Enabling Constraint-based Binary Reconfiguration by Binary Analysis

Daniel Baldin, Stefan Grösbrink, Simon Oberthür

Design of Distributed Embedded Systems research group
Heinz Nixdorf Institute, University of Paderborn
Fuerstenallee 11, D-33102 Paderborn, Germany
dbaldin@upb.de, stefan.groesbrink@hni.upb.de, oberthuer@upb.de

***Abstract*—Today's adaptable architectures require configurability and adaptability to be supported already at design level. However, modern software products are often constructed out of reusable but non-adaptable and/or legacy software artifacts (e.g. libraries) to meet early time-to-market requirements. Thus, modern adaptable architectures are rarely used in commercial applications, because the effort to add adaptability to the reused software artifacts is just too high. In this paper, we propose a methodology to semi-automatically configure existing binaries on a given set of constraints. It is based on building the annotated control flow graph to identify and remove unused code on static basic block level depending on different execution requirements given as a set of constraints. This allows for adaptation of binaries after compile time without the use of source code. We then propose a way of adding additional reconfiguration support to these configured binary objects. With this approach, adaption and reconfiguration can be added with a low effort to non-adaptive software.**

***Index Terms*—basic block, binary adaptation, legacy code optimization, reconfiguration**

I. Introduction

Different architectures have been proposed in order to support configurable as well as reconfigurable binaries [1], [2]. However these approaches assume e.g. that all components are built with configurability support on source code level. Hence, the systems do not support legacy code. In order to speed up the development of software products, reuse of libraries is essential in today's industry to reach a short time-to-market. If we want to use existing libraries, which do not support configurability and can not be modified at source code level, a new approach is required.

In this paper we introduce a methodology which adds configurability to binary objects by an approach that semi-automatically optimizes the binary with respect to a given set of constraints. The configuration is based on building and using the annotated control flow graph of the binary to optimize the binary on static basic block level. We assume the software to be already optimized by current state of the art link time code optimizers [3], [4] and/or any other kind of global optimization technique as we are not performing standard link time optimization. We then propose a way of using the extracted configurations as reconfigurable sets using the Flexible Resource Manager approach described in Section IV-B to add reconfiguration support to the software product.

The approach requires only minimal source code information. Specifically we need method signatures to identify higher level expressions that are used for the optimization process. This is only a small restriction since even proprietary libraries come shipped with header files containing structure and method signatures for interface methods. If this would not be the case the entire library would not be usable by any higher level language as the interfaces would be unknown.

II. Related Work

Many algorithms and approaches have been proposed to optimize a program on binary level. Link Time Optimization has become common in most compiler tool chains. Algorithms for different kind of optimization goals exist, as e.g., Dead-Code Elimination, Loop Unrolling, Live Register Optimization, etc. Despite the overall benefit of these approaches to globally optimize the application for performance and/or memory consumption, there is up to now no tool which allows the binaries to be adapted on a constraints basis without high level modifications on the source code of the binaries. The authors of [5] propose the creation of so called "adaptable binaries" by adding information to the binaries, which may then be used to modify the binary later on. The approach in [6] is based on using new architectures and creating adaptable and reloadable components on source code level. A promising approach has been shown in [7] by creating so called "delta files", which contain the byte streams of the adaptations to be made on binary level. However, the delta files are created by compiling the adaptations from source code for the different kinds of configurations.

All these approaches have in common that they cannot be used with proprietary libraries that already have been compiled and may not be rebuilt with these kind of information or adaptation support.

III. Binary Basic Block Optimization

Configuration on basic block level can be a valuable technique since it allows adaptation even for legacy code. Table I compares some valuable features and techniques that may

DOI: 10.5176_2010-2283_1.4.96

```
var = namespace::func(&(char*)var2,var3 << 4 ^ 0xff)
    != sizeof(unsigned int)
```

Listing 1. Hard to evaluate example C-style expression.

be needed for automatic configuration of systems that are not built to support configuration or even reconfiguration. The evaluation shall demonstrate which features or technique are more or less expensive to support on different code levels as there are Basic Block Level (BBL), Intermediate Language Level (ILL) and Higher Language Level (HLL).

	BBL	ILL	HLL
Platform Independence	- -	++	+
Expression Parsing	++	+	-
Data Flow Analysis	-(-)	+	++
Legacy Code Support	++	- -	- -
Tool Support	+	-	+

TABLE I
FEATURES OR TECHNIQUES NEEDED FOR AUTOMATIC OPTIMIZATION ON DIFFERENT CODE LEVELS COMPARED ("-" MEANS MORE DIFFICULT, "+" MEANS EASIER).

Any automatic or semi-automatic approach will need to do some expression parsing and data flow analysis at some point of the optimization process. While expressions can easily be parsed on basic block level since most architectures consist primarily of binary and unary operations parsing higher level languages gets more and more complicated as they support complex expressions. For example parsing C++ expressions as seen in Listing 1 needs far more effort than parsing a block of assembler instructions as we also need to face ambiguities and other problems. On the other hand data flow analysis can be complex on basic block level [8] while parsing higher level languages easily allows to identify variables and references and thus data flows in general. Platform Independence and legacy code support are an important factor as well. While the high platform dependence on basic block level may be compensated by a higher implementation effort, legacy code support can not be achieved efficiently by some means on ILL layer or HLL layer since this layer is often not available for legacy code. Especially the support of legacy code renders the basic block layer an interesting layer for new configuration and reconfiguration approaches.

