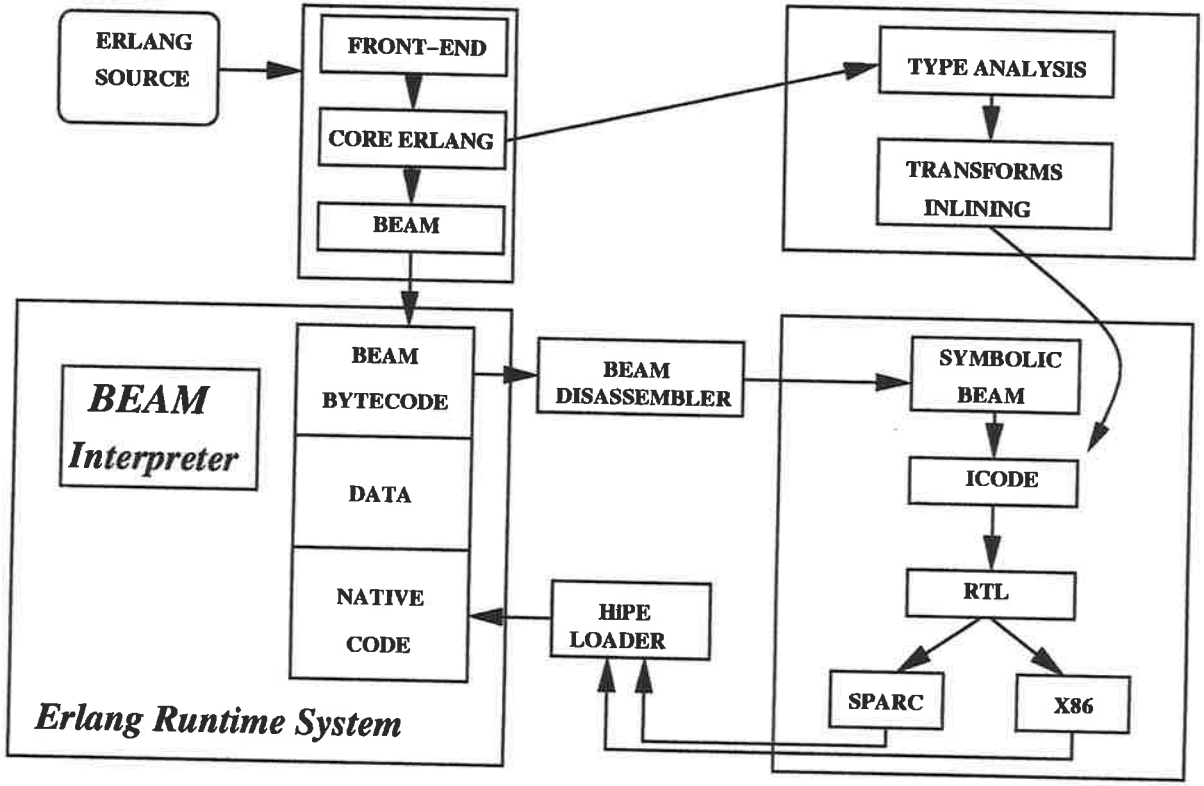
K. Sagonas, M. Pettersson, R. Carlsson,
P. Gustafsson, T. Lindahl

The High-Performance Erlang Group
Computing Science Department
Uppsala University, Sweden

Contents

- Brief overview of HiPE's architecture.
- HiPE–HOWTO: compiling and loading code, compiler options
- Recent improvements.
- Planned extensions.
- HiPE do's and dont's.
- (The few) Known incompatibilities and limitations.

HiPE's architecture



HiPE–HOWTO

Simply add the `native` flag to the usual BEAM compilation tools!

```
1> c(Module, [native]).
```

compile and load interactively

```
2> compile:file(Module, [native]).
```

compile to `.beam` file with both BEAM and native code; native code in a `.beam` file is automatically enabled at load-time

```
erlc +native Module.erl
```

in a Unix shell or Makefile

HiPE–HOWTO, continued

For HiPE-specific options, add `{hipe,Optionlist}` to the list of options (in addition to `native`):

```
c(Module, [native, {hipe, [o3]}]).
erlc +native '+{hipe, [o3]}' Module.erl
-compile({hipe, [o3]}).
```

- `o0`, `o1`, `o2`, `o3` Choose optimisation level. `o2` is default.
- {`regalloc,linear_scan`} Specify the use of a specific register allocator. `coalescing` is default on x86 (and SPARC at `o3`), but it can be slow on large functions.

HiPE–HOWTO, continued

To find out what options there are and what they do, use `hipe:help()`, `hipe:help_options()`, and `hipe:help_option(Option)`. For example:

`hipe:help_option(regalloc)` tells you what register allocators you can choose from.

`hipe:help_option(o3)` tells you what internal compiler options `o3` enables.

Recent improvements

Bit syntax

- In R9B, HiPE recognised special-cases of BEAM's bit-syntax instructions, and implemented them with native code. Other cases became calls to C functions. Speedup over BEAM generally between 1.0–2.0.
- In R9C, HiPE identifies the entire block of BEAM instructions for a binary matching or creation expression, and compiles it as a single unit. This results in better code and fewer calls to C functions. Speedup over BEAM in the 1.5–4.0 range.

Bit syntax speedups in R9C

Benchmark	speedup SPARC	speedup x86
bsextract	3.30	4.13
bsdecode	1.23	1.76
ber_decode	2.44	2.57
bsencode	1.53	2.30
ber_encode	1.91	2.08

Recent improvements

Floating-point arithmetic

- A single f.p. operation becomes: type test, move value from heap to FPU, do operation, allocate heap, move value from FPU to heap, tag pointer.
- Starting with R9B, BEAM recognises blocks of f.p. operations, and uses new instructions for unboxed f.p. arithmetic. BEAM now skips the allocate, store, tag, test, load sequence between pairs of f.p. operations.
- HiPE goes further and keeps intermediate f.p. values in the FPU rather than in memory "registers". This reduces memory traffic and delays between f.p. instructions.

