TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

Design And Implementation of the GNU INSEL-Compiler gic

Markus Pizka

TUM-I9713
SFB-Bericht Nr. 342/09/97 A
April 97

TUM–INFO–04-I9713-100/1.–FI

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

Design and Implementation of the GNU INSEL-Compiler *gic*

Markus Pizka
Technische Universität München
Institut für Informatik
pizka@informatik.tu-muenchen.de

Abstract

The syntax of the object–based language INSEL is derivated from abstract and formal concepts developed in a language–based and top–down oriented approach to construct distributed systems. The concepts of INSEL serve as the starting point for all resource management steps required to transform the source code into an efficient running systems. A language–based approach allows to tailor the resource management system to the language concepts. This in turn allows to automatically exploit application specific properties based on the language concepts and therefore improves efficiency. Obviously, the success of such an approach highly depends on the abilities of the compiler to extract language-level properties and exploit the analyzed information to transform source code into an efficient target representation.
In contrast to comparable projects and due to experiences with prototypes, the INSEL compiler *gic* does not use an existing high–level language such as C as an intermediate language but interfaces with a modified version of the well–known GNU C compiler *gcc*. This report describes the architecture of the compiler and provides important information on the interfaces of *gcc*. Syntax processing and most parts of semantic checking is accomplished by a well structured INSEL front-end. The internal representations "'RTL" and "trees" of the GNU C compiler are used to transform abstract INSEL syntax trees in a structured and flexible way into the target representation.
This strategy allows the construction of a fast, portable and optimizing compiler, provides reusability of existing tools such as debuggers and allows for the flexibility needed in our research project without the necessity to reinvent and re-implement existing and successful techniques.

Contents

1 Introduction

Currently, a broad spectrum of research activities is focusing on the transition from sequential and centralized processing to distributed, parallel and cooperative computing. To support the construction of complex but high-quality and efficient distributed systems, appropriate software environments have to be provided. These environments have to fulfill at least two somehow contradictory goals. On the one hand, they should significantly ease distributed programming by hiding as many details of the distributed nature of the hardware configuration as possible. On the other hand, performance has to be enhanced by providing adaptability and scalability without introducing distinct management overhead.

New resource management systems, comprising languages and software such as compiler, linker and operating system (OS) kernels are required to meet these requirements. We argue that the implementation of these tools does not have to start from scratch. Existing software can be modified to meet the demands of distributed computing [PE97a].

To provide the desired simplicity of distributed programming we chose a language-based approach. The programming language INSEL provides concepts [SEL$^+$96] to construct parallel and cooperative applications on a high-level of abstraction. The distributed nature of the execution environment is completely hidden for the programmer. The development of a new programming language supporting parallelism and cooperation eases distributed computing significantly by transferring the task of resource management completely to the system level encompassing the OS and management tools.

Therefore, the importance of the construction of software tools is twofold. First, their implementation demands tremendous efforts. Second, the quality of the tools determines the success of the system. This report demonstrates, that by modifying but basically reusing an existing compiler both aspects can be addressed to develop a high-quality compiler with acceptable effort.

1.1 Programming Language INSEL

INSEL [RW96, Win96] provides language concepts to develop distributed applications without knowledge about details of the underlying distributed hardware configuration. It is a high-level, type-safe, imperative and object-based programming language, supporting explicit task parallelism.

INSEL objects support encapsulation and can dynamically be created during program execution as instances of class describing objects, called "generators". To prevent dangling pointers, objects are automatically deleted according to a conceptually defined life-time [PE97b]. In contrast to class concepts known, as for instance in C++ [Str91] generators are integrated into the system in the same way as other objects and can be nested within other generators or instances and vice versa.

The generator also defines whether objects created as instances of this generator are active, called "actors" or passive ones. An actor defines a separate flow of control and performs concurrently to its creator. Actors are dynamically and explicitly created in the path of computation just like any other (passive) object without any references to the execution environment, such as a specific node or virtual memory address. Concurrency being a language and class property has important advantages relative to pure OS or runtime concepts such as multi–threading. It eases the task of static analyzes of the flow graphs and allows the compiler to generate dedicated code for actor classes to support concurrency at runtime.

INSEL objects may communicate directly in a client-server style (message passing paradigm) as well as indirectly by accessing shared passive objects (shared memory paradigm). All requests to objects are served synchronously. In addition to the concepts listed in table 1.1, INSEL also provides common building blocks known from other imperative languages, such as loops, case statements and blocks.

concept	**performs**	**comment**
m–actor	active	concurrent yet synchronized subprogram
c–actor	active	object performing its canonic operation concurrently
c–order	passive	procedure performed synchronously by two actors in a rendezvous
ps/fs–order	passive	procedure/function
depot	passive	containers that might serve as typed modules or data objects with [a]synchronized access orders

Table 1.1: Major concepts of INSEL

Arguments are passed either `IN`, `OUT` or transient `INOUT`. The semantic of `IN` is "copy-in" [ASU88a] and `OUT` determines to "copy-out" the results to the caller on return of the subprogram. `INOUT` therefore determines "copy-restore" semantics. In contrast to "call-by-reference" the concept of `OUT` parameters warrants, that the values of arguments passed between sequential and concurrently executing computations are always well-defined either holding the value before or after the call. Furthermore, concurrent computations do not interfere unexpectedly because of `OUT` parameters being passed.

All components (actors, orders, depots, simple data objects and generators) of an INSEL system are "elaborated" at the time computation reaches their declaration. Elaboration can be regarded as fixing the properties of the component and has to be prepared by the compiler and completed at runtime. For example, the elaboration of an array generator with statically unknown boundaries is completed by determining the layout of this generator at runtime as soon as the computation reaches its declaration. Furthermore all components of an INSEL system perform a "canonic operation" that consists of elaborating the declaration part and executing the statement part inherited from the generator. Naturally, simple data objects such as integers do not have a declaration or statement part resulting in an empty canonic operation.

1.2 Particular Advantages of the INSEL Concepts

The specific properties of the language concepts, such as nesting, argument passing, cooperation and the conceptually defined lifetime of all objects, implicitly establish strong dependencies between the objects of an INSEL system. This kind of structuring information has several benefits. First, it reflects application-level properties and can therefore be exploited to enforce automated application-specific resource management. Second, it is implicitly determined by the programmer by employing the language concepts without the burden of having to specify hints to the resource management system. And third, since most of these dependencies are based on class properties, they are easy to predetermine by software tools such as the compiler.

Most important of these structures is the termination dependency defining a partial order on the termination and deletion of objects. The lifetime of each INSEL-object depends conceptually on exactly one other object in a way ensuring, that no object is deleted as long as it is accessible. In particular, a component of either kind can only be deleted as soon as its termination dependent objects are terminated. Among others, it has the consequence, that created actors have to terminate before the creating component can be deleted. In practice this does not impose a major restriction for the programmer but has major beneficial aspects for the OS.

1.3 Redefinition of the Term "Operating System"

To enforce transparent, scalable and adaptable distributed resource management, we developed the architecture of a cooperative distributed management system [Gro96, GP97]. Based on the termination dependency, INSEL objects are clustered to actor-contexts (ACs) forming essential units of resource management. An AC comprises exactly one actor and all its termination dependent passive objects. With each AC, exactly one abstract manager is associated, being responsible for performing AC-specific resource management, that is to fulfill all requirements of the actor-context. Besides fundamental tasks such as allocating memory for the stack, heap and code of the objects within the AC, the manager might also have to provide facilities to maintain consistency of replicated objects, enforce access restrictions or perform load balancing. Conflicts, such as stack collisions, arising from different managers performing their tasks in parallel are solved by communication between managers according to application-level structural dependencies between the ACs. This management scheme is scalable as it does not have a potential central bottleneck and is adaptable because resource management is performed based on characteristics of application-level objects. For instance, the resource management system implements actors in a non-uniform manner. There is no single mapping of actors to for example UNIX processes or threads with a fixed size stack portion.

Definition 1.3.1 (Cooperating Managers) *The management of the distributed system splits up into multiple actor-context managers performing the task of global resource management cooperatively.*

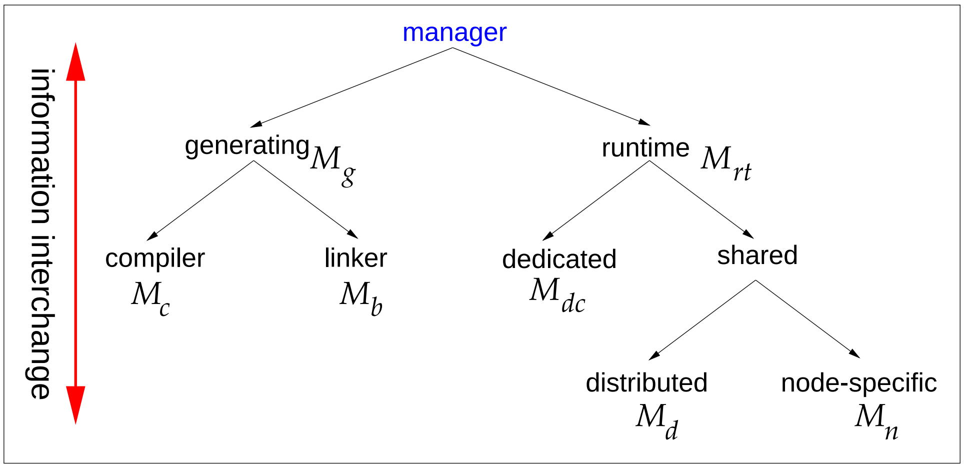

Figure 1.1: Software instances used to implement AC managers

It should be evident, that the straight forward approach of a rigid implementation of managers as objects defined in a runtime library would lead to an unacceptable overhead at runtime. In fact, the result of this idea would be closely related with an interpreter for INSEL [Wei97] with very similar performance characteristics. Instead, all software tools involved in management must be considered in an integrated way as the means to implement the abstract managers. The approach taken is to systematically incorporate manager functionality into software instances related with management. Each manager may individually be constructed by combinations of the capabilities of the software instances used. Hence, an implemented manager might solely consist of stack managing code inlined by the compiler or it may itself be a complex object comprising further activities. The functionality and granularity of the manager is tailored to the requirements of its AC.

Figure 1.1 illustrates software instances used to implement management facilities as well as it emphasizes the tight integration of all implementation techniques. Dedicated management instances (M_{dc}) are created specifically for one AC or eventually even for a single component. A common but not single implementation technique for M_{dc} instances is inlining. M_d denotes management functionality that is itself implemented as part of distributed system and jointly usable by more than one AC manager. Finally, M_n is used to classify node specific management (e.g. TLB management) mostly implemented in some kind of an OS kernel.

Of major importance among these implementation alternatives are naturally the compiler and the OS kernel, as the goal of the resource management system is to improve execution speed while reducing the size of the target representation. Hence, the basic strategy is to incorporate management functionalities into the compiler or the OS kernel instead of employing inlining techniques or runtime libraries.

Definition 1.3.2 (Management Instances) *The management functionality of abstract managers is implemented by several instances of an integrated management tool set.*

A main issue of the approach taken is to exploit information concerning overall system behavior as well as application-specific information gained from static and dynamic analysis to achieve adaptive resource management. Information is systematically exchanged between the managers [GR97] of the system and interchanged between the management instances.

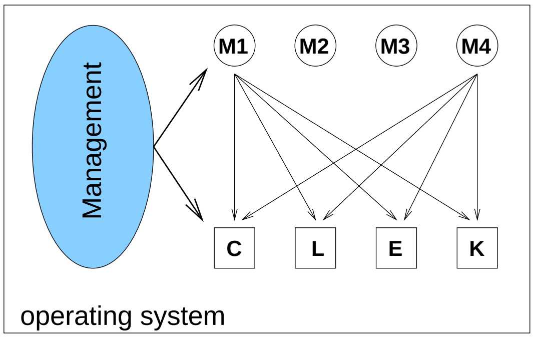

Figure 1.2: Structure of the operating system

As illustrated in figure 1.2, the "operating system" splits up into two dimensions. First the architecture of AC specific managers (`M1`–`M4` in the example) and second, their implementation techniques (e.g. compiler, linker, runtime environment and kernel). Cooperation and coordination among all management units is needed to achieve holistic distributed resource management. The crucial issues of this approach is the correct mapping of management tasks to the software instances and to establish information exchange and interchange.

Definition 1.3.3 (Operating System) *The* operating system *is the management of the computing system. It consists of cooperating actor context managers that are implemented by an integrated tool set.*

As a consequence of this approach, distributed and parallel processing (DPP) will be fully integrated into the architecture of this OS instead of consisting of adaptional layers that inherently introduce overhead. The accumulated overhead of all management instances due to support for DPP determines a lower bound for the effectiveness of the distributed OS. Scalability of all management techniques determines the upper bound for the effectiveness of the distributed OS. To extract the chance of performance benefits due to the utilization of distributed hardware resources, the overhead of runtime management has to be kept as low as possible which can only be reached by a thorough design of "static" management — the compiler.

1.4 Goals of the *gic* Project

Respective the above explained OS architecture, the compiler dominates the resource management system. First, it is the most important instance to analyze application-specific properties by reading the source code. Second, decisions made by the compiler are of major impact on the decisions made by the resource management in general. The role of the compiler as part of the targeted cooperative OS architecture is defined as producing suitable resources for further processing. A "resource" in this general sense is either information or executable code.

As the context of the project *gic* is the construction of a distributed OS, the methods investigated in theory of language design and compiler construction are of secondary interest. Instead, the goals of *gic* derivate from the dominant role of the compiler for the management system. Most relevant is:

- Establishing effective information interchange between the compiler and other management instances.
- Maximum flexibility to adapt the management to the requirements of INSEL, including decisions such as ordering of machine instructions, register allocation and stack management.
- Performance of executable code produced by the INSEL compiler has to be comparable to the efficiency of an existing language and an existing industrial strength compiler.

Besides these goals, it has to be reconsidered, that INSEL as part of a research project is still an experimental language. Some concepts might change while others will be added or removed. Hence, maintainability of the INSEL compiler has strong influences on the implementation techniques to choose. Some other aspects, such as portability, the performance of the compiler itself are respected but not the objective target of the project *gic*. Furthermore, it can not be neglected that the availability of a development environment is of major importance for the success of a new language. Hence, tools such as a debugger and a profiler either have to be developed in addition to the compiler or some means to enforce reusability of existing tools are mandatory.

1.5 Terminology and Typography

In this report, several terms meaning different things to different readers will be used frequently. Following definitions should be respected to avoid confusion:

"gcc" The "GNU C compiler" distribution consists of numerous header files, libraries, executables and their source files. With "GNU C compiler" or "*gcc*" in emphasized letters we refer to the entire distribution.

"gcc-based compiler" Such a compiler is constructed using the concepts and source codes of *gcc*. Well-known examples are the compilers for C, C++ and Objective-C.

"front-end" With "front-end" we denote the part of the compiler that performs syntactic and semantic analyzes.

"back-end" A generic "back-end" performing optimization and generating assembler output is shipped with *gcc* and linked to *gcc*-based compilers.

"gic" The term "*gic*" is used to identify the project with the goal to develop a *gcc*-based compiler for INSEL.

"RTL" This acronym stands for "Register Transfer Language" that is the most important intermediate representation of *gcc*-based compilers.

"tree" With "*tree*" or "*tree node*" in emphasized font the data structure provided by *gcc* as the interface for language front-ends is denoted.

Terms printed in typewriter font, such as "`IN`","`gperf`","`gic1`" or "`i-init.c`" are either names of executable programs or source files or keywords of INSEL.

