Estimation of Static Memory Consumption for Systems Built from Source Code Components

E.M. Eskenazi, A.V. Fioukov, D.K. Hammer, M.R.V. Chaudron

Department of Mathematics and Computing Science, Eindhoven University of Technology,
Postbox 513, 5600 MB Eindhoven, The Netherlands
+31 (0)40 – 247 4449
{ e.m.eskenazi, a.v.fioukov, d.k.hammer, m.r.v.chaudron }@tue.nl

Abstract
The quantitative evaluation of certain quality attributes – memory consumption, timeliness, and performance – is important for component-based embedded systems. We propose an approach for the estimation of static memory consumption of software components. The approach deploys the Koala component model, used for embedded software in TV sets. There are two main parts in the method: specification of the memory demand of components and estimation of memory demand for systems built of these components. The proposed method allows flexible trade-off between estimation effort and achievable precision, yet requiring no changes in the tools supporting the Koala component model. The method may be extensible to include other resource attributes as well.

1. Introduction
Nowadays, component-based engineering [4], [6] actively enters the area of product families for resource-constrained embedded systems. For example, Philips Electronics is deploying a proprietary component model, called Koala [5].

The Koala component model focuses on the following points: (1) diversity handling to build different products from existing components for supporting the development of product families and (2) efficiency in resource-constrained systems, since it is applied to the high-volume electronics domain where product costs are the driving factors.

Koala does not yet provide any explicit mechanism for the quantitative assessment of the memory demand for code and static data. The most challenging issue here is to predict the memory demand for component compositions, given the demands of the constituents. Since our goal is the support of the early product creation phases (feasibility study, architecting etc.), we choose for static evaluation techniques. As we aim at reasoning about quality attributes of Koala components, we concentrate on source code components.

2. Problem analysis
This section describes the objectives of the approach, complications to be tackled, and assumptions made.

2.1 Requirements
We formulated the following requirements for the approach for memory consumption estimation:
1. The approach should be compositional. This means that the memory consumption of the component composition should be expressed in terms of the memory demands of the constituents1.
2. The approach should be tunable with respect to the estimation accuracy. There is a trade-off between the estimation effort and the accuracy.
3. The approach should support budgeting. It should be possible to take into account the estimates of memory demands for the components that are not developed yet.

2.2 Complications
There are some Koala language-specific features [5] influencing the memory consumption of a component:
1. Diversity and optional interfaces. Diversity interfaces are used to tune reusable components for specific needs of product family members. The components can be configured with diversity parameters via diversity interfaces. Optional interfaces contain one Boolean diversity parameter determining whether interface is present in a particular product or not.
2. Function binding. It is possible to substitute a function of an interface with an expression (a piece of code) specified in the component description file.

1 In principle, this requirement should hold for any quality attribute. For more detail, the reader is referred to [1], [2], [3].
Consequently, some C-language expressions are injected into the component code, and prediction of the code size becomes dependent on that.

3. **Interface binding.** During the build process, the Koala compiler monitors the use of the provides interfaces of the components. The component is not reachable and excluded from building if none of its provides interfaces are connected.

There are additional complicating factors, such as:

1. C Compiler optimizations. Modern compilers can optimize object code to decrease the amount of the required memory. The results of these optimizations may be very context dependent and are consequently hard to predict.
2. Platform dependency. The necessary amount of memory for a component depends on target hardware platform due to bus width and data alignment.
3. Mapping of memory regions. After compiling, the object code can be allocated to different types of memory, e.g., internal ROM (IROM), external ROM (XROM), external RAM (XRAM), etc. The allocation is defined by a locator configuration file. As the Koala compiler cannot access this file, it is more difficult to account for the contribution to a particular memory region. Modification of this file would require complete recalculation of memory consumption for different regions.

2.3 **Assumptions**

The following assumptions were made:

1. Function binding is ignored.
2. Compiler options are not changed, i.e., compiler optimizations are considered to be fixed.
3. Platform dependencies are not accounted for.
4. The distribution of memory types according to the locator configuration is fixed.

3. **Memory consumption model**

This section introduces mathematical basis for the approach and describes its implementation within the Koala framework.

3.1 **Analytical expression**

In general, the size of component code and static data can be calculated by the following formula:

\[
\text{size}(c,E) = \sum_{i \in \text{mod}(c)} \text{size}(i,F_i(E)) + \sum_{i \in \text{mod}(c) \cap \text{reachable}(c,E)} \text{size}(i,F_i(E)) + \sum_{i \in \text{mod}(c) \cap \text{non}(c,E)} \text{size}(i,F_i(E)), \tag{3.1}
\]

where \( E \) is a set of interfaces\(^2 \) of component \( c \); \( E \) is a set of interfaces of sub-component \( i \); \( F_i : E \otimes E_i \) is the function that specifies how the interfaces of sub-component \( i \) are bound within component \( c \) (this also includes mapping of the diversity parameters of component \( c \) onto the ones of sub-component \( i \)); \( \text{sub}(c) \) is the set of all the sub-components of \( c \); \( \text{reachable}(m,E) \) is the set of all the run-time switches\(^3 \) of a component \( c \); \( \text{size}(x,E) \) is the function that calculates the size of a (sub)-component \( x \), module \( x \), or run-time switch \( x \), taking into account the interfaces \( E \); \( i \) denotes a sub-component \( i \) of component \( c \), and \( m \) denotes a module \( m \) of the component \( c \).