Floating-point benchmark speedups

Benchmark	speedup SPARC	speedup x86
float_bm	3.66	2.57
barnes-hut	2.39	2.51
fft	1.88	1.95
wings	1.12	1.36
raytracer	1.07	1.12
pseudoknot	2.16	2.71

Recent improvements

Local type analysis

- Propagates type information from guards and other operations that yield known types.
- Works on a single function at a time.
- Eliminate redundant type tests.
- Specialise overloaded operations (e.g. +) when types are known.
- No performance results yet, but code size is reduced, which also makes remaining compiler passes faster.

Recent improvements

Miscellaneous

- Apply (`M:F(...)` calls) is now implemented natively. Major speedup in apply-intensive code. Also improves "generic server".
- Mailbox operations in receives are now inlined. 10–15% improvement in receive/timer-intensive benchmarks.
- Compile-time literals are merged, reducing their space usage.
- A source of excessive compile-times has been fixed.

Planned extensions

Module-global type analysis

Extends the local type analysis with information about actual parameters and return values from function calls.

- More opportunities for type check removal and specialising overloaded operations.
- Tuples can be returned in registers or on the stack.

The precision of the analysis depends the amount of code it sees at a time, the size of the external interfaces, and on programming style.

Planned extensions

Module-global compilation from source

Instead of compiling a function at a time, via its BEAM code, we are working on compiling from Erlang source, via Core Erlang.

- Type analysis is potentially more precise when done at a level closer to normal Erlang.
- Easier to utilise high-level meta-data (e.g. types) for things like function inlining and optimised calling conventions.
- Avoiding BEAM code gives us better control over constructs like pattern-matching and bit syntax operations.

Do's and Dont's

- Basic fact #1: Compiling Erlang code to native code makes *it* run faster, but doesn't change anything else.
- Basic fact #2: Switching modes (from native to BEAM and vice versa) is more expensive than a simple function call.

Do's and Dont's, continued

Do compile your code if it performs a lot of computations.

- Traditional functional code with lots of function calls, data construction, and pattern matching, clearly benefits from being compiled to native code.

  Deep recursions are several times faster in native code due to explicit garbage collection support for deep native stacks.

- Bit-syntax operations on binaries may be several times faster in native code.

- Blocks of floating-point operations may be several times faster in native code.

Do's and Dont's, continued

Helping the compiler:

- Bit-syntax optimisations rely on segments being 8-bit aligned. Variable-length segments should use a multiple-of-8 unit.

  32-bit floats in binaries aren't handled efficiently.

- Monomorphic code is more likely to be translated to good native code. Using guards, e.g., `is_float/1`, may help the compiler detect and utilise relevant type restrictions.

- Order clauses so that more likely cases precede less likely cases.

Do's and Dont's, continued

Helping the compiler, continued:

- Largish self-contained modules with narrow external interfaces help type analysis and inlining decisions.

  The `-compile(export_all)` directive makes the external interface wide-open, which limits the precision of type analysis and makes inlining less likely to occur.

  At a remote call, the compile cannot in general make any assumptions about the called function or what it will return.

Do's and Dont's, continued

While in native code, avoid leaving it.

- Avoid frequent calls to BEAM-mode functions.
- Ensure that Erlang libraries you use frequently are compiled to native code. This includes standard libraries.
- If you call function closures in native code, ensure that they also come from native code.
- It's best to first profile the application, and then compile the most frequently used parts to native code.

Do's and Dont's, continued

Don't compile your code if time is mostly spent elsewhere.

- Some applications are "BIF bound", being mostly glue around things like ETS tables or drivers.
- Some applications are "scheduling bound", being mostly send and receive agents that don't do much actual computation.

Compiling these can only yield limited performance improvements.

Known incompatibilities

- Stack dumps at exceptions are less detailed than in BEAM. Assume `bug(X) -> X+1`. Then calling `bug(a)` generates

  `{badarith,[]}`

  if `bug/1` is in native code, but

  `{badarith,[{Module,bug,1},...]}`

  if `bug/1` is in BEAM code.

- `Fun = {M,F}`, `Fun()` is not supported. Use proper `funs` or `M:F()` calls instead.

- Floating-point on x86 uses higher precision for intermediate values and calculations than BEAM does, so HiPE may deliver slightly different (more accurate) results.

Current limitations

- Native code is never freed. Repeatedly loading newer versions of the code leads to a (bounded) space leak.

  Fixing this requires a redesign of the HiPE object code loader.

- There is a *single* global area for compile-time constant terms, and this area may overflow if large amounts of native code is loaded.

  This is a consequence of the design of Erlang/OTP's memory management system. We are hoping that R10 will fix this.

PROFILE-DRIVEN INLINING FOR ERLANG

Thomas Lindgren
e-mail: thomasl_erlang@yahoo.com

Abstract

Inlining of function calls is a common optimization for high-level languages. While *static* inlining has been described for functional languages, *profile-driven* inlining of functional programs has to our knowledge not been explored. In this paper, we describe a simple and powerful algorithm for function inlining using profile data and give some tentative performance data.

INTRODUCTION

Inlining is a straightforward, common and useful optimization for functional, object-oriented and imperative languages. Inlining a function call f(X1,...,Xn), or *call site* means replacing the call with a copy of the function body of f/n and binding the formal parameters to the actual parameters X1, ..., Xn.

Doing so enables the following optimizations:

- The cost of the function call can be removed.
- Duplicating the function body means the code can be specialized with regard to the actual parameters. For example, operations on constant parameters can be simplified.
- Compiler optimizations often work only on a single function at a time, so enlarging the function may help.
- Native code optimizations such as code motion, instruction scheduling and register allocation work better with larger sections of code.

Unlimited inlining would yield too large program, negating all performance advantage by increasing cache misses, or would not terminate at all. For reasonable performance, the inlining algorithm has to decide which call sites to attack, and when to stop. The interesting issue with inlining, then, is what heuristics are used.

There are two main methods of doing this: static inlining, which examines the program structure to decide (e.g., inlining only calls to small functions), and dynamic or profile-driven inlining, which uses profiling information to focus inlining on the frequently executed, hot, regions of code.