1.6 Outline

The rest of this report is organized as follows. In chapter 2 important questions about the design of *gic* are discussed. Section 2.1 compares the alternative approaches of developing an INSEL compiler using an existing language as intermediate representation or writing a complete source to assembler compiler. These considerations are followed by an overview of *gcc* in section 2.1.2 and an explanation of the structure of the INSEL compiler in section 2.3. Chapter 3 elaborates details of the implementation of *gic* and is intended to serve as a starting point for developers of *gic* and might also be helpful to implement other *gcc*-based compilers. Afterwards, information on how to obtain, install and use *gic* is given in chapter 4. The report will conclude in chapter 5 with the reconsideration of results of the *gic* project, information about the current state and future objectives. Technical information about *gcc*, *gic* and INSEL such as important files, function, grammar, etc. is listed in the appendix.

2 Design of *gic*

Driven by the goals stated in 1.4, the design of the INSEL compiler focuses on maximum flexibility of management decisions, advanced analysis techniques, information interchange and maintainability. Although its design is based on the GNU C compiler consisting in total of more than half a million lines of code, it is well structured into mostly pipelined passes with well-defined functionality and interfaces. The most discriminative design issue compared to other *gcc*-based compilers is its additional abstract syntax tree (AST) representation and attribute evaluation method encompassing the handling of the symbol table.

2.1 Choosing the Target Representation

A question that has to be answered when constructing a compiler for a new language is the selection of the target representation. It is not obvious that the target representation must equal executable binary code. Numerous other intermediate representations for further processing are conceivable. Analyzing the benefits and deficiencies of the choices is a prerequisite and will be sketched in the following paragraphs.

2.1.1 C/C++ as a Portable Intermediate Representation

An often performed simplification in the development of the compiler in a language–based approach is to choose an existing language as intermediate representation and use an unmodified compiler to generate target code. Examples for this approach using C or C++ are the compilers constructed in the project Diamonds [NC96] and our own prototypical INSEL implementations EVA [Rad95] and AdaM [Win95]. The compiler for Napier [Dea87] goes a few steps beyond this translation scheme by exploiting extensions of GNU C, to for example place certain data in fixed hardware registers. The inherent deficiencies common to these approaches is, that overall management is not integrated due to a lack of flexibility to tune decisions made by the compiler. Eventually even with the result of inconsistencies but at least either limiting the success of static optimization or of runtime management. Calling conventions, register and stack allocation and optimization techniques have strong interferences with management techniques such as distributed shared memory (DSM) [Li86] or mapping of the virtual address space. Lacking coordination of the capabilities of the compiler and other management instances leads to considerable performance degradations that can hardly be compensated with distributed execution.

In the project EVA we experienced a drastical performance degradation of 700% for INSEL relative to C. The reason is, that the level of abstraction of the intermediate C++ code produced is too high to serve as a good starting point to produce an efficient executable. Similar performance experiences were gained with compilation via low–level C in AdaM. Here, the reason is, that the low–level of the code produced — integrated stack management, etc. — spoils the potentialities of the C optimizer.

Besides these performance experiences, it is also worth noticing that existing developing tools such as source level debugger or profilers can not be reused without major modifications in these approaches. Although it is possible to insert line number information into C/C++ code, complete associations between the code generated by the C/C++ compiler and the primary INSEL source code can hardly be established as needed to allow advanced handling of the running program, such as source level investigation of stack frames.

The obvious solution to these problems is to write a complete optimizing native source to binary compiler. But, the effort required to fulfill this task is unacceptable in a research project that is concerned with the development of distributed OS technology. A promising compromise is to choose an existing compiler available in source code and adapt it to INSEL.

2.1.2 *gcc* as a Retargetable Code-Generator

Due to its outstanding properties concerning portability, documentation, optimization and most of all support for more than a single language, the GNU C compiler was selected as the foundation for the INSEL compiler.

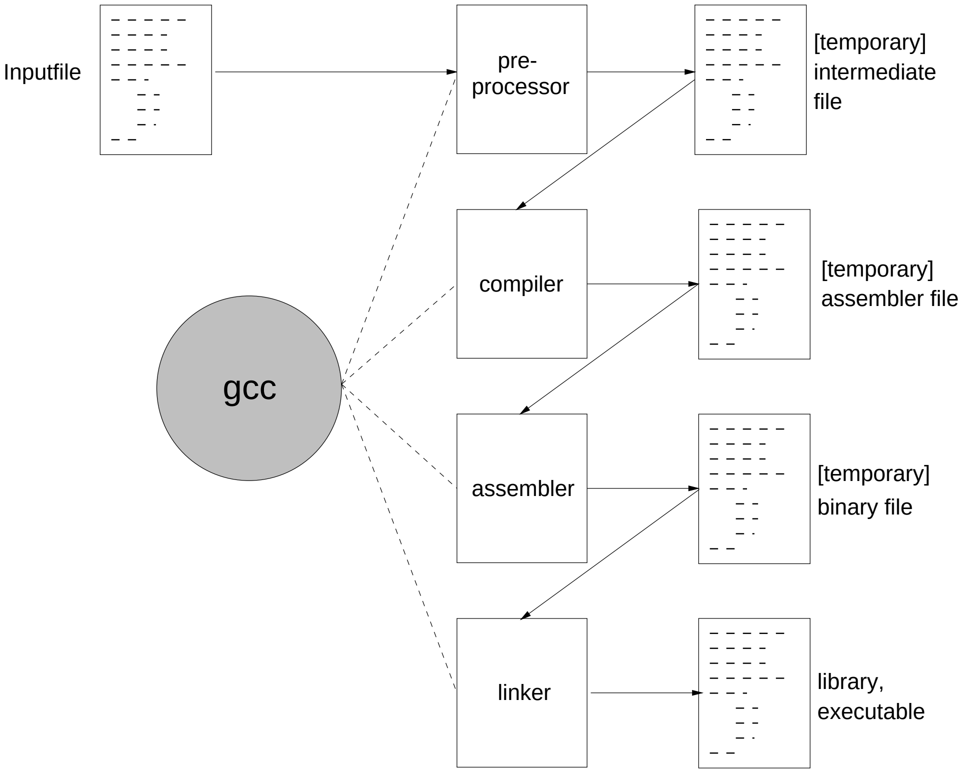

Figure 2.1: Source to target transformation using *gcc*

GNU is a Unix-compatible operating system, being developed by the Free Software Foundation and distributed under the GNU Public License (GPL). GNU software is always distributed with its sources, and the GPL enjoys anyone who modifies GNU software and redistributes the modified product to supply the sources for the modifications as well. In this fashion, enhancements to the original software benefit the software community in large. The GNU C compiler is the centerpiece of the GNU software. It is a retargetable and rehostable compiler system with multiple front-ends and a large number of hardware targets. The crucial asset of *gcc* is its mostly independency from languages and targets. It produces excellent code for both CISC and RISC machines. The machine dependent source code represents only 10% of the total. New targets can be added by giving an algebraic description of each machine instruction. The leverage of constructing a front-end for *gcc* is thus enormous: currently, more than 200 configurations of hardware architectures and OSs are supported. Optimization techniques developed and integrated by a large community are reused by all front-ends without additional effort and most of the tools of the GNU development environment such as the debugger `gdb` can be fully reused with hardly any modifications.

Best known of *gcc* is the "compiler-driver" `gcc`. As shown in figure 2.1 the user usually starts the compiler-driver to request a source to target transformation instead of directly calling a compiler. In fact, the program `gcc` analyzes command line options and calls various other executables to perform the translation. Based on the suffix of the input file names, language specific processing usually consisting of preprocessing and source to assembler translation is performed by calls of the language specific compilers. If not excluded with command line options, `gcc` afterwards calls an assembler and the linker to produce an executable.

Compilers based on *gcc* are structured into a "front-end" for language-specific processing and a generic "back-end" for optimization and target code generation. Both parts have to be statically linked to build a source to assembler compiler for one language.

Front-Ends

Language-specific processing is the transformation of source text into the machine-independent *tree* representation accepted as input by the *gcc* back-end. Hence, the front-end usually encompasses scanning, parsing, semantic analyzes and finishes with the synthesis of *gcc trees*. Table 2.1 lists some of the known languages for which front-ends are available and the name of the respective compilers. Other

language	**compiler**
C	`cc1`
C++	`cc1plus`
Objective-C	`cc1obj`
Fortran 77	`f77`
Pascal	`gpas`
Ada	`gnat1`

Table 2.1: Front-ends and compiler based on the GNU C compiler

front-ends for languages such as for Java are currently under development. Due to similarities of the language INSEL with Ada, work performed in the context of the Ada compiler *gnat* [CGS] stimulated the project *gic*. Ada also provides explicit tasking parallelism but lacks support for transparent distributed execution.

Back-End

Generation of optimized assembler output is performed by a generic back-end common to all *gcc*-based compilers. The front-end passes *trees* to the back-end and steers the compilation by calling procedures. The *trees* received are first translated into the machine-dependent lisp-like internal representation RTL [Sta95] (Register Transfer Language). All further optimization steps operate on RTL code before it is translated into the final assembler output. The features of the back-end are not limited to the concepts of C. Instead, it already offers special support for nesting, dynamic arrays, objects and other concepts not existing in C. Further support is added with the integration of new front-ends. As a result, the expressiveness of the back-end outclasses the alternative of using C as intermediate. We compared the performance of back-end support for nesting with the solution used in the PASCAL to C compiler `p2c`. Basic tests with a loop calling a nested function that accesses non-local variables demonstrated, that the *gcc* back-end integrated support for nesting outperforms the alternative most efficient solution using C by considerable 30%, although the back-end does not yet use “displays” but a chain of static predecessors. This simple experiment already demonstrates the significant benefit of extended management flexibility.

2.2 Attributed Abstract Syntax Trees - MAX

To be able to provide the desired advanced analysis and information interchange facilities while still preserving maintainability, an intermediate representation supporting flexible attribute evaluation in a separate compilation pass was inserted between the parser and the *gcc* back-end. Usually *gcc*-based compilers such as the C compiler directly call procedures of the *gcc* back-end in the semantic actions of the parser specification to construct *trees* and steer code generation. Although it might deliver peak performance this approach has several disadvantages:

1. The design of the grammar influences attribute evaluation and vice versa.
   (a) Some syntactical errors have to be treated as if they were semantic errors.
   (b) Tendency to decline analyzes due to difficult integration into the parser.
2. Re-evaluation of the attributes due to new information collected by runtime monitoring is not possible without parsing the source code.
3. Maintainability of both, the grammar and attribute evaluation is distinctively aggravated.

Besides these inherent disadvantages it is also debatable whether hand-code syntax-driven semantic analyzes with complex symbol table handling and attribute evaluation realized with cumbersome techniques like “back-patching” [ASU88b] delivers performance benefits. Attribute evaluation created by a well designed compiler compiler can be expected to outperform hand-coded versions if they are not optimized with strong effort.

Compilers developed as part of research projects in the field of distributed processing often create a separate abstract syntax tree (AST) as a tree of C++ objects [NC96]. Compared to a hand-coded tree of C++ objects, tool supported generation of such an AST representation in general reduces memory consumption, provides better performance and eases this task considerably. Nowadays, several compiler construction toolkits such as ELI [Gro94] and the Cocktail tool box [GE90] offer tools that allow to specify AST properties and attribute evaluation on a high level of abstraction. Because of its simplicity, integrated support for concrete to abstract syntax tree transformation and automatic attribute evaluation we decided to use the tool MAX [PH]. Among its major concepts are:

- A *tuple*, *alternatives* and *list* notation to specify the abstract grammar,
- a *functional language* augmented with *pattern matching* and an interface to C to operate on the AST and
- a *predicate logic* to specify *context conditions* for semantic checking.

MAX does not impose any restrictions on the order attributes have to be evaluated. Furthermore, AST nodes can itself be referenced by attribute values. Therefore,

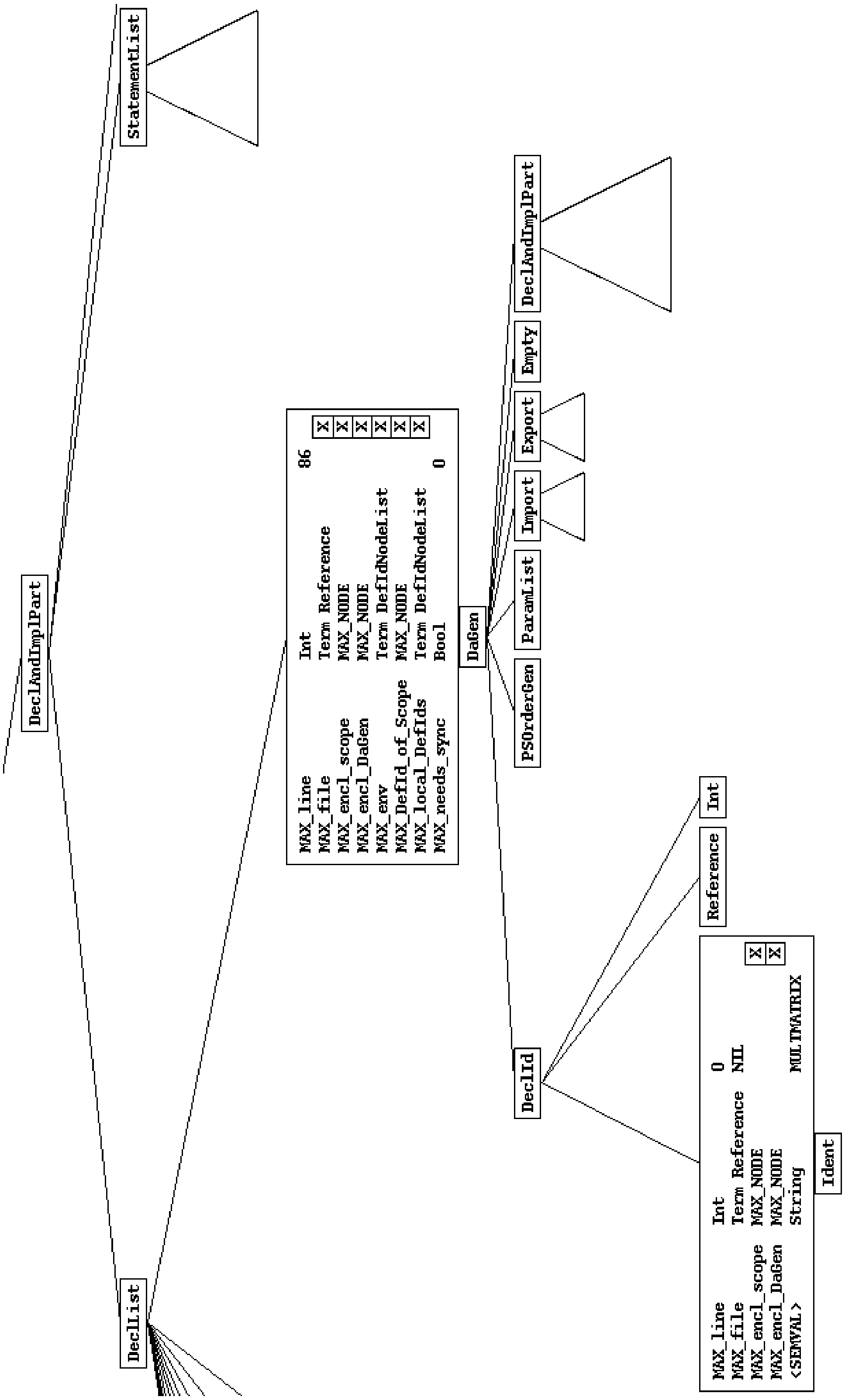

Figure 2.2: Part of a MAX browser screen shot

maximum flexibility for static analyzes of the source code is achieved. Understanding and debugging of the decorated AST is supported by an interactive browser, that visualizes the AST with its evaluated attributes (see figure 2.2).

The decomposition of our prototypical compiler, that used to employ syntax-driven semantic analyzes, into separate functional units for syntax checking and semantic analyzes using MAX, proofed to tremendously reduce the amount and complexity of source code as well as increased flexibility and speed.

