Integration of restricted design spaces and partial evaluation in a developing kernel for embedded systems

Vladimir Támara Patiño
tamara@informatik.uni-kl.de

February 2002

06/01

Sonderforschungsbereich 501

Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
67653 Kaiserslautern
Germany

Abstract

We present how we integrated partial evaluation and restricted design spaces in an experimental kernel for embedded systems as well as some measurements of the resulting performance. The kernel has been developed with flexibility in mind, we use restricted design spaces to manage its variability and partial evaluation to make it efficient for particular requirements.

Acknowledgments

I want to express my gratitude specially to:

- God,
- the Professor Jürgen Nehmer for his financial support and his permission to release this document to the public domain and the source code that sustain it under a BSD-style license,
- the reviewers of this article: Melissa Giraldo, Prof. Klaus Madlener, Prof. Jürgen Nehmer,
- Henning Makholm and Jens Peter Secher, developers of C-Mix who helped us,
- the professor Calton Pu who suggested us partial evaluation for the tailoring of kernels.

Contents

1 Introduction **1**

2 Overview of concepts and tools **2**
2.1 GCRKTES 2
2.2 Partial Evaluation 4
2.2.1 C-Mix 5
2.3 Restricted design spaces 8
2.3.1 Problem in previous definition and solution 9

3 Integration **11**
3.1 Changes to the sources of GCRTKES 11
3.2 Extensions to restricted design spaces 13
3.3 Changes to specialization 14
3.3.1 Removing some unnecessary variables 15
3.3.2 Basis for the elimination of unnecesary variables 17

4 Measurements **20**
4.1 Microbenchmarks 20
4.2 Product of Laurent series 22
4.3 Proxy server 23

5 Partial evaluation for system software **27**

6 Conclusions **28**

1 Introduction

As part of the project SFB 501 "Development of large systems with generic methods", we want to generate correct kernels for embedded systems based on generic components, furthermore we need that those kernels fulfill some functional and nonfunctional requirements imposed by the application and the platform. To achieve this, we are exploring methods and concepts of flexibility and composition and applying them in the development of experimental kernels for embedded systems.

Our main goal by experimenting with restricted design spaces and partial evaluation is to manage the variability of a kernel written in C for embedded systems, while maintaining its efficiency. In our study we have been using:

- a toy kernel, GCRTKES[1], that is being developed as part of our study,
- the tool C-Mix developed at the University of Copenhagen (see [22]) and
- tools that we are developing to organize the variability of the sources through restricted design spaces (see [29]).

The rest of this report is organized as follows: in the section 2 we present briefly the concepts and tools used, in the section 3 we present results of some experiments, some projects related to this work are presented in the section 4 and in the section 5 our conclusions.

[1] Available as well as other tools developed for this experiment at http://wwwagss.informatik.uni-kl.de/Mitarbeiter/tamara/index.html

2 Overview of concepts and tools

Depending on the programming language there are several mechanisms to give flexibility during compilation or during runtime to a program, for the C language some of them are: constructions of the language (e.g. `if`, `switch`), preprocessors and embedding C code in other languages [21]. For this work we decided to implement as much as possible runtime flexibility in GCRTKES by using the usual constructions of the language, and we tried to achieve customization before compilation with partial evaluation. Our motivations are:

1. There are promising results in the application of partial evaluation to portions of certain operating systems, where the speedup quotients reach 3.9 [27].

2. By using the constructions of the language our kernel can be as flexible as possible during runtime, and can also be specialized during compilation for certain environments with possible gains in efficiency given by the partial evaluation.

2.1 GCRKTES

GCRTKES is a toy kernel written in C[2] which is a reimplementation of a generic kernel written in C++ (GCRTK - 2000) that in turn was a reengeniering of another generic kernel (RTK - 1998 [15]), all of them developed in the group of operating systems of the University of Kaiserlautern.

Currently GCRTKES can be used as a threads library for Linux, OpenBSD and DOS or as a multitasking kernel with its own bootloader for the processor i8088. The compilation can be done natively on Linux and OpenBSD while for DOS and i8088 with the cross-compiler `Dev86/bcc`[3] [12].

Its sources have annotations that are checked statically with LCLint [13] as well as assertions to check preconditions and invariants during runtime

[2]We had to rewrite GCRTK in C since we couldn't find an open source, cross compiler for C++ that could generate code for the processors Intel 8051 and Intel 8088 (those are target platforms of our experiment).

[3]For this work we have modified Dev86/bcc. Our modifications for the version 0.15.4 add support for `#pragma` and for the preprocessor operator `#`. The patch is available along with the sources of GCRTKES at
http://wwwagss.informatik.uni-kl.de/Mitarbeiter/tamara/index.html

(other facilities for development offered to the applications and used by the kernel are logging and tracing as well as regression tests for each layer). The commentaries in the sources allow to extract the documentation automatically (with Doc++ [2]).

This kernel is organized in layers, each one with its own variabilites. The figure 1 shows a simplified view of the inclusion graph[4], as well as the organization of the sources in directories (each cluster of nodes is a directory whose name is in a grayed box).

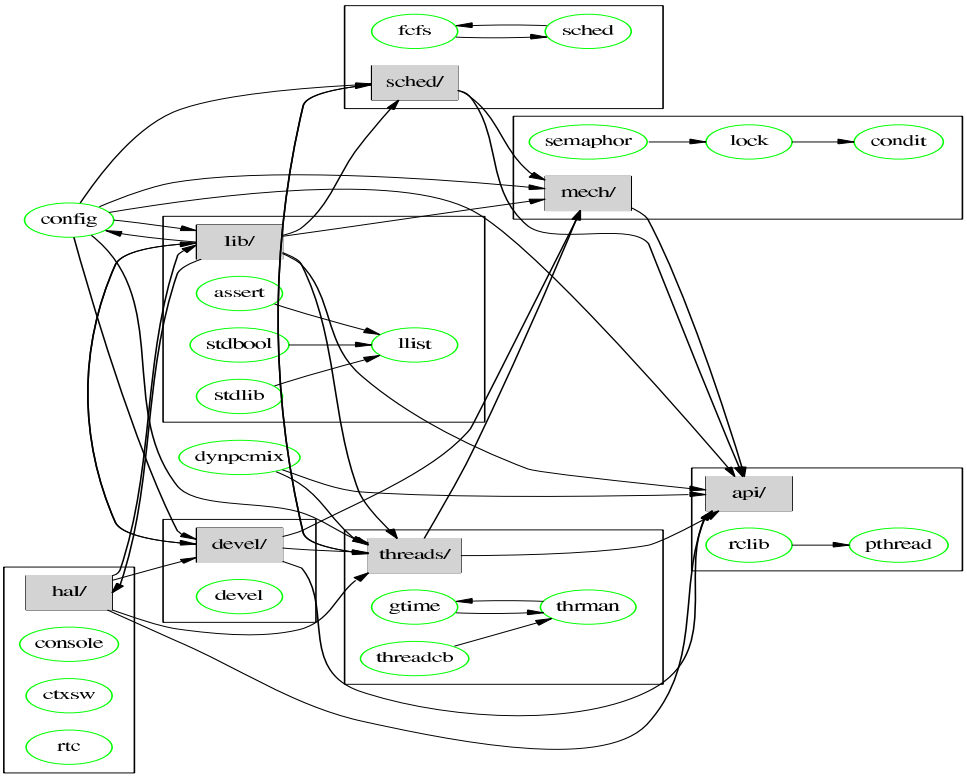

Figure 1: Modules of GCRKTES

We have identified a few functions that are platform dependant and on top of which the rest of the kernel has been constructed (we call them "hal"), they are: context switch , real time clock management [16] and device drivers

[4]Ovals represent source files, grayed boxes are directories, arrows represent the relation "included by". Inclusions of one file into another in a different directory is shown with an arrow from the directory of the first in the directory of the second.

