A Case Study on Partial Evaluation in Embedded Software Design

Michael Jung, Ralf Laue, and Sorin Alexander Huss

Integrated Circuits and Systems Lab.
Department of Computer Science
Technische Universität Darmstadt
{mjung, laue, huss}@iss.tu-darmstadt.de

Abstract

Source code generators are often applied in embedded systems design to combine the flexibility necessary for reusability with the performance of highly specialized software. However, designing generators adds additional complexity to the software development process. Partial Evaluation promises to provide some means of automation in generator development. This paper reports on our experiences with the implementation of a generator for an embedded operating system, utilizing the partial evaluator C-Mix/II.

1 Introduction

The consistent reuse of already created, tested and possibly field proven software modules is one important approach to achieve a high level of development efficiency. The importance of this approach is indicated by the massive research effort spent in the field of software component technologies.

A software module has to exhibit a high degree of flexibility in order to be reusable in varying contexts. This is typically achieved by the application of variability mechanisms (e.g., parameters, inheritance, delegation) and integration techniques (e.g., observer pattern, connectors). All of these techniques, however, impose a certain overhead on run-time resources. The configuration of a software module, e.g., has to be stored somewhere and it usually has to be evaluated and applied at system startup. Both memory resources as well as CPU cycles are allocated in order to achieve configurability.

It is appropriate for software development in the server and workstation domains to accept this kind of overhead in order to achieve improved qualities. However, in the embedded systems world one often has to face special economical constraints. Embedded systems usually have clearly defined functional requirements and are produced in high volume. Thus, the fraction of costs related to software development is not very large, which results in the necessity to keep the per device costs, i.e. for hardware resources, as low as possible, while still complying to the functional requirements. The often cited statement that next year the costs for the same hardware will only be half as high does not give real relief here: The competitor’s products will also be available at half the price then.

These conditions have led to the fact that the current state of the art in embedded systems software development rarely incorporates reuse methodologies: Embedded software is in the majority of cases highly specialized for the application and hardware at hand, which hampers reuse in different contexts.

However, there is no denying it that successful software modules for the embedded market exist, the most prominent examples of which are embedded operating systems. It is characteristic that many of those software modules are not distributed exclusively as source code, but rather apply the concept of a generator. These generators are programs, which compute an optimized version of the software module based on a given system configuration. The system configuration consists of all the design parameters, which may vary between different systems, but which are constant for a given one.

Partial evaluation, first suggested as an approach in computer science in [2], is based on the idea that given a program $p$ and some of its input $d_1$ a specialized residual program $p_{res}$ can be computed that behaves identically to $p$ on the remaining input. Since all computations that solely depend on $d_1$ can be evaluated during the specialization process, the residual program can potentially be smaller as well as faster as the original [3].

This paper reports on our experiences utilizing the partial evaluator C-Mix/II [5] for the implementation
of an embedded operating system generator, which obeys the OSEK OS Specification [7].

2 Partial Evaluation

Partial evaluators usually follow a two step approach (see figure 1): First, for the software module that is to be specialized, the user has to specify which of its parameters shall be assumed to be static (i.e., the values, which will be bound to these parameters, will already be known at system-build time). Based on this information the generator generator performs the binding time analysis (BTA), which derives for each expression and statement in the code, whether it can be evaluated once the values of the static parameters are known. This information can directly be used to build the generating extension of the original program, which basically is a program that, given the values for the static parameters, computes in a second step the residual program. Note that the generating extension is in fact a generator, which is synthesized semi-automatically by the partial evaluator. Applying this approach for the implementation of generators has several advantages:

- Without the application of partial evaluation techniques, the designer has to decide for the generator to be implemented, which parameters shall be static (and therefore can not be changed during runtime) and which have to be dynamic. Should it be necessary to change this division when the project evolves or when the software component is reused in another project, the generator basically has to be rewritten. This is not the case with partial evaluation: Generating extensions for all \(2^p\) parameter divisions of a component with \(p\) parameters can automatically be synthesized.

