Quality-based Software-Selection and Hardware-Mapping as Model Transformation Problem

Sebastian Götz, Johannes Mey, Rene Schöne and Uwe Aßmann
{first.last}@tu-dresden.de, sebastian.goetz@acm.org

Software Technology Group
Technische Universität Dresden

Abstract

In this TTC case, we describe the computation of an optimal mapping from software implementations to hardware components for a given set of user requests as a model transformation problem. Further, contracts specify dependencies between components in terms of non-functional properties. Different approaches of this case can be compared in terms of their validity, performance, scalability and, quality w.r.t. the real optimal deployment.

1 Introduction

One of the goals of software engineering is to enable the development of efficient software, i.e., software which provides the best possible utility for the least possible cost. By now, this goal has been addressed at several levels of abstraction: from compiler construction [Kil73] in the 70s to cloud computing [BAB12].

To optimize the tradeoff between utility and cost, at least two interrelated standard problems are typically addressed:

- Resource allocation (aka. register allocation in compilers), i.e., the mapping of software implementations to hardware components leading to the least cost
- Variant selection (aka. instruction selection in compilers), i.e., selecting the software implementation, which provides the best utility

These problems have been investigated for compilers since almost 50 years, where they are called code generator optimizations. However, both problems are strongly intertwined. Together, they form an NP-complete problem with exponential cost, but typically are carried out in phases [WG12]. Considering both problems to be independent from each other allows to use efficient optimization approaches, running in polynomial time [HG06]. But, an immanent problem of solving variant selection and resource allocation in isolation is that invalid solutions might result. Hence, still new approaches to compute solutions for the joint selection and mapping problem, are required.

This case provides a generic metamodel for the combined problem of resource allocation and variant selection. Both problems are interrelated by user requests specifying minimum requirements on the non-functional properties provided (i.e., minimum utility), whilst searching for a selection and mapping both maximizing utility and minimizing cost. To show progress over state of the art, these approaches need to be evaluated w.r.t. their correctness, performance, solution quality and scalability. Correctness denotes that only solutions not violating the minimum requirements of the users are considered valid. The performance of an approach describes how fast
a solution can be computed for a given problem. The solution quality quantifies how close the computed solution is to the optimal solution. Finally, scalability is represented by the size of the largest problem, for which a valid solution can still be computed.

We describe a reference implementation using a reference attribute grammar as metamodel [BKWA11] to translate the above described selection and mapping problem into an integer linear program, whose solution is then translated into a deployment description.

2 Case Description

In the following, we first provide a detailed description of the case by specifying the metamodel shared by both, source and target models, thus resulting in an endogenous transformation. Afterwards, in Section 2.6, we give an overview of our reference implementation.

As mentioned in the previous section, the problem to be solved is selecting variants of software components and a mapping of those to suitable hardware resources based on user requests. To adequately cover this problem, we designed a combined metamodel comprised of four views: hardware, software, expressions, and their combination.

2.1 Hardware metamodel

Figure 1a depicts the hardware view, specifying a HardwareModel, which comprises hierarchically structured ResourceType and Resources as instances of these types, following the type-object pattern. Thus, the hardware model composes static knowledge about resources (types) and runtime knowledge (instances). Certain resource types are able to run software, i.e., are a valid mapping targets for software implementations. To mark such types, the attribute container is used.

Moreover, resource types are further characterized by a set of properties. Resources then specify concrete values for these properties. As an example, one could define the resource type CPU with a Property frequency having the unit Mhz and mark it as a container. An instance of this type would be CPU0 with a current frequency of 200. A full example is shown in Figure 4a.

2.2 Expression metamodel

Figure 1b depicts the expressions metamodel, which allows the specification of simple expressions. The hardware view uses LiteralExpressions to specify the current value of a resource property. BinaryExpressions are required by the software view to describe formulas for required and provided properties. Designators are variables used in expressions referring either to a meta-parameter, or to a property of an instance. The latter can target both resource and software component instances. An example of such a formula can be seen in Figure 4b at line 16 and is explained later in detail.