2.3 Structure of the Compilation Process

The INSEL front-end is decomposed into units with well-defined tasks and interfaces. Tool support is deployed where possible. Figure 2.3 illustrates the internal

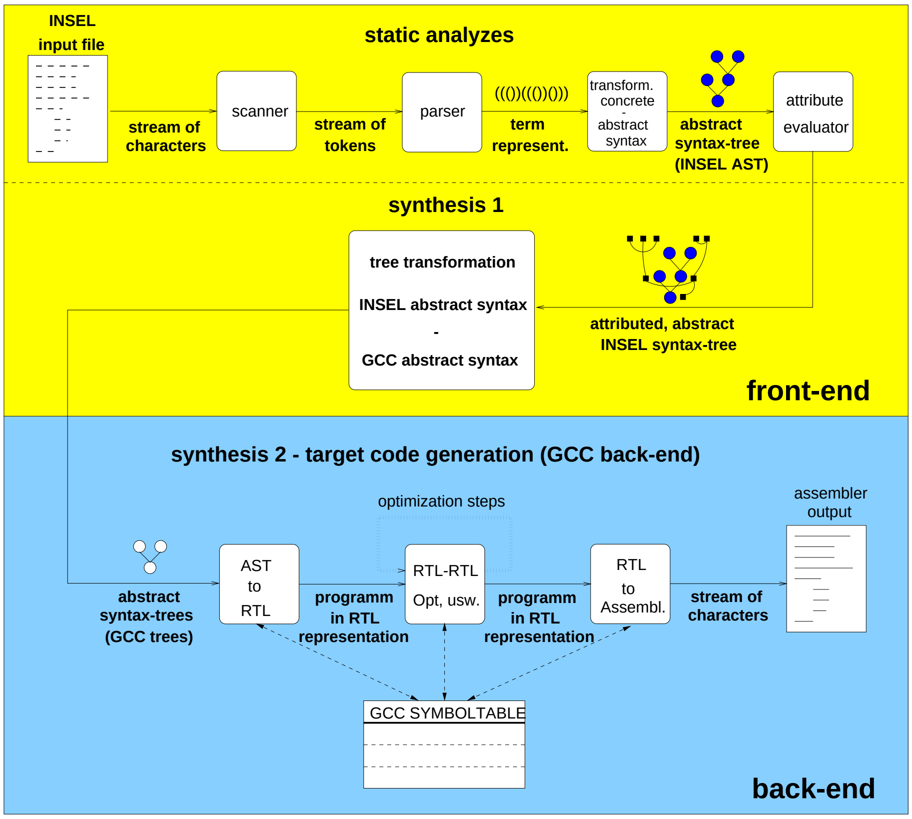

Figure 2.3: GIC Compilation Process

structure of the GNU INSEL compiler *gic* as a result of the decisions explained above. The INSEL front-end parses the input file using common syntax checking techniques. With procedures generated by MAX, based on the AST specification, a "term" representation is produced by the parser and passed to the "term to AST" transformer also generated by MAX. The AST representation is decorated with attributes representing compile-time as well as run-time properties. The final task of the INSEL front-end is to transform the decorated AST into the GNU *tree* representation by traversing the AST and calling procedures of the generic back-end of *gcc*. The GNU back-end manages an own symbol table and performs several RTL to RTL transformations before producing the final assembler code.

2.4 Information Interchange

The decisions to on the one hand use *gcc*'s back-end for code generation and therefore gaining the possibility to completely control the code produced and on the other to fit the AST in between parsing and semantic analyzes, deliver a sound foundation to establish information interchange between the compiler and other management instances.

In the path of semantic checking performed by MAX generated code, attributes of the AST reflecting application level properties are evaluated. These attributes are first used as usual in the synthesis step to decide about target representations to produce. In contrast to common compilation techniques, relevant attribute values are later on not annihilated but forwarded to the linker and the runtime management system in one of two ways. In most cases information is forwarded by "inlining" data into dedicated management code. A simple example are inlined argument values determining the required stack size being used in calls to stack allocating code to support the creation of actors with adequate stack portions. Besides inlining, attribute values are passed as extensions of the symbol information created with the assembler code. Hence, no additional files and associations between attribute data and target code has to be managed, neither by the compiler nor the linker.

The basic approach to establish the reverse flow of information from runtime monitoring to the compiler is based on attribute rereading and dynamic reevaluation in the AST representation. Values are either transfered via the augmented symbol information if they reflect class properties or directly transfered from the stack frames of components in execution if instance specific management has to be performed. The exploitation of reverse flow of information is subject to future research.

3 Implementation

Due to its complexity, a complete documentation of the implementation would exceed the length of this report. Instead, the subsequent sections plot significant aspects of *gic*'s implementation. The issues addressed below are intended to sketch the effort that has to be invested to implement a *gcc*-based compiler and to punctuate the results gained.

3.1 Interfacing with *gcc*

To be able to interface with the *gcc* back-end, several data structures as well as conventions comprising naming of files and functions have to be respected. Unfortunately, the otherwise excellent guide "Using and Porting GNU GCC" [Sta95] consisting of more than 500 pages of detailed information on *gcc*, does not describe how to add a new language-specific front-end. In fact, there seems to be no document elaborating this capability of *gcc* besides a collection of 146 slides [Ken95].

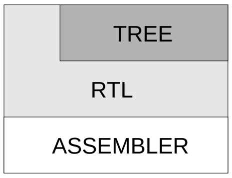

Figure 3.1: Intermediate representation in the GNU C compiler

As shown in figure 3.1, the GNU C compiler uses internally two intermediate representations: an abstract syntax tree (short "*tree*") data structure and the "Register Transfer Language" RTL. The interface between a language specific front-end and the back-end is mostly defined by the *tree* data structure. The RTL layer is not completely hidden by the *tree* representation. In fact, constructing a front-end that omits the *tree* representation is feasible but would contradict the objectives of the front-end/back-end architecture resulting in awkward properties such as increased complexity, lack of portability and compatibility. Instead, RTL is reasonably accessed and generated in front-ends as a short cut to, for example emit library calls or implement new language-specific *tree nodes*. Because RTL is described in detail in [Sta95] the rest of the explanations of the front-end/back-end interface will concentrate on *trees*.

3.1.1 Directory Structure and Files

Except for C, the source code of a language-specific front-end is kept in a separate subdirectory of the *gcc* source tree, e.g. subdirectory "`cp`" for the C++ front-end. Hence, to integrate a new front-end into *gcc*'s build process, a subdirectory with preferably the name of the language has to be created. The files listed in table 3.1 must exist within this directory to allow *gcc*'s build process to recognize and compile the new front-end. Calling `configure` in the *gcc* root directory calls the `config-lang.in` files of all front-ends, creates all Makefiles and C header files which include the language-specific header files listed in the table.

File	Content
`Makefile.in`	Makefile to compile the front-end.
`Make-lang.in`	Makefile fragment copied into the parent Makefile. It describes how to call the Makefile of the front-end.
`config-lang.in`	Called during configuration of *gcc* and used to prepare compilation. For example, announcing the name of the language and the compiler, applying patches or setting of platform specific options.
`lang-options.h`	Defines a list of strings of language-specific options to be added to existing options.
`lang-specs.h`	Specification of the file name suffix for this language and how to compile such files including calls of the assembler and linker. Documentation is only inlined in the source code of the compiler driver (`gcc.c`).

Table 3.1: Files expected by *gcc*

3.1.2 Front-End Interface — The *Tree* Data Structure

Most important for the compiler writer but poorly documented is the *gcc* internal *tree* data structure. It consists of multiply linked *tree nodes* and access macros to operate with these *nodes*. It is a common misconception that *gcc* would build *trees* for entire functions or even files. In reality, the front-end interface is mostly procedural and *trees* only exist for:

- types (or INSEL: generators),
- variables,
- expressions and
- blocks.

The data structure *tree* is a C union type consisting of fields common to all kinds of nodes and extensions for the different possible kinds, such as a reference to the string name in case of an identifier node or a field describing the size of the frame for function nodes. Furthermore, the tree structure can be extended for language-specific processing. The kind of a *tree* node is determined in the field `TREE_CODE` in its common area. About 127 different codes currently exist (see `tree.def`) comprising codes for all constructs available in C and additionals needed in other languages, front-ends were already implemented for, such as C++ and PASCAL. Many of the bits of the *tree* structure are used for different purposes depending on the *tree* code. Therefore, it is a strong recommendation to use the macros provided for convenient and efficient access of a *tree* node.

A front-end constructs *trees* by calling a small subset of the procedures of the back-end (see `tree.h`). It fills important fields, eventually performs simple optimization such as constant folding, passes the *tree* back to the back-end and requests its immediate expansion or to finish the compilation of the current declaration. The back-end then creates RTL code, optimizes the RTL code and outputs assembler code augmented with additional information for debugging or profiling, if activated.

Besides the interface to code generation, *gcc* also facilitates standard tasks as a convenience and to standardize behavior of development environments. Among the services offered are for example, error reporting functions graded into error, warning and sorry (not implemented) messages. With these services error reporting is alleviated with automatic counting of messages, a message layout that is understood by GNU development environments and automatic tracking of include stacks.

3.2 The INSEL Front-End

As explained in section 2.2, the INSEL front-end does not use the *tree* representation for its own purposes but solely for code generation. Although the *tree* framework is powerful enough to support all semantic actions we decided to clearly separate language-specific processing from code-generation. In the following paragraphs we will first discuss issues of syntactic and semantic analyzes before focusing on some special aspects of code generation to support parallel and distributed processing.

3.2.1 Scanner and Parser

Because maximum performance of the compiler is not our primary interest, syntactical analyzes are constructed using table generated keyword hashing, scanning and parsing by utilizing the GNU tools `gperf`, `flex` and `bison`. The INSEL grammar complies to LALR(1) and consists of 211 rules and 43 keywords. Experiences so far demonstrate, that performance of the constructed syntactic analyzes is sufficient to compile large units of source code.

3.2.2 Abstract Syntax Tree Representation

INSEL's abstract grammar is derivated from its concrete syntax by removing "syntactical sugar" (keywords, etc.) and further making adjustments to simplify attribute evaluation and code generation. Since the AST is not analyzed but constructed by calling procedures, it must not comply to LALR(1) and is therefore easier to specify and handle relative to the pars tree. The abstract grammar is defined with a high-level specification serving as input to MAX, which produces C code to construct and traverse the AST. The code generated by MAX is first used in the semantic actions of the parser to construct a "term"-representation. After parsing is finished, the resulting bracketed term is transformed into the AST representation by MAX generated code. In contrast to terms, ASTs can be freely traversed and decorated with attributes. This property is exploited in the project *gic* to perform all semantic actions on the INSEL AST. A MAX specification in general and INSEL's in particular consists of three parts:

1. Definition of the abstract grammar using tuple, list and variant productions.
2. Attribute part and
3. predicates and context conditions to define semantic rules.

Supplementary functions, written in MAX's functional language or imported from other languages can be added to support attribute evaluation and semantic checking.

The concept of logical predicates and context conditions eases the task of semantic checking significantly. Example 3.2 is taken from the INSEL AST specification and illustrates some of the concepts mentioned. Each AST node of sort `UsedId` is decorated with the attribute `DefId` referencing the node within the AST that defines this identifier. By using MAX's powerful pattern matching feature, the node of sort `Name` containing the `UID` node searched is retrieved and analyzed. Evaluation of this attribute commences with the nodes matched, the value of further attributes (e.g. `encl_scope`) and supplemental functions (e.g. `lookup_DefId`). Notice, that the required order of attribute evaluation is determined by MAX. In the context condition `UsedId`, attribute `def` is used in the predicate to check that no node of sort `UsedId` without a definition of the corresponding identifier exists. If the predicate fails, the supplementary function `IC_error` is called which in turn calls error reporting functions of *gcc*. Again, the order of checking context conditions is determined by MAX.

```
// def          evaluates the DefId-node (either SpecId or DeclId)
//              that defines the used id within the syntax tree.
//              For the distinction of whether the used id is within a
//              name or not, see explanation above ("decl").

ATT def( UsedId@ UID ) DefId@ :
  IF Name@<*, NameItem@ NI, UsedId@ UID, *> :
        lookup_DefId( id(UID), local_DefIds( encl_scope ( basic_gen(
                        type( NI ) ) ) ) )
  ELSE lookup_DefId( id(UID), env(encl_scope(UID)) )

// This condition checks whether an used id has been defined before

CND UsedId@ UID : def(UID) # nil()
| LET E==IC_Error(file(UID), line( UID )):
  '"Identifier \"" namepartstr( UID ) "\" not defined."'
```

Figure 3.2: Example of a MAX attribute and context condition

The approach, to specify target code generation (*gcc trees*) as an automatically evaluated attribute was aborted, because the otherwise most pleasant property of automatic determination of the order attributes are evaluated is awkward in this case. Naturally, the order to evaluate the code attribute is most important. Additionally, importing and exporting all required interfaces between *gcc* and the MAX specification proofed to be too complicated. Instead, it was decided to construct the AST and evaluate all attributes within MAX and traverse the AST separately in C to steer synthesis using the *gcc* back-end.

The combination of MAX with *gcc* further required to redirect MAX's standard error reporting method using "`stderr`" to calls of error reporting services provided by the *gcc* back-end.

3.2.3 Symbol Table

All information about symbols of the source code referring to abstract properties is kept in the AST and its attributes. The front–end does not maintain a separate symbol table besides the AST. In contrast to the front-end, the *gcc* back–end maintains separate symbol information with its *tree* representation. The technique used in the back-end to record *trees* forms a symbol table holding all information about the properties of symbols needed to generate target code, such as the assembler name of a declaration or the sizes of stack frames. In addition to this target code related information, the *gcc* symbol table is also capable of storing all other semantic properties needed for compilation. As elaborated above, this feature is not used by the INSEL front–end. But, since other front–ends make extensive use of the symbol table features provided by *gcc* and each front-end at least has to support it with procedures called by the back-end, understanding its basic structure is mandatory.

Figure 3.3 illustrates the organization of the symbol table as maintained by the back–end. The *gcc tree* nodes are organized in linked lists and "binding levels". The links plotted in the figure are the `TREE_CHAIN` links chaining nodes in the same binding level[1]. Fast access to the *tree nodes* and their fields is achieved by a hashing mechanism that associates identifiers with their corresponding nodes. With the service `get_identifier` an identifier *tree* node is retrieved or newly allocated if not

[1]In INSEL, binding levels represent lexical scopes.

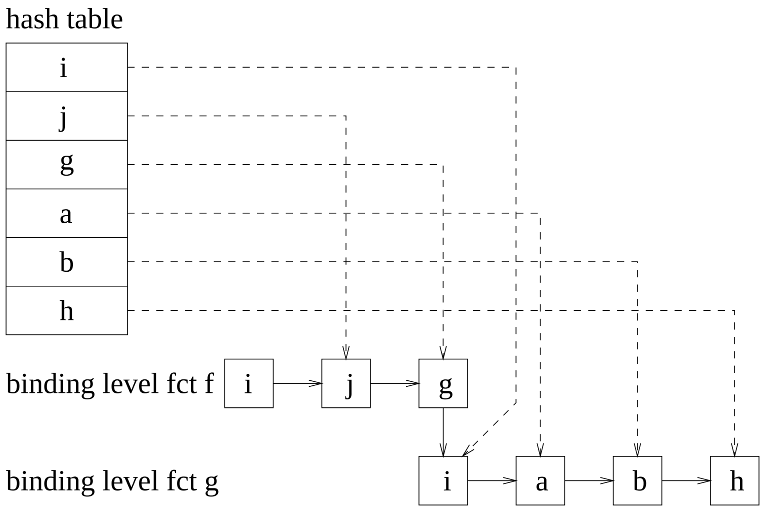

Figure 3.3: *gcc* symbol handling

yet existent. Each block, function and aggregate generator defines a new binding level temporarily shading the meaning of the identifiers on previous binding levels. Actions to take when entering or leaving a binding level vary between languages and must therefore be implemented with each front-end.

3.2.4 Synthesis: AST to *tree* Transformation

Final task of the front-end compiling an INSEL source text into optimized assembler output is to traverse the decorated AST and call procedures of the back-end to generate, pass and expand *gcc trees*. The concepts of the GNU C compiler offer a broad spectrum of alternatives for this transition. Some of the less trivial transformational actions are explained in the following paragraphs.