IV. FOUNDATION

The desired configurability at basic block level is realized by following the TEReCS concept, which allows the synthesis of valid configurations based on the given requirements like binary size. The approach of puppet configuration is introduced. Finally the online configurability at basic block level is realized by applying our flexible resource management (FRM) approach. The FRM approach allows to define different profiles with different configurations encapsulated as basic block combinations. At runtime the FRM can then decide which profile needs to be activated to provide the desired service as e.g. IPv4 or IPv6.

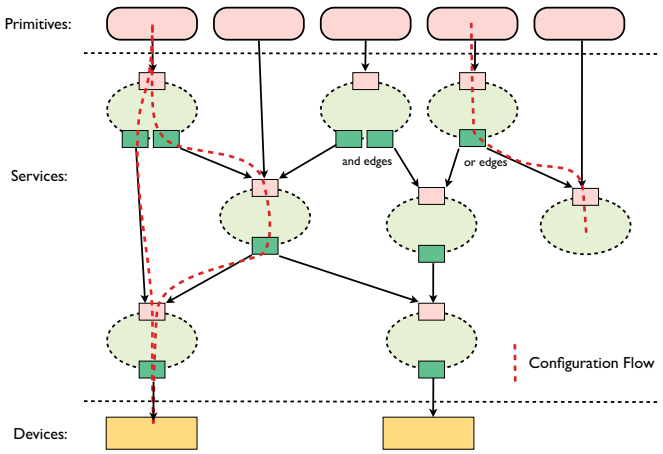

Fig. 1. TEReCS's design space description from system primitives via services down to hardware devices (from [10]).

A. TEReCs

TEReCS[1] is a methodology to synthesize and configure operating systems for distributed embedded applications developed at the University of Paderborn [9], [10]. The approach strictly distinguishes between knowledge about the application and expert knowledge about the customizable operating system. Knowledge about the application is considered as a requirement specification. This requirement specification is an input to the configurator.

The complete and valid design space of the customizable operating system is specified by a so-called AND/OR service dependency graph [11]. This domain knowledge contains options, costs, and constraints and defines an over-specification by containing alternative options. The configuration process removes some domain specific knowledge by exploiting knowledge about the application. Thereby, a configuration for the run-time platform will be generated.

The complete valid design space of the configurable operating system is specified by an AND/OR graph as depicted in Figure 1:

- Nodes represent *services* and are the smallest atomic items, which are subject of the configuration
- Mandatory dependencies between services are specified by the AND edges
- Optional or alternative dependencies between services are specified by the OR edges
- *Constraints* (preferences, prohibitions, enforcements) for the alternatives can be specified
- Root nodes of the graph are interpreted as *system primitives*

The main objective of the configuration process is to remove all OR dependencies from the graph (over specification → complete and non-ambiguous specification). The configuration can be interpreted as a sub-graph without any alternatives.

The algorithm works as follows: A path can be found through the complete graph from the sending primitive down

[1]**T**ools for **E**mbedded **Re**al-Time **C**ommunication **S**ystems

to the sending device. The primitives can be considered as the strings of a puppet. Depending on which strings are pulled, the "configuration" of the puppet will change accordingly. The service dependencies can be compared to the joints of the puppet. Therefore, the algorithm is named *"Puppet Configuration"*.

B. Flexible Resource Management (FRM)

The FRM [12] was developed to improve the resource utilization by putting temporary unused but for worst case resource consumptions reserved resources at other applications' disposal. In case of a conflict, it is solved under hard real-time constrains. The FRM schedules the resource demands of multiple applications. Each agent is equipped with a set of possible profiles and transitions between them. Each profile contains information about maximum and minimum resource requirements, switching conditions and a profile quality, which is used to indicate which agents to prefer when resources can be freed. The FRM is in charge of deciding in which of their profile the applications are running. The profiles can be semi-automatically generated out of hybrid reconfiguration charts [13].

The FRM approach is also applied to the operating system (OS) itself. The resource usage implies the services that the applications require from the OS. Reconfiguration of the OS means supporting on demand services or the possibility of degrading services. The FRM was used to extend an offline customizable OS in order to be dynamically reconfigurable during run-time. Thus with this extension the operating system is aware of the current required services.

V. Evaluation Scenario

For evaluation purposes and as an illustration of the methodology, we implemented an Internet Protocol(IP)-Stack on an integrated circuit card (ICC) with an S3FS9CI AT91 ARM processor. The approach has thus been evaluated on the ARMv4(T) Instruction Set Architecture (ISA). We extended the lightweight Internet Protocol Stack (lwIP Stack) [2] with IPv6, ICMPv6, and TLS functionalities which allows the IP-Stack to be used in a broad set of configurations. The overall IP-Stack contains several modules and hundreds of methods which makes it possible to test our approach with a realistic scenario.