The BEAM compiler already performs inlining of some code, in R9C0 using a static algorithm proposed by Wadell and Dybvig for Scheme [5] and redeveloped for Erlang by Richard Carlsson. The performance improvements on application code are, however, so far slight [2].

For functional languages, research has focussed on static inlining, in particular connected to higher-order functions; the author is at the time of writing unaware of any previous work on profile-guided inlining of functional languages. In this paper, we introduce a simple profile-driven algorithm for cross-module inlining of Erlang.

INLINING ALGORITHM

An example

We begin by describing the intuition of our algorithm with an example. Consider the following example program, where each function is annotated with the number of times it was executed, and each call site is annotated with the number of times it was visited during profiling.

We assume that there is no code size restriction, but that call sites must be visited at least 201 times to be considered for inlining.

```
%% f/1 called 1000 times
f({a,X}) -> g(X);   %% visited 700 times
f({b,X}) -> h(X).   %% visited 300 times

%% g/1 called 700 times
g({c,Y}) -> g1(Y);  %% visited 500 times
g({d,Y}) -> h1(Y).  %% visited 200 times

%% h/1 called 300 times
h({e,Z}) -> g1(Z);  %% visited 200 times
h({f,Z}) -> h1(Z).  %% visited 100 times

%% g1/1 called 500+200 times
g1(yes) -> true;
g1(_) -> false.

%% h1/1 called 200+100 times
h1(no) -> true;
h1(_) -> false.
```

We want to inline the most frequently visited function calls. We begin by ordering all of the call sites by number of visits, then consider each entry in priority order.

```
g/1 in f/1:  700
g1/1 in g/1: 500
h/1 in f/1:  300
h1/1 in g/1: 200
g1/1 in h/1: 200
```

We begin by inlining g/1 in f/1, which yields

```
f({a,X}) ->
  case X of
    {c, Y} -> g1(Y);
    {d, Y} -> h1(Y)
  end;
f({b,X}) -> h(X).
```

Note that two new call sites have appeared: g1/1 in f/1 and h1/1 in f/1. What are their call frequencies? In general, we can only estimate this value, which we do by taking the ratio of the call site visits divided by the total function visits in the original function. This is then multiplied with the number of visits to the call site.

*(internal call site visits/function visits) * call site visits*

In this case, the first ratio is (500/700) for g1/1 in g/1, meaning the ratio is 0.714. We then multiply the actual number of visits to the call site (700 calls), to yield 500. (Not such a coincidence, since only f/1 calls g/1.) So the call site g1/1 in f/1 gets priority 500. In the same vein, h1/1 in f/1 gets priority 200.

```
g1/1 in g/1: 500
g1/1 in f/1: 500
 h/1 in f/1: 300
h1/1 in g/1: 200
g1/1 in h/1: 200
h1/1 in f/1: 200
```

Two sites have the same priority 500, so we use a tie-breaker to select the next site. Let us assume it is g1/1 in g/1; we inline it to yield:

```
g({c,Y}) ->
   case Y of
      yes -> true;
      _ -> false
   end;
g({d,Y}) -> h1(Y).  %% visited 200 times
```

No new call sites appear, so we proceed to the next site, g1/1 in f/1, which yields:

```
f({a,X}) ->
  case X of
    {c, Y} ->
       case Y of
          yes -> true;
          _ -> false
       end;
    {d, Y} -> h1(Y)
  end;
f({b,X}) -> h(X).
```

Again, no new call sites appear. We then inline h/1 in f/1:

```
f({a,X}) ->
  case X of
    {c, Y} ->
       case Y of
          yes -> true;
          _ -> false
       end;
    {d, Y} -> h1(Y)
  end;
f({b,X}) ->
  case X of
     {e, Z} -> g1(Z);
     {f, Z} -> h1(Z)
  end.
```

In h/1 there are two call sites: g1/1 and h1/1, which have the ratios 2/3 and 1/3 respectively. The inlined call site h/1 in f/1 had priority 300, so the new call sites get priority 200 and 100, respectively. The new priority queue looks like:

```
h1/1 in g/1: 200
g1/1 in f/1: 200
g1/1 in h/1: 200
(1) h1/1 in f/1: 200
(2) h1/1 in f/1: 100
```

Note that there now are two different calls to h1/1 in f/1, which have been distinguished by an index. At this point, however, no call site has a priority of 201 or more, so inlining stops.

The algorithm

We now want to implement the intuitive algorithm of the example. This is done in three parts.

1. A preprocessing step that computes call site ratios and function sizes.
2. A loop over a priority queue to decide which call sites to inline.
3. A code generation step that actually performs the inlining.

Preprocessing. We traverse each function to yield a mapping *call_sites* from MFA to (Call_site, Ratio), where Ratio is a value $0 \leq \mathit{Ratio} < 1$. This denotes the ratio of how many times call site c in function f is visited compared to how many times f itself is visited. This initial ratio is then adjusted downwards to get rid of some unfortunate special cases, yielding a final ratio.

First, consider the function `f() -> g()`. The call to g/0 would have a ratio of 1.0 since it is always invoked when f/0 is executed. We adjust such call sites to a lesser value, currently 0.99.

For a heavily executed loop, the recursive call will have a ratio of nearly 1.0, which means it will be inlined heavily. (In one case, we saw our system unroll a recursive loop 68 times.) However, this is seldom productive and overstates the benefit of inlining a recursive call. We heuristically *scale down* the ratios of such call sites by a factor 0.5.

Hence, for each call site in f, we now have a ratio indicating how often the call site on average will be executed when f is executed. For function f with call sites $c_1, \ldots, c_n$ with associated ratios $r_1, \ldots, r_n$, we define:

$$\mathit{call\_sites}(f) = \{(c_1, r_1), \ldots, (c_n, r_n)\}$$

Inlining must also restrict **code growth**, since the benefits of inlining would otherwise be cancelled by worse cache performance. We therefore also estimate the code size of each function as a mapping $\mathit{size}(f)$ from MFA to integers, always yielding a code size greater than or equal to one.