(for the moment GCRTKES only has a driver for the console to receive or send a character). For the i8086 they are written in assembler, on Linux and OpenBSD the context switch is done with `setjmp/longjmp` [18] and the interruptions of the real time clock are simulated with the alarm signal.

The preemption of the thread manager can be configured, the scheduler can be changed easily —although in the moment of this writing we have only implemented a first come first served strategy—, it is possible to sleep a task during certain time or to block it with a synchronization mechanisms: semaphores, locks and monitors.

GCRTKES offers its own API as well as a partial implementation of the interface "POSIX 1003.1c Threads Extension"[5], based on [26], [25] and [1] (currently it supports only initialization, termination, thread creation, mutex and condition variables).

2.2 Partial Evaluation

An informal definition of partial evaluation (also known as specialization) given in [28] is:

> Specialization is an approach which consists of adapting a generic program component to a given usage context. A specialization context can be composed of various things such as machine features, hardware or software configurations, values or properties of program inputs. Specialization is aimed at evaluating the computations which rely on the information provided by the usage context. Therefore, this process can lead to considerable performance gains.

In programming terms partial evaluation is the process of constructing from a program P with inputs x_1, x_2, ..., x_n, y_1, y_2,...,y_m another program $P_{X_1,X_2,...,X_n}$ with inputs y_1, y_2,...,y_m semantically equivalent to P with fixed inputs $x_1 = X_1$, $x_2 = X_2$,...$x_3 = X_3$. Theoretically, as expressed in [19], partial evaluation can be identified with the application of the S_m^n theorem of Kleene [17] or with the evaluation of expressions in Lambda calculus[5].

The field of partial evaluation has been under active development for several years resulting in tools for different languages[6] (e.g. Scheme, C, Java)

[5]In our research group there is also ongoing work on other APIs: RTX51 and OSEK.

[6]Some partial evaluators are listed in http://www.diku.dk/research-groups/topps/activities/PartialEvaluation.html and http://compose.labri.fr/prototypes/

and applications in several areas (e.g ray tracing [4] and software engineering [9]). There are even experiments of manual specialization and of automatic specialization before runtime (off-line specialization) and during runtime (on-line specialization) to sources of system software (see section 5).

For the C language we only know two tools for automatic off-line specialization: C-Mix and Tempo. Although there is already work in system software with Tempo (see the section 5), we have used C-Mix because for our experiments we prefer tools distributed along with their sources in order to fix or modify them when needed (in the moment of this writing the sources of Tempo are not available in the distribution site).

2.2.1 C-Mix

C-Mix has been developed by several people in the University of Copenhagen ([3], [4], [22]), the version available in the moment of this writing (2.0.12) is written in C++, it is well documented and there is enough support for the users from the developers. That version has some bugs[7] and limitations but gives enough flexibility and the sources to overcome them.

To specialize a program with C-Mix, the following data must be provided:

- a subset of the sources in C[8],
- the names of functions to specialize along with the parameters that must be evaluated partially (but not the values).
- optionally, annotations to specify binding time for some local and global variables, and certain features of functions and types not visible during specialization[9].

From this information, C-Mix transforms the program to a restricted version of C (Core C) and makes a binding time analysis. This analysis decides which portions of the expressions and statements of the sources will

[7] The workarounds are described in section 3.3.

[8] The version of C-mix available in the moment of this writing (2.0.12) supports a lot of C90 but not all, for example it doesn't support: certain function pointers as arguments in function calls (when the body of the function passed as argument is known by C-Mix and the function must be residual); variadic parameters; `setjmp`; `longjmp`.

[9] The annotations for C-mix can be done in the sources with the preprocessor directive `#pragma` or in a separate file with a simple syntax.

be needed during specialization (static) or during execution (dynamic), trying to maximize the static portion and satisfying the binding times specified by the user —when it is not possible C-Mix reports the problem.

Given a programm where all the expressions and statements have binding time annotations, doing a partial evaluation means evaluating the portions annotated as static and leaving the rest to be executed during runtime. Doing such evaluation is the role of a program P_{gen} or **generating extension** generated by C-Mix after the binding time analysis. P_{gen} is generated in C, when run, if the user specified one function to specialize, it receives as inputs the values for the parameters of the function to evaluate partially, executes the static portion and generates the dynamic portion also in C (the program generated is called **residual program**). If the user specified more than one function to specialize he/she must provide the function `main` of P_{gen}.

Optionally C-Mix can generate a description of the binding time analysis that can be consulted, for example with a browser, to understand some specializations (including viewing why a variable has certain binding time).

An example extracted from the sources of GCRTKES before and after specialization is presented in the figure 2, with the static portions in boldface(the specialized code shown was produced with C-Mix and an auxiliar tool presented in section 3.3.1).

The code shows the implementation of the `P` operation in a semaphore, the specialized version was obtained by evaluating partially GCRTKES along with the application presented in section 4.2[10], with the variability shown in figure 3 except tracing=No. Note that:

1. the calls to the functions `th_test_and_enter`, `TCB_setPrevState`, `TCB_setState` and `sem_addBlockedThread` were judged as dynamic by the BTA and inlined after specialization (they were not declared as inline functions),

2. the calls to the functions `assert` and `j_trace` are eliminated after specialization because assertion checking and tracing were turned off in the RDS,

3. the specialized version of `sem_wait` doesn't receive parameters, because the application with which the kernel was specialized used just one — through the global variable `lock_f` — thus during specialization all the

[10]The partial evaluation was repeated three times and the resulting code was transformed as we describe in the section 3.3.