For the deployment of this protocol stack, it is desirable to configure the stack application-specifically. Especially the high amount of independent functionalities and sub-functionalities inside the protocol implementation makes this software program a perfect candidate for our configuration approach. Depending on the application many parts of the protocol stack may not be used. One may imagine a scenario in which the IP-Stack is deployed in an IPv4-only infrastructure. In this example the parts of the application implementing IPv6

[2]http://savannah.nongnu.org/projects/lwip/

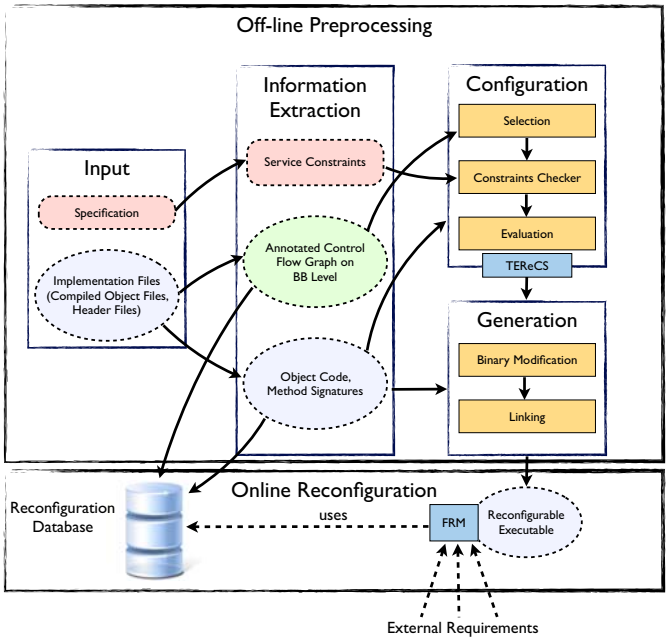

Fig. 2. Configuration Methodology.

would not be needed. On an even more fine granular basis configuration could also be applied to sub-functionalities as the support for multicast addresses. An deployment scenario may change from a non-multicast environment to a multicast using environment. Automatic adaptation of the software program to support these functionalities would be a major benefit in terms of resource efficiency.

VI. Methodology

The overall offline optimization process involves three major steps as depicted in Figure 2: Binary Analysis, Configuration and Modification. First of all, the binary has to be analyzed. This analysis works on the object files which are used inside the linking process. Using these object files, a global annotated control flow graph is derived. This graph construction is described in the next section.

A. Binary Analysis

Before we can remove parts of the binary, we need to identify the control flow and the semantics of the program. In the first step we parse the binary to derive the control flow graph on static basic block level which is described in Section VI-A1. In the next step we extract the conditions for transitions to take place between basic blocks of the program using data flow analysis techniques as described in Section VI-A2.

1) Control Flow Graph Generation: By disassembling the binary code we identify the static basic blocks and the control flow between these blocks of the program. A static basic block is a sequence of instructions that has exactly one entry point and one exit point. We use the basic block as the smallest representation and optimization unit since it describes a linear flow of instructions. A non-linear control flow appears only

at the end of a basic block. Each instruction that is a target of a branch instruction defines a new basic block. In general, every program can be uniquely partitioned into a set of non-overlapping static basic blocks.

Whenever it is possible to remove an instruction inside a basic block, it is possible to remove the complete block. Figure 3 depicts the first four basic blocks of the disassembled `ip6_input` method. Using these blocks we can derive a graph representing the possible control flow of the processor (called control flow graph) as seen on the right side of Figure 3. Each node defines a basic block and the edges represent conditional control flow (dashed edges) and unconditional control flow (solid egdes) between these blocks. Each control flow edge models a dependency between the basic blocks, as reaching one basic block means that we may also reach the successors of it.

The analysis of binary code is a non-trivial task. Disassembling and interpreting binary files is fraught with problems, e.g., the Code Discovery Problem. Many Instruction Set Architectures (ISA) allow binary data to be mixed up with executable instructions and vice versa. Not being able to distinguish between instructions and data may invalidate the whole optimization process since some control flows may not be discovered or data may be misinterpreted. However for our evaluation platform this problem does not exist, since the ARM Embedded Applications Binary Interface (EABI) forces all EABI conform Embedded Linker File (ELF) object files to provide information on all occurrences of data and instruction blocks by special mapping symbols inside the symbol table (see Section 4.6.5 in [14] for the symbol definition).

Another problem with control flow detection arises if indirect control flow instructions are used inside the binary. Most of the indirect control flows are due to jump tables that are generated by the compiler to speed up switch/case statements. The targets of these jumps can be computed with high precision as it was shown in [15]. The basic idea is to use expression substitution, similar to the approach in Section VI-A2, to allow the expression to be checked against branch normal forms. Other sources of indirect control flows are method pointers, available in most high-level languages, e.g., to implement inheritance or to allow dynamic program behavior. The targets of these kind of indirect control flows are very hard to compute and to the best of our knowledge no approach exists which can guarantee the precise detection of all targets. Using the approach proposed in [16] however, we may overestimate the jump targets by introducing a so called ”hell node”. The estimation uses the complete set of relocatable symbols, which is the union of all relocatable symbols of all object files, as the target for every indirect jump. The approach may not be as tight as possible but it ensures the correctness of the following optimization process.