Note that formula (3.1) holds both for code and static data size.

3.2 **Specification**

This section describes a method for the specification of memory demand for a component.

We introduce an auxiliary provides interface \( IResource \) specifying memory consumption of a component (Figure 1). The members of this interface correspond to particular types of memory. Each component is attached with a formula for the estimating the memory size of each type. This formula employs constants, expressions related to Koala features (e.g., diversity parameters), and arithmetic operations.

\[
\begin{align*}
\text{div}(\text{XROM}) & \text{ sizes } \text{IDRAM}\text{ sizes } \text{XRAM}\text{ sizes } \text{IROMCODE}\text{ sizes } \text{XROMDATA}\text{ sizes } \\
\text{div} & = (3.1) + (8+((\text{MaxTasks}+1)/2)*2)*(\text{MaxTasks} + 2) \\
\end{align*}
\]

\[
\text{div} = (3.1) + (8+((\text{MaxTasks}+1)/2)*2)*(\text{MaxTasks} + 2)
\]

**Figure 1. Example of "IResource" interface.**

\(^2\) \( E \) denotes the set of diversity, optional and provides interfaces. Note that actual dependence on \( E \) may involve only a subset of \( E \), e.g., only component’s provides interfaces.

\(^3\) A module is a code block implementing interface functions [5].

\(^4\) A run-time switch occurs whenever a non-constant expression controls the switch. For more detail, the reader is referred to [5].
The formula for calculation of the sizes is an expression over diversity parameters, optional interface connections, and sizes (similar formulas) of the sub-components. It also can contain some constants for denoting the sizes of the inner modules.

The component “ClxCmx” (Figure 1) includes the sub-component “CMgCmx” and the module “m”. The specification of external RAM (XRAM) size consists of the following parts.

1. Contribution of the module “m”:

\[ 34 + (8 + (\text{res.MaxTalos}+1)/2)*2*(\text{res.MaxTasks}+2) \]

This formula contains a constant part and a variable part which depends on the diversity parameters MaxTalos and MaxTasks.

2. Contribution of the sub-component “CMgCmx”:

\[ \text{mgcmx.req.iPresent()} \ ? \ \text{mgcmx.req.XRAM\_size} \ : \ 0 \]

The expression \( \text{mgcmx.req.iPresent()} \) indicates whether any module of “CMgCmx” is reachable. The module is reachable if the \( \text{provides} \) interface implemented with this module is needed for any other component. If \( \text{mgcmx.req.iPresent()} \) is true, then the size of “CMgCmx” is added to the size of “ClxCmx”. For the component “CMgCmx” the similar interface “req” is specified, and \( \text{mgcmx.req.XRAM\_size} \) provides the size of “CMgCmx” to account for in the formula for “ClxCmx”.

When using this specification technique, the memory consumption estimates for component compositions can be calculated automatically by the Koala compiler.

### 4. Memory consumption estimation

This section describes two approaches for memory consumption estimation, based on the specification technique from the previous section. Both approaches are illustrated with the experimental results.

#### 4.1 Two possible approaches

Three types of components can be distinguished in the Koala model [5]: (1) basic components that do not contain other components, (2) compound components that may contain other components, forming a hierarchy (see Figure 2), and (3) configurations that are top-level components without any \( \text{provides} \) and \( \text{requires} \) interfaces. The configuration is a set of components assembled together to form a product.

For estimation of the component size, two approaches were considered: (1) an \textit{exhaustive} bottom-up approach and (2) a \textit{selective} top-down one. These two approaches trade estimation accuracy against estimation effort.

In the \textit{exhaustive} approach, all diversity and optional \textit{requires} interfaces of all compound and all basic components are taken into account (see Figure 2). The component hierarchy is traversed in a bottom-up way, starting from the basic components up to ones at the defined level of the hierarchy. The formula is constructed for each component, until a formula for the entire configuration is determined.

The \textit{selective} approach deals only with diversity and optional interfaces of the compound components located at some fixed level of the composition hierarchy (see Figure 2, e.g. only components C1, C2, and C3). If it is impossible to obtain a sufficiently accurate formula at this level, then also the sub-components need analyzing and constructing formulas for them. The considered degree of nesting should be as deep as necessary for achieving a sufficiently accurate formula.