- Testing of the software component is easier with the partial evaluation based approach: One can write test cases for the original source code and, under the assumption that the partial evaluator works correctly, be sure that the specialized code is as correct as the original. It will be hard to validate that a hand written generator will produce correct software components for each possible parameter set, which basically means that each generated source code has to be validated apart.

Partial evaluation was successfully applied to functional languages, but real world imperative languages like C, which are important in the embedded software world, seem to be a hard target. This is due to the incorporation of pointers in C, which results in a complex BTA (see [4],[8]). With C-Mix/II [5] a working partial evaluator for the complete ANSI C standard was implemented for the first time.

3 OSEK Operating System

The OSEK specification for embedded operating systems [7], developed as a joint effort by universities and automotive companies, shows that it is recognized by both industry and the scientific world that generator technology offers substantial advantages in the embedded software domain. OSEK applies the OSEK Implementation Language (OIL) as a configuration DSL (Domain Specific Language), which specifies features such as the number of tasks and resources, stack sizes, which tasks have access to which resources and events, extent of error condition checking and reporting, and
so on (see figure 2). The configuration, formulated in OIL, is then passed over to the system generator, which produces source code that is compiled and linked to the OSEK kernel and the actual application binaries. This approach allows parameters to be static, which usually have to be dynamic (e.g., the task ready queue). OSEK is therefore a very efficient and scalable real time OS: It runs even on 8 bit microcontrollers with just a few kByte of memory as target platforms.

4 Implementation

Our OSEK OS prototype implementation is targeted at a Motorola 68336 MCU evaluation board. The applied development tools are the GNU compiler “gcc” version 2.95.3 (in two configurations for m68k cross-compilation and for host machine compilation) and version 2.0.12 of the partial evaluator C-Mix/II.

As stated before, the OSEK OS generator reads system configurations written in a language called OIL. The OIL syntax is similar to those of markup languages like, e.g., XML. Instead of implementing a special parser, the OIL syntax is mapped to an XML document type definition and system configurations are expressed in XML files. The Libxml2 [11] library is applied to parse those files.

Apart from the necessary machine dependent assembler code for task switching and interrupt handling, all source code is written in C. All API code, which in a traditional implementation would be synthesized by the OSEK generator, meets the following structure in our implementation: In addition to the function parameters specified by the standard, all API functions have a parameter config, which holds the parsed tree of the XML'ed OIL configuration file. The first part of the function body extracts the relevant information from this tree (usually by calling helper functions), which is used in the core function, e.g., to control the degree of error condition checking and reporting.

When the source code is processed by C-Mix/II, the config parameter is assessed to be static. Based on this information, the partial evaluation process determines that the part, which extracts relevant information from the tree, can already be evaluated at run-time of the generating extension. Based on this information, the generating extension will only generate code, which is relevant given the static configuration.

Figure 3 gives an example of a small API code fraction. Function read_COM_xml called in line 4 extracts configuration information and puts it into global variables, like, e.g., COM_cmix_use_parameter_access. The code in lines 5–11 stores information for extended error reporting in global last_error variables, but only if this code is requested in the configuration. If the OS is run in an extended mode, then API function parameters are sanity checked, which is implemented in lines 12–27. There is furthermore the possibility to specify a hook function, which is optionally called in case of a detected error condition (see lines 18–24.) The API's unspectacular core functionality is implemented in lines 28–30. Given this code (plus the necessary support code)
and the information that the `config` function parameter is static, while the others are dynamic, C-Mix/II synthesizes a generating extension. Assuming that this generator is called with a configuration, which requests no error reporting and no hook function callback support, but sanity checks to be performed, the source code given in figure 4 is generated.

```c
TaskRefType *OSEK_ready_queues_head;
TaskRefType *OSEK_ready_queues_back;

void OSEK_init_ready_queue(xmlNodePtr config)
{
    unsigned int *priority_mapping, maxPriority;

    priority_mapping = create_priority_mapping(config);
    maxPriority = priority_mapping[max_priority(config)] + 1;
    OSEK_ready_queues_head = malloc(sizeof(TaskRefType)*maxPriority);
    OSEK_ready_queues_back = malloc(sizeof(TaskRefType)*maxPriority);
    for (i=0; i<maxNeededPriority; i++) {
        OSEK_ready_queues_head[i] = INVALID_TASK;
        OSEK_ready_queues_back[i] = INVALID_TASK;
    }
}
```