2) Graph Annotation: We are using the common approach of forward substitution, as described in [17], [18], to derive higher level expressions from low level expressions, which

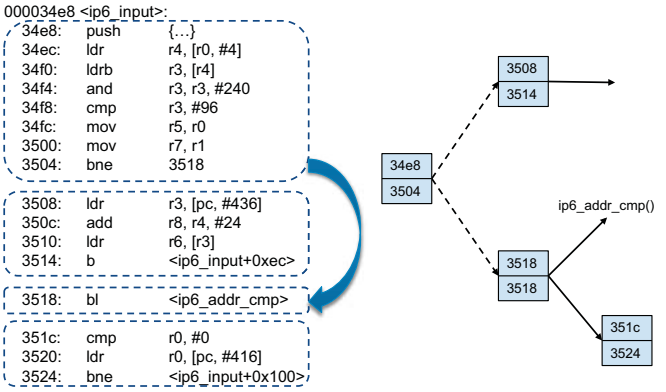

Fig. 3. Control flow graph construction using the basic blocks of the disassembled binary.

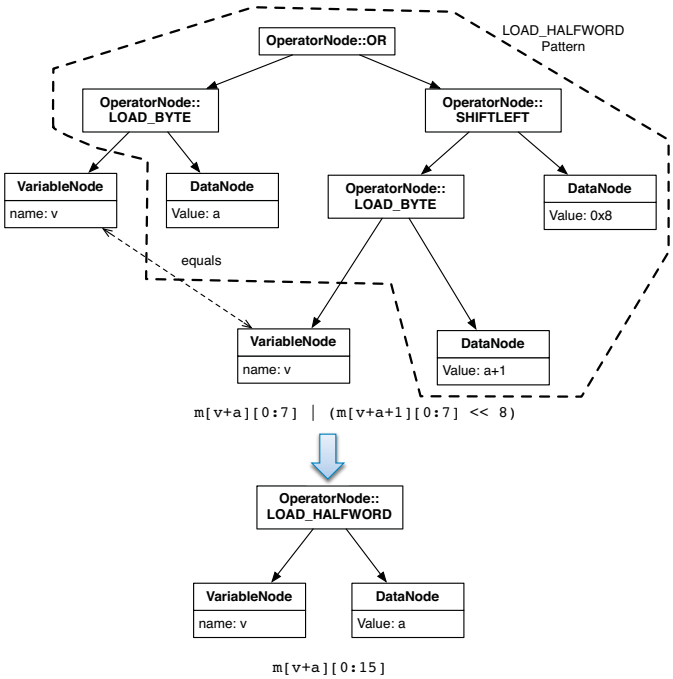

Fig. 4. Expression Representation and Reduction.

in our case are the assembler instructions of the object files. For assembly code one can express the contents of a register r in terms of a set a_k at instruction i as $r = f_1(\{a_k\}, i)$. If the definition at instruction i is the unique definition of a register r that reaches an instruction j along all paths in the program, without any of the registers a_k being redefined, one can forward substitute the register definition at instruction j with $s = f_2(\{r\}, j)$, resulting in:

$$s = f_2(\{f_1(\{a_k\}, i)\}, j)$$

After analyzing a basic block, the content of each register can be computed by forward substitution as an intermediate level representation based on the Register Transfer Lists (RTL) model by [19]. This model describes the effect of machine instructions as a list of register transfers and is general enough to support different kinds of architectures. For performance reasons we represented these expressions by a tree of literals.

```
1 struct pbuf {
2   /** next pbuf in linked pbuf chain */
3   struct pbuf *next;
4   /** pointer to the actual data */
5   void *payload; }
```

Listing 2. pbuf structure definition as defined inside pbuf.h

These literals can be represented as nodes as seen in Figure 4. Binary and unary operations are represented by an Operator-Node with the corresponding operation (e.g., bit shift, binary or/and/not or memory accesses). Unknown register contents and variables are represented as VariableNodes, numerical values as DataNodes. The binary operators implicitly define an order on the tree, e.g., the memory access operator's left child defines the base address whereas the right child defines the offset.

As an example for the expression generation, we may analyze the first basic block, which has address $0x34e8$, in Figure 3. After forward substitution of the instructions, the contents of the modified registers may be represented (in textual RTL notation and using word memory accesses) as follows:

```
1 *32* r[4]=m[r[0]+4]
2 *32* r[3]=m[m[r[0]+4]][0:7] & 0xF0
3 *32* r[5]=r[0]
4 *32* r[7]=r[1]
```

Listing 3. RTL register expressions after analysis of basic block at address $0x34e8$ of Figure 3

Inside these expressions we search for patterns of literals that are either given by the system designer or automatically generated from the header files, that contain the corresponding structure information and/or method headers. Using gcc-xml we are able to derive the byte layout, the members of structures and the method signatures, which are specified in the provided header files. At this point it is important that the implementation conforms to some known ABI, so that a well defined correlation between input registers and method header exists. For the following example let the structure `pbuf` be defined as in Listing 2. Given this structure definition we know whenever we are accessing the word at position zero of this structure that we are accessing the `next` pointer. Accordingly the word at position four stores the `payload` pointer. As the basic block at address $0x34e8$ is the first block of the method `ip6_input(struct pbuf *p)`, the term `m[r[0]+4]` may be substituted with the term `p->payload`. This step completely depends on the underlying ABI and the processor architecture as the layout and byte order of the structure can vary on different architectures.