Creation of Actors Similar to the common nomenclature of “caller” and “callee” used for subprograms, we will use “creator” and “createe” to designate a creating component (actor, order, depot, etc.) and the newly created actor. Naturally, *gcc* does not yet offer support for “Create-Statements” similar as for “Call-Statements” since its concepts are still bound to languages that do not offer parallelism as a language concept. This shortage is currently compensated by techniques integrated into the INSEL front-end that may later on be moved to the back-end.

At least two assembler level functions are generated for each actor generator: a *stub* function implementing the functionality to prepare the createe and a *compute* function that performs the statement part of the actor generator. To be able to distinguish both functions at the assembler level, the suffix “.T”[2] is added to the assembler name of the compute function. The signature of the stub function is equivalent to the signature of the actor generator on the abstract level. Hence, actors are created by calling their stub function in the same way ordinary subprograms are called. The signature of the compute function complies to the interface of the call to create new threads on the selected platform.

The stub function of an actor generator is interspersed with two calls of the INSEL supportive environment. First, `ACTOR_ALLOCATE` is called to have the run-

[2]The suffix “.T” stands for “thread”. The dot was selected to avoid conflicts with user definable symbols.

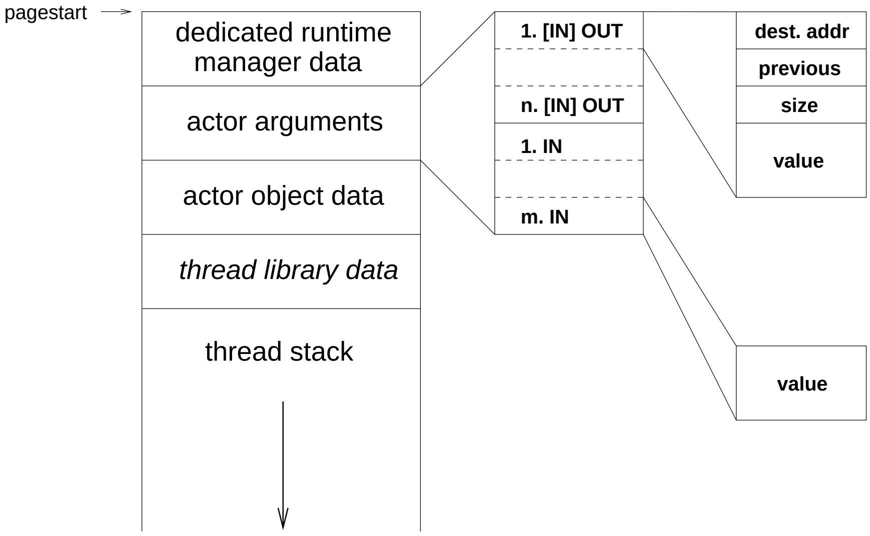

Figure 3.4: Initial memory layout of an actor

time manager provide initial virtual memory for the createe. The arguments passed in this call are the sizes of:

- the new manager,
- the arguments of the createe and
- the initial stack frame needed to start the computation.

After allocation of memory, control returns to the dedicated management portion inlined into the stub function by the compiler. The current arguments are passed to the createe (see below) and important fields of its manager, such as the address of the compute function and the size of its arguments are initialized. At the end of the stub function, `ACTOR_VITALIZE` is called to finish local initialization of the createe and to create a new flow of control (thread) involving load balancing and network communication. If a new thread is created, the only argument passed to the compute function via `pthread_create` (currently used thread interface) is the address of the manager holding all other relevant information. If the load management system decides to initiate the new flow of control on a remote node, initial virtual memory pages depending on the size of arguments and initial frame are transfered to the selected node.

Argument Passing To implement INSEL's `IN` and `OUT` parameter modes in a uniform way, all values of aggregate types[3] are passed by reference. Solely `IN` arguments of simple types such as `INTEGER` or `CHARACTER` are directly passed by value. In case of an order (subprogram) being called with arguments of a non-trivial type, the callee creates itself local place-holders for the arguments and uses these place-holders for its computation. For arguments passed with `IN` semantics the callee copies the values to its local place-holders before starting its computation. For `OUT` arguments the callee copies the values of the place-holders to their destination defined by the caller after finishing its computation. Naturally, for `INOUT` arguments, both copy steps are performed. This uniform method sometimes designated as

[3]used as a synonym for *generators*

"callee-copies" provides similar performance to the more common "caller-copies" method. In case of "callee-copies" the callee can itself optimize argument passing by omitting the creation of copies for unused objects. "Caller-copies" on the other hand would improve register utilization in the context of argument passing.

Argument passing to actors is more complex (see figure 3.4). Actors are essential units of management with an own separate stack and are often executed on a remote node wherefore costs for communications between the createe and the creator have to be kept as low as possible. Furthermore, the service used to create the new flow of control for an actor often only allows to pass a single argument of a pointer type. As a result, *gic*'s support for the creation of actors aims to reduce network communication and page faults, minimizes local copying and accommodates to the thread currently used. The creator of a new actor allocates virtual memory for <u>all</u> arguments of the actor within the initial stack of the createe. It further copies the values of `IN` arguments directly into the allocated space. Handling of `OUT` additionally requires that the destination address is stored with each `OUT` argument inside the createe. To correctly and efficiently implement INSEL's conceptually defined finish synchronization in combination with `OUT` arguments of actors, the createe must not copy back the values of `OUT` arguments by itself. Instead, with each `OUT` argument, two additional fields `size` and `previous` are stored and used by the creator to fetch the results during finish synchronization from the createe. `INOUT` arguments are passed in the same way as `OUT` arguments with the difference, that the creator also copies the input value to the space allocated for the `OUT` value of the argument. It is important to notice, that the computation of the createe directly operates on the space allocated for the arguments by the creator and no additional copies are made. As a beneficial effect of this compiler supported method to create actors, expensive heap management techniques to pass arguments to actors are omitted.

Arguments of "depots" or "c-actors" — objects encapsulating data — are handled similar to field components of the object. The component comprising the parameterized depot or c-actor simply copies the argument values to and from argument fields of the object.

It is worth mentioning, that the *gcc* back-end also has built-in support for "callee-copies" using "invisible references" to implement call-by-value. In fact, it should be sufficient to define the macros `FUNCTION_ARG_PASS_BY_REFERENCE` and `FUNCTION_ARG_CALLEE_COPIES` in the machine depended part of *gcc* to activate the "callee-copies" alternative without any changes to the front-end. We decided to integrate this technique into the front-end for two reasons. First, `IN` and `OUT` parameters need different handling and second, not to confuse other front-ends.

Start and Finish-Synchronization The INSEL concept of start and finish synchronization defines regulations for the creation/call and the deletion of an INSEL component. Basic regulations are for example the semantics for argument passing. In case of actors, "start-sync" and "finish-sync" must additionally ensure that a creator is not deleted before its createes. Hence, the management system has to keep track of all concurrently performing actors. First, for scalable decentralized management, the task of globally recording parallelism (π-structure) is split among the managers of actor contexts (AC). Each manager only keeps track of the actors created within its AC. Second, the synchronization concept enables to perform start and finish-sync for actors stack alike with the difference, that createes may terminate at any time. Figure 3.5 illustrates an efficient solution implemented in the context of *gic*. The path of computation of AC 1 managed by manager 1 has reached a certain call level and performs a sequential computation on the current stack frame. By incrementing and decrementing the field `current_comp_id`, the

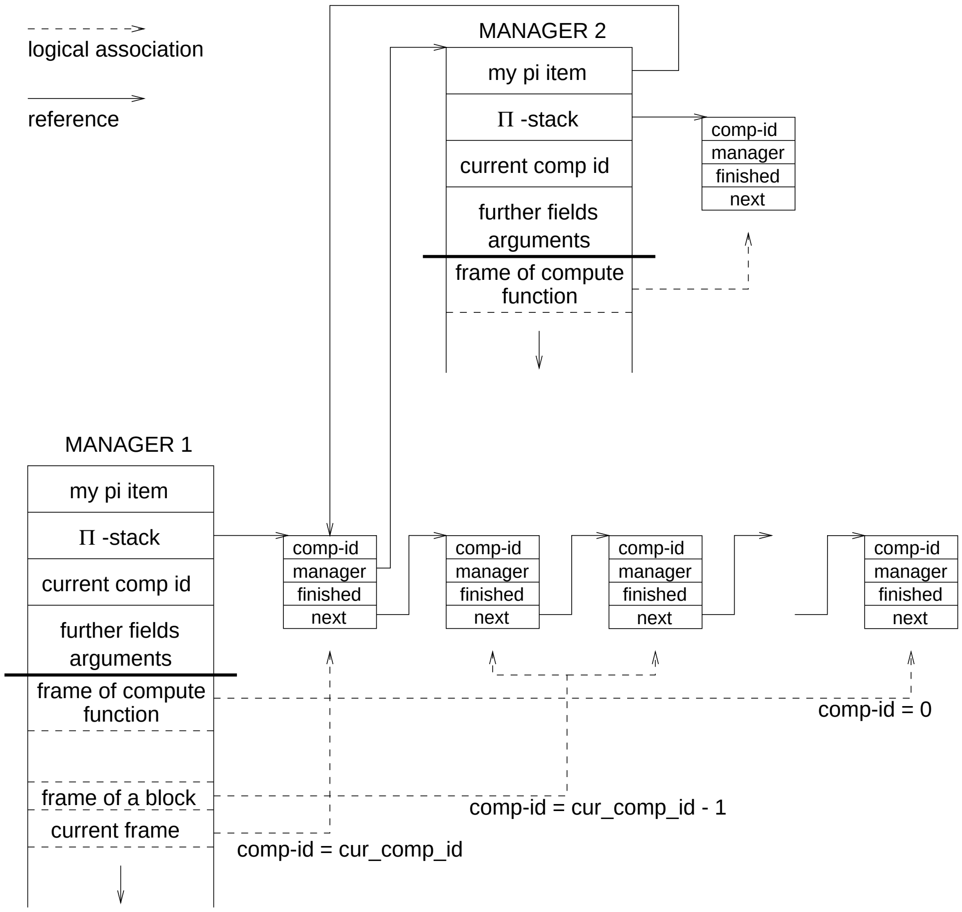

Figure 3.5: Organization of π-stacks

AC manager keeps track of the sequential call levels and blocks entered and left. If a component creates an actor, a new π-item is pushed on top of the π-stack maintained by the manager. The createe notifies the creator about its termination with setting the `finished` flag in its π item. Before leaving a call level or block, the AC manager is requested to perform finish-sync with all actors created by the current component. It in turn uses `current_comp_id` to wait on all π-items in the π-stack having the same value in `comp_id` to be `finished`. Since, all finished π-items of a component are deleted in the order of termination to reduce memory consumption, the organization of π-items is not truly a stack.

This strategy is itself mostly implemented in INSEL. The compiler simply emits calls to `PI_START_SYNC` and `PI_FINISH_SYNC` into the prologue and epilogue of relevant components. To not introduce infinite recursion these calls have to be omitted when compiling their INSEL definition. In fact, any INSEL component involved in start or finish-sync makes such an exception. A sound solution to this general problem of recursive definition is in this case reached with the attribute `needs_sync` determining the necessity to synchronize with createes. It is set in the AST representation based on the property whether a component creates actors[4] and used in the synthesis step to omit π-synchronization at runtime if the compiler already knows, that no createes will exist. With this strategy the problem of recursion

[4]Note, that only local knowledge is necessary for this decision.

in case of start and finish-sync is solved because, naturally `PI_START_SYNC` and `PI_FINISH_SYNC` do not create actors. Additionally, the performance of the system in general is enhanced, since many more unneeded calls are avoided.

Symbol	**Code of *gcc tree* node**	**Size/Value**
CHARACTER	CHAR_TYPE	HOST_BITS_PER_CHAR
INTEGER	INTEGER_TYPE	HOST_BITS_PER_INT
REAL	REAL_TYPE	HOST_BITS_PER_WORD
MANAGER_T	RECORD_TYPE	*runtime*
STRING_T	RECORD_TYPE	*runtime*
TRUE	INTEGER_CST	1
FALSE	INTEGER_CST	0
NULL	INTEGER_CST	0

Table 3.2: Built–in and joint symbols

Built-in Generators and Constants
Fundamental predefined generators and constants are handled as "built–in" by the compiler. The constants `HOST_BITS_PER_CHAR` and `HOST_BITS_PER_INT` are usually set to 8 and 32. A new data type for 64 bit (`LONG`) will be added as soon as full hardware and OS support is available.

The integration of the generators `MANAGER_T` and `STRING_T` into the compiler differs from the technique used to define the rest of the symbols listed in table 3.2. The corresponding generator definitions are not hard-coded into the compiler but written in INSEL for themselves and read by the compiler in a certain order ensuring that none of these is used before it was compiled. The only information hard-coded into the compiler are the names of identifiers used to denote these generators and fields that are to be accessed by the compiler. The advantage of this strategy is twofold. First, the flexible integration of these types allows for rapid and frequent changes that are due to the progress of the research project. Second, manager records are intensively accessed by INSEL management code outside the compiler and string operations are also written in INSEL. Hence, both generators have to be defined in the runtime environment, anyway. Additionally, hard-coding these generators in the compiler would introduce the risk of inconsistencies while hardly improve the performance of the compiler. Symbols agreed between the compiler and other mainly management instances of the system are called *joint* symbols. The concept of *joint* data structures further emphasizes the tight integration of the compiler into the resource management system. Other *joint* symbols are functions defined in INSEL and called by the compiler to perform complex management tasks as for example the creation of actors explained above.

3.3 Modifications to the Back–End

In order to meet the requirements of the new concepts of INSEL and the aim to support distributed processing, some parts of the back–end of *gcc* had to be adapted. Although most tasks could also be performed in the front–end, modifying the back–end is advantageous wherever the task would be awkward or inefficient to perform in the front-end. As a beneficial side–effect, compilers for other languages also profit from these changes that mainly reflect necessities of parallel and distributed processing.

3.3.1 Non-Contiguous Stacks

Due to its well-known advantages concerning persistency and mobility of objects, we employ a single 64 bit virtual address space for our system. A major problem in such parallel computing environments with fine-grain parallelism is adequate memory management for multiple activities within the single non-segmented address space. First, the management has to be performed decentralized to avoid bottlenecks and second, the stack size required for a parallel activity can not be statically predicted. A mechanism is needed that automatically handles stack growths, collisions and overflows.

In fact, hardware should provide advanced means to monitor stack evolution of multiple threads and the OS has to be prepared to expand and shrink stack sizes transparently. Since hardware support is not available, we have to integrate stack checks into the compiler. Whenever stack space is (de-)allocated, the stack-pointer has to be checked against upper and lower bounds of the current stack segment. If these limits are exceeded, the runtime manager has to (de-)allocate stack segments by splitting or merging free segments. To avoid expensive reorganizations of the stack space, the newly allocated stack segment does not have to be contiguous with existing ones, establishing a fragmented stack organization. According to the fragmentation of stack space the addressing scheme of the *gcc* back-end had to be changed. For example on a SUN Sparc arguments are addressed via a constant offset from the frame-pointer (`%fp`). We modified the addressing scheme to use local register `%l0` as an explicit argument pointer. For further details on virtual memory management for INSEL see [GPR97].

3.3.2 Trampoline

For compatibility reasons, *gcc* implements pointers to nested subprograms via a trampolining technique. If the address of a nested function g is taken within function f, a portion of code, that sets up information about static predecessors before branching to g is inserted in the stack frame of f and the address of the trampoline is used in place of the address of g. This technique allows to use existing libraries, such as pthreads [OSF92] without modifications together with languages that support nesting.