2.3 Software metamodel

Figure 2 shows the SoftwareModel with its main element Component representing a certain functionality (e.g., sorting). Implementations provide this functionality, requiring other components and/or resources to
fulfill their work. For example, a software component sort could have the implementations Radix sort and Counting sort, where the component specifies that it requires a ReadFile and a WriteFile component to read the unordered list and write the ordered list. The in- and output components have different implementations, too, as sketched in Figure 4b. A ComponentRequirement states that the referenced component is called during the execution of this implementation. Requirements for components and resources are specified on type level (i.e., refer to components and resource types). Their relation to the runtime model is expressed by referencing a set of Instances, which are either Resources or instantiated Implementations. Implementations have a contract specified as a number of Clauses describing provisions to and requirements on required instances. A clause comprises a left hand side (LHS), which is a designator, a comparator (one of $=, \neq, \leq, <, >, \geq$), a type (provision or requirement), and a right hand side (RHS), which is an expression. For example, the predicted minimum runtime of an implementation can be expressed as a provision clause coupled with a requirement clause, which specifies a certain CPU frequency for which the provision holds. An example contract is shown in Figure 4b. In this example contract, requirements on components as well as resource types are declared (cf. lines 8-11). Later, in line 16, a provision clause is defined with energy as the SoftwareDesignator for the left hand side, EQ as ClauseComparator and an AddExpression as right hand side.

### 2.4 Combination of metamodels

Figure 3a depicts the complete metamodel showing how the three former parts in Figures 1a, 1b and 2 are connected. Its main concepts are the objective function, user requests, a solution to the problem (show in Figure 3b), and all used enumerations.

The Objective specifies how to compute the objective value of a solution, i.e., for which value(s) the problem should be optimized. Therefore, a property to optimize for, and an objective function defining how to aggregate all values of this property are selected. Currently, the aggregation function can either be sum or maximum.

A Request represents a requirement of a user, who specifies what algorithm with which parameters and requirements should be executed. Requests contain their functional requirement by referencing a target software component, constraints on non-functional requirements using clauses, and an abstract characterization of the input (i.e., parameters) using MetaParameters. A meta-parameter describes possible values of the actual parameter. As an example, when passing a list of items to a sort component, a possible meta-parameter for this list is its length.

### 2.5 Solution metamodel

The class Solution is to be computed by the solvers. It contains a number of Assignment, each selecting one implementation of the given problem model and assigning every instance in the contract of the implementation to either another assignment or a resource. With this, a concrete implementation is selected for each required component along with their required resources. The value of topLevel is true if the assignment corresponds to a requested component. An example solution is shown in Figure 6.

A solution is valid, if for each request a) an implementation for the target component is deployed, b) an implementation is deployed for each required component, c) all required (non-functional) property clauses are fulfilled (including the request constraints), and d) at most one implementation is deployed on each resource. A solution is optimal, if it is valid and no other solution has a better objective value.
To preserve readability, the following inheritance relation was not included in the Figures 1a, 1b, 2 and 3a: All of Instance, ResourceType, Resource, Component, Implementation, Property, MetaParameter inherit from the class ModelElement, which defines a single attribute “Name” of type String, to uniquely identify those model elements, e.g., while parsing the input model.

2.6 Reference Implementation using ILP

This section describes the reference implementation for the presented problem based on an attribute grammar-based model-to-text transformation and integer linear programming (ILP). The approach is a variant of Multi-Quality Auto-Tuning [GKP+14, SGAB16] which directly uses the metamodel presented in the previous subsection.

The general approach of our reference implementation to the case is depicted in Figure 5. We first transform the problem model into an integer linear program, whose solution is computed using a standard solver. Then, we interpret the solution of the solver, and reconstruct a Solution object as described in Figure 3b.

However, there are restrictions on the complexity of the model: Only ≤, ≥, = are permitted as expression clause comparators, resource types can only be nested to a depth of one, and every component may not be required more than once per request. Models generated for this case as described in Section 3.1 adhere to those restrictions.