In the next step we normalize the expressions, as depicted in Figure 4, with respect to a set of reduction patterns. The figure shows how a pattern inside the expression tree may be used to reduce the tree to a smaller but semantically identical expression tree. In this specific example the second expression tree essentially describes the same halfword memory access as the first expression tree, which is realized by two consecutive byte memory accesses. This kind of behavior can be observed frequently within binaries, mainly due to alignment restrictions of the hardware platform for memory accesses. The reduction takes place as long as there exists a pattern that can be applied. The normalization is mandatory as there exist an infinite amount of possible combinations for the same expression which makes the computation overhead huge and the comparison of two expressions hard, if not impossible.

```
1 ip4_input():
2   // dont support multicast
3   ip4_header.src_addr[0] > 239
4   ip4_header.src_addr[0] < 224
5 ethernet_input():
6   // do not support ipv6
7   eth_header.type != 0x806
```

Listing 4. Exemplary constraint set

If enough information is given by the header files we may then use the normal-form high-level expressions to annotate the conditional edges as seen in Figure 5. The approach is limited by the availability of information that can be extracted from the set of header files. Edges which are not annotated cannot be checked against the constraints of the following optimization process. However even a single annotated edge may allow the removal of huge parts of the binary, reachable over this edge as we will see in the section VII.

B. Configuration

Using the annotated control flow graph we are now able to identify under which conditions control flow occurs between basic blocks. Precisely the expression of a conditional edge describes the condition that must be fulfilled for the edge to be taken. The basic idea for the configuration step is to use a set of constraints, given by the user who wants the program to be adapted, check them against the conditions of the conditional edges and thus identify which basic blocks are not reachable using the constraints.

1) Constraints: The set of constraints for the optimization process contains constraints on expressions that are used inside the binary. An example of such a constraint set, for the example scenario, is given in Listing 4. Constraints are specified on input parameters of methods. Since parameters may have the same name for different methods, it is mandatory to specify the method as well. Line two and three state that the value of the first byte of the supported IPv4 source addresses inside the method `ip4_input` will be lower than 224 and greater than 239. This essentially represents all non multicast addresses. Line five states that ethernet packets that contain IPv6 frames shall not be supported inside the method `ethernet_input`. All other ethernet packets would be valid if the type field inside the ethernet header would be different to the value of `0x806`.

The subsequent optimization mechanism now searches the annotated control flow graph for conditional control flow edges. Selected edges are forwarded to the constraint checker

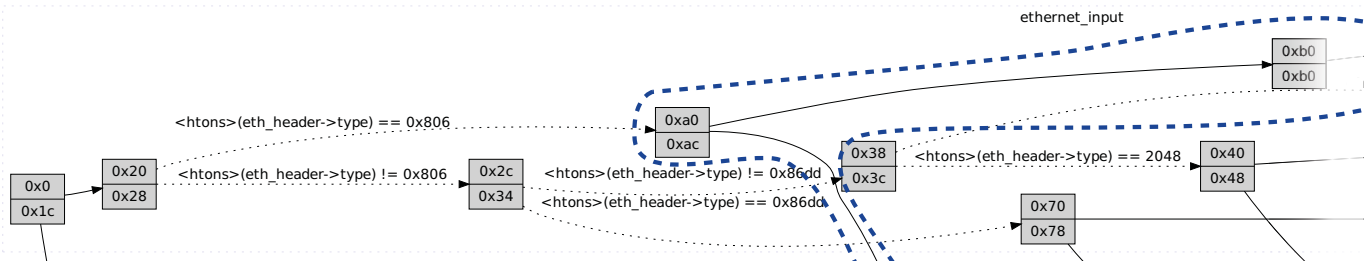

Fig. 5. The annotated control flow graph of the method $ethernet\_input(struct\ pbuf * p)$. The selected parts define a partition that may be removed for the constraint $eth\_header- > type! = 0x806$.

which uses the set of constraints to check whether the expression of the edge will be fulfilled or not. Edges containing only one variable that is related to at least one constraint are tested using the Algorithm 1. The algorithm checks whether the condition of an edge will only be fulfilled for constraint values. If the condition is fulfilled for values different to the ones of the constraints, the edge cannot be removed, since the edge may be taken for values that are not specified as constraints.

Algorithm 1 Edge evaluation algorithm

```
proc isRemovable(Edge e,Constraints Set c_i) ≡
  isRemovable := false;
  do if condition(e) only contains one variable v
       comment: test all possible values of the variable
       for value := minval(v) to maxval(v) do
         do if value is a constraint
              comment: check if the constraint is fulfilled
              if evaluate(e, value) == true
                isRemovable = true; fi
         else
              comment: edge must evaluate to false
              if evaluate(e, value) == true
                return false; fi
            od
         od
       od
       return isRemovable;
  else
       return false; fi od.
```

As the expression evaluation is a non-trivial task, the amount of expressions that can be evaluated by using the constraints heavily depends on the application and the data analysis technique. Some expressions may even contain function calls, which makes the evaluation ambiguous. As an example, consider the expressions in Figure 5. All of the expressions contain function calls to the method `htons` which modifies the input parameter `eth_header->type`. However evaluating the expression is not impossible. One may inline the method into the expression, use summary functions ([20]) or even emulate the expression under certain circumstances. Anyhow, the quality of the optimization scales with the amount of expressions that can be evaluated. Fortunately even being able to evaluate only small parts of the binary may still lead to very good optimization results.