Priority-based inlining. The data structure to keep track of what inlinings to do is an *inline forest*. Each function is represented by a tree in the forest. The tree originally consists of the function node and its immediate call sites. The nodes and leaves of every tree are call sites, annotated with the estimated number of visits.

We define a function $\mathit{expand}(c, n)$ which, given a call site c and priority n to c, yields a list of call sites with associated priorities.

$$expand(c,n) = [(c, rn) \mid (c,r) \leftarrow call\_sites(function\_of(c))]$$

Now for the algorithm, shown below in pseudo-code. The main loop examines each call site in priority order, and decides whether to inline it. A call site is inlined if it (a) has sufficient priority, and if (b) inlining it does not break a predefined code growth limit.

If inlined, the corresponding node in the inline tree is extended with its call sites as new leaves, and the new call sites are added to the priority queue.

```
compute size(c) for all functions mfa;
compute expand(c,n) for all functions mfa;
seed inline forest with all functions;
initialize priority queue with initial call sites;
currsize := <estimated total size>;
maxsize := currsize * <predefined value>;
minprio := <predefined value>;

while priority queue non-empty
  let (c,n) be the call site c with highest priority n;
  if (n < minprio) then stop;
  if (size(called_function(c)) + currsize > maxsize) then <skip call site>
  else <expand call site, update inline forest and priority queue>
end

return inline forest;
```

Note that the call sites may include recursive calls. Recursion means the inline tree can potentially be extended infinitely, say, by inlining a recursive call repeatedly. Our algorithm overcomes this problem, as follows. Expanding a site with priority n always yields call sites with lower priorities $n' < n$ due to the definition of ratios, eventually growing smaller than `minprio`. Since $size(c) \geq 1$, `currsize` furthermore always increases, eventually approaching `maxsize`. Hence, the algorithm terminates.

Code generation. The final code generation phase is straightforward. We have a forest of inline trees, each of which will generate one function. (An example inline tree is shown in the Appendix.)

We traverse each tree recursively from root to leaves. For a node f with children $g_1, \ldots, g_n$, we first generate code for each child g_i, which yields a fun λ_i. We then substitute e_i for the function at the proper call site, converting $g_i(x)$ into $\lambda_i(x)$. When all children of f have been inlined into the body of f, we convert f into a fun λ and return λ. On returning to the root of the tree, code generation for the corresponding function is finished.

Each function is then simplified heuristically; in particular, the inlined code has a number of fun:s applied to known arguments. These are simplified using beta reduction, which ideally opens up further possibilities for simplification.

The simplified functions are then collected into modules, compiled and executed.

A short example. Here is a short example of the mechanics of inlining a single function. The initial function is this:

```
minimum_octets(0,Acc) ->
    Acc;
minimum_octets(Val,Acc) ->
    minimum_octets(Val bsr 8,[Val band 255|Acc]).  %% site 1
```

Let us assume minimum_octets/2 is called 1000 times and that call site 1 is called 900 times. The ratio for site 1 is then 0.9. The size of minimum_octets/2 is estimated by considering that it has some pattern matching, two arithmetic operations and a list construction. (The precise estimated value is not interesting; our current method is fairly naive.)

The inliner subsequently decides to inline call site 1 (that is, "unrolling the loop" by one iteration). This is done by converting the called function into a fun and substituting it for the called function.

Here is the fun:

```
(fun (0,_5_Acc) ->
        _5_Acc;
    (_5_Val,_5_Acc) ->
        minimum_octets(_5_Val bsr 8,[_5_Val band 255|_5_Acc])
end)
```

We substitute the fun for the function call to minimum_octets/2. The inlined but unsimplified code is then the following:

```
minimum_octets(0,Acc) ->
    Acc;
minimum_octets(Val,Acc) ->
    (fun (0,_5_Acc) ->
            _5_Acc;
        (_5_Val,_5_Acc) ->
            minimum_octets(_5_Val bsr 8,[_5_Val band 255|_5_Acc])
    end)(Val bsr 8,[Val band 255|Acc]).
```

Our system then simplifies the code into the following:.

```
%% after simplify:
minimum_octets(0,Acc) ->
    Acc;
minimum_octets(Val,Acc) ->
    case {Val bsr 8,[Val band 255|Acc]} of
        {0,_5_Acc} ->
            _5_Acc;
        {_5_Val,_5_Acc} ->
            minimum_octets(_5_Val bsr 8,[_5_Val band 255|_5_Acc])
    end.
```

The BEAM compiler can now perform further optimizations. In the compiled code, shown below, we can see that most of the potential overhead has disappeared.

```
{function, minimum_octets, 2, 2}.
  {label,1}.
    {func_info,{atom,oct},{atom,minimum_octets},2}.
  {label,2}.
```

```
    {test,is_eq_exact,{f,3},[{x,0},{integer,0}]}.
    {move,{x,1},{x,0}}.
    {'%live',1}.
    return.
  {label,3}.
    {test_heap,2,2}.
    {bif,'bsr',{f,0},[{x,0},{integer,8}],{x,2}}.
    {bif,'band',{f,0},[{x,0},{integer,255}],{x,0}}.
    {put_list,{x,0},{x,1},{x,0}}.
    {test,is_eq_exact,{f,4},[{x,2},{integer,0}]}.
    return.
  {label,4}.
    {test_heap,2,3}.
    {bif,'bsr',{f,0},[{x,2},{integer,8}],{x,3}}.
    {bif,'band',{f,0},[{x,2},{integer,255}],{x,2}}.
    {put_list,{x,2},{x,0},{x,1}}.
    {move,{x,3},{x,0}}.
    {'%live',2}.
    {call_only,2,{f,2}}.
```

EVALUATION

We have evaluated the inlining algorithm on four benchmarks taken from OTP, similar to those in Ref [4]. While the workload and interface remains the same in this paper, the older paper used Erlang version R7B2, while this paper uses Erlang R9C0. There are substantial differences in implementation between the two versions, including the underlying code of the benchmarks, and the results are thus not comparable for these purposes. Furthermore, the current program in general selects more modules for optimization than in the previous paper.