Unfortunately, since trampoline code is statically produced by the compiler, this strategy hampers dynamic extensibility. Trampolines can not be dynamically placed on stack frames of existing functions at the time new functionality is to be integrated into the running system. To overcome this deficiency we replaced the trampolining mechanism with a customized addressing scheme for nested functions.[5]

3.4 Interoperability

The INSEL compiler allows to inter-operate with functionality written in languages other than INSEL. This section will elaborate on interfacing between INSEL and C although most predications also comply to other languages. Except for union types and bit fields, INSEL allows to construct most of the types available in C. Direct exchange of global data between INSEL and C is not supported.

As long as the calling conventions of INSEL (see 1.1) are considered, global INSEL orders can be called from C. Orders are global if they are defined either on the outermost nesting level or in a not nested depot. When calling orders of a depot, the first argument has to be a pointer to the depot data. Calling of C functions from INSEL is possible by defining their interfaces in the INSEL system.

[5]These changes affect *gcc*'s files `expr.c` and `function.c`

The method of creating new actors as explained in 3.2.4 allows to create new actors from within C. Currently neither start nor finish synchronization is automatically performed for functions written in other languages than INSEL, wherefore synchronization of the created actors is performed with the INSEL component that called the C component. `PI_START_SYNC` and `PI_FINISH_SYNC` may be explicitly called from within C to provide synchronization of created actors with the creating C function.

4 Installing and Using *gic*

The compiler and its source code is available for interested readers. Please contact the author (see 5.3) to obtain an up to date snapshot.

4.1 Portability and Tested Platforms

Currently, the only platform dependent code of *gic* is the selection of hardwired registers for the argument pointer and to hold the address of the current runtime manager which is done in the machine dependent part of the back–end. Besides the selection of these registers, the front–end of *gic* itself does not impose further portability restrictions. Therefore, the compiler should be portable to most of the more than 200 configurations supported by *gcc*. More important portability issues are determined by the INSEL supportive environment. It strongly relies on `pthreads`, TCP/IP sockets, signal handling and the possibility to compute the manager register in the signal handler. Additionally, the integrated browser for the AST and its evaluated attributes can only be compiled and used on platforms with a X11 window system. On other platforms, this feature must be omitted.

Implementation of *gic* started on HP PA–RISC workstations running HP–UX 9.x. Later on, due to the requirement of the runtime system, the project migrated to the SUN UltraSparc architecture with SUN Solaris 2.5.1. This is the only platform currently tested. As the project advances, *gic* will be ported to Linux on x86 processors, UltraLinux and back to HP–UX on PA–RISC processors. As the goal of the MoDiS and INSEL approach is to develop a stand alone distributed operating system we are also working on a new micro–kernel called DyCoS [Cze97]. The long term target is to port *gic* and its supportive environment to DyCoS which will together with other tools such as an incremental linker form the distributed OS as explained in 1.3.

4.2 Installation

To compile *gic* from source, several tools besides a C compiler, linker and `make` have to be installed on the build platform:

- The perfect hash generator `gperf` to generate the hash table for keyword hashing.
- To generate the scanner, `flex` has to be available although `lex` should be sufficient with minor adaptions of the scanner specification.
- The parser generator `bison`; with adaptions of the parser specification `yacc` will work but was not tested, yet.

Modification of the keyword list, parser, abstract grammar or attribute evaluator is only possible, if *noweave* is installed, too. All files related in the definition of the INSEL syntax are written using WEB to allow automatic generation of syntax documentations. MAX and its X11 browser will be made available together with *gic*.

The process of installation of *gic* is straight forward. First, a *gcc* source distribution has to be obtained and unpacked. Next step is to unpack the *gic* source distribution inside the *gcc* root directory. After this, the steps described in the file `INSTALL` shipped with *gcc* have to be performed to build and install the compilers and supportive applications for the selected languages.

Simultaneous development of a *gcc*–based compiler by multiple developers tends to utilize tremendous amounts of disk space. Instead of every developer using an own dedicated version of the *gcc* build tree ($\approx$ 100MB), it is advisable to configure and compile *gcc* with the *gic* patches applied once and to create local working copies using links for all *gcc* files that are not part of the INSEL front–end.

4.3 Using *gic*

After compilation and installation of *gcc* and *gic*, INSEL files can be compiled using the compiler driver `gcc`. The compiler driver recognizes the language of source files according to the suffix of the file name. For "`.insel`" files, the INSEL compiler `gic1` is called to produce assembler code.

Besides language independent options of *gcc* that can be used for INSEL files as well, *gic* adds two new options to the command line:

-fiv is a debugging option that produces line number information as well as output about the AST node currently processed on `stdout`.

-fib activates the INSEL browser. After successful parsing and attribute evaluation, the X11 AST browser is started as a separate process.

5 Conclusion

To support the development of complex but high-quality distributed systems, new programming environments are required. The foundation of the environment presented are high-level language concepts offered by INSEL. Concepts and techniques for an innovative distributed resource management system that tightly integrates compiler, linker and OS functionalities are derivated from the language concepts in a top-down oriented manner. We argue, that the complete set of software tools involved in the transformation of a parallel program into an efficient distributed executable, has to be tailored to the requirements of distributed computing. The alternative to construct layers above existing unmodified tools inherently introduces overhead and even conflicts, limiting the potential efficiency of distributed processing.

Although new tools have to be constructed, existing and successful techniques integrated and implemented in available software can and should be reused. Modifications instead of re-inventions are necessary to decrease the development effort and at the same time increase the success of distributed and parallel processing. We elaborated on the construction of the INSEL compiler *gic* which is based on the GNU C compiler *gcc* to demonstrate our general strategy. The tight integration of the modified *gcc* into our resource management system eliminated the expensive need to implement a new code-generator while still preserving full flexibility for the source to target transformation and the opportunity to make use of a large collection of advanced compilation techniques.

Several features of the INSEL compiler constructed would either not be possible without the approach taken or extremely awkward and costly to realize. Following is a incomplete list of some important features described in the text:

1. The exact frame size needed for the compute function of an actor is delivered as a beneficial side-effect. Optimized register allocation and eliminations of unused expressions are automatically considered. Hence, adaptive memory management for concurrent actors is supported in a way hardly possible without the unique design of *gic*.

2. Sound starting point for extended management flexibility with dynamic attribute re-evaluation and dynamic re-compilation.

3. Full control over the assembler output produced which is important to construct the incremental linker.

4. The already existing method to forward application-specific information as symbol information to tools such as the linker and debugger can easily be extended to enforce information interchange between the compiler and other management instances of the cooperative management system.

5. Support for source level debugging and profiling without hardly any additional effort.

The compiler clearly separates different steps of compilation to gain flexibility for techniques such as dynamic re-compilation and increase maintainability significantly to meet the requirements of our research project. The utilization of compiler compilers and most of all interfacing with the GNU C compiler reduced the effort invested by some orders of magnitude.

5.1 Distributed and Parallel Processing

Several details of code generation were discussed. Some important changes and sanctions explained, directly reflect necessities of the distributed and parallel nature of the execution environment and account for the integration of the compiler *gic* in the cooperative manager architecture presented in 1.2.

The selection of a hardwired global register to hold the address of the associated runtime manager was only possible by modifying the machine–dependent code of the compiler. It greatly simplifies the handling of the manager in the compiler and the runtime system, provides compatibility with other front–ends and enhances the performance of the system relative to otherwise required passing of a manager argument. Using a hardware register determines a well–defined interface for the kernel to operate with managers as well as it allows to communicate the concerning manager between the kernel and runtime system using signals. It serves as a good example for a minor sanction with the major effect of an efficient integration of all management instances.

Of major impact is also the automatic creation of stub functions and in general, the method used to create actors. By interfering compiler and runtime functionality it is possible to create concurrently and remotely executing actors in a way that minimizes page faults and supports migration due to a packed implementation of an actor, its arguments and its manager. Integration of this method directly into the code generator further ensures advantageous symbiosis with existing techniques like register allocation.

Other important particularities are decisions to integrate management facilities such as dynamic stack checking as much as possible into the prologue and epilogue of functions to support incremental extensibility of the distributed system at runtime.

New attributes such as `non-local` and `needs-sync` are used to steer and enhance optimization according to the changed execution environment that comprises multiple threads and distributed shared memory.

5.2 Current State and Future Work

Currently, *gic* supports about 80% of the INSEL syntax. The runtime environment is partially written in C and INSEL because INSEL does not support system programming. Distributed execution is currently supported on SUN UltraSprac with Solaris 2.5.1. Experiences so far are promising as on the one hand INSEL programs proof to be far less complex than multi-threaded applications written in C with explicit message passing and on the other hand, overall management overhead is still low ($\approx$ 10%) comparing sequential performance of INSEL and C.

Besides the necessity to support the complete INSEL syntax, several other major development steps are objected. Of course, further modifications to the back-end of *gcc* will be made reflecting the change of paradigm from centralized and sequential to distributed and parallel processing. An important example is the integration of displays [ASU88b] to implement access to non-local data for languages that support nesting. Currently, the GNU C compiler only uses a chain of static predecessors. Assuming that levels of nesting are low and program execution is centralized, this scheme delivers sufficient performance. In case of distributed execution, performance is unacceptable, because tracing the chain might result in memory violations of the distributed shared memory and messages being sent for each level of nesting.

Although *gic* does and will run as a UNIX process, it is planed to further integrate the compiler into the INSEL system and support the incremental construction of the system with some kind of incremental compilation. An important milestone will be to use INSEL depots as input and output for *gic* instead of UNIX files.

Another important step will be to enhance interoperability of INSEL with other languages by providing tools to automatically convert interface definitions from one language to the other. INSEL will form the common ground for all other languages and serve as a sort of interface definition language to avoid the necessity to implement $O(n^2)$ converters.

Furthermore, advanced methods to establish reverse flow of information from runtime to the compiler will be developed and its capabilities investigated. It is aimed to use the information returned also to recompile existing instances of objects if runtime monitoring indicates its necessity.

Finally, we are aware, that in the field of parallel programming, numerous new languages and compilers (e.g. PSather [MFLS93]) with specific optimizations for parallel processing are developed. We aim to incorporate major results of work performed in this field into our general approach to resource management.

5.3 Contacting the Author

Feel free to contact the author of this report if you have questions, suggestions or want to join the project.

email: pizka@informatik.tu-muenchen.de
WWW: http://www.informatik.tu-muenchen.de/~pizka

5.4 Acknowledgment

This project is sponsored by the DFG (german research council) as part of the project SFB 342: “Tools and Methods for the Utilization of Parallel Architectures”.

Without the help and commitment of several people, the design and the current state of the project would be far from where it is. Many thanks go to Richard Kenner who helped us to get started with his collection of 146 slides on “Targetting and Retargetting the GNU C Compiler” [Ken95]. At the time of writing, these slides seem to be the only written information on how to add front–ends for new languages to *gcc*, besides inlined comments. Arndt Poetzsch–Hefter greatly simplified and speeded–up the development of semantic processing including attribute evaluation by providing his tool MAX. Jürgen Rudolph contributed a lot of work in the design and implementation of syntactic analyzes as well as he integrated the valuable intermediate representation of abstract syntax trees using MAX. Special thanks go to Christian Strobl who spent a lot of time in reading huge amounts of inlined comments in the source code to pave the way for the project and finally implemented considerable parts of the compiler. Last but not least, Claudia Eckert deserves greatest thanks for hours of helpful discussions and also for reading and correcting this document.

A INSEL - Syntax

A precise explanation of the INSEL syntax is subject to a special report on using INSEL. In contrast to this, the subsequent sections should give insights in the syntax for the interested reader and compiler writers. As the project advances, the syntax will change.

A.1 Keywords

ACCEPT	**ACCESS**	**AND**	**ARRAY**
BEGIN	**BLOCK**	**CACTOR**	**CASE**
CONSTANT	**DEPOT**	**ELSE**	**ELSIF**
END	**ENTRY**	**ENUM**	**EXIT**
EXPORT	**EXTERN**	**FALSE**	**FOR**
FUNCTION	**GENERIC**	**IMPORT**	**INCOMPLETE**
LOOP	**MACTOR**	**MOD**	**NEW**
NONE	**NOT**	**NULL**	**OTHERS**
OUT	**PROCEDURE**	**RECORD**	**RETURN**
RTMANAGER	**SELECT**	**SPEC**	**TERMINATE**
THEN	**TRUE**	**TYPE**	**WHEN**
WHILE	**WITH**	**XOR**	

Table A.1: INSEL reserved words

A.2 INSEL Syntax

The following list of INSEL grammar rules was directly produced from the bison parser specification. The semantic actions with calls of functions provided by MAX to construct the term representation are omitted for better readability.

1. compilation-unit
   : declaration-part[2]
   ;
2. declaration-part
   : /* empty */
   | declaration-part[2] declaration[3] ';'
   | declaration-part[2] error ';'
   ;
3. declaration
   : de-generator[35]
   | da-generator[4]
   | generic-generator[30]
   | generic-generator-incarnation[33]
   | object-declaration[46]
   ;
4. da-generator
   : specification-part[5]
   | implementation-part[9]
   ;

5. specification-part
: interface-specification[6]
| no-interface-specification[7]
| function-specification[8]
;

6. interface-specification
: interface-generator-type[13] SPEC IDENTIFIER
formal-in-parameter-part[24]
limitation-part[17]
interface-part[15]
;

7. no-interface-specification
: no-interface-generator-type[14] SPEC IDENTIFIER
formal-parameter-part[20]
limitation-part[17]
;

8. function-specification
: FUNCTION SPEC IDENTIFIER
formal-in-parameter-part[24]
limitation-part[17]
RETURN name[95]
;

9. implementation-part
: interface-implementation[10]
| no-interface-implementation[11]
| function-implementation[12]
;

10. interface-implementation
: interface-generator-type[13] IDENTIFIER
formal-in-parameter-part[24]
limitation-part[17]
declaration-and-implementation-part[16]
;

11. no-interface-implementation
: no-interface-generator-type[14] IDENTIFIER
formal-parameter-part[20]
limitation-part[17]
declaration-and-implementation-part[16]
;

12. function-implementation
: FUNCTION IDENTIFIER
formal-in-parameter-part[24]
limitation-part[17]
RETURN name[95]
declaration-and-implementation-part[16]
;

13. interface-generator-type
: CACTOR
| DEPOT
;

14. no-interface-generator-type
: MACTOR

```
                | PROCEDURE
                | ENTRY
                ;

15. interface-part
                : /* empty */
                | IS
                        declaration-part[2]
                        END opt-identifier[29]
                ;

16. declaration-and-implementation-part
                : IS
                        declaration-part[2]
                        BEGIN
                        statement-part[52]
                        END opt-identifier[29]
                | IS EXTERN STRING-LITERAL
                ;

17. limitation-part
                : import-part[18] export-part[19]
                | limitation-part[17] error ';'
                ;

18. import-part
                : /* empty */
                | IMPORT NONE ';'
                | IMPORT name-list[28] ';'
                ;

19. export-part
                : /* empty */
                | EXPORT NONE ';'
                | EXPORT identifier-list[27] ';'
                ;

20. formal-parameter-part
                : /* empty */
                | '(' formal-parameter-list[21] ')'
                ;

21. formal-parameter-list
                : formal-parameter[22]
                | formal-parameter-list[21] ';' formal-parameter[22]
                | error ';'
                ;

22. formal-parameter
                : identifier-list[27] ':' parameter-mode[23] name[95]
                ;

23. parameter-mode
                : IN
                | OUT
                | IN OUT
                | /* empty default: IN ! */
                ;

24. formal-in-parameter-part
                : /* empty */
                | '(' formal-in-parameter-list[25] ')'
                ;
```

25. formal-in-parameter-list
: formal-in-parameter[26]
| formal-in-parameter-list[25] ’;’ formal-in-parameter[26]
| error ’;’
;

26. formal-in-parameter
: identifier-list[27] ’:’ IN name[95]
| identifier-list[27] ’:’ name[95]
;

27. identifier-list
: IDENTIFIER
| identifier-list[27] ’,’ IDENTIFIER
;