Edges which do not fulfill the constraints are so called "Configuration Points" and candidates for removal. However not all configurations points improve the overall performance or reduce the overall code size if removed. Thus an evaluation step may filter the edges for the following steps.

2) Graph Partitioning: In general the graph can be partitioned into sets of basic blocks based on the configuration points. In our case, if a set of blocks is identified to be only reachable by a specific edge these blocks are grouped into one partition. In Figure 5 the control flow graph of the method `ethernet_input(struct pbuf *p)` of our IP-Stack implementation has been annotated and the marked part has been identified to be only reachable by an edge that can be removed from the binary using the constraints given in Listing 3.

Using the following algorithm we can calculate the partitions of basic blocks for all configuration points:

Let the application be given as a graph: $G = (N, E)$ with N being the set of nodes (static basic blocks) of the graph and E the set of edges of the graph, set $S \subseteq N$ of start nodes (entry points) and set $R \subseteq E$ of configuration points.

1) Define the set T of nodes that can be reached from the start nodes without taking any removable edge: $T = \{n \in N : \exists w = (w_1, w_2, ..., w_n), w_1 \in S \wedge (w_i, w_{i+1}) \in E \backslash R \wedge w_n = n\}$. This essentially defines the set of basic blocks that are mandatory given a specific execution environment.
2) For every removable edge $r_i \in R$ with $r_i = (n_1, n_2)$ create the set N_{r_i} of nodes that can be reached over the removable edge without reaching a node that is mandatory (inside set T): $N_{r_i} = \{k \in N : \exists w = (w_1, w_2, ..., w_n) \wedge w_1 = n_2 \wedge (w_j, w_{j+1}) \in E \wedge w_j \notin T \wedge w_n = k\}$. These sets define the configurations.
3) For any two sets N_{r_i} and N_{r_j} calculate $K_{ij} = N_{r_i} \cap N_{r_j}$. Every edge going from any of the two sets into the intersection K_{ij} needs to be treated specially during code reconfiguration/reloading. (There can not be any edges going from K_{ij} into N_{r_i} or N_{r_j} since these nodes would then be part of K_{ij}. This can be seen if we reconsider the construction of N_{r_i} and N_{r_j}.)

Each of the sets $N_{r_i} \backslash K_{ij}, N_{r_j} \backslash K_{ij}$ and K_{ij} itself may then be used inside the configuration process and may be removed from the binary. The removed configurations are stored and

linked separately. Thus each basic block inside a configuration needs to be updated with the correct memory addresses of depending configurations and symbols inside the final binary. This can be done off-line so reloading the code can be done without any kind of online address translation. Each configuration defines a node inside the TEReCS configuration process. Dependencies between the static binary and the configurations is modeled by corresponding AND/OR edges.

As we now obtained the sets N_{r_i}, we can replace the first basic block inside the set that is reached by the edge r_i with a call to the FRM. This needs to be done for edges r_i that are evaluated by the configuration process to be removed from the initial configuration. Taking one of these edges during runtime triggers the reconfiguration process of the FRM. Using a connection to the configuration database the configuration and all depending configurations (given by the TEReCS configuration tree) are loaded. The execution of the program then continues at the reloaded configuration.

C. Binary Modification

Given the partitions of basic blocks, we can now remove those sets of blocks N_{r_i} from the object files. We parse the basic blocks of the object file in a linear manner and remove the basic blocks that are specified to be removed. Thus all following basic blocks move to lower addresses. This process continues until all basic blocks are parsed. Using the standard libelf library we now modify the executable .text area, the symbol and the relocation tables of the object files to reflect the changes made on the control flow graph. The major part of the rewriting process involves modifying all instructions that reference other basic blocks inside the object files. For the ARMv4(T) ISA this involves changing the following set of eleven different instructions specified in Table II (see [21] for details on the instruction types).

Instruction	Encoding Type
Branch B	T1-B, T2-B, A1-B
Branch and Link BL	T1-BL, A1-BL
Load Register LDR	T1-LDR, A1-LDR
Load Byte LDRB	A1-LDRB
Load Halfword LDRH	A1-LDRH
Load Signed Byte LDRSB	A1-LDRSB
Load Signed Halfword LDRSH	A1-LDRSH

TABLE II
INSTRUCTIONS THAT NEED TO BE MODIFIED INSIDE THE BINARY REWRITING PROCESS.

For each of the instructions the corresponding offset to the basic block referenced needs to be recalculated and changed. However this only needs to be done for object files that need to be modified.

Additionally the symbol and relocation tables need to be changed. Symbols and relocatable instructions may now be defined at different positions inside the text area. Thus the table entries are updated with the new positions. Some symbols and relocation entries may even be removed since the basic blocks, which referenced these symbols, do not exist any more, thus removing the dependency between these object files containing these basic blocks. The result of such a rewriting process on a relocation table can be seen in table III. During the process, entries five, six and eight have been removed from the relocation table as the basic blocks containing these relocatable instructions have been deleted. For all other entries the offset has been updated. The basic block removal also results in symbols to be changed or removed inside the symbol table of the binary, as seen in table IV. This also means that linking this object no longer depends on the removed symbols, which had to be provided in some other object files.