The textual representation of the selection and mapping problem as an ILP comprises an objective function, a set of linear constraints and a set of variables used both in the objective function and the constraints [Man87]. Each combination of a request, resource and implementation is represented by one binary variable in the ILP stating whether or not this implementation is deployed on that resource for the respective request.

The objective function is represented as minimization of either a weighted sum or the maximum of all variables, depending on Objective in the current model. The weights for each variable are the effect the decision represented by the respective variable has on a selected non-functional property. If, for example, the objective is to optimize the performance, the weights represent the effect on the total runtime.

Furthermore, the ILP comprises three types of constraints:

**Architectural constraints** ensure that each request is fulfilled, exactly one implementation per component is
// type and property definitions
container resource type ComputeNode {
  resource type CPU {
    property frequency [Hz]
    property load [%]
  }
  resource type RAM {
    property totalRam [MB]
    property freeRam [MB]
  }
  resource type DISK {
    property totalDisk [MB]
    property freeDisk [MB]
  }
  resource type NETWORK {
    property latency [ms]
    property throughput [kB/s]
  }
}

// resource definitions
resource resource0:ComputeNode {
  resource cpu0_0:CPU { frequency = 200 load = 0 }
  resource ram0:RAM { totalRam = 885 freeRam = 885 }
  resource disk0:DISK {
    totalDisk = 15472 freeDisk = 15472
  }
  resource network0:NETWORK {
    latency = 14 throughput = 23466
  }
}

// meta parameter and properties
meta size
property energy [J]

// components and implementations
component Sort {
  using property quality
  implementation CountingSort {
    requires resource compute_resource_0 of type ComputeNode
    requires resource cpu_1 of type CPU
    requiring input.quality ≥ 95
    requiring output.quality ≥ 55
    requiring cpu_1.frequency ≥ 2159
    providing quality = 300
    providing energy = ((0.59 * (size ^ 2)) + cpu_1.frequency)
  }
  implementation RadixSort { /* ... */ }
}

component ReadFile {
  using property quality
  implementation BufferedInputStream { /* ... */ }
}

// requests and objective
request request1 for Sort {
  meta size = 873.0
  requiring quality ≥ 95.0
}
minimize sum(energy)

(a) Hardware model of the scenario generator.
(b) Example of two implementation contracts.

Figure 4: Model specification using our concrete syntax.

Request constraints ensure that components for each request are chosen such that the requested non-functional properties are provided.

Negotiation constraints guarantee that non-functional requirements are met by depending implementations

The reference implementation uses non-terminal attributes [VSK89] to allow caching of the computed representation of the ILP containing the objective function, used variables and all constraints. Already at the time of generation of those constraints, only valid deployments of configuration on hardware resources w.r.t. contracts of implementations are considered. Variables of invalid combinations are set to be ignored when solving.

We used GLPK ¹ to solve the resulting ILP. Calling GLPK was done in two different ways: The first way comprises pretty-printing the subtree to a file, calling the solver via command line, and parsing the result from the created solution file. As a second way, we used the Java-Binding of GLPK² to programmatically create constraints, solve the problem and read the solution.

In both cases, the solution can be easily interpreted by iterating over all variables having value one. By this, we can recursively reconstruct the assignments for each request.

The reference implementation together with a benchmark framework is publicly available online³.

3 Task Description

The general task is to solve the interleaved problems of software variant selection and hardware mapping. To ease implementing an approach to solve this problem, we provide a problem generator (cf. Section 3.1), a concrete set of example problems to be solved (cf. Section 3.2) and a benchmark environment (cf. Section 3.3). The underlying metamodel was built using JastAdd [EH07], a tool to define Reference Attribute Grammars. Once the definitions are parsed and built, the resulting Java classes can be used normally without any dependencies. Model elements can be manipulated using Java methods, however it is possible that caches of results from attribute evaluation are updated automatically upon changes.