28. name-list
: name[95]
| name-list[28] ’,’ name[95]
;

29. opt-identifier
: /* empty */
| IDENTIFIER
;

30. generic-generator
: GENERIC
formal-generic-parameter-part[31]
da-generator[4]
;

31. formal-generic-parameter-part
: /* empty */
| formal-generic-parameter-part[31] formal-generic-parametert[32] ’;’
| formal-generic-parameter-part[31] error ’;’
;

32. formal-generic-parameter
: WITH IDENTIFIER
| WITH FUNCTION IDENTIFIER
formal-parameter-part[20]
RETURN name[95]
| WITH PROCEDURE IDENTIFIER
formal-parameter-part[20]
;

33. generic-generator-incarnation
: NEW interface-generator-type[13] IDENTIFIER IS
name[95] actual-generic-parameter-part[34]
;

34. actual-generic-parameter-part
: /* empty */
| ’(’ name-list[28] ’)’
;

35. de-generator
: TYPE IDENTIFIER
IS type-part[48]
;

```
36. type-constructor
                : array-type-constructor[37]
                | enumeration-constructor[38]
                | pointer-type-constructor[39]
                | range-constructor[40]
                | record-type-constructor[43]
                ;

37. array-type-constructor
                : ARRAY '[' range-part-list[41] ']' OF type-part[48]
                ;

38. enumeration-constructor
                : ENUM '(' identifier-list[27] ')'
                ;

39. pointer-type-constructor
                : ACCESS type-part[48]
                ;

40. range-constructor
                : simple-expression[87] RANGE-POINTS simple-expression[87]
                ;

41. range-part-list
                : range-part[42]
                | range-part-list[41] ',' range-part[42]
                ;

42. range-part
                : name[95]
                | range-constructor[40]
                ;

43. record-type-constructor
                : RECORD
                        field-declaration-list[44]
                        END RECORD
                ;

44. field-declaration-list
                : field-declaration[45] ';'
                | field-declaration-list[44] field-declaration[45] ';'
                ;

45. field-declaration
                : identifier-list[27] ':' type-part[48]
                ;

46. object-declaration
                : identifier-list[27] ':' constant-part[47] type-part[48] init-part[51]
                ;

47. constant-part
                : /* empty */
                | CONSTANT


48. type-part
                : name[95] actual-parameter-part[49]
                | type-constructor[36]
                ;
```

```
49. actual-parameter-part
                    : /* empty */
                    | '(' expression-list[50] ')'
                    ;

50. expression-list
                    : expression[85]
                    | expression-list[50] ',' expression[85]
                    ;

51. init-part
                    : /* empty */
                    | ASSIGN-OP expression[85]
                    ;

52. statement-part
                    : /* empty */
                    | statement-part[52] statement[53] ';'
                    | statement-part[52] error ';'
                    ;

53. statement
                    : da-related-statement[54]
                    | compound-statement[55]
                    | simple-statement[56]
                    ;

54. da-related-statement
                    : call-statement[57]
                    | accept-statement[58]
                    | select-statement[59]
                    | block-statement[65]
                    ;

55. compound-statement
                    : loop-statement[66]
                    | if-statement[69]
                    | case-statement[73]
                    ;

56. simple-statement
                    : assignment[79]
                    | return-statement[80]
                    | exit-statement[81]
                    | incomplete-statement[82]
                    | empty-statement[83]
                    ;

57. call-statement
                    : name[95] actual-parameter-part[49]
                    ;

58. accept-statement
                    : ACCEPT IDENTIFIER
                    ;

59. select-statement
                    : SELECT
                          select-alternatives[60]
                          select-else-part[64]
                          END SELECT
                    ;
```

```
60. select-alternatives
                    : select-alternative[61]
                    | select-alternatives[60] OR select-alternative[61]
                    ;

61. select-alternative
                    : when-part[62] accept-alternative[63]
                    | when-part[62] TERMINATE ';'
                    ;

62. when-part
                    : /* empty */
                    | WHEN condition[84] DO
                    ;

63. accept-alternative
                    : accept-statement[58] ';'
                          statement-part[52]
                    ;

64. select-else-part
                    : /* empty */
                    | ELSE statement-part[52]
                    ;

65. block-statement
                    : BLOCK IDENTIFIER IS
                          declaration-part[2]
                          BEGIN
                          statement-part[52]
                          END opt-identifier[29]
                    ;

66. loop-statement
                    : label-part[67] for-while-part[68]
                          LOOP
                          statement-part[52]
                          END LOOP opt-identifier[29]
                    ;

67. label-part
                    : /* empty */
                    | IDENTIFIER ':'
                    ;

68. for-while-part
                    : /*empty */
                    | FOR IDENTIFIER IN range-part[42]
                    | WHILE condition[84]
                    ;

69. if-statement
                    : IF condition[84] THEN statement-part[52]
                          elsif-part[70]
                          else-part[72]
                          END IF
                    ;

70. elsif-part
                    : /* empty */
                    | elsif-part[70] elsif[71]
                    ;
```

71. elsif
: ELSIF condition[84] THEN statement-part[52]
;

72. else-part
: /* empty */
| ELSE statement-part[52]
;

73. case-statement
: CASE expression[85] IS
case-alternatives[74]
END CASE
;

74. case-alternatives
: case-alternative[75]
| case-alternatives[74] case-alternative[75]
;

75. case-alternative
: WHEN choices-or-others[76] DO statement-part[52]
;

76. choices-or-others
: choices[77]
| OTHERS
;

77. choices
: choice[78]
| choices[77] choice[78]
;

78. choice
: expression[85]
| range-constructor[40]
;

79. assignment
: name[95] ASSIGN-OP expression[85]
;

80. return-statement
: RETURN expression[85]
;

81. exit-statement
: EXIT opt-identifier[29]
;

82. incomplete-statement
: INCOMPLETE
;

83. empty-statement
: NULL
;

84. condition
: expression[85]
;

85. expression
: relation[86]
| expression[85] logical-operator[97] relation[86]
;

86. relation
: simple-expression[87]
| simple-expression[87] relational-operator[98] simple-expression[87]
;

87. simple-expression
: term[88]
| simple-expression[87] adding-operator[99] term[88]
;

88. term
: factor[89]
| term[88] multiplying-operator[100] factor[89]
;

89. factor
: operand[90]
| factor[89] exponentiating-operator[101] operand[90]
;

90. operand
: primary[91]
| unary-operator[102] operand[90]
;

91. primary
: literal[92]
| variable-or-function-call[94]
| generating-expression[96]
| '(' expression[85] ')'
;

92. literal
: CHARACTER-LITERAL
| STRING-LITERAL
| INTEGER-LITERAL
| REAL-LITERAL
| boolean-literal[93]
| NULL
;

93. boolean-literal
: TRUE
| FALSE
;

94. variable-or-function-call /* includes type conversion */
: name[95] actual-parameter-part[49]
;

95. name
: IDENTIFIER
| name[95] '.' IDENTIFIER
| name[95] DEREF
| name[95] '[' expression-list[50] ']'
| RTMANAGER
;

96. generating-expression
    : NEW name[95] actual-parameter-part[49]
    ;

97. logical-operator
    : AND
    | OR
    | XOR
    ;

98. relational-operator
    : '='
    | '<'
    | '>'
    | NE
    | LE
    | GE
    ;

99. adding-operator
    : '+'
    | '−'
    | '&'
    ;

100. multiplying-operator
    : '*'
    | '/'
    | MOD
    ;

101. exponentiating-operator
    : EXP
    ;

102. unary-operator
    : '+'
    | '−'
    | NOT
    ;

103. sign-part
    : '+'
    | '−'
    | /* empty */
    ;

A.3 Abstract INSEL Grammar

1. **CompUnit** (DeclList[2])
2. **DeclList** ∗ Decl[3]
3. **Decl** = DaGen[4] | DeGen[29] | GenGen[40] | GenGenIncarn[41] | ObjectDecl[42]
4. **DaGen** (DefId[121] DaGenType[19] ParamList[7] Import[13] Export[16] TypePart[31] DaBody[5])
5. **DaBody** = String[133] | DeclAndImplPart[6]
6. **DeclAndImplPart** (DeclList[2] StatementList[47] OptUsedId[118])

7. **ParamList** * Param[8]
8. **Param** (DeclIdList[120] ParamMode[9] TypeName[32])
9. **ParamMode** = In[10] | Out[11] | InOut[12]
10. **In** ( )
11. **Out** ( )
12. **InOut** ( )
13. **Import** (ImportPart[14])
14. **ImportPart** = All[18] | TypeNameList[15]
15. **TypeNameList** * TypeName[32]
16. **Export** (ExportPart[17])
17. **ExportPart** = All[18] | DeclIdList[120]
18. **All** ( )
19. **DaGenType** = AkteurGen[20] | DepotGen[23] | OrderGen[24]
20. **AkteurGen** = MAkteurGen[21] | KAkteurGen[22]
21. **MAkteurGen** ( )
22. **KAkteurGen** ( )
23. **DepotGen** ( )
24. **OrderGen** = SOrderGen[25] | KOrderGen[28]
25. **SOrderGen** = FSOrderGen[26] | PSOrderGen[27]
26. **FSOrderGen** ( )
27. **PSOrderGen** ( )
28. **KOrderGen** ( )
29. **DeGen** (DefId[121] TypePart[31])
30. **TypePartList** * TypePart[31]
31. **TypePart** = TypeName[32] | DeGenType[33]
32. **TypeName** (Name[114] ExpList[78])
33. **DeGenType** = Array[34] | Enum[35] | Pointer[36] | Range[37] | Record[38] | Empty[125]
34. **Array** (TypePartList[30] TypePart[31])
35. **Enum** (DeclIdList[120])
36. **Pointer** (TypePart[31])
37. **Range** (Exp[79] Exp[79])
38. **Record** * FieldDecl[39]
39. **FieldDecl** (DeclIdList[120] TypePart[31])
40. **GenGen** (DeclId[123] DeclList[2] DaGen[4])
41. **GenGenIncarn** (DeclId[123] DaGenType[19] TypeName[32] TypeNameList[15])

42. **ObjectDecl** = VarObjectDecl[43] | ConstObjectDecl[44]
43. **VarObjectDecl** (DeclIdList[120] TypePart[31] InitPart[45])
44. **ConstObjectDecl** (DeclIdList[120] TypePart[31] InitPart[45])
45. **InitPart** = Empty[125] | Exp[79]
46. **Predeclared** (DeclList[2])
47. **StatementList** * Statement[48]
48. **Statement** = CallStatement[72] | AcceptStatement[61] | SelectStatement[54]
49. **IfStatement** (IfRuleList[50] ElsePart[52])
50. **IfRuleList** * IfRule[51]
51. **IfRule** (Cond[53] StatementList[47])
52. **ElsePart** (StatementList[47])
53. **Cond** (Exp[79])
54. **SelectStatement** (SelectList[55] ElsePart[52])
55. **SelectList** * SelectItem[56]
56. **SelectItem** (OptCond[57] TermAcceptPart[58])
57. **OptCond** = Empty[125] | Cond[53]
58. **TermAcceptPart** = Terminate[59] | AcceptPart[60]
59. **Terminate** ( )
60. **AcceptPart** (AcceptStatement[61] StatementList[47])
61. **AcceptStatement** (UsedId[124])
62. **BlockStatement** (DeclId[123] DeclList[2] StatementList[47] OptUsedId[118])
63. **LoopStatement** (OptDeclId[119] ForWhilePart[64] StatementList[47] OptUsedId[118])
64. **ForWhilePart** = Empty[125] | For[65] | While[66]
65. **For** (DeclId[123] TypePart[31])
66. **While** (Cond[53])
67. **CaseStatement** (Exp[79] CaseList[68])
68. **CaseList** * CaseItem[69]
69. **CaseItem** (ChoiceList[70] StatementList[47])
70. **ChoiceList** * ChoiceItem[71]
71. **ChoiceItem** = Exp[79] | Range[37]
72. **CallStatement** (Name[114] ExpList[78])
73. **Assignment** (Name[114] Exp[79])
74. **ReturnStatement** (Exp[79])
75. **ExitStatement** (OptUsedId[118])
76. **IncompleteStatement** ( )

77. **EmptyStatement** ( )
78. **ExpList** ∗ Exp[79]
79. **Exp** = Literal[80] | VarOrFctAppl[81] | Operation[88] | GenExp[82]
80. **Literal** = Int[131] | Char[132] | String[133] | Real[83] | TrueVal[84] | FalseVal[85]
81. **VarOrFctAppl** (Name[114] ExpList[78])
82. **GenExp** (TypeName[32])
83. **Real** (String[133])
84. **TrueVal** ( )
85. **FalseVal** ( )
86. **NilVal** ( )
87. **ManagerVal** ( )
88. **Operation** (Operator[89] ExpList[78])
89. **Operator** (OpType[90] File[126] LineNo[127])
90. **OpType** = LogOpType[91] | RelOpType[92] | ArithOpType[93] | StringOpType[94]
91. **LogOpType** = AndOp[95] | OrOp[96] | XorOp[97] | NotOp[98]
92. **RelOpType** = EqOp[99] | LessOp[100] | GreaterOp[101] | NeqOp[102] | LeqOp[103] | GeqOp[104]
93. **ArithOpType** = AddOp[105] | SubOp[106] | MultOp[107] | DivOp[108] | ModOp[109]
94. **StringOpType** = ConcOp[113]
95. **AndOp** ( )
96. **OrOp** ( )
97. **XorOp** ( )
98. **NotOp** ( )
99. **EqOp** ( )
100. **LessOp** ( )
101. **GreaterOp** ( )
102. **NeqOp** ( )
103. **LeqOp** ( )
104. **GeqOp** ( )
105. **AddOp** ( )
106. **SubOp** ( )
107. **MultOp** ( )
108. **DivOp** ( )
109. **ModOp** ( )
110. **ExpOp** ( )
111. **PosOp** ( )

112. **NegOp** ( )

113. **ConcOp** ( )

114. **Name** ∗ NameItem[115]

115. **NameItem** = UsedId[124] | Deref[116] | ArrayAppl[117] | ManagerVal[87]

116. **Deref** ( )

117. **ArrayAppl** (ExpList[78])

118. **OptUsedId** = Empty[125] | UsedId[124]

119. **OptDeclId** = Empty[125] | DeclId[123]

120. **DeclIdList** ∗ DeclId[123]

121. **DefId** = SpecId[122] | DeclId[123]

122. **SpecId** (Ident[130] File[126] LineNo[127])

123. **DeclId** (Ident[130] File[126] LineNo[127])

124. **UsedId** (Ident[130] File[126] LineNo[127])

125. **Empty** ( )

126. **File** = Reference[134]

127. **LineNo** = Int[131]

128. **LimitPart** (Import[13] Export[16])

129. **Constant** ( )

130. **Ident** ( )

131. **Int** ( )

132. **Char** ( )

133. **String** ( )

134. **Reference** ( )

B INSEL Example Programs

The following INSEL sample programs are listed to demonstrate some of the concepts of INSEL and the current abilities of the compiler.

B.1 Nested Function

```
-- $Id: function.insel,v 1.2 1997/09/12 16:37:33 pizka Exp $

MACTOR System IS

  PROCEDURE WriteChar(c: IN character) IS EXTERN "putchar";
  PROCEDURE WriteInt (c: IN integer)   IS EXTERN "WriteInt";

  a: integer := 5;
  b: integer;

  FUNCTION dummy(x,y : IN integer) RETURN integer IS
    lokal: integer;
  BEGIN
    lokal := x+y+a;
    RETURN lokal;
  END dummy;

BEGIN
  b := 1;
  FOR I IN 1..1000000 LOOP
    b := dummy(a,I);
  END LOOP;
  WriteInt (b);
  WriteChar('\n');
END System;
```

Although `function.insel` is only a simple example, it already demonstrates some of the capabilities of the INSEL compiler:

- Actors: Based on the source text `MACTOR system ...` the actor generator `system` is elaborated at runtime. Each instance of an actor created on behalf of this generator performs its computation concurrently to its creator.
- Nesting of generators: The Order–Generator `dummy` is nested within actor instances of generator `system`.
- Access to non–local variables as done with integer `a` in `dummy`.
- Integration of external functions written in other languages such as C is demonstrated with `WriteChar` and `WriteInt`.