Nr	Offset	Type	Sym. Name
1	000010	R_ARM_THM_CALL	htons
2	00002c	R_ARM_THM_CALL	ethar_ip_input
3	000036	R_ARM_THM_CALL	pbuf_header
4	00004a	R_ARM_THM_CALL	ip4_input
5	000054	R_ARM_THM_CALL	ethar_ip_input
6	00005e	R_ARM_THM_CALL	pbuf_header
7	00006e	R_ARM_THM_CALL	libprintf
8	000078	R_ARM_THM_CALL	ip6_input
1	000010	R_ARM_THM_CALL	htons
2	00002c	R_ARM_THM_CALL	ethar_ip_input
3	000036	R_ARM_THM_CALL	pbuf_header
4	00004a	R_ARM_THM_CALL	ip4_input
5	000058	R_ARM_THM_CALL	libprintf

TABLE III
RELOCATION TABLE BEFORE AND AFTER THE BINARY REWRITING PROCESS.

Nr	Value	Bind	Ndx	Name
...	...	...	...	...
19:	00000001	GLOBAL	1	ethernet_input
20:	00000000	GLOBAL	UND	htons
21:	00000000	GLOBAL	UND	ethar_ip_input
22:	00000000	GLOBAL	UND	pbuf_header
23:	00000000	GLOBAL	UND	ip4_input
24:	00000000	GLOBAL	UND	libprintf
25:	00000000	GLOBAL	UND	ip6_input
...	...	...	...	...
19:	00000001	GLOBAL	1	ethernet_input
20:	00000000	GLOBAL	UND	htons
21:	00000000	GLOBAL	UND	ethar_ip_input
22:	00000000	GLOBAL	UND	pbuf_header
23:	00000000	GLOBAL	UND	ip4_input
24:	00000000	GLOBAL	UND	libprintf
25:	00000000	GLOBAL	UND	

TABLE IV
SYMBOL TABLE BEFORE AND AFTER THE BINARY REWRITING PROCESS.

VII. EVALUATION

We evaluated the approach on the IP-Stack and the hardware platform described inside the illustration scenario. We compiled the whole IP-Stack to contain all functionality although not all functions would be used to communicate with other peers.

In a first evaluation we tried to remove the TCP support from the IP-Stack using the constraints shown in line one to four of listing 4. The first line states that the lower 16 bit of the `_ttl_proto` field may not contain the value six. This essentially states that there should not be a TCP header after a IPv4 header. For our second evaluation we tried to remove the IPv6 support by using the constraints in line three

```
1 ip4_input():
2   ip4_header._ttl_proto & 0x00ff != 0x06
3 ethernet_input():
4   eth_header.proto != 0x806
```

Listing 5. Constraints set for removing the TCP and IPv6 support

and four given in Listing 5. The IP-Stack object files have been analyzed, optimized and rewritten using the constraint set given above. The complete analysis, annotation, configuration and rewriting process of 80 Kilobytes of assembler code took about two minutes on a general purpose linux machine (Core2 Duo, 2,4 GHz, 1 GB Ram). The most time-consuming part (about 60%) is the annotation phase as the annotation step involves analyzing every basic block (sometimes multiple times for loops). The reduction inside each of the components is depicted in Figure 6. The complete TCP component is

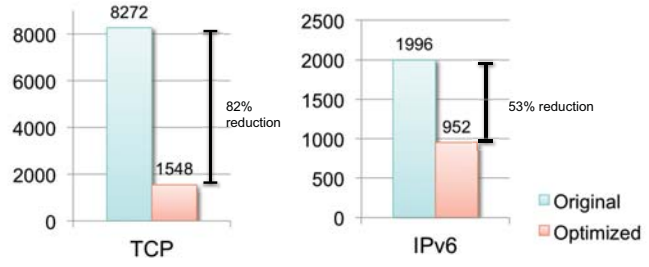

Fig. 6. Binary size reduction for each component.

about eight Kilobytes in size. Using the simple constraint set in Listing 5 it was possible to remove 82% of the TCP implementation using the optimization approach in this paper. The remaining bytes of the implementation may be removed with a more sophisticated constraint set as not all control flows are covered by the set in Listing 5. A similar statement holds for the IPv6 reduction case in Figure 6 although only 53% could be removed by the single constraint in line four of Listing 5. This is due to the fact that the constraint only restricts control flow from the lower ethernet packet layer. Control flow from higher layers, as e.g., the application layer, was not considered by the constraint set. This is however absolutely possible.

VIII. Conclusions

In the last sections we proposed a way of configuring and reconfiguring a software product on binary level which was not designed to support configurability in general. The proposal is based on the idea of analyzing the existing software product with decompilation techniques in order to partition the code into sets of mandatory and optional static basic blocks. The evaluation showed that the approach can be used to automatically remove huge parts of the binary using only small sets of constraints. We then proposed a way to use these removable sets as configurations by the FRM approach to add reconfiguration to the software product, which allows the freed memory resources to be used for other components of the system. By this methodology it is possible to adapt binary objects to a requirement specification, given as a set of constraints, and allow the reconfiguration of the binary whenever the specification changes at runtime.

References

[1] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, ser. WOSS '04. ACM, 2004, pp. 28–33.

[2] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, pp. 54–62, 1999.