Before

```
void
sem_wait(Semaphore *s)
{
  bool in=th_test_and_enter();
  assert(TCB_getState(th_current)
          ==stateRunning);
  j_trace(4,"sem_wait\n",0);
  s->val--;
  if (s->val < 0 ) {
       TCB_setPrevState(th_current,
                        stateRunning);
       TCB_setState(th_current,
                    stateSemBlocked);
       sem_addBlockedThread(s);
       th_reSched();
  }
  th_test_and_exit(in);
}
```

After

```
void
sem_wait(void)
{
  int insideKernelAtBegin;
  int iTmp1;
  struct sLList_S *psLtmp;
  iTmp1 = j_isTickInterruptEnabled();
  insideKernel=!iTmp1;
  j_tickInterruptDisable();
  lock_f.lock.s.val--;
  if (lock_f.lock.s.val < 0) {
     (th_current)->prevState = 1;
     (th_current)->state = 3;
     psLtmp=&(th_current)->ListElement;
     ll_insertAtEnd1(psLtmp);
     th_reSched();
  };
  th_test_and_exit(inKernelAtBegin);
}
```

Figure 2: Specialization of the primitive P of a semaphore in GCRTKES

functions that were judged as dynamic by the binding time analyzer and that received a semaphore as parameter, were transformed to use directly that semaphore.

The first transformation can be done by a compiler with help of the user (if he/she declares the functions as `inline`), the second could be achieved with the C preprocessor (with the differences that a macro and a function imply) but the third one is not perfomed by a classical optimizing compiler since it has to maintain the prototypes of the functions and because to achieve it, an intermodular analysis is required and traditional C compilers don't do this.

With the nature of the method used by C-mix to evaluate partially a program, it is necessary to restrict the binding time of some variables used in cycles or recursion to be dynamic: (1) variables of cycles that make many iterations or that appear in deep recursions must be dynamic to avoid code explosion (since cycles and recursion are "unrolled"), and (2) infinite loops should have as guard a constant variable annotated as dynamic (otherwise the execution of the generating extension will not terminate).

2.3 Restricted design spaces

To isolate the variability of a program along with some interdependences and to represent them graphically, our research group has been developing the notion of design space (originally presented in [31] and further developed, for example in [6], [14]).

The particular definition of design space that we are using in this work (restricted design spaces) as well as the supporting tools are described in [29]. A restricted design space (RDS) is a set of sentences in propositional logic with a restricted syntax that can be identified with a XML format and a graphical representation. The supporting tools are prototypes that allow us: to check the consistency of a RDS with a theorem prover (STeP [7]), to translate one representation into other, to browse the graphical representation and manage through it the variability of a program, and to map the RDS into the source code as well as into a configuration script.

To illustrate informally a RDS, the figure 3 shows the graphical representation of the RDS of GCRTKES.

The boxes (called dimensions) represent variabilities, each variability has a set of possible options that are mutually exclusive (just one can be used in a configuration). The arrows and the other nodes (diamonds and ellipses)

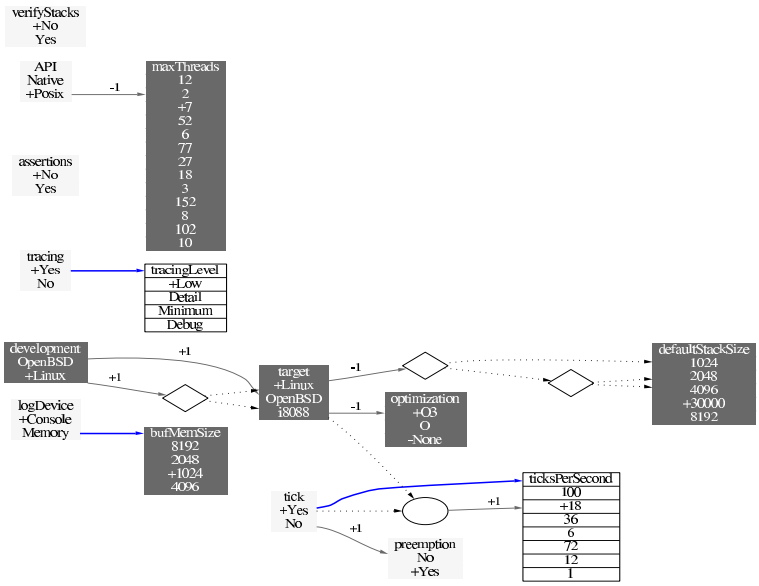

Figure 3: Restricted design space of GCRKTES

represent constraints. For example the arrow between "No" in the dimension "tick" to the option "No" in "preemption" indicates that if there is not a periodic interruption it is not possible to implement a preemptive scheduler, diamonds and ellipses represent "or" and "and" respectively. Thick arrows indicate dimensions that are required only under certain conditions, for example the size of a buffer for logging messages (`bufMemSize`) is only meaningful if the logging device is the memory. We say that the RDS of the example is fully profiled because in every dimension an option has been choosen.

2.3.1 Problem in previous definition and solution

We have found a problem in the definition of hierachy given in [29], since it would judge as inconsistent the RDS shown in figure 4 (two hierarchies on the dimension `C`, one from `A.a` and the other from `B.b1`), and however such configuration could occur in practice.

The condition for hierarchies proposed in [29] is

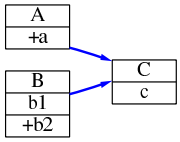

Figure 4: RDS considered inconsistent with previous definition

$$(e \rightarrow ExactlyOne(v_1, v_2, ..., v_n)) \wedge$$
$$(\neg e \rightarrow None(v_1, v_2, ..., v_n))$$

where $ExactlyOne(v_1, v_2, ..., v_n)$ is true iff there is an i $(0 \leq i \leq n)$ such that v_i is true while the others sentential symbols v_j (with $j \neq i$) are false, and $None(v_1, v_2, ..., v_n)$ is true iff for all i $(0 \leq i \leq n)$ v_i is false. We propose to relax it as:

$$(e \rightarrow ExactlyOne(v_1, v_2, ..., v_n)) \wedge$$
$$(\neg e \rightarrow None(v_1, v_2, ..., v_n) \vee ExactlyOne(v_1, v_2, ..., v_n))$$

This change doesn't affect the expressivity of RDS neither the decibility of the consistency nor its complexity. Besides making consistent the RDS of figure 4, the new definition allows to have hierarchies that go to dimensions with static binding time, as will be presented in the section 3.2.

3 Integration

To integrate the tools presented in the section 2, we have observed the following requirements in the sources of GCRTKES:

- Maintain compatibility with ANSI C (to use the sources without C-Mix).
- Encode each variabilities of the RDS with a global variable whenever it can be done.

The method used is depicted in the figure 5, where we wan to express that the whole kernel is configured with data of `config.h`, `config.c` and `inst.sh`, data that is modified by the restricted design space tools, and after the modifications all the sources (application along with kernel) are specialized and then compiled.