In a previous paper [4], we described a number of techniques to improve cross-module inlining opportunities: apply optimization, outlining and module merging. For clarity, none of these extra optimizations were performed on the applications in this paper. However, we here assume that module merging is done after inlining (cf. Future Work below for a further discussion on this issue).

The evaluation was performed on an Athlon 1300+ PC with 512 MB memory running Erlang R9C0. The inlining algorithm per se was implemented in 460 lines of Erlang, though that number excludes a substantial support library. The benchmarks were executed ten times each, and the fastest results used for comparison. Speedup for each benchmark was computed as:

$$Speedup = T_{orig}/T_{opt}$$

We used the following applications:

- ldapv2. Generating and parsing ASN.1 data for the LDAPv2 protocol.
- gen_tcp. Socket communication using short and long messages.
- beam. BEAM compiler working on a number of source files.
- mnesia. Mnesia simulating simple HLR-like traffic.

The applications can be summarized by measuring the number of modules, functions and call sites. The number of local call sites are those accessible to a per-module algorithm. We also measured the number of call sites that were visited during profiling; only visited call sites were even considered for inlining.

Application	Modules	Functions	Call sites	Local sites	Visited sites
gen_tcp	13	658	1546	989	202
ldapv2	5	321	1038	616	140
beam	51	2347	9669	7594	2653
mnesia	63	4207	13390	8435	984

Local call sites are 60% to 78% of the total; the remainder are remote call sites and higher-order function calls.

It is interesting to note that only 7% (mnesia) to 27% (beam) of all call sites are ever visited during profiling. This suggests that profiling can eliminate large numbers of functions from being considered for inlining.

Performance results. We have only very preliminary performance results at the time of writing. The two larger benchmarks could not be executed, due to bugs in the code emitted by the inliner. Of the two remaining programs, the gen_tcp benchmark yields a small speedup of 4% at a size cost of 1%; examination shows that the inliner chose just to inline three functions in prim_inet. Ldapv2 shows a better speedup of 10% at a greater size cost.

In both benchmarks, we set the maximum code growth to 50%, which was not attained; our size estimation heuristic may be the culprit.

Application	Speedup (emu)	Size increase
gen_tcp	1.04	1.01
ldapv2	1.10	1.33

Finally, since native code compilation could hypothetically take greater advantage of inlined code, we tried to measure native code performance of the benchmarks. Unfortunately, the native compiler choked and we were unable to complete this task.

Our tentative conclusion is that the inlining algorithm can provide some speedup on a sizeable application. However, substantial work remains to debug, tune and enhance the inlining algorithm. This is further discussed below.

CONCLUSION AND FUTURE WORK

We have shown how to exploit profiling information to drive function inlining. Our algorithm performs cross-module inlining and recursive inlining. While our evaluation is incomplete, the inlined programs tentatively show reasonable speedups.

Cross-module inlining must handle Erlang's hot code loading feature. One approach to doing this at a near-zero runtime cost is described in Ref [3]: modules are merged into larger units of code, and former cross-module calls inside the merged entity are guarded to preserve the code loading semantics.

In a previous article [4], we described how to form *module aggregates* by merging the modules that called each other most frequently; inlining was then done inside each aggregate. The current approach to inlining instead suggests post hoc module aggregation, by first observing the actual cross-module inlinings, then merging the modules where cross-module inlinings occurred. The details of this approach remain to be worked out.

What has been presented here is in some sense work in progress. We have, for example, conducted a very basic evaluation in this paper. When considering the interactions between multiple inlining parameters, multiple optimizations, multiple benchmarks and multiple workloads, a more thorough performance evaluation seems necessary. It is likely that such experience would also lead to refinements to the heuristics described in this paper.

For instance, our heuristic of always choosing the most visited call site unless too large could be generalized to consider cost and benefit more explicitly [1], which could lead to better results.

STOCKHOLM, OCTOBER 2003

References

[1] M. Arnold, S. Fink, V. Sarkar, P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, ACM 2000.

[2] R. Carlsson, personal communication. October 2003.

[3] T. Lindgren. *Module merging: aggressive optimization and code replacement in highly-available systems.* Technical report 154, Computing Science Department, Uppsala University, 1998.

[4] T. Lindgren. Cross-module optimization. In Proc. Seventh Erlang User Conference. September 2001.

[5] O. Waddell, R. K. Dybvig. Fast and Effective Procedure Inlining. In Proc. Fourth International Symposium on Static Analysis (SAS '97), Springer LNCS 1302, 1997.

A Inline tree for mnesia

The inliner performs 802 inlines on the mnesia benchmark, where one of the core activities is mnesia:dirty_write/2. We show the inline tree for that function here, which involves inlining across a total of eight modules. The original version of dirty_write/2 is two lines of code; the inlined version of dirty_write/2 is somewhat more than 1000 lines of Erlang.