**Figure 5. Unspecialized Initialization Code**

```c
TaskRefType OSEK_ready_queues_head[3] =
{ INVALID_TASK, INVALID_TASK, INVALID_TASK };
TaskRefType OSEK_ready_queues_back[3] =
{ INVALID_TASK, INVALID_TASK, INVALID_TASK };
```

**Figure 6. Specialized Initialization Code**

Figure 5 gives another example. This time the code is no API implementation, but rather initialization code executed at system startup. In contrast to the previous example, in which the generator basically only selectively included source code fragments into the resulting output, some simple computations are performed at generator run-time here: In the OIL specification, each of a system’s tasks is assigned a priority. For an efficient implementation it is beneficial to map those to a contiguous range, which is done in `create_priority_mapping`. This function is completely evaluated during generator run-time. If the generator is called with a configuration that specifies three different task priorities, then the code given in figure 6 is generated\(^1\). Note that the dynamic memory allocation can completely be omitted in the final code.

\(^1\)The given code is edited for clarity, but semantically equivalent to the C-Mix/II generated code.

Basically the complete OSEK OS is implemented in the style exemplified above. The source code is then processed with C-Mix/II, which is given the information that all the `config` parameters are static while all other parameters are dynamic. The resulting generating extension reads an XML’ed OIL specification and synthesizes an optimized version of the operating system.

5 Results

This section gives some numbers on the sizes of source code and compiled binary images for both the original source code as well as for different specialized versions. Furthermore, some drawbacks with the proposed approach that became apparent during implementation work are discussed.

5.1 Source and Binary Code Sizes

As stated before the original source code, which still includes the configuration evaluation code and all possible features of the operating system, can be compiled without applying C-Mix/II at all. The size of this source code, as well as that of the binary object file compiled for the i386 target are given in Table 1 (a).

As some features still used by this unprocessed source code (e.g., the XML related code) prevent the compilation for the actual target platform, m68k specific numbers cannot be given here.

<table>
<thead>
<tr>
<th>Table 1. Source and Binary Code Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
</tr>
<tr>
<td>artifact</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>source code</td>
</tr>
<tr>
<td>obj. code (i386)</td>
</tr>
<tr>
<td>obj. c. (m68k)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>artifact</td>
</tr>
<tr>
<td>flashable image</td>
</tr>
</tbody>
</table>

Table 1 (a) gives the sizes of specialized versions of the operating system code. STANDARD mode means, that in the OIL specification all sanity checking, error reporting and hook functionality are disabled, while the contrary holds for EXTENDED mode.
As can be seen from the figures, the generating extension synthesized by C-Mix/II from the original source code generates code which is considerably smaller than the original full blown version. A certain fraction of this savings is due to the configuration interpretation code, which is not present in the resulting source code. However, the comparison of two different configurations shows that code fragments from the core functionality can be pre-evaluated as well.

Table 1 (b) compares the size of binary flashable images of two versions of a complete small application, which contains the operating system functionality for a certain configuration as well as some application code. We do have an implementation of the OSEK OS generator, which is written by hand in the traditional way. The numbers were derived once while applying this traditional implementation and once for a generator implemented with C-Mix/II, but both times with equivalent configurations and the same application code. While it is coincidence that both sizes are almost identical, these numbers indicate that it is possible to achieve comparable results with the partial evaluation based approach.

5.2 Observed Weaknesses

Based on our experiences we consider it reasonable to implement generators for embedded systems software with C-Mix/II already today. However, there are still a lot of weaknesses with this approach. Many of them only exist because some features are not implemented in C-Mix/II while the required theory already exists [5].