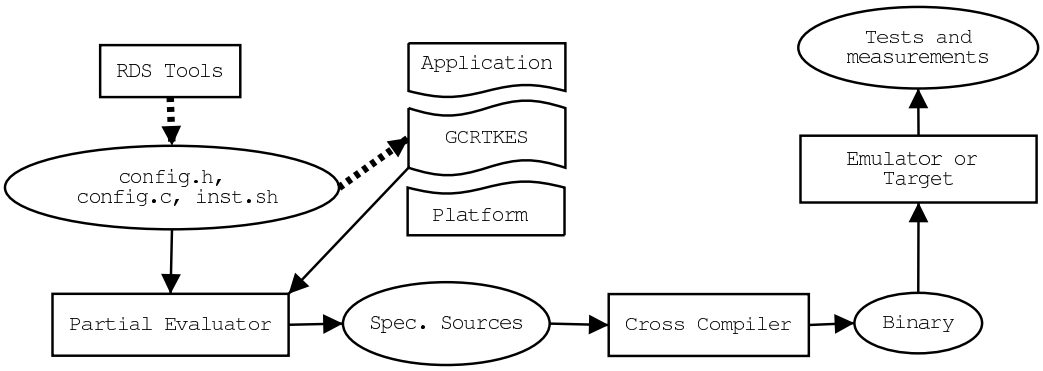

Figure 5: Integration of tools

3.1 Changes to the sources of GCRTKES

We have reimplemented the flexibility of GCRTKES depending only on:

- Some global variables and preprocessor macros[11] that are defined in

[11] We have tried to avoid using the preprocessor but in some cases it was indispensable, for example `target` that identifies the platform where GCRTKES is running, or `maxThreads` that indicates the maximum amount of threads. `maxThreads` must be a preprocessor macro, since it is used as size for a static array that must be allocated during compilation (in this case, using `const` is equivalent but not supported by dev86/bcc).

just one module: `config.h`[12] + `config.c` (see the figure 1).

- A shell script that handles variabilities that cannot be handled with C or with the preprocessor (e.g using the appropriate compiler, compilation flags and source files depending on the host and target platform).

To take advantage of partial evaluation, the binding time of the variables in `config.c` is annotated explicitly depending on the requirements of the platform and the application that will use the kernel. The variables that will not change during runtime are annotated as `spectime` and the ones that can change as `residual`[13]. this gives the following classification for the variabilities in the sources:

- static (implemented as preprocessor macros),
- specializable static (implemented with variables annotated as `spectime`)
- specializable dynamic (implemented with variables annotated as `residual`)

To give an idea of the effort required to specialize the sources we present some syntactical metrics (measured over headers, modules and regression tests of GCRTKES):

- Lines with code: 4684
- Lines with comments: 1942
- Lines with directives for specialization: 68

[12] Although the default name for the configuration header used by Autoconf [35] is also `config.h`, the contents of that file in our case as well as our approach are different. Autoconf is suited for static configuration of applications, it relies on the C preprocessor and detects automatically features of the hosting platform, our approach is suited for dynamic and static configuration producing only the dynamic part of a program and precomputing the static one (however we still have not worked on automatic detection of features, and the approach of Autoconf is very appropriate for that).

[13] This approach is not equivalent to using `const`, since a constant variable cannot be assigned in any function, while with Cmix it can be assigned and during the binding time analysis it will be checked that the assignment doesn't occurr during runtime.

3.2 Extensions to restricted design spaces

Each variability in the sources has been identified with a dimension in the RDS, giving place to "binding time" as an extension for the concept of RDS (see section 3.2).

- static: the variability cannot change during runtime
- dynamic: the variability can change during runtime
- dynamic static: in the sources it is treated as dynamic but because of the requirements or the platform it will be considered static.

We have implemented this extension only in the tools (graphical browser and translators) but no yet in the formal definition.

The graphical representation has been extended by filling the static dimensions with a color (dark grey for static, light gray for dynamic static and white for dynamic, see figure 3), the user can change with a mouse click between dynamic and dynamic static; in the XML representation we have added an attribute `binding` to the dimensions (it can be `static`, `dynamic` or `dynstat`); in the translation to CML2 the options in dynamic dimensions are treated like modules[14]; in the translation to a shell script we have added variables `var_`*id*, where *id* is the identification of a dimension, that can be: 0 if the dimension is static or dynamic static and 1 if it is dynamic.

To map into code, the annotations and values of variables and preprocessor macros in `confing.c` and `config.h` are changed by a tool[15] in accordance to the RDS. Analogously a tool generates a shell script `inst.sh`, with variables that indicate the option selected in each dimension, and a sequence of `if` statements that semidecides[16] the consistency and describes eventual problems (if it judges a RDS as inconsistent it is sure there are inconsistencies).

[14]The notation in CML2 is designed to configure parts of the Linux kernel, a part of the kernel can be left out (n), compiled along with the kernel (y) or linked during runtime as a module (m).

[15]Other possible way would be adapting the generator extension produced by C-Mix to receive the values of the variables associated with the RDS.

[16]In general deciding the consistency of a RDS is a NP-complete problem as shown in [29], however when it is fully profiled and each dimension has one option selected, checking the consistency means simply to verify that the hierarchies and correlations hold, and that can be accomplished in time $O(n)$ where n is the amount of connectives in the correlations of the RDS (taking as basic operation the evaluation of a connective over atomic values)

In the extension to the formal definition that must be done, we consider that the following informal requisite should be observed:

> The restricted design space must be consistent during runtime (except during changes in the configuration)

To verify this property during runtime we have implemented a translator from RDS to a function in C that decides if the configuration is valid with a sequence of `if`. The function generated by the new translator (we call it **configuration invariant**) can be used to check during runtime the viability of a change in the configuration, or to make sanity checks (for example checking preconditions with `assert`).

3.3 Changes to specialization

We have not implemented profund changes to this procedure but just invented tricks to overcome bugs or improve the code generated by C-Mix:

- Minor bugfixes in the code generated by C-Mix.
- By using the preprocessor and by renaming before and after specialization some variables, we could use `setjmp`, `longjmp` as well as pointers to residual functions whose body is known by C-Mix (we have not implemented the specialization that these constructions require).
- We use tools at lexical level to leave certain functions in the sources after specialization (C-Mix allows it but requires manual modification of the function `main` in the generating extension). These tools remove also unnecessary wrappers generated for those functions in the residual program.
- Since in just one pass the partial evaluator cannot specialize completely some functions, we are feeding C-Mix with its own output two or three times. To support this we use tools that work at lexical level, reinserting the annotations after each pass and preparing them to be specialized again[17].

[17]We have prefered not to specify the annotations in a separate file since: 1) It was easier for us to maintain the annotations near the code affected, 2) some annotations must be changed after the first specialization (e.g variables annotated as `spectime`) hence even a separate file with annotations would have to be modified before a second specialization.

- After specialization some unnecessary variables are introduced. For system software the usage of the stack is very delicate and in embedded systems a very limited memory is the rule. To overcome this we have implemented two solutions: (1) not specializing some functions (the entire "hal" and the function to start a thread[18]) and (2) a tool that analyzes syntactically and modifies the output produced by C-Mix to remove some unnecessary variables (see section 3.3.1).