[3] B. De Bus, D. Kästner, D. Chanet, L. Van Put, and B. De Sutter, "Post-pass compaction techniques," *Commun. ACM*, vol. 46, pp. 41–46, August 2003. [Online]. Available: http://doi.acm.org/10.1145/859670.859696

[4] B. De Sutter, L. Van Put, D. Chanet, B. De Bus, and K. De Bosschere, "Link-time compaction and optimization of arm executables," *ACM Trans. Embed. Comput. Syst.*, vol. 6, February 2007.

[5] R. Wahbe, S. Lucco, and S. L. Graham, "Adaptable binary programs," IN, Tech. Rep., 1994.

[6] S. Kogekar, S. Neema, and X. Koutsoukos, "Dynamic software reconfiguration in sensor networks," in *Proceedings of the 2005 Systems Communications*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 413–420.

[7] R. Keller and U. Hölzle, "Binary component adaptation," in *Proceedings of the 12th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 1998, pp. 307–329.

[8] C. Cifuentes, "Reverse Compilation Techniques," PhD thesis, Queensland University of Technology, Brisbane, Australia, 1994.

[9] C. Böke, "Software Synthesis of Real-Time Communication System Code for Distributed Embedded Applications," in *Proc. of the 6th Annual Australasian Conf. on Parallel and Real-Time Systems (PART)*. Melbourne, Australia: IFIP, IEEE, Dec. 1999.

[10] C. Böke, "Automatic Configuration of Real-Time Operating Systems and Real-Time Communication Systems for Distributed Embedded Applications," PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, Paderborn University, Paderborn, Germany, 2003.

[11] R. P. Chivukula, C. Böke, and F. J. Rammig, "Customizing the Configuration Process of an Operating System Using Hierarchy and Clustering," in *Proc. of the 5^{th} IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*. Crystal City, VA, USA: IFIP WG 10.5, 29 April – 1 May 2002, pp. 280–287, iSBN 0-7695-1558-4.

[12] H. S. Lichte and S. Oberthür, "Schedulability Criteria and Analysis for Dynamic and Flexible Resource Management," *Electron. Notes Theor. Comput. Sci.*, vol. 200, no. 2, pp. 3–19, 2008.

[13] S. Burmester, M. Gehrke, H. Giese, and S. Oberthür, "Making Mechatronic Agents Resource-aware in order to Enable Safe Dynamic Resource Allocation," in *Proc. of Fourth ACM International Conference on Embedded Software 2004 (EMSOFT 2004), Pisa, Italy*, B. Georgio, Ed. ACM Press, September 2004, pp. 175–183.

[14] ARM Ltd., "ELF for the ARM Architecture," 2009.

[15] C. Cifuentes and M. V. Emmerik, "Recovery of jump table case statements from binary code," in *Science of Computer Programming*, 1999, pp. 2–3.

[16] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen, "On the static analysis of indirect control transfers in binaries," in *In PDPTA*, 2000, pp. 1013–1019.

[17] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to high-level language translation," in *In Int. Conf. on Softw. Maint*. IEEE-CS Press, 1998, pp. 228–237.

[18] C. Cifuentes, "Interprocedural data flow decompilation," *Journal of Programming Languages*, vol. 4, pp. 77–99, 1996.

[19] C. Cifuentes and S. Sendall, "Specifying the semantics of machine instructions," in *Proceedings of the 6th International Workshop on Program Comprehension*, ser. IWPC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 126–.

[20] S. Gulwani and A. Tiwari, "Computing procedure summaries for interprocedural analysis," in *European Symp. on Programming, ESOP 2007*, ser. LNCS, R. De Nicola, Ed., vol. 4421, 2007, pp. 253–267.

[21] ARM Ltd., "ARM Architecture Reference Manual," 2009.

Daniel Baldin, born 1983 in Germany, is a research assistant, working at the research group ”Design of Distributed Embedded Systems” of Prof. Dr. rer. nat. Franz Josef Rammig at the University of Paderborn since 2009. He is an associated member of the software quality lab (s-lab) at the University of Paderborn. His research interest covers embedded real-time virtualization, binary optimization and reconfigurable embedded systems. Mr. Baldin finished his master thesis in 2009 with the title Proteus, a hybrid Virtualization Platform for Embedded Systems. Since 2010 he is working on a research project targeted at binary reconfiguration for smart card systems.

Stefan Groesbrink was born in Germany in 1983. He studied computer science with electrical engineering at the University of Paderborn (Germany) and the Carleton University Ottawa (Canada). Since 2011, he is member of the research staff of the group ”Design of Distributed Embedded Systems” of Prof. Dr. rer. nat. Franz Josef Rammig and associated member of the s-lab, a multi-private-public partnership institute for knowledge and technology transfer between academia and industry. His research focus are scheduling for embedded real-time systems and system virtualization.

Dr. Simon Oberthuer was born 1977 in Steinheim (Westphalia), Germany and is scientific staff at the workgroup ”Design of Distributed Embedded Systems” of Prof. Dr. rer. nat. Franz Josef Rammig since 2002. He has studied computer science at the University of Paderborn. His research focus is dynamic resource management for real-time systems. His dissertation is titled Towards an RTOS for Self-optimizing Mechatronic Systems and was submitted in 2009.