The following sections discuss some important issues, for which a solution still has to be developed.

5.2.1 Reflection

Amongst others, an OIL specification declares the tasks, which the system shall execute at run-time. The actual task functionality is specified by the developer in a C function and the name of this function is referenced in the configuration’s task declaration.

The problem now is to derive a pointer to a function from a character string, which gives the function’s name. While it is easy for a traditional generator to produce a line of code like ‘extern void FuncName(void);’ and then reference the function pointer via the FuncName identifier, this is not directly possible with partial evaluation.

This problem can be solved with ad-hoc workarounds. However, an elegant solution would have to provide some means of reflection capabilities. The developer would use functions like ‘void* GetFuncPtr(const char *name)’, which the generating extension would have to implement in the way described in the previous paragraph.

5.2.2 Cross Partial Evaluation

What is actually needed in the context of embedded systems is ‘cross partial evaluation’: Whereas the generator generator as well as the generating extension are executed on the developer’s host machine, the specialized code will be executed and has to be correct with respect to the target machine’s environment. This is especially problematic for ANSI C, since the standard does not fully specify the language semantics, but leaves some degree of freedom with respect to the target machine (e.g., bitwidth of the int type, endianess.)

While it is possible with carefully crafted code to prevent the rise of problems from these facts, it would be preferable if the generating extension would be synthesized to be linked to an abstraction layer library, which implements the behaviour of the target platform. E.g., a statically evaluated addition ‘a+b’ would be executed in the generating extension via a function like ‘add_m68k(a,b)’.

5.2.3 Code Bloat

It is possible that the specialized version of a program is considerably larger than the original. This is the case when a single function is present in the residual program in differently specialized versions, or when a loop is unrolled because it’s boundaries can be computed statically, but it’s body has to be executed at runtime. However, for the case of software component configuration, where the parameters in question are mainly component properties, this is probably not a problem. These properties are typically set at system startup in initialization code, which is only called once from one point in the program. It should thus not often be the case that different specialized versions of the initialization function will be synthesized. A component’s configuration code furthermore is often mainly composed of switch- and nested if-then-else statements, the provably unused branches of which can be eliminated during specialization. Complex computations, which typically are build on complex looping constructs, are less common in configuration code. For these reasons it is not to be expected that specialized configuration code will suffer code bloat problems. Nevertheless, this subject needs further investigation.
5.2.4 Function Signatures

A problem with C-Mix/II is that it changes the function signatures of partially evaluated functions, which is not desireable if reusable software modules are to be specialized. One reason for this behaviour is the fact, that it is not necessary to pass pre-evaluated static function parameters to an already partially evaluated function. Therefore, those are removed from the function signature. While this would be relatively easy to fix with a small overhead on run-time resources, another reason is more fundamental. A function, which is called in two different contexts in the code to be specialized, may have a parameter which is static in one context, but dynamic in the other. C-Mix/II will produce two different specialized versions of this function, with two different signatures and names.

For embedded systems, it would probably be necessary to conservatively unite both versions of the function. This would yield worse specialization results, but prevent code bloat and function signature problems. The theory for this is not much developed yet, however.

6 Related Work

[6] reports on the application of the partial evaluator Tempo [1] on five instances of different architectural software patterns. In contrast to this, our study applies a single yet larger real world example from the embedded software domain.

Many component models were proposed for embedded software development [12][10][9][13]. Most of them consider only the instantiation and interfacing of modules as the static configuration. A partial evaluation based approach considers a module’s core functionality in addition. The Koala component model [10] utilizes the C preprocessor in a clever way to make the binding of constant values to parameters explicit. This allows an optimizing compiler to pre-evaluate some statements, which could be considered limited partial evaluation.

7 Conclusions

This paper reports on our experiences gathered during the implementation of an embedded operating system generator, applying partial evaluation techniques. While we have demonstrated that it is already feasible and reasonable to choose this approach, we have also identified some serious weaknesses. We are currently working on a partial evaluator, which considers special requirements of embedded systems software design.

References