We consider that besides integrating the mentioned features directly into C-Mix[19], maintaining the structure of the original code as much as possible is a desirable enhancement for it —since the programmer can use certain constructions that work better for the available compiler[20]. Other features that could be useful for system software include preserving the original stack usage in some functions and specializing some fields in structures (for example by removing unused ones).

3.3.1 Removing some unnecessary variables

For its operation C-Mix translates the sources that receives to a subset of C called "Core C" [22], in the process some temporal variables are introduced to avoid lateral effects, and some constructions are replaced with others (for example cycles and conditionals to jumps and a restricted conditionals). After the partial evaluation C-Mix tries to convert some jumps and conditionals to `if` and `while`, resulting in functions that use only the following statements:

- assignment of an expression that doesn't produce lateral effects to a variable or to an address,
- function call or assignment of the returning value of a function call,
- `goto`, `label`, `return`, `if`, `while`

[18] We still can refine more this, since the portion to leave unspecialized depends on the platform (in 8088 the entire HAL cannot be specialized with C-Mix, but in Linux and OpenBSD probably part of it can be specialized).

[19] In the moment of this writing, just one bug fix has been made to C-Mix 2.0.12 (such patch will be included in future versions of C-Mix).

[20] C-Mix tries to restructure the code after the specialization to make it more readable e.g. removing unnecessary labels and translating certain `if` and `goto` to `while`. However the resulting code doesn't use, necessarily, the constructions of the initial sources.

During these transformation C-Mix introduces new intermediate variables, that in several cases are not necessary. For example, the following code is produced by C-Mix after the specialization of the P operation presented in the left side of figure 2:

```
void
sem_wait(void)
{
  int iTmp;
  int inKernelAtBegin;
  short sTmp;
  int insideKernel;
  int iTmp1;
  struct sThreadCB *tcb;
  struct sThreadCB *tcb1;
  struct sThreadCB *tcb2;
  struct sLList_S *psLTmp;
  struct sThreadCB *tcb3;
  int a;
  iTmp1 = j_isTickInterruptEnabled();
  insideKernel =  !iTmp1;
  j_tickInterruptDisable();
  iTmp = insideKernel;
  inKernelAtBegin = iTmp;
  tcb = th_current;
  sTmp = tcb->state;
  a = (int )sTmp == 1;
  lock_f.lock.s.val = lock_f.lock.s.val - 1;
  if (lock_f.lock.s.val < 0) {
    tcb1 = th_current;
    tcb1->prevState = (short )1;
    tcb2 = th_current;
    tcb2->state = (short )3;
    tcb3 = th_current;
    psLTmp = &tcb3->ListElement;
    ll_insertAtEnd1(psLTmp);
    th_reSched();
  }
  th_test_and_exit(inKernelAtBegin);
  return ;
}
```

A common pattern in the code generated by C-Mix is to introduce variables that are assigned just once and are not used or used once in the function's body. Note that in the example the only local variable required is

`iTmp1` and the developed tool eliminates almost all the unnecessary ones as shown in the right side of figure 2.

To eliminate automatically some of the unnecessary variables we developed a tool[21] in Ocaml that parses a source generated by C-Mix[22], analyzes each local variable in each function and removes those that are judged as unnecessary. This simple enhancement gives some gains in performance, in the absence of a highly optimizing compiler, as compared in the columns 'S-' and 'S' of the table 3 (the example of this section was taken from the application measured in that table).

3.3.2 Basis for the elimination of unnecesary variables

To analyze the situation, we consider the following piece of abstract code:

s_1;
...
s_p;
$v = E$;
s_{p+1};
...
s_q;
$s[v]$;
s_{q+1};
...
s_n;

assuming there is only one assignment of v (line with $v = E$;) and only one usage in $s[v]$, we could remove it and replace the usage with E i.e $s[v]_E^V$ if we are sure the semantics is maintained. If a variable is used twice or more, we don't judge it as unnecessary (it could be more efficient in time to leave it).

Definition 3.1. *In the function body of a program produced by C-Mix a variable is called unnecessary if it is assigned exactly once, it is used at most*

[21] The tool developed is availabe along with other sources that sustain this report at: http://wwwagss.informatik.uni-kl.de/Mitarbeiter/tamara/index.html

[22] The parser accepts almost the entire ANSI C 90 (except full support for preprocessor directives) but the analysis, at least, doesn't support some lateral effects (e.g `a=b=1`) and some non-constant initialization during declaration (e.g `int a=var`).

once and if its eventual usage can be replaced with the assigned expression maintaining the semantics of the function.

To semidecide which variables are unnecessary the developed tool checks the conditions of the following theorem —these conditions could be more elaborated although in their current form remove several unnecessary variables introduced by C-mix.

Theorem 3.1. *In a function body produced by C-Mix where an expression E is assigned once to a variable v and the value of v is used at most once, if all the following conditions hold, the variable is unnecessary:*

- *E is not a function call,*
- *if v is used, E doesn't contain neither dereferences (arrays or operators* `* ->`*) nor volatile variables,*
- *if v is used its usage occurs in a statement in the same block of the assignment. Such statement is an assignment or a conditional (*`if`*,* `switch`*,* `case`*) that has only conditionals and assignments.*
- *v is not used or all the statements between the assignment and the usage are of the form:*
  - *An assignments $w = F$ such that: w is not a dereferenced pointer neither v neither a function call and it doesn't appear as subexpression in E.*
  - *A statement* `if (`F`)` S *where S is a block of statements where these conditions hold also.*
  - *An statement* `if (`F`)` $S1$ `else` $S2$ *where $S1$ and $S2$ are blocks of statements where these conditions hold.*
  - *A statement* `while (`F`)` S *or* `do` S `while (`F`)` *where S is a block of statements where these conditions hold.*
  - *An statement* `for (`$S1$`;`F`;`$S2$`)` $S3$ *where $S1$, $S2$ and $S3$ are blocks of statements where these conditions hold.*
  - *A statement* `do` S `while (`F`)` *where S is a block of statements where these conditions hold.*
  - *A statement* `case` F `:` S *where S is a block of statements where these conditions hold.*

– *A statement* `switch (`F`)` S *where* S *is a block of statements where these conditions hold.*

Proof. The first and second conditions ensure that E doesn't produce lateral effects and that its value in the usage point doesn't depends on values that can change outside the sequential control flow neither on memory dereferenced (memory that could be altered by other variables or pointers). The third and fourth condition ensure that the control flow between the assignment and the expression where v is used don't have jumps neither function calls nor can be reached from a jump in another part of the function. Hence the value of v could change only in one of the statements between the assignment and the usage as seen syntactically. Given that the assertion $\{V = E\}$ holds after the assignment we only have to check that for each statement s_i appearing between the assignment and the usage: $\{V = E\}s_i\{V = E\}$. To prove it we use structural induction on the statements s_i, the base case is the assignment and the induction hypotheses are $\{V = E\}S\{V = E\}$, $\{V = E\}S1\{V = E\}$ and $\{V = E\}S2\{V = E\}$ (where S, $S1$ and $S2$ are statements):