¹https://www.gnu.org/software/glpk/
²http://glpk-java.sourceforge.net/
³https://git-st.inf.tu-dresden.de/stgroup/ttc18
3.1 Problem Generator

For the presented case, we developed a generator that is able to create random models of arbitrary size. However, the software and hardware component structures are fixed to ensure comparability and some level of realism. The resource types defined in the hardware model are designed to resemble regular computer hardware. A compute node consists of one or more CPUs, RAM memory, disk, and a networking interface. Each of these components has properties that specify their capabilities. Figure 4a shows the resource types used in the case and their respective properties. The number of resources generated can be specified as a parameter represented as the ratio $r$ of the minimally required resources.

The software model has a simple tree structure. That is, all requests have a single meta-parameter ($size$), refer to the same software component and each software component may require other components.

While the generator is able to create sparse trees, for simplicity and comparability reasons, we only generate dense trees with a fixed branching factor of two, such that all possible solutions fulfilling a request require the same number of software components. For all software components, one property, $quality$, is specified. Requirements and provisions of this property are randomly generated for each implementation.

Finally, for each software component the requirement of exactly one container is generated. Clauses in every implementation use the meta-parameter size as part of a randomly generated formula (i.e., expression). For approaches not directly using Java, our benchmark framework offers the ability to implement visitor methods, which are called by the problem generation and, thus, can be used to create another problem model representation. Alternatively, approaches can reuse the existing concrete syntax to write out the model.

3.2 Scaling the Problem

To test the scalability of an approach, we generate a set of different problems. The difficulty of a problem is scaled using a set of parameters as shown in Table 1. Naturally, the number of software variants (per software component) and the number of requests have a lower impact on the problem complexity than depths of the component tree and the number of available resources. Since the number of problems that can be generated within the given parameter ranges is large, we propose a set of example problems, which are listed in Table 2.

In addition to the four above mentioned parameters, the generator takes a seed as a fifth parameter to ensure that for the same seed and same values for the other four parameters always the exact same problem is generated.

3.3 The Benchmark Environment

To simplify benchmarking, an easily adoptable process has been defined. Our framework can be used to run the scenarios presented in Section 3.2. If the framework is used, results and timing data are automatically taken and
Table 2: Parameters of the benchmark generator

<table>
<thead>
<tr>
<th>ID</th>
<th>Variants</th>
<th>Requests</th>
<th>Tree Depth</th>
<th>Resources</th>
<th>Impl’s</th>
<th>Compute Resources</th>
<th>Scenario Class</th>
<th>Scenario Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>minimal</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1.5</td>
<td>6</td>
<td>5</td>
<td>small</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5.0</td>
<td>6</td>
<td>15</td>
<td>small</td>
<td>much hardware</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>1.5</td>
<td>62</td>
<td>47</td>
<td>small</td>
<td>complex software</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>15</td>
<td>2</td>
<td>1.5</td>
<td>30</td>
<td>68</td>
<td>medium</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>15</td>
<td>2</td>
<td>5.0</td>
<td>30</td>
<td>225</td>
<td>medium</td>
<td>much hardware</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>1.5</td>
<td>155</td>
<td>465</td>
<td>medium</td>
<td>complex software</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>20</td>
<td>2</td>
<td>1.5</td>
<td>60</td>
<td>90</td>
<td>large</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
<td>20</td>
<td>2</td>
<td>5.0</td>
<td>60</td>
<td>300</td>
<td>large</td>
<td>much hardware</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>20</td>
<td>5</td>
<td>1.5</td>
<td>310</td>
<td>930</td>
<td>large</td>
<td>complex software</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>50</td>
<td>2</td>
<td>1.5</td>
<td>150</td>
<td>225</td>
<td>huge</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>50</td>
<td>50</td>
<td>2</td>
<td>5.0</td>
<td>150</td>
<td>750</td>
<td>huge</td>
<td>much hardware</td>
</tr>
<tr>
<td>12</td>
<td>20</td>
<td>50</td>
<td>5</td>
<td>1.5</td>
<td>620</td>
<td>2325</td>
<td>huge</td>
<td>complex software</td>
</tr>
</tbody>
</table>

stored for comparison.

