The GNU 64-bit PL8 compiler: Toward an open standard environment for firmware development

W. Gellerich
T. Hendel
R. Land
H. Lehmann
M. Mueller
P. H. Oden
H. Penner

For two decades, large parts of zSeries® firmware have been written in the PL8 programming language. The existence of a large amount of mature zSeries firmware source code and our excellent experience with PL8 for system programming suggest keeping this language. However, the firmware address space of today's zSeries servers may exceed 2 GB, raising the need for a new 64-bit PL8 compiler, since the original implementation, developed at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, supports only 32-bit platforms. The GNU compiler collection (GCC) (GNU is a freeware UNIX®-like operating system) has been used to translate those parts of firmware written in C for some years and has also proved successful in compiling Linux™ for zSeries. This fact, combined with the highly modular GCC design, suggested reimplementing PL8 within the GCC framework. In this paper, we report on the extension of PL8 to support 64-bit addressing, its implementation as a GCC front end, and the validation of the new compiler. We also evaluate PL8 as a language for highly reliable low-level programming and give some performance data. The paper documents the high level of quality achieved by the GCC open-source project and how such software fits into the traditional IBM software development processes.

Introduction and overview

zSeries* firmware is the software layer between the operating system (OS) and hardware. Its development is characterized by two elements: a very high demand for correct behavior, because firmware bugs could crash any application program running on any OS, and the fact that it requires low-level programming, i.e., accessing specific addresses and dealing with individual bits. Firmware has many interfaces to hardware registers and to assembler-written routines implementing low-level services.

The next section of this paper gives an overview and some background information concerning zSeries firmware development and puts the GNU PL8[1] compiler project into a larger context. We then discuss PL8 in more detail, giving an overview of its features, its history, and a more detailed discussion of its advantages for reliable low-level programming. The section that follows is about the project itself and gives details about the GNU compiler collection (GCC), and in particular the PL8 compiler front end. The important task of validating correctness and performance

[1] The GNU Project was launched in 1984 to develop a complete UNIX-like operating system which is free software: the GNU system. (GNU is a recursive acronym for "GNU's Not UNIX"; it is pronounced "guh-noo." See *http://www.gnu.org/*.) Since there are a few differences between the original pl.8 and the current GNU implementation, we avoid confusion by referring to the latter as PL8.

0018-8646/04/$5.00 © 2004 IBM

of the new compiler is addressed in the next section