Many nested inlinings are done. Starting from the top, mnesia:dirty_write/2 contains an inlined call to mnesia:do_dirty_write/3, which contains an inlined call to mnesia_tm:dirty/2, which contains an inlined call to mnesia_tm:rec_dirty/2; one inlined call to mnesia_tm:async_send_dirty/4; one to mnesia_tm:val/1, and one to mnesia_tm:prepare_items/5. All but one of which themselves contain inlined calls.

```
mnesia:dirty_write/2
.mnesia:do_dirty_write/3
..mnesia_tm:dirty/2
...mnesia_tm:rec_dirty/2
...mnesia_tm:async_send_dirty/4
....mnesia_tm:async_send_dirty/6
.....mnesia_tm:async_send_dirty/6
......mnesia_tm:async_send_dirty/6
.......mnesia_tm:do_dirty/2
........mnesia_tm:do_commit/2
.........mnesia_tm:do_commit/3
..........mnesia_tm:do_update/4
..........mnesia_tm:do_update/4
..........mnesia_tm:do_update/4
..........mnesia_tm:do_snmp/2
..........mnesia_dumper:update/3
........mnesia_log:log/1
```

```
.........mnesia_dumper:incr_log_writes/0
..........mnesia_lib:incr_counter/2
.........mnesia_log:append/2
..........disk_log:alog/2
...........disk_log:notify/2
............disk_log_server:get_log_pids/1
.............disk_log_server:do_get_log_pids/1
.........mnesia_monitor:use_dir/0
......mnesia_tm:do_dirty/2
.......mnesia_tm:do_commit/2
........mnesia_tm:do_commit/3
.........mnesia_tm:do_update/4
..........mnesia_tm:do_update_op/3
...........mnesia_lib:db_put/3
...........mnesia_tm:commit_write/6
.........mnesia_tm:do_update/4
..........mnesia_tm:do_update_op/3
...........mnesia_lib:db_put/3
...........mnesia_tm:commit_write/6
.........mnesia_tm:do_update/4
..........mnesia_tm:do_update_op/3
...........mnesia_lib:db_put/3
...........mnesia_tm:commit_write/6
.........mnesia_tm:do_snmp/2
.........mnesia_dumper:update/3
.......mnesia_log:log/1
........mnesia_dumper:incr_log_writes/0
.........mnesia_lib:incr_counter/2
........mnesia_log:append/2
.........disk_log:alog/2
..........disk_log:notify/2
...........disk_log_server:get_log_pids/1
............disk_log_server:do_get_log_pids/1
........mnesia_monitor:use_dir/0
.....mnesia_tm:do_dirty/2
......mnesia_tm:do_commit/2
.......mnesia_tm:do_commit/3
........mnesia_tm:do_update/4
.........mnesia_tm:do_update/4
.........mnesia_tm:do_update_op/3
..........mnesia_lib:db_put/3
..........mnesia_tm:commit_write/6
........mnesia_tm:do_update/4
.........mnesia_tm:do_update/4
.........mnesia_tm:do_update_op/3
..........mnesia_lib:db_put/3
..........mnesia_tm:commit_write/6
........mnesia_tm:do_update/4
.........mnesia_tm:do_update/4
.........mnesia_tm:do_update_op/3
..........mnesia_lib:db_put/3
..........mnesia_tm:commit_write/6
........mnesia_tm:do_snmp/2
........mnesia_dumper:update/3
```

```
......mnesia_log:log/1
.......mnesia_dumper:incr_log_writes/0
........mnesia_lib:incr_counter/2
.......mnesia_log:append/2
........disk_log:alog/2
.........disk_log:notify/2
..........disk_log_server:get_log_pids/1
...........disk_log_server:do_get_log_pids/1
.......mnesia_monitor:use_dir/0
...mnesia_tm:val/1
...mnesia_tm:prepare_items/5
....mnesia_tm:check_prep/2
....mnesia_tm:do_prepare_items/7
.....mnesia_tm:prepare_nodes/5
......mnesia_tm:prepare_nodes/5
.......mnesia_tm:prepare_nodes/5
........mnesia_tm:prepare_node/5
........mnesia_tm:pick_node/4
.......mnesia_tm:prepare_node/5
........mnesia_tm:prepare_node/5
.......mnesia_tm:pick_node/4
......mnesia_tm:prepare_node/5
.......mnesia_tm:prepare_node/5
........mnesia_tm:prepare_node/5
......mnesia_tm:pick_node/4
.....mnesia_tm:prepare_snmp/7
....mnesia_tm:val/1
....mnesia_tm:val/1
```

What's new in R9C

Kenneth Lundin

ERICSSON

What's new in Erlang/OTP R9C

This presentation will highlight some of the new and changed functions in the latest release.

Outline of the presentation

- New and changed things
  - ERTS
  - Applications
  - Misc
- Incompatible things

ERICSSON

New and Changed things (ERTS)

- Memory management improvements with a number of different allocators which can be configured
  - **temp_alloc**, **eheap_alloc**, binary_alloc, ets_alloc, sl_alloc, **ll_alloc**, fix_alloc, sys_alloc, mseg_alloc

  Changed flags for E-node start

  erl +MB true +ME true +MEas aobf +r

  - E:\Program\erl5.3\erts-5.3\doc\html\erl.html
  - E:\Program\erl5.3\erts-5.3\doc\html\erts_alloc.html
- Improved memory instrumentation

ERICSSON

New and Changed things (ERTS)

- New! System flag **+Bi** starts an E-node which ignore break signals (i.e cannot be terminated with CTRL-C)
- New! **erlang:hibernate(Mod, Func, ArgList)**, reduce a process memory footprint as much as possible. Can be used instead of saving a process state in ets-tables

Note! **erl +P 262144** (default 32768 number of Erlang processes)

- New! **erlang:send/3** supersedes erlang:send_nosuspend
- New! Support for IPv6 in the gen_tcp, gen_udp and inets modules.

ERICSSON

New and Changed things (kernel/stdlib)

- New! integer_to_list(Integer, Base)

```
3> erlang:integer_to_list(10,16).
"A"
```

- New! list_to_integer(CharList, Base)

```
4> erlang:list_to_integer("A",16).
10
```

- New format characters for io:format/2 ~b, ~B, ~x, ~X, ~+ and ~#
- New format charaters for io:fread/2 ~u, ~- and ~#

New and Changed things (stdlib)

- New! The Erlang shell enables the user to write functions which can restrict the execution of certain functions.
- New! The modules **erl_tar** and **filelib** are now documented and supported

New and Changed things (compiler)

- Change! When updating a record the record tag and size is now **always checked**.
- New! In the bit syntax a size field can be bound during the same matching

```
<<Size, B:Size/binary,Rest/binary>> =
<<2,"AB",3,"CDE">>
```

ERICSSON

New and Changed things (crypto/ssl)

- Change! Crypto requires dynamically linked OpenSSL libraries that the user has to install (used via SNMP v3)
- Change! SSL is now based on OpenSSL

ERICSSON

New and Changed things (applications)

- Change! IDL compiler: The CORBA stub/skeleton-files generated are reduced in size and less dependent on the interface repository.
- New! ASN.1 compiler have support for partial decode for BER (can be used to improve performance depending on application)

New and Changed things (misc)

- New! Erlang Reference Manual
- New! **crashdump_viewer** for browsing Erlang crashdumps.