- If s_i is an assignment $w = F$, where w is a variable and not a dereferenced pointer. Since F doesn't produce lateral effects (because it is not a function call and C-mix generates expressions without lateral effects), it can only change the state of execution in the variable w. Then we can use as postcondition $\{v = E\}$ and the assignment rule of Hoare calculus [11]: $\{(v = E)^w_F\}w = F\{v = E\}$. Since v is not w and w doesn't appear in E we have $(v = E)^w_F \equiv (v = E)$.
- We want to prove $\{V = E\}$ `if ( `F` ) { `S` }` $\{V = E\}$ since F doesn't produce lateral effects (C-Mix generates function calls only in assignements) we need $\{V = E \wedge F\}S\{V = E\}$ and $\{V = E \wedge \neg F\}S\{V = E\}$ both true since $(V = E \wedge F) \rightarrow V = E$ and by induction hypotheses $\{V = E\}S\{V = E\}$.
- The other possible statements can be proved analogously to the previous one.

□

4 Measurements

The measurements presented in this section were done on the following platforms:

- Linux: Pentium II/450 Mhz with 128MB in RAM, 112MB in swap, running Debian 2.2 (kernel 2.2.18) as a user. GCRTKES was configured as shown in figure 3 and it was compiled with gcc 2.95.2 (optimization=O3).
- 8088: with Xmess we emulated[23] a 8088 of 4MHz and 640K in RAM (although currently GCRTKES uses at most 64K on this platform). GCRKTES was configured as presented in figure 3, except because target=8088, optimization=O, defaultStackSize=2048 and maxThreads=12. As compiler we used bcc 0.15.4.
- OpenBSD: Pentium II/450 Mhz with 128MB in RAM, 112MB in swap, running OpenBSD 2.8. GCRTKES was configured as shown in figure 3 except host=OpenBSD, target=OpenBSD. The compiler is gcc 2.95.3 with optimization -O3.

4.1 Microbenchmarks

The results of some measurements inspired from [25] are presented in tables 1, 2 and 3. For each supported platform we present the amount of times that each test could be repeated in one second[24] with and withouth specialization, as well as the ratio of the specialized over the not specialized version[25].

The metrics were: calling an empty function several times (control enters to the kernel and to a null thread periodically); locking and unlocking a mutex withouth and with contention (in the case of contention two threads share the lock and one has to wait for the other to release it); giving control to null thread; creation of a thread along with switching to it and deleting it.

[23] The measurements presented were taken on an emulator but we verified that they run also on real hardware.

[24] To measure the time, we used the internal counter of GCRTKES that relies on the clock interruption (i.e hardware tick interruption in 8088 and signals on Linux and OpenBSD).

[25] Each measurement was repeated 5 times the data presented for the non-specialized version corresponds to the best performance, the data presented for the specialized case corresponds to the worst performance in the 5 measurements.

Metric	NS [t/s]	S [t/s]	S/NS
empty	11791077	15309889	1.2
lock/unlock	207263	235803	1.1
shared mutex	13125	14812	1.1
yield	184667	204968	1.1
thread	32771	36591	1.1
size	26536	14140	1.8

Table 1: Microbenchmarks for Linux (measured in tests per second).

Metric	NS [t/s]	S [t/s]	S/NS
empty	10412388	14366457	1.3
lock/unlock	185609	213299	1.1
shared mutex	8207	9065	1.1
yield	93240	102409	1.1
thread	20107	22052	1.1
size	36864	24576	1.5

Table 2: Microbenchmarks for OpenBSD.

Metric	NS [t/s]	S- [t/s]	S [t/s]	S/NS
empty	3733	4193	4504	1.2
lock/unlock	748	1134	1314	1.6
shared mutex	296	32	36	1.4
yield	56	416	473	1.4
thread	23	77	87	1.4
size	25528	10668	8792	2.9

Table 3: Microbenchmarks for i8088.

	Time [s]			Size [b]			Comp. Time [s]		
Products	I	C	Ratio	I	C	Ratio	I	C	Ratio.
1	.03	.01	3.00	26312	17724	1.48	2.83	43.77	.06
5	.16	.18	.88	26408	17884	1.47	2.82	43.75	.06
10	.32	.18	1.77	26376	17820	1.48	2.86	44.06	.06
25	.83	.46	1.80	26376	17820	1.48	3.00	50.01	.05
50	1.63	.90	1.81	26472	17916	1.47	2.95	45.86	.06
75	2.54	1.41	1.80	26472	17916	1.47	2.90	44.83	.06
100	3.61	1.82	1.98	26376	17884	1.47	3.15	49.60	.06

Table 4: Products in Linux.

The differences in the ratios from one platform to other is consequence of compiler and hardware optimization (Dev86/bcc doesn't support many optimization strategies that gcc implements).

4.2 Product of Laurent series

To calculate the response of some discrete regulation algorithms used in embedded systems [24] and to calculate the response of FIR and IIR filters via the Z transform, it is required to calculate products of Laurent series of the form $H(z) = \sum_{i=0}^{N} h_i z^{-i}$, which is analogous to multiply two polynomials.

We have implemented a program to compute simultaniously several products of Laurent series with integer coefficients and we used it to measure the performance of the specialization. Each product is computed in a thread, the operands of each product are a Laurent serie with arbitrary coefficients computed during runtime (the signal) and a Laurent serie of 13th grade with fixed coefficients (the filter).

The table 4 presents measurements[26] on Linux for several amounts of simultaneous products, using a signal of length 100.000. The figure 6 shows only the running time of the specialized and the non-specialized versions. Note that the filesize also decreased in the specialized version.

[26]In that table Comp. Time is the time required for generating the program from the sources i.e for the non-specialized case it corresponds to compiling, for the specialized case it corresponds to specializing and and compiling.

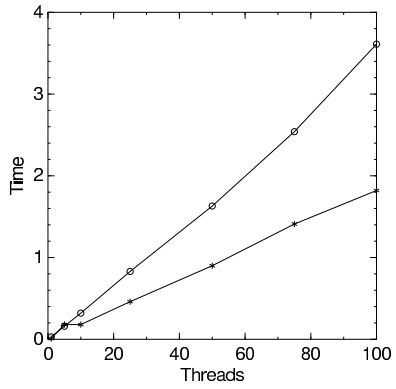

Figure 6: Speed of products in Linux

	Time [s]			Size [b]			Comp. Time [s]		
Prod.	I	C	Ratio	I	C	Ratio	I	C	Ratio
1	.63	.40	1.57	25784	20948	1.23	2.76	46.29	.05
4	2.40	1.50	1.60	25784	20948	1.23	1.16	42.28	.02
6	3.57	2.24	1.59	25760	20908	1.23	1.16	42.10	.02
8	4.82	3.05	1.58	25784	20948	1.23	1.15	42.30	.02
10	6.04	3.84	1.57	25784	20948	1.23	1.15	43.20	.02

Table 5: Products in i8088.

