Delivering Optimized Portable SBC Software

John Hogg
CTO
Zeligsoft
Embedded Systems Industry Facing Software Crisis

Key Contributing Factors:
2. Ever increasing set of complex features
3. Powerful, but complex multiprocessor platforms and SoC
4. Exponential growth of software complexity
5. Ever increasing pace of new product introduction

<table>
<thead>
<tr>
<th></th>
<th>1996</th>
<th>2006</th>
<th>2008</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Product</strong></td>
<td>2G</td>
<td>3G</td>
<td>4G, WiMAX</td>
</tr>
<tr>
<td><strong>Features</strong></td>
<td>basic telephony, no data</td>
<td>advanced telephony, multiple standards, multiple data services</td>
<td>Converged standards, interactive services</td>
</tr>
<tr>
<td><strong>Platform lifecycle</strong></td>
<td>8 to 10 years</td>
<td>5 years</td>
<td>3 years</td>
</tr>
<tr>
<td><strong>Development time</strong></td>
<td>4 years</td>
<td>3 years</td>
<td>2 years</td>
</tr>
</tbody>
</table>

The problem cannot be solved by adding more people.....
A new way of developing embedded software is mandatory
Component-Based Development Conundrum

Component-based development delivers:
- Encapsulation
- Reuse

Perceived benefits:
- Decreased resource requirements
- Increased quality
- Decreased time-to-market

Perceived issue:
- Performance

Key Problem
- How can we maintain the benefits of components without paying a performance cost?
Agenda

- Component architecture
- Traditional development lifecycle
- Deployment-Aware Generation™ (DAG™)
- Summary
Component Architecture

- Component-based behavior: how the component interacts
- Functional behavior: what the component does
- The two typically come from different sources and must be merged
Component APIs

- Component outer API: interface of component-based to external system
  - Requires knowledge of communication system

- Component internal API: interface of functional code to packaging
  - Requires knowledge of messages
  - Both APIs provide opportunities for integrating generated code
Component Architectures

- “Who speaks to whom”
- Relationships between elements
  - Components in application
- Devices in platform
Hand-Optimized Non-Component Lifecycle

- Hand-optimized lifecycle without components:
  - Deploy before generation, implementation and building
  - Enables system optimization
  - Severely limits component portability
Traditional Component-Based Lifecycle

- Traditional development lifecycle with components:
  - Deploy after generation, implementation and building
  - Enables component portability
  - Limits component optimization
Optimizing in the Traditional Lifecycle

- The contract of the generated code is the contract of the model
  - Implicitly code-centric approach
  - (Equivalently, binary-centric approach)
- Optimization strategies are based on middleware
  - Generate generic invocation code for communication (or other aspects)
  - Optimize in the implementation of the middleware
- Can’t optimize to same level as hand-coding
Reorganizing the Lifecycle

- Optimal lifecycle:
  - Deploy before generation, implementation and building
  - Enables component portability
  - Enables component optimization
  - Deployment-Aware Generation™ (DAG)
Deployment-Aware Optimization

- DAG-generated code depends not just on the target (platform-aware generation) but on the targets of related components
- The contract of the generated code is *separated from* the contract of the model
  - Model elements are reusable in other contexts
- Generated code is context-specific
  - Based on clear division between component-based and functional code
- Optimization strategies are based on using right middleware
  - Same approach used to optimize hand-coded applications
Traditional Generation

- Consider component’s deployment only
  - Packaging code matches constant external and internal API
  - Generated code is independent of other components’ deployments
DAG Generation

- Consider other components’ deployments
  - Packaging code matches constant internal and context-dependent external API
  - Generated code depends on other components’ deployments
DAG Generation

- Consider other components’ deployments
  - Packaging code matches constant internal and context-dependent external API
  - Generated code depends on other components’ deployments
Deployment Aware Generated Code

void seeBase_Antenna_IFConfigurator::config();
{
    // Component is deployed on device Main
    // Setting up a user connection between me (BA_IF_21) and BC_11 through port "BaseIF_Port"
    // The two components are not collocated, therefore setting up a TIPC connection
    foo2InterfaceTIPC *conn = new foo2InterfaceTIPC(16686, 17, 16688, 17);
    servant_out_BaseIF_Port->connectTo(conn);
}

void seeBase_Antenna_IFConfigurator::config();
{
    // Component is deployed on device Main
    // Setting up a user connection between me (BA_IF_21) and BC_11 through port "BaseIF_Port"
    // The two components are collocated, therefore setting up a Shared memory connection
    foo2InterfaceSharedMem *conn = new foo2InterfaceSharedMem(156, 123);
    conn->Init(&Client);
    servant_out_BaseIF_Port->connectTo(conn);
}
Applying DAG to Other Aspects

- Communication and middleware are natural applications of DAG
- Other aspects can also be optimized:
  - Services
    - Timing
    - Log
    - Accelerators
    - ...
  - Encryption
  - ...

Heterogeneous Environment Support

- Embedded platforms are typically heterogeneous
- Functionality on GPPs, DSPs, FPGAs, ASICs
- Represent software in a uniform way as communicating components and devices
- Specify deployment
- Create deployment-aware connections

```cpp
void sceBase_antea_IFConfigurator::config()
{
    // Component is deployed on device main
    // Setting up a user connection between (SA_IP_21) and SC_11 through port "BaseIF_Port"
    // The two components are not collocated, therefore setting up a TIPC connection
    void *conn = new TIPCConn("TIPC:16888,17,18888,17");
    servant_out_baseIF_port->connectTo(conn);
}
```
Summary

- Hand-coding maximizes optimization, but limits portability
- Traditional component-based SBC development maximizes portability, but limits optimization
- Deployment-Aware Generation maximizes portability and optimization
  - A precise platform model is the cornerstone of DAG
- DAG generates optimized components
Thank You

www.zeligsoft.com    +1 819 684 9639
hogg@zeligsoft.com