New and Changed things (misc)

- New! Now possible to build Open Source Erlang also on Windows, New Installation program for Windows
- HiPE improved performance for bit syntax matchings
- Erlang mode for EMACS now documented + bugs fixed + new feature for aligning arrows.
- Cover
  - new WebCover interface
  - export and import of cover data
  - multi node support
  - analyse_to_file can produce nice html output

ERICSSON

Crashdump Viewer

- HTML based tool for browsing Erlang crash dumps
- Displays any crashdump from OTP R7B01 and newer
- Available in ERV vob now for OTP R8B and R9B.
- Warning if dump is truncated
- example of dump

ERICSSON

Crashdump Viewer, general info

General information
Processes
Ports
ETS tables
Timers
Fun table
Atoms
Distribution information
Loaded modules
Internal Tables
Memory information
Documentation

Load New Crashdump

General information Help

Slogan	init terminating in do_boot ()
Node name	unknown
Crashdump created on	Mon Nov 4 04:20:07 2002
System version	Erlang (BEAM) emulator version 5.0.2.11.1
Compiled	Compiled on Thu Dec 20 09:43:34 2001
Atoms	3217
Processes	31
ETS tables	11
Funs	0

ERICSSON

Crashdump Viewer, process info

Process Information Help

Pid	Name	Spawned as	State	Reductions	Stack+heap	MsgQ Length
<0.63.0>	mnesia_locker	proc_lib:init_p/5	Running	4081077	233	0
<0.0.0>	init	otp_ring0:start/2	Waiting	404904	4181	0
<0.2.0>	erl_prim_loader	erl_prim_loader:start_it/4	Waiting	464792	610	0
<0.4.0>	error_logger	proc_lib:init_p/5	Waiting	4620161	832040	0
<0.5.0>	application_controller	proc_lib:init_p/5	Waiting	176517	2584	0
<0.7.0>		proc_lib:init_p/5	Waiting	45	377	0
<0.8.0>		application_master:start_it/4	Waiting	113	233	0
<0.9.0>	kernel_sup	proc_lib:init_p/5	Waiting	29752	610	0
<0.10.0>	rex	proc_lib:init_p/5	Waiting	3911	377	0
<0.11.0>	global_name_server	proc_lib:init_p/5	Waiting	303615	377	0
<0.12.0>		global:init_the_locker/1	Waiting	789	610	0
<0.13.0>	inet_db	proc_lib:init_p/5	Waiting	7335	233	0

ERICSSON

Crashdump Viewer, process details

Help

Process <0.63.0>			
Name	mnesia_locker	Spawned as	proc_lib:init_p/5
State	Running	Last scheduled in for	mnesia_locker:check_queue/5
Started	unknown	Spawned by	unknown
Reductions	4081077		
Stack+heap	233	OldHeap	233
Heap unused	85	OldHeap unused	233
Number of heap fragments	unknown	Heap fragment data	205 words
Msg queue length	0		
Link list	[<0.60.0>]		
Dictionary	[('$initial_call',(mnesia_sp,init_proc, [mnesia_locker,mnesia_locker,init,[<0.60.0>]])), ('$ancestors',[mnesia_kernel_sup,mnesia_sup,<0.57.0>])]		
StackDump	Expand (1017 bytes)		

ETS tables owned by this process Timers owned by this process

Incompatibilities

- **Erl_interface** , internal and interface header files separated if internal header files are used the application might not compile without changes.
- When updating **records** , **tag and size are always checked**
- The deprecated module **unix** is removed
- See the documentation

Erlang/OTP User Conference 2003 - Participants

Chairman and speakers				
Johan Blom	Mobile Arts	Stockholm, Sweden	johan.blom_at_mobilearts.se	Pub
Göran Båge	Mobile Arts	Stockholm, Sweden	goran.bage_at_home.se	Pub
Richard Carlsson	Uppsala university	Uppsala, Sweden	richardc_at_csd.uu.se	Pub
Bjarne Däcker	cs-lab.org	Stockholm, Sweden	bjarne_at_cs-lab.org	Pub
Per Gustafsson	Uppsala university	Uppsala, Sweden		Pub
John Hughes	Chalmers university	Gothenburg, Sweden	rjmh_at_cs.chalmers.se	Pub
Tobias Lindahl	Uppsala university	Uppsala, Sweden		Pub
Thomas Lindgren		Stockholm, Sweden	thomasl_erlang_at_yahoo.com	Pub
Kenneth Lundin	Ericsson	Stockholm, Sweden	kenneth_at_erix.ericsson.se	Pub
Mikael Pettersson	Uppsala university	Uppsala, Sweden	mikpe_at_csd.uu.se	Pub
Mickaël Rémond		Paris, France	mickael.remond_at_erlang-fr.org	Pub
Kostis Sagonas	Uppsala university	Uppsala, Sweden	kostis_at_user.it.uu.se	Pub
Juan José Sánchez Penas	University of Corunha	Coruña, Spain	juanjo_at_dc.fi.udc.es	Pub
Danie Schutte	Teba Bank	Midrand, South Africa	DANIESC.SCHUTTE_at_tebabank.com	Pub
Hal Snyder	Vail Systems	Chicago, USA	hal_at_vailsys.com	Pub
Torbjörn Törnkvist	Nortel	Stockholm, Sweden	tobbe_at_bluetail.com	
Ulf Wiger	Ericsson	Stockholm, Sweden	ulf.wiger_at_ericsson.com	Pub
Jesper Wilhelmsson	Uppsala university	Uppsala, Sweden	jesperw_at_it.uu.se	Pub
Mike Williams	Ericsson	Stockholm, Sweden	michael.williams_at_ericsson.com	Pub
Participants				
Peter Andersson	Ericsson	Stockholm, Sweden		Pub
Ingela Anderton	Ericsson	Stockholm, Sweden	ingela_at_erix.ericsson.se	Pub
Gunilla Arendt	Ericsson	Stockholm, Sweden	gunilla_at_erix.ericsson.se	