The formula for the top-level component is a sum of the formulas of its constituents. To define formulas depending on the diversity parameters and optional interfaces, the investigated component is wrapped with an auxiliary configuration. The formulas can be built in an empiric stepwise way: their extrapolations are obtained by sequential compiling of the wrapper with various values of diversity parameters and different sets of connected optional interfaces. All relevant components contained in the top-level one also need wrapping to construct their own formulas. Building of the formulas can be facilitated by code observation (e.g. when a diversity parameter defines the size of an array).

Note that both approaches support budgeting, i.e. the expected memory demands of non-existing components can be involved into a formula.

![Figure 2. Approaches at different levels of component hierarchy.](image)

The main differences between these approaches are the estimation accuracy and annotation effort.

The exhaustive approach ensures the required level of accuracy for the entire composition if all components are annotated with sufficiently accurate formulas (in the general case). However, this implies huge amount of effort.

The selective approach may not ensure the defined level of accuracy. Achieving the appropriate level of accuracy may require analysis of deeper levels of the component hierarchy, while considering only selected components may reduce the amount of effort needed for annotation.

---

5 For the exhaustive approach, all components are relevant.
4.2 Estimation examples

To demonstrate both memory estimation approaches, we applied them to two different component configurations taken from the existing software stack for TV sets. The first configuration consisted of seven components was used for checking the exhaustive approach, while the second one consisted of 22 components was used for checking the selective approach.

The software stack for the case study was implemented for a 16-bit derivative of the popular Intel 8051 microcontroller. This microcontroller differentiates several types of memory. For each type of memory, the estimates were compared with the actual sizes, considering different sets of diversity parameters and connections of different optional requires interfaces (see Table 1 and Table 2).

Table 1. Estimates and actual sizes for exhaustive approach.

<table>
<thead>
<tr>
<th>Type of memory</th>
<th>Real size (bytes)</th>
<th>Estimated size (bytes)</th>
<th>Relative error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XROM Data</td>
<td>166</td>
<td>166</td>
<td>0,00</td>
</tr>
<tr>
<td>XROM Code</td>
<td>19429</td>
<td>19477</td>
<td>0,25</td>
</tr>
<tr>
<td>IROM Code</td>
<td>3363</td>
<td>3425</td>
<td>1,80</td>
</tr>
<tr>
<td>IROM Data</td>
<td>379</td>
<td>379</td>
<td>0,00</td>
</tr>
<tr>
<td>IDRAM</td>
<td>572</td>
<td>572</td>
<td>0,00</td>
</tr>
<tr>
<td>SRAM</td>
<td>145</td>
<td>145</td>
<td>0,00</td>
</tr>
<tr>
<td>XRAM</td>
<td>2123</td>
<td>2123</td>
<td>0,00</td>
</tr>
</tbody>
</table>

Table 2. Estimates and actual sizes for selective approach.

<table>
<thead>
<tr>
<th>Type of memory</th>
<th>Real size (bytes)</th>
<th>Estimated size (bytes)</th>
<th>Relative error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XROM Data</td>
<td>12379</td>
<td>12479</td>
<td>0,81</td>
</tr>
<tr>
<td>XROM Code</td>
<td>70996</td>
<td>71409</td>
<td>0,58</td>
</tr>
<tr>
<td>IROM Code</td>
<td>21353</td>
<td>21401</td>
<td>0,20</td>
</tr>
<tr>
<td>IROM Data</td>
<td>1705</td>
<td>1703</td>
<td>0,12</td>
</tr>
<tr>
<td>IDRAM</td>
<td>796</td>
<td>796</td>
<td>0,00</td>
</tr>
<tr>
<td>SRAM</td>
<td>544</td>
<td>544</td>
<td>0,00</td>
</tr>
<tr>
<td>XRAM</td>
<td>84471</td>
<td>84607</td>
<td>0,15</td>
</tr>
</tbody>
</table>

5. Conclusions

We have proposed a method that allows estimating the memory consumption for Koala component compositions. The proposed method is illustrated with examples taken from the existing software stack.

We have described the mechanism for specification of the component memory demand for code and static data. This mechanism employs standard constructions of the Koala component definition language.

The suggested specification mechanism is compositional and hierarchical: the memory demands of a compound component are specified in terms of memory requirements of its constituents, and each component can be used in another context without changing the specification of its memory consumption. When using this technique, the memory consumption estimates for component compositions can be calculated automatically by the Koala compiler.

This mechanism also supports budgeting; i.e. the expected sizes of the components being developed can be incorporated into the specification.

Two approaches for the estimation were proposed: exhaustive and selective. Each approach was validated with a case study. High estimation accuracy can be achieved for both approaches.

Further research will be directed towards additional validation and generalization of the proposed technique. Firstly, the thorough validation of the approach with more experiments will be performed. Secondly, the possibility to generalize and apply this approach to other component models will be considered. Finally, the different ways to specify the memory consumption and other resource attributes (particularly, in XML-based description language) will be investigated.

6. Acknowledgements

We are grateful to Rob van Ommering and Christiene Aarts for a valuable contribution in this work. We also thank Marc Stroucken for constructive suggestions and critical feedback.

7. References