Analogous measurements for the processor i8088 with a signal of length 300 are presented in the table 5, the running times are compared graphically in the figure 7.

The measurements under OpenBSD are presented in the table 6 and figure 8 (the case of 100 products didn't fit in memory running the test as a normal user).

Since this application is appropriate for specialization (because the filter is static), the ratios are better than the ratios of the microbenchmarks.

4.3 Proxy server

To test GCRTKES as a possible user level library of POSIX threads we have implemented a proxy server for TCP/IP. This program mediates in

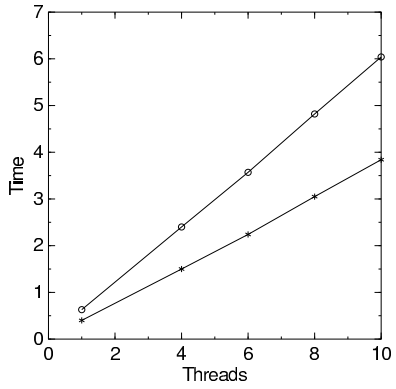

Figure 7: Speed of products in i8088

	Time [s]			Size [b]			Comp. Time [s]		
Prod.	I	C	Ratio	I	C	Ratio	I	C	Ratio
1	.08	.03	2.66	36864	28672	1.28	5.57	68.04	.08
5	.22	.12	1.83	36864	28672	1.28	5.04	67.83	.07
10	.39	.21	1.85	36864	28672	1.28	4.98	67.90	.07
25	.90	.49	1.83	36864	28672	1.28	5.04	69.29	.07
50	1.77	1.00	1.77	36864	28672	1.28	5.13	68.64	.07
75	2.66	1.52	1.75	36864	28672	1.28	5.52	70.20	.07

Table 6: Products in OpenBSD.

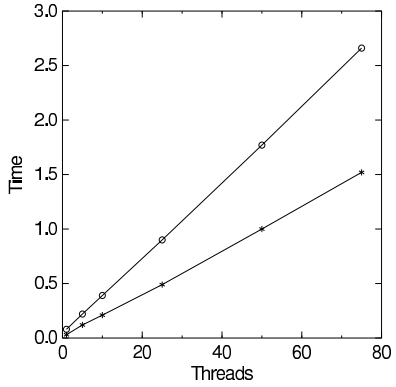

Figure 8: Speed of products in OpenBSD

the communication between a client and a server, forwarding all the data produced by the server to the client and all the data produced by the client to the server. The proxy uses the usual primitives for sockets of the C library[27].

The speed of different variants of the proxy running on the same hardware[28] are presented in the tables 7 and 8.

Threads	Time [s]	LinuxThreads	NS	S	R
1	10	.019	.031	.030	1.033
5	30	.103	.123	.118	1.042
10	50	.216	.237	.223	1.062
20	100	.435	.468	.437	1.070
30	150	.718	.711	.662	1.074
40	200	.967	.956	.901	1.061

Table 7: Proxy speed on Linux (measured in seconds per Kbit).

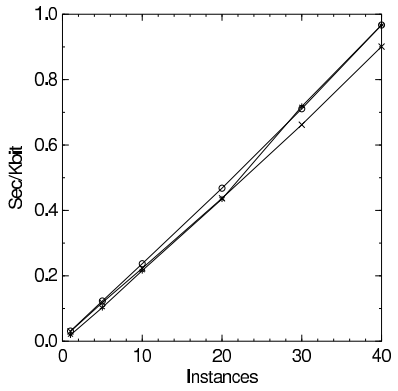

Figure 9: Proxy speed on Linux.

In the first table, the 'LinuxThreads' column correspond to the proxy run-

[27]Since the clock interruption of GCRTKES on Linux is simulated with the timer signal we wrapped the blocking primitives for sockets with cycles that check the possible error code returned and repeat the operation if it is `EINTR` —this relies on POSIX semantics but it is not required in BSD.

[28]To avoid lateral effects produced by a real network or protocol, we have implemented a test protocol with client and server inside the proxy (the server serves consecutive characters and a client receives and verifies them while measures speed).

ning on top of the POSIX threads library of Linux [20] using blocking I/O and a preemptive scheduler, the column 'NS' for the proxy on top of GCRTKES with non-blocking I/O and collaborative[29] and preemptive scheduling, the column 'S' is the same case as GCRTKES but with application specialized along with GCRTKES (the scheduling and blocking policies give the best results we could achieve in each case).

The performance of GCRTKES compared with LinuxThreads is typical of user level threads packages as noted in [8] and [25].

Threads	Time [s]	NS	S	R
1	10	.062	.061	1.016
2	15	.120	.117	1.025
3	20	.178	.175	1.017
5	30	.293	.287	1.020
10	50	.582	.574	1.013

Table 8: Proxy speed on OpenBSD (measured in seconds per Kbit).

Note that this application is not suited for specialization since all its data is known at runtime, the small improvement of the specialized version over the non-specialized one is gained when the control enters routines of GCRTKES that are specialized. The collaborative scheduling —usage of `sched_yield()`— and the nonblocking I/O in the variant with GCRTKES plays a decisive role, since GCRTKES is a user level threas library that uses the alarm signal to preempt threads, and blocking primitives would block all the threads during a timeslice of the alarm signal.

[29] Each thread gives the processor to other thread with `sched_yield()`.

5 Partial evaluation for system software

The articles[30] and projects about partial evaluation applied to system software that we know of are:

- [10] where the idea of incremental partial specialization is presented (the code is specialized in several stages: compile time, link time, boot time, run time) based on the experience with the kernel "Synthesis" which had hand-coded templates that generated specialized routines. The project Synthetix was a descendant of Synthesis, where manual specialization (static and dynamic) was applied to part of the filesystem of HP-UX [30].
- In the thesis [33], the author presents a study of two cases of specialization of real system software: Sun RPC and IPC of the Chorus Microkernel as well as a declarative method to specify specialization opportunities for Tempo.
- [27] and [34] could be considered predecessors of [33], in the second one in addition to an introduction to the specialization of Sun RPC, a technique to apply specialization for copying buffers is presented.
- In [32] a transition between manual specialization and automatic specialization with Tempo is described. In accordance to the authors Tempo has some features that are not implemented in C-Mix (e.g partially static data structures, use sensivity, runtime specialization), and in accordance to [27] it has been targeted towards system software.
- [23] presents tools to implement on-line and off-line specialization for operating systems based on Tempo, as experiments presents the specialization of Sun RPC, BSD Packet filters and the implementation of signals in Linux.
- Some of the articles here referenced (as well as Tempo), were developed in the context of the Compose[31] project "Design and Development of Adaptive Programs and Systems", which is active and focused in system software.