Joe Armstrong	SICS	Stockholm, Sweden	joe_at_sics.se	Pub
Thomas Arts	IT-university	Gothenburg, Sweden	thomas.arts_at_ituniv.se	Pub
Gösta Ask		Stockholm, Sweden	g.ask_at_telia.com	Pub
Robert Balla	Ericsson	Budapest, Hungary	qballro_at_cbe.ericsson.se	Pub
John-Olof Bauner	Ericsson	Budapest, Hungary	John-Olof.Bauner_at_ericsson.com	
Clara Benac Earle	University of Kent	Canterbury, England	cb47_at_kent.ac.uk	Pub
Per Bergqvist	Synapse	Stockholm, Sweden	per_at_synap.se	Pub
Johan Bevemyr	Nortel	Stockholm, Sweden	jb_at_bluetail.com	Pub
Éva Bihari	Ericsson	Budapest, Hungary	eva.bihari_at_ericsson.com	Pub
Martin Björklund	Nortel	Stockholm, Sweden	mbj_at_bluetail.com	Pub
Pascal Brisset	Cellicium	Bagneux, France	pascal.brisset_at_cellicium.com	Pub
Hans Bolinder	Ericsson	Stockholm, Sweden		
Francesco Cesarini	Erlang Training & Consulting	London, England	francesco_at_erlang-consulting.com	Pub
Mats Cronqvist	Ericsson	Budapest, Hungary	mats.cronqvist_at_eth.ericsson.se	Pub
Lars-Åke Fredlund	Uppsala university	Uppsala, Sweden	fred_at_sics.se	
Magnus Fröberg	Nortel	Stockholm, Sweden	magnus_at_bluetail.com	Pub
Luke Gorrie	Nortel	Stockholm, Sweden	luke_at_bluetail.com	Pub
Pär Grandin	Ericsson	Stockholm, Sweden		
Joakim Grebenö	Nortel	Stockholm, Sweden	jocke_at_bluetail.com	Pub
Dan Gudmundsson	Ericsson	Stockholm, Sweden		Pub
Björn Gustavsson	Ericsson	Stockholm, Sweden	bjorn_at_erix.ericsson.se	Pub
Gordon Guthrie	Scottish Enterprise Network	Glasgow, Scotland	gordon.guthrie_at_scotent.co.uk	
Per Hallin	Synapse	Stockholm, Sweden	per.hallin_at_synap.se	Pub
Siri Hansen	Ericsson	Stockholm, Sweden		Pub
Per Hedeland	Nortel	Stockholm, Sweden	per_at_bluetail.com	Pub
András Horváth	Ericsson	Budapest, Hungary	andras.horvath_at_ericsson.com	Pub
Bertil Karlsson	Ericsson	Stockholm, Sweden		

Mikael Karlsson	Creado Systems	Stockholm, Sweden	mikael.karlsson_at_creado.com	
Bengt Kleberg	Ericsson	Stockholm, Sweden	Bengt.Kleberg_at_ericsson.com	
Tord Larsson	Nortel	Stockholm, Sweden	tlarsson_at_nortelnetworks.com	
Fredrik Linder	CellPoint AB	Kista, Sweden	fredrik.linder_at_cellpoint.com	Pub
Matthias Läng	Corelatus	Stockholm, Sweden	matthias_at_corelatus.se	Pub
Luca Manai	Ericsson	Stockholm, Sweden	luca.manai_at_ericsson.com	Pub
Håkan Mattsson	Ericsson	Stockholm, Sweden	hakan_at_erix.ericsson.se	Pub
Peter Nagy	Ericsson	Budapest, Hungary	peter.nagy_at_ericsson.com	Pub
Hans Nilsson	Ericsson	Stockholm, Sweden	hans.r.nilsson_at_ericsson.com	Pub
Raimo Niskanen	Ericsson	Stockholm, Sweden		Pub
Annika Nordqvist	Ericsson	Stockholm, Sweden	annika.nordqvist_at_ericsson.com	
Patrik Nyblom	Ericsson	Stockholm, Sweden		Pub
Jan Henry Nyström	Heriot-Watt University	Edinburgh, Scotland	jann_at_macs.hw.ac.uk	Pub
Denes Pazmany	Ericsson	Budapest, Hungary	Denes.Pazmany_at_eth.ericsson.se	Pub
Akos Princzinger	Ericsson	Budapest, Hungary	qprakos_at_cbe.ericsson.se	Pub
Arthur Quinn	Scottish Enterprise Network	Glasgow, Scotland		
Claus Reinke	University of Kent	Canterbury, England	C.Reinke_at_kent.ac.uk	Pub
Tony Rogvall	Synapse	Stockholm, Sweden	tony_at_rogvall.com	Pub
Sebastian Strollo	Nortel	Stockholm, Sweden	seb_at_bluetail.com	Pub
Per Einar Strömme		Stockholm, Sweden	stromme_at_telia.com	Pub
Biro Szabolcs	Ericsson	Budapest, Hungary	Szabolcs.Biro_at_eth.ericsson.se	Pub
Simon Thompson	University of Kent	Canterbury, England	S.J.Thompson_at_kent.ac.uk	
Zoltan Peter Toth	Ericsson	Budapest, Hungary	Zoltan.Toth_at_eth.ericsson.se	
Laszlo Vadkerti	Ericsson	Budapest, Hungary	laszlo.vadkerti_at_ericsson.com	
Jane Walerud		Stockholm, Sweden	jane_at_walerud.com	Pub
Claes Wikström	Nortel	Stockholm, Sweden	klacke_at_bluetail.com	Pub
Chris Williams	Ericsson	Stockholm, Sweden	chris.williams_at_ericsson.com	Pub

3

Updated 2003-11-12	**To send a mail replace the "_at_" by an "@"**

4

Aquarelle White DFS, 12 mm for 91-120 sheets
www.bindomatic.com