B.2 Primes

This example of a naive generator for prime numbers illustrates the use of INSEL's passive objects called "depots", dynamic data structures (pointer generators and the NEW operator) and an actor generator PrimeTest to create actors of extremely fine granularity. Notice, that the program only determines, that actor incarnations of PrimeTest may perform concurrently to their creator. It is the task of the distributed cooperative management system including the functionality of the compiler to enforce efficient execution of the computation. In the case of the prime generator, it demands the production of alternative code representations from the compiler. The alternatives needed for PrimeTest actors allow to perform the computation of PrimeTest as a usual subprogram without a new thread or by one thread performing the computations of a set of PrimeTest actors.

```
--
-- primes.naive.insel
--

MACTOR System IS

PROCEDURE OutChar(c: IN CHARACTER) IS EXTERN "putchar";
PROCEDURE OutInt (c: IN INTEGER)   IS EXTERN "WriteInt";

  isPrim : boolean;                   -- shared result variable

DEPOT listmanager is                  -- Passive object generator

  TYPE listpointertype IS ACCESS listitem;

  TYPE listitem IS
    RECORD
      next : listpointertype;
      item : integer;
    END RECORD;

  TYPE myrecord IS
    RECORD
      laenge   : integer;
      listhead : listpointertype;
    END RECORD;

  help, helpto, helpact : listpointertype;
  counter : integer;
  tmp : myrecord;

  PROCEDURE PrintList IS
  BEGIN
    help := tmp.listhead;
    WHILE help /= NULL LOOP
      OutInt(help^.item);
      OutChar('>');
      help := help^.next;
    END LOOP;
    OutChar('\n');
  END PrintList;

  FUNCTION GetNextItem RETURN INTEGER IS
  BEGIN
    help := helpact;
    IF helpact^.next /= NULL THEN
      helpact := helpact^.next;
    ELSE
      helpact := tmp.listhead;
```

```
    END IF;
    RETURN (help^.item);
  END GetNextItem;

  FUNCTION Count RETURN INTEGER IS
  BEGIN
    counter := 0;
    help := tmp.listhead;
    WHILE help /= NULL LOOP
      counter := counter + 1;
      help := help^.next;
    END LOOP;
    tmp.laenge := counter;
    RETURN counter;
  END Count;

  PROCEDURE Addnext(add : IN integer) IS
  BEGIN
    helpact := tmp.listhead;
    help := tmp.listhead;
    helpto := NULL;
    WHILE help /= NULL LOOP
      helpto := help;
      help := help^.next;
    END LOOP;
    helpto^.next := NEW listpointertype;
    helpto^.next^.item := add;
    helpto^.next^.next := NULL;
    tmp.laenge := tmp.laenge + 1;
  END Addnext;

  PROCEDURE Createfirst IS
  BEGIN
    tmp.listhead := NEW listpointertype;
    tmp.listhead^.item := 2;
    tmp.listhead^.next := NULL;
    tmp.laenge := 1;
    helpact := tmp.listhead;
  END Createfirst;
BEGIN                                   -- Canonic operation of the depot
  tmp.laenge := 0;
  tmp.listhead := NULL;
END listmanager;
                                        -- Actor generator !
MACTOR Primtest(c : IN integer; p : IN integer) IS
BEGIN
  IF c MOD p = 0 THEN
    isPrim := false;
  END IF;
END Primtest;

  prim   : listmanager;                 -- variable local to SYSTEM
  length : integer;
  cand   : integer := 3;
  wurzel : integer;

BEGIN
  prim.Createfirst;
```

```
  WHILE prim.Count <= 200 LOOP
    isPrim := true;
    BLOCK PrimeTestBlock is              -- Block syncing the testers
    BEGIN
      FOR i IN 1 .. prim.Count LOOP
         wurzel := prim.GetNextItem;
         IF wurzel * wurzel <= cand THEN
            Primtest(cand, wurzel);
         END IF;
      END loop;
    END PrimeTestBlock;                  -- end of sync block
    IF isPrim THEN
      prim.Addnext(cand);
    END IF;
    cand := cand + 2;
  END LOOP;
  prim.PrintList;
END System;
```

Another concept illustrated in this example is the use of a `block` to synchronize multiple actors. Block `PrimeTestBlock` at the end of the source text inside the statement part of `SYSTEM` synchronizes all test actors created inside the for–loop. The computation does not leave the block before all actors created inside the loop are terminated.

C Interface of the gcc Back-End

C.1 Important Files

File	Contents
`tree.def`	Contains the definitions and documentation for the different kinds (distinguished by codes) of *tree* nodes available and used in the GNU C compiler.
`tree.h`	Front-end tree definitions for GNU compilers. Defines the union type *tree* and access macros. Declarations of functions to create and expand *trees*. Incomplete list of symbols that a front–end has to define.
`toplev.c`	Top level of GNU compilers. Comprises `main` and some other very useful functions, such as `error`, `warning`, `sorry` and `rest_of_decl_compilation`.

Table C.1: Important files of *gcc*

See `insel/gcc.h` for other relevant files and functions that are needed or helpful to develop a GNU based compiler.

C.2 Symbols Front–Ends Have to Define

Following functions and variable identifiers have to be declared and defined by each front–end. Documentation about the expected semantics can partially be found in `tree.h` and must otherwise be extracted from the source codes of existing compiler, e.g. *gic*.

- Initialization and parsing
  - `init_lex()`
  - `init_decl_processing()`
  - `lang_decode_option()`
  - `lang_init()`
  - `lang_finish()`
  - `lang_identyfy()`
  - `yyparse()`
- Management of lexical scopes
  - `pushlevel()`
  - `poplevel()`
  - `insert_block()`
  - `set_block()`
  - `pushdecl()`
  - `getdecl()`
  - `global_bindings_p()`
  - `kept_level_p()`
  - `copy_lang_decl()`
- Handling of types
  - `incomplete_type_error()`
  - `type_for_size()`
  - `type_for_mode()`
  - `signed_type()`
  - `unsigned_type()`
  - `signed_or_unsigned_type()`
  - `truthvalue_conversion()`

- convert()
- mark_addressable()

- Language specific output
  - print_lang_decl()
  - print_lang_type()
  - print_lang_identifier()
  - print_lang_statistics()
- Expected data
  - language_string
  - error_mark_node
  - integer_type_node
  - char_type_node
  - viod_type_node
  - integer_zero_node
  - integer_one_node
  - current_function_decl
  - flag_traditional

C.3 Important Functions Provided by *gcc*

Following list of C functions represents a selection of important operations provided by the generic back–end. For further information consult the source code of *gcc*, especially `tree.h, tree.c, expr.c and stmt.c`.

- Access to the symbol table:
  - get_identifier()
- To start compilation of a function:
  - push_function_context()
  - announce_function()
  - make_function_rtl()
  - init_function_start()
  - expand_function_start()
  - expand_start_bindings()
- Finishing compilation of a function:
  - expand_end_bindings()
  - expand_function_end()
  - rest_of_compilation()
  - pop_function_context()
- Compilation of declarations
  - expand_decl()
- Expressions
  - expand_expr_stmt()
- Code generation for loop statements
  - expand_end_loop()
  - expand_exit_loop()

  - `expand_exit_loop_if_false()`

- Conditions
  - `expand_start_cond()`
  - `expand_start_elseif()`
  - `expand_start_else()`
  - `expand_end_cond()`

- Error reporting
  - `error()`
  - `warning()`
  - `sorry()`

List of Figures

Bibliography

[ASU88a] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilerbau (part 1)*. Addison–Wesley (Germany) GmbH, 1988. "dragon book".

[ASU88b] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilerbau (part 2)*. Addison–Wesley Verlag (Germany) GmbH, 1988. "dragon book".

[CGS] C. Comar, F. Gasperoni, and E. Schonberg. The gnat project: A GNU-Ada9X compiler. Technical report, Courant Insitute of Mathematical Science, N.Y. University.

[Cze97] C. B. Czech. Architektur und Konzept des Dycos–Kerns. Technical report, TU München, 1997. SFB–Bericht 342/12/97 A TUM–I9717 (german only).

[Dea87] Alan Dearle. Constructing compilers in a persistent environment. In *Proc. of 2nd Int. Workshop on Persistent Object Systems*, Appin, Scotland, 1987.

[GE90] J. Grosch and H. Emmelmann. A tool box for compiler construction. Technical Report 20, GMD, University of Karlsruhe, January 1990.

[GP97] S. Groh and M. Pizka. A different approach to resource management for distributed systems. In *Proc. of PDPTA'97 – International Conference on Parallel and Distributed Processing Techniques and Applications*, July 1997.

[GPR97] S. Groh, M. Pizka, and J. Rudolph. Shadow Stacks - a hardware-supported dsm for objects of any granularity. In *Proc. of IEEE 3rd Int. Conf. on Alogrithms and Architectures for Parallel Processing*, Melbourne, Australia, December 1997.

[GR97] S. Groh and J. Rudolph. On the efficient distribution of a flexible resource management. In *Proc. of EuroPDS'97*, June 1997.

[Gro94] Compiler Tools Group. *Guide for New Eli Users*. University of Colorado, Boulder, CO, USA, 1994.

[Gro96] S. Groh. Designing an efficient resource management for parallel distributed systems by the use of a graph replacement system. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pages 215–225, August 1996.

[Ken95] Richard Kenner. Targetting and retargetting the GNU C compiler. slides, November 1995.

[Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of CS, Yale University, New Haven, CT, October 1986.

[MFLS93] S. Murer, J. A. Feldman, Chu-Cheow Lim, and Martina-Maria Seidel. psather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA, June 1993 November 1993.

[NC96] N. Nagaratnam and G. L. Craig. Fuzzy–based dynamic program reconfiguration. In *Florida AI Research Symposium*. Mai 1996.

[OSF92] OSF. *Introduction to OSF DCE*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[PE97a] M. Pizka and C. Eckert. Evolving software tools for new distributed computing environments. In Prof. H. Arabnia, editor, *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications - PDPTA'97*, Las Vegas, NV, July 1997.

[PE97b] M. Pizka and C. Eckert. A language-based approach to construct structured and efficient object-based distributed systems. In *Proc. of the 30th Hawaii Int. Conf. on System Sciences*, volume 1, pages 130–139, Maui, Hawai, January 1997. IEEE CS Press.

[PH] A. Poetzsch-Heffter. Programming language specification and prototyping using the MAX system. Cornell University, NY.

[Rad95] R. Radermacher. *Eine Ausführungsumgebung mit integrierter Lastverteilung für verteilte und parallele Systeme*. PhD thesis, Technische Universität München, 1995.

[RW96] R. Radermacher and F. Weimer. INSEL Syntax-Bericht. Technischer Bericht TUM-I9617, Technische Universität München, March 1996. german only.

[SEL+96] P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Rader macher, F. Weimer, and H.-M. Windisch. Sprachkonzepte für die Konstruktion Verteilter Systeme. Technischer Bericht TUM-I9618, Technische Universität München, März 1996.

[Sta95] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, November 1995.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, MA, 2nd edition, 1991.

[Str96] C. Strobl. Integration of the language INSEL into gcc. Thesis, June 1996. german only.

[Wei97] Frank Weimer. DAViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL–Programmen. Technical report, TU München, April 1997. SFB–Bericht 342/15/97 A TUM–I9721.

[Win95] H.-M. Windisch. Improving the efficiency of object invocations by dynamic object replication. In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications — PDPTA*, pages 680–688, November 1995.

[Win96] H.-M. Windisch. The distributed programming language INSEL - concepts and implementation. In *High-Level Programming Models and Supportive Environments*, pages 17–25, Honolulu, Hawaii, April 1996. IEEE CS Press.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

342/1/90 A Robert Gold, Walter Vogler: Quality Criteria for Partial Order Semantics of Place/Transition-Nets, Januar 1990

342/2/90 A Reinhard Föẞmeier: Die Rolle der Lastverteilung bei der numerischen Parallelprogrammierung, Februar 1990

342/3/90 A Klaus-Jörn Lange, Peter Rossmanith: Two Results on Unambiguous Circuits, Februar 1990

342/4/90 A Michael Griebel: Zur Lösung von Finite-Differenzen- und Finite-Element-Gleichungen mittels der Hierarchischen Transformations-Mehrgitter-Methode

342/5/90 A Reinhold Letz, Johann Schumann, Stephan Bayerl, Wolfgang Bibel: SETHEO: A High-Performance Theorem Prover

342/6/90 A Johann Schumann, Reinhold Letz: PARTHEO: A High Performance Parallel Theorem Prover

342/7/90 A Johann Schumann, Norbert Trapp, Martin van der Koelen: SETHEO/PARTHEO Users Manual

342/8/90 A Christian Suttner, Wolfgang Ertel: Using Connectionist Networks for Guiding the Search of a Theorem Prover

342/9/90 A Hans-Jörg Beier, Thomas Bemmerl, Arndt Bode, Hubert Ertl, Olav Hansen, Josef Haunerdinger, Paul Hofstetter, Jaroslav Kremenek, Robert Lindhof, Thomas Ludwig, Peter Luksch, Thomas Treml: TOPSYS, Tools for Parallel Systems (Artikelsammlung)

342/10/90 A Walter Vogler: Bisimulation and Action Refinement

342/11/90 A Jörg Desel, Javier Esparza: Reachability in Reversible Free- Choice Systems

342/12/90 A Rob van Glabbeek, Ursula Goltz: Equivalences and Refinement

342/13/90 A Rob van Glabbeek: The Linear Time - Branching Time Spectrum

342/14/90 A Johannes Bauer, Thomas Bemmerl, Thomas Treml: Leistungsanalyse von verteilten Beobachtungs- und Bewertungswerkzeugen

342/15/90 A Peter Rossmanith: The Owner Concept for PRAMs

342/16/90 A G. Böckle, S. Trosch: A Simulator for VLIW-Architectures

342/17/90 A P. Slavkovsky, U. Rüde: Schnellere Berechnung klassischer Matrix-Multiplikationen

342/18/90 A Christoph Zenger: SPARSE GRIDS

342/19/90 A Michael Griebel, Michael Schneider, Christoph Zenger: A combination technique for the solution of sparse grid problems

342/20/90 A Michael Griebel: A Parallelizable and Vectorizable Multi- Level-Algorithm on Sparse Grids

342/21/90 A V. Diekert, E. Ochmanski, K. Reinhardt: On confluent semi-commutations-decidability and complexity results

342/22/90 A Manfred Broy, Claus Dendorfer: Functional Modelling of Operating System Structures by Timed Higher Order Stream Processing Functions

342/23/90 A Rob van Glabbeek, Ursula Goltz: A Deadlock-sensitive Congruence for Action Refinement