3.4 Task Description

The task is to solve the given set of scenarios listed in Table 2. Since the size of the scenarios varies, it might not be possible to find the optimal solution, so any valid solution suffices.

The solution is to be represented within the original problem model by instantiating the respective classes (i.e., Assignment, ComponentMapping, and HardwareMapping).

A possible textual representation of the solutions is shown in Figure 6. The depicted exemplary solution is to be interpreted as follows. For the request request0, the CountingSort implementation of Sort shall be used. This configuration is composed of one hardware mapping (lines 4-9) and two variant selections (lines 11-26). The hardware mapping specifies that the designator (i.e., the variable) compute resource0 is to be mapped to the physical resource named resource0. Moreover, the designators like ram 1 defined within the scope of compute resource0 are mapped to the respective physical subresources. The variant selections specify that for request_0 the BufferedInputStream implementation of ReadFile and the RandomAccessFile implementation of WriteFile shall be used. For both variant selections, the respective mapping to resources is described analogously to the example in lines 4-9.

Note, that the solution not only contains the information which implementation is assigned to which hardware resources, but also how the software instances from the contract are mapped; thus, the solution describes a tree structure.

In addition to the found solution, the approach is to be evaluated as described in the following section.

4 Evaluation Criteria

To evaluate approaches for the above described case, the following three criteria shall be used: performance, solution quality, and scalability. While solution quality is measured on an ordinal scale, performance and scalability are measured on a rational scale. Both solution quality and performance are considered separately for each scenario, while scalability is defined globally over all scenarios. To avoid differences between the measures taken for evaluation due to different hardware, we encourage the authors to evaluate their approach in the SHARE environment.

4.1 Solution Time (Performance)

To assess the performance of a presented approach, the time required to compute a solution for a given problem model shall be measured. Approaches translating the problem model to another technical space, must include

https://fmt.ewi.utwente.nl/redmine/projects/grabats/wiki
the times required to transform the problem and to interpret the solution. For this, approaches can write out the solution to disk and use the benchmark framework to check it against the initial model. As an example, for the reference implementation, the times needed to transform the model to an ILP representation, to solve the ILP and to interpret the ILP solution are summed up.

The value for this criterion is to be represented as a measured execution time. For each measured execution time, its median and standard deviation have to be shown.

4.2 Solution Quality

The most important criterion to be met by an approach to our case is the validity of the computed solution(s). For this, the computed solutions have to be checked against the constraints described in the problem model. However, the selection and mapping problem often has multiple valid solutions, i.e., only the constraints are fulfilled by the respective solutions. To compare different approaches, first the validity of the computed solution is checked. Failure in this check results in zero points. All valid solutions are then compared to each other w.r.t. their objective value. The approach that has found the best solution gets the most points, depending on how many approaches exist. Others will get fewer points according to their ranking of objective value.

4.3 Scalability

Quality-based Selection and Mapping is a combinatorial optimization problem and, hence, computing the best solution with an exact approach leads to a combinatorial explosion, i.e., does not scale. Approximate approaches can trade solution quality for performance and scalability, i.e., by not ensuring that the best solution is found, but maybe a near-optimal one, the solution can be found faster and larger problems can be solved at all.

To compare different approaches, their scalability shall be assessed in terms of the above described generator parameters. Using Table 1, the largest solvable scenario class can be determined. Approaches able to solve any problem of the class huge are awarded 4 points, and one point less per unsolvable class (large 3, medium 2 and small 1). Solving only the minimal scenario does not result in points for this category.

Acknowledgements

This work has been funded by the German Research Foundation within the Collaborative Research Center 912 Highly Adaptive Energy-Efficient Computing, within the Research Training Group Role-based Software Infrastructures for continuous-context-sensitive Systems (GRK 1907) and the research project “Rule-Based Invasive
Software Composition with Strategic Port-Graph Rewriting” (RISCOS) and by the German Federal Ministry of Education and Research within the project “OpenLicht”.

References