[30] Two sites in Internet dedicated to partial evaluation are http://compose.labri.fr/documentation/papers/ and http://www.diku.dk/research-groups/topps/activities/PartialEvaluation.html

[31] http://compose.labri.fr/

6 Conclusions

With the current version of GCRTKES specialized with C-Mix and with the examples tried, the effort of the specialization and the increment in "compilation" time was compensated with the size and speed of the final binaries. Although in some situation a classical highly optimizing compiler is enough, we consider an off-line partial evaluator specially good when the application is appropriate for specialization (i.e it has a considerable static part) and when it can be fully integrated with the kernel —as it usually occurrs in embedded systems.

Managing flexibility with partial evaluation can remove, in several cases, the difference between static and dynamic variabilities from the implementation point of view, allowing the programmer to treat the variabilities as dynamic and still achive good performance. We consider that the design of the kernel can be flexible and well structured in layers (or ADTs) using the constructions of the language withouth big penalties in performance, since the layers will be integrated (if desired along with the application) through partial evaluation.

In the practice, we extended restricted design spaces with binding times, making it possible to map the variability expressed in a RDS to the sources of a program specializable with C-Mix.

Although other experiences have invested efforts in specializing portions of existing and well known kernels that are much more complex and mature than GCRTKES, we consider that our experience has new features: application of automatic partial evaluation to a small kernel with an application on top of it, use of C-Mix for system software, simple comparision of specialization of the same code on different platforms.

Improvements in the tools are required as well as a formalization of the binding time extension for restricted design spaces. Possible improvements:

- C-Mix: integration of the workarounds used for this work (specially to avoid the generation of unnecessary variables).
- RDS: integration of RDS of application with RDS of kernel and support to visualize implications of a configuration in the performance of the final program.
- GCRTKES: porting and testing on a typical embedded processor (i.e i8051), using different scheduling strategies and adding functionality.

References

[1] Posix.1c/d10 summary. Technical report, Sun Microsystems, http://www.npac.syr.edu/projects/pcrc/cpswt-e/CPSWT.106/106_2/pthread.html, 1995.

[2] Dragos Acostachioaie. *Doc++: A documentation System for C, C++ and Java.* http://docpp.sourceforge.net/, 2001.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[4] Peter Holst Andersen. Partial evaluation applied to ray tracing. Unpublished, 1993.

[5] H.P. Barndregt. *The Lambda Calculus: Its syntax and semantics.* North Holland Publishing, 1984.

[6] Lothar Baum, Lars Geyer, Georg Molter, Steffan Rothkugel, and Peter Sturm. Architecture-centric software development based on extended desing. In *Second Internation Workshop on Development and Evolution of Software Architectures for Product Families*, 1997.

[7] Nikolaj S. Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, 16(3):227–270, June 2000.

[8] W. Cohen, C. Patel, and A. Seshagiri. Cost of user and kernel level threads operations on Linux, 1998.

[9] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. Partial evaluation for software engineering. *ACM Computing Surveys (CSUR)*, September 1998.

[10] C. Consel, C. Pu, and J. Walpole. Incremental partial evaluation: The key to high performance, modularity and portability in operating systems, 1993.

[11] Edsger W. Dijkstra and W.H.J. Feijen. *Methodik des Programmierens.* Addison-Wesley Verlag, 1985.

[12] Burce Evans. *C compiler.* http://www.cix.co.uk/ mayday/, 2001.

[13] David Evans. Annotation-assisted lightweight static checking. In *First International Workshop on Automated Program Analysis, Testing and Verification (ICSE 2000)*, 2000.

[14] Andreas Gilbert. Design space-basierte knofiguration des betriebssystems qnx. Master's thesis, Fachbereich Informatik, Universität Kaiserslautern, 1999.

[15] Marcel Ianizzi. Generisch betriebsystemimplementierung am beispiel eines realzeit-schedulers. Master's thesis, Universität Kaiserslatern, 1998.

[16] Peter Sturm Jürgen Nehmer. *Systemsoftware: Grundlagen moderner Betriebssysteme*. Dpunkt Verlag, 1998.

[17] S.C. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[18] T. Kramp and R. Koster. Flexible event-based threading for QoS-supporting middleware. In *Proceedings of the Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*, July 1999.

[19] Ksheerabdhi Krishna. Specializing code via partial evaluation. Technical Report Technical Report No. 94-06, The University of New Mexico, ftp://ftp.cs.unm.edu/pub/krishna/CS94-06.ps.Z, 1994.

[20] X. Leroy. The linux threads library, 2001.

[21] M. Lösch. Beschreibung generischer komponenten mit xml. Technical report, AG Systemsotware, Universität Kaiserslautern, 2000.

[22] Henning Makholm. Specializing C: an introduction to the principles behind C-Mix/II, 1999.

[23] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transaction on Computer Systems*, (2), May 2001.

[24] R. Merz and L. Litz. Regelungstechnisches tutorial. Technical Report 06/2000, Sonderforschungsbereich 501, Teilprojekt D2, Universität Kaiserslautern, 2000.

[25] Frank Mueller. A library implementation of POSIX threads under Unix. In *Proceedings of theWinter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, USA, 1993.

[26] Frank Mueller. Pthreads library interface. Technical report, Institut für Informatik, Humboldt-Universität zu Berlin, 1999.

[27] G. Muller, R. Marlet, E.-N. Volanschi, C. Consel adn C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceeding of the 19th IEEE International Conference on Distributed Computing Systems*, 1998.

[28] G. Muller, C. Pu, and C. Consel. *Encyclopedia of Distributed Computing*, chapter Specialization. Kluwer Academic Publisher, 1999.

[29] Vladimir Támara Patiño. Restricted design spaces: Visualization and consistency tools. Technical Report 03/01, Sonderforschungsbereich 501, Teilprojekt B5, Universität Kaiserslautern, http://www-madlener.informatik.uni-kl.de/publications/2001/sfb501-rds.ps.gz, 2001.

[30] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, 1995.

[31] Lane T.G. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18, Carnegie Mellon University, 1990.

[32] E.N. Volanschi, G. Muller, and C. Consel. Safe operating system specialization: The RPC case study, 1996.

[33] Eugen N. Volanschi. *Une approche automatique à la spécialisation de composants système*. PhD thesis, l'Université de Rennes 1, February 1998.

[34] Eugen-Nicolae Volanschi, Gilles Muller, Charles Consel, Luke Hornof, Jacques Noyé, and Calton Pu. A uniform and automatic approach to copy elimination in system extensions via program specialization. Technical Report 1021, IRISA. Institut de Recherche en Informatique et Systémes Aléatories, June 1996.

[35] Erez Zadok. *Autoconfiscating AMD: Automatic Software Configuration of the Berkeley Automounter*. http://www.cs.columbia.edu/ ezk/research/amu/autoconfiscating_amd.html, 1998.