Reihe A

342/24/90 A Manfred Broy: On the Design and Verification of a Simple Distributed Spanning Tree Algorithm
342/25/90 A Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Peter Luksch, Roland Wismüller: TOPSYS - Tools for Parallel Systems (User's Overview and User's Manuals)
342/26/90 A Thomas Bemmerl, Arndt Bode, Thomas Ludwig, Stefan Tritscher: MMK - Multiprocessor Multitasking Kernel (User's Guide and User's Reference Manual)
342/27/90 A Wolfgang Ertel: Random Competition: A Simple, but Efficient Method for Parallelizing Inference Systems
342/28/90 A Rob van Glabbeek, Frits Vaandrager: Modular Specification of Process Algebras
342/29/90 A Rob van Glabbeek, Peter Weijland: Branching Time and Abstraction in Bisimulation Semantics
342/30/90 A Michael Griebel: Parallel Multigrid Methods on Sparse Grids
342/31/90 A Rolf Niedermeier, Peter Rossmanith: Unambiguous Simulations of Auxiliary Pushdown Automata and Circuits
342/32/90 A Inga Niepel, Peter Rossmanith: Uniform Circuits and Exclusive Read PRAMs
342/33/90 A Dr. Hermann Hellwagner: A Survey of Virtually Shared Memory Schemes
342/1/91 A Walter Vogler: Is Partial Order Semantics Necessary for Action Refinement?
342/2/91 A Manfred Broy, Frank Dederichs, Claus Dendorfer, Rainer Weber: Characterizing the Behaviour of Reactive Systems by Trace Sets
342/3/91 A Ulrich Furbach, Christian Suttner, Bertram Fronhöfer: Massively Parallel Inference Systems
342/4/91 A Rudolf Bayer: Non-deterministic Computing, Transactions and Recursive Atomicity
342/5/91 A Robert Gold: Dataflow semantics for Petri nets
342/6/91 A A. Heise; C. Dimitrovici: Transformation und Komposition von P/T-Netzen unter Erhaltung wesentlicher Eigenschaften
342/7/91 A Walter Vogler: Asynchronous Communication of Petri Nets and the Refinement of Transitions
342/8/91 A Walter Vogler: Generalized OM-Bisimulation
342/9/91 A Christoph Zenger, Klaus Hallatschek: Fouriertransformation auf dünnen Gittern mit hierarchischen Basen
342/10/91 A Erwin Loibl, Hans Obermaier, Markus Pawlowski: Towards Parallelism in a Relational Database System
342/11/91 A Michael Werner: Implementierung von Algorithmen zur Kompaktifizierung von Programmen für VLIW-Architekturen
342/12/91 A Reiner Müller: Implementierung von Algorithmen zur Optimierung von Schleifen mit Hilfe von Software-Pipelining Techniken
342/13/91 A Sally Baker, Hans-Jörg Beier, Thomas Bemmerl, Arndt Bode, Hubert Ertl, Udo Graf, Olav Hansen, Josef Haunerdinger, Paul Hofstetter, Rainer Knödlseder, Jaroslav Kremenek, Siegfried Langenbuch, Robert Lindhof, Thomas Ludwig, Peter Luksch, Roy Milner, Bernhard Ries, Thomas Treml: TOPSYS - Tools for Parallel Systems (Artikelsammlung); 2., erweiterte Auflage
342/14/91 A Michael Griebel: The combination technique for the sparse grid solution of PDE's on multiprocessor machines
342/15/91 A Thomas F. Gritzner, Manfred Broy: A Link Between Process Algebras and Abstract Relation Algebras?
342/16/91 A Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Thomas Treml, Roland Wismüller: The Design and Implementation of TOPSYS
342/17/91 A Ulrich Furbach: Answers for disjunctive logic programs
342/18/91 A Ulrich Furbach: Splitting as a source of parallelism in disjunctive logic programs

Reihe A

342/19/91 A Gerhard W. Zumbusch: Adaptive parallele Multilevel-Methoden zur Lösung elliptischer Randwertprobleme

342/20/91 A M. Jobmann, J. Schumann: Modelling and Performance Analysis of a Parallel Theorem Prover

342/21/91 A Hans-Joachim Bungartz: An Adaptive Poisson Solver Using Hierarchical Bases and Sparse Grids

342/22/91 A Wolfgang Ertel, Theodor Gemenis, Johann M. Ph. Schumann, Christian B. Suttner, Rainer Weber, Zongyan Qiu: Formalisms and Languages for Specifying Parallel Inference Systems

342/23/91 A Astrid Kiehn: Local and Global Causes

342/24/91 A Johann M.Ph. Schumann: Parallelization of Inference Systems by using an Abstract Machine

342/25/91 A Eike Jessen: Speedup Analysis by Hierarchical Load Decomposition

342/26/91 A Thomas F. Gritzner: A Simple Toy Example of a Distributed System: On the Design of a Connecting Switch

342/27/91 A Thomas Schnekenburger, Andreas Weininger, Michael Friedrich: Introduction to the Parallel and Distributed Programming Language ParMod-C

342/28/91 A Claus Dendorfer: Funktionale Modellierung eines Postsystems

342/29/91 A Michael Griebel: Multilevel algorithms considered as iterative methods on indefinite systems

342/30/91 A W. Reisig: Parallel Composition of Liveness

342/31/91 A Thomas Bemmerl, Christian Kasperbauer, Martin Mairandres, Bernhard Ries: Programming Tools for Distributed Multiprocessor Computing Environments

342/32/91 A Frank Leßke: On constructive specifications of abstract data types using temporal logic

342/1/92 A L. Kanal, C.B. Suttner (Editors): Informal Proceedings of the Workshop on Parallel Processing for AI

342/2/92 A Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: The Design of Distributed Systems - An Introduction to FOCUS

342/2-2/92 A Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: The Design of Distributed Systems - An Introduction to FOCUS - Revised Version (erschienen im Januar 1993)

342/3/92 A Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems

342/4/92 A Claus Dendorfer, Rainer Weber: Development and Implementation of a Communication Protocol - An Exercise in FOCUS

342/5/92 A Michael Friedrich: Sprachmittel und Werkzeuge zur Unterstüt- zung paralleler und verteilter Programmierung

342/6/92 A Thomas F. Gritzner: The Action Graph Model as a Link between Abstract Relation Algebras and Process-Algebraic Specifications

342/7/92 A Sergei Gorlatch: Parallel Program Development for a Recursive Numerical Algorithm: a Case Study

342/8/92 A Henning Spruth, Georg Sigl, Frank Johannes: Parallel Algorithms for Slicing Based Final Placement

342/9/92 A Herbert Bauer, Christian Sporrer, Thomas Krodel: On Distributed Logic Simulation Using Time Warp

342/10/92 A H. Bungartz, M. Griebel, U. Rüde: Extrapolation, Combination and Sparse Grid Techniques for Elliptic Boundary Value Problems

342/11/92 A M. Griebel, W. Huber, U. Rüde, T. Störtkuhl: The Combination Technique for Parallel Sparse-Grid-Preconditioning and -Solution of PDEs on Multiprocessor Machines and Workstation Networks

342/12/92 A Rolf Niedermeier, Peter Rossmanith: Optimal Parallel Algorithms for Computing Recursively Defined Functions

Reihe A

342/13/92 A Rainer Weber: Eine Methodik für die formale Anforderungsspezifkation verteilter Systeme
342/14/92 A Michael Griebel: Grid– and point–oriented multilevel algorithms
342/15/92 A M. Griebel, C. Zenger, S. Zimmer: Improved multilevel algorithms for full and sparse grid problems
342/16/92 A J. Desel, D. Gomm, E. Kindler, B. Paech, R. Walter: Bausteine eines kompositionalen Beweiskalküls für netzmodellierte Systeme
342/17/92 A Frank Dederichs: Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen
342/18/92 A Andreas Listl, Markus Pawlowski: Parallel Cache Management of a RDBMS
342/19/92 A Erwin Loibl, Markus Pawlowski, Christian Roth: PART: A Parallel Relational Toolbox as Basis for the Optimization and Interpretation of Parallel Queries
342/20/92 A Jörg Desel, Wolfgang Reisig: The Synthesis Problem of Petri Nets
342/21/92 A Robert Balder, Christoph Zenger: The d-dimensional Helmholtz equation on sparse Grids
342/22/92 A Ilko Michler: Neuronale Netzwerk-Paradigmen zum Erlernen von Heuristiken
342/23/92 A Wolfgang Reisig: Elements of a Temporal Logic. Coping with Concurrency
342/24/92 A T. Störtkuhl, Chr. Zenger, S. Zimmer: An asymptotic solution for the singularity at the angular point of the lid driven cavity
342/25/92 A Ekkart Kindler: Invariants, Compositionality and Substitution
342/26/92 A Thomas Bonk, Ulrich Rüde: Performance Analysis and Optimization of Numerically Intensive Programs
342/1/93 A M. Griebel, V. Thurner: The Efficient Solution of Fluid Dynamics Problems by the Combination Technique
342/2/93 A Ketil Stølen, Frank Dederichs, Rainer Weber: Assumption / Commitment Rules for Networks of Asynchronously Communicating Agents
342/3/93 A Thomas Schnekenburger: A Definition of Efficiency of Parallel Programs in Multi-Tasking Environments
342/4/93 A Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, Christoph Zenger: A Proof of Convergence for the Combination Technique for the Laplace Equation Using Tools of Symbolic Computation
342/5/93 A Manfred Kunde, Rolf Niedermeier, Peter Rossmanith: Faster Sorting and Routing on Grids with Diagonals
342/6/93 A Michael Griebel, Peter Oswald: Remarks on the Abstract Theory of Additive and Multiplicative Schwarz Algorithms
342/7/93 A Christian Sporrer, Herbert Bauer: Corolla Partitioning for Distributed Logic Simulation of VLSI Circuits
342/8/93 A Herbert Bauer, Christian Sporrer: Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving
342/9/93 A Peter Slavkovsky: The Visibility Problem for Single-Valued Surface (z = f(x,y)): The Analysis and the Parallelization of Algorithms
342/10/93 A Ulrich Rüde: Multilevel, Extrapolation, and Sparse Grid Methods
342/11/93 A Hans Regler, Ulrich Rüde: Layout Optimization with Algebraic Multigrid Methods
342/12/93 A Dieter Barnard, Angelika Mader: Model Checking for the Modal Mu-Calculus using Gauß Elimination
342/13/93 A Christoph Pflaum, Ulrich Rüde: Gauß' Adaptive Relaxation for the Multilevel Solution of Partial Differential Equations on Sparse Grids
342/14/93 A Christoph Pflaum: Convergence of the Combination Technique for the Finite Element Solution of Poisson's Equation
342/15/93 A Michael Luby, Wolfgang Ertel: Optimal Parallelization of Las Vegas Algorithms

Reihe A

342/16/93 A Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, Christoph Zenger: Pointwise Convergence of the Combination Technique for Laplace's Equation

342/17/93 A Georg Stellner, Matthias Schumann, Stefan Lamberts, Thomas Ludwig, Arndt Bode, Martin Kiehl und Rainer Mehlhorn: Developing Multicomputer Applications on Networks of Workstations Using NXLib

342/18/93 A Max Fuchs, Ketil Stølen: Development of a Distributed Min/Max Component

342/19/93 A Johann K. Obermaier: Recovery and Transaction Management in Write-optimized Database Systems

342/20/93 A Sergej Gorlatch: Deriving Efficient Parallel Programs by Systemating Coarsing Specification Parallelism

342/01/94 A Reiner Hüttl, Michael Schneider: Parallel Adaptive Numerical Simulation

342/02/94 A Henning Spruth, Frank Johannes: Parallel Routing of VLSI Circuits Based on Net Independency

342/03/94 A Henning Spruth, Frank Johannes, Kurt Antreich: PHIroute: A Parallel Hierarchical Sea-of-Gates Router

342/04/94 A Martin Kiehl, Rainer Mehlhorn, Matthias Schumann: Parallel Multiple Shooting for Optimal Control Problems Under NX/2

342/05/94 A Christian Suttner, Christoph Goller, Peter Krauss, Klaus-Jörn Lange, Ludwig Thomas, Thomas Schnekenburger: Heuristic Optimization of Parallel Computations

342/06/94 A Andreas Listl: Using Subpages for Cache Coherency Control in Parallel Database Systems

342/07/94 A Manfred Broy, Ketil Stølen: Specification and Refinement of Finite Dataflow Networks - a Relational Approach

342/08/94 A Katharina Spies: Funktionale Spezifikation eines Kommunikationsprotokolls

342/09/94 A Peter A. Krauss: Applying a New Search Space Partitioning Method to Parallel Test Generation for Sequential Circuits

342/10/94 A Manfred Broy: A Functional Rephrasing of the Assumption/Commitment Specification Style

342/11/94 A Eckhardt Holz, Ketil Stølen: An Attempt to Embed a Restricted Version of SDL as a Target Language in Focus

342/12/94 A Christoph Pflaum: A Multi-Level-Algorithm for the Finite-Element-Solution of General Second Order Elliptic Differential Equations on Adaptive Sparse Grids

342/13/94 A Manfred Broy, Max Fuchs, Thomas F. Gritzner, Bernhard Schätz, Katharina Spies, Ketil Stølen: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems

342/14/94 A Maximilian Fuchs: Technologieabhängigkeit von Spezifikationen digitaler Hardware

342/15/94 A M. Griebel, P. Oswald: Tensor Product Type Subspace Splittings And Multilevel Iterative Methods For Anisotropic Problems

342/16/94 A Gheorghe Ştefănescu: Algebra of Flownomials

342/17/94 A Ketil Stølen: A Refinement Relation Supporting the Transition from Unbounded to Bounded Communication Buffers

342/18/94 A Michael Griebel, Tilman Neuhoeffer: A Domain-Oriented Multilevel Algorithm-Implementation and Parallelization

342/19/94 A Michael Griebel, Walter Huber: Turbulence Simulation on Sparse Grids Using the Combination Method

342/20/94 A Johann Schumann: Using the Theorem Prover SETHEO for verifying the development of a Communication Protocol in FOCUS - A Case Study -

342/01/95 A Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids

Reihe A

342/02/95 A Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law
342/03/95 A Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space
342/04/95 A Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes
342/05/95 A Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems
342/06/95 A Maximilian Fuchs: Formal Design of a Model-N Counter
342/07/95 A Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology
342/08/95 A Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies
342/09/95 A Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness
342/10/95 A Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design
342/11/95 A Thomas Schnekenburger: The ALDY Load Distribution System
342/12/95 A Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm
342/13/95 A Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming
342/14/95 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks
342/15/95 A Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs
342/16/95 A Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik
342/17/95 A Georg Stellner: Using CoCheck on a Network of Workstations
342/18/95 A Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications
342/19/95 A Thomas Schnekenburger: Integration of Load Distribution into ParMod-C
342/20/95 A Ketil Stølen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication
342/21/95 A Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control
342/22/95 A Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes
342/23/95 A Petr Jančar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation
342/24/95 A M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems
342/01/96 A Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries
342/02/96 A Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs
342/03/96 A Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes
342/04/96 A Thomas Huckle: Efficient Computation of Sparse Approximate Inverses
342/05/96 A Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS — On-line Monitoring Interface Specification
342/06/96 A Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components

Reihe A

342/07/96 A	Richard Mayr: Some Results on Basic Parallel Processes
342/08/96 A	Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht
342/09/96 A	P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme
342/10/96 A	Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PF-SLib – A File System for Parallel Programming Environments
342/11/96 A	Manfred Broy, Gheorghe Ştefănescu: The Algebra of Stream Processing Functions
342/12/96 A	Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete
342/13/96 A	Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Many-to-Many Data-flow Networks
342/14/96 A	Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project
342/15/96 A	Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time μ -Calculus
342/16/96 A	Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen
342/17/96 A	Richard Mayr: Model Checking PA-Processes
342/18/96 A	Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States
342/01/97 A	Tobias M"uller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsf"ahigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken
342/02/97 A	Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations
342/03/97 A	Bernhard Mitschang: Technologie f"ur Parallele Datenbanken - Bericht zum Workshop
342/04/97 A	nicht erschienen
342/05/97 A	Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik
342/06/97 A	Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump
342/07/97 A	Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming
342/08/97 A	Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement
342/09/97 A	Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B Wolfgang Reisig: Petri Nets and Algebraic Specifications

342/2/90 B Jörg Desel: On Abstraction of Nets

342/3/90 B Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems

342/4/90 B Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme

342/1/91 B Barbara Paech1: Concurrency as a Modality

342/2/91 B Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox - Anwenderbeschreibung

342/3/91 B Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen

342/4/91 B Werner Pohlmann: A Limitation of Distributed Simulation Methods

342/5/91 B Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis

342/6/91 B Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme

342/7/91 B W. Reisig: Concurrent Temporal Logic

342/1/92 B Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support
Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support

342/2/92 B Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen

342/1/93 B Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung

342/2/93 B Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick

342/1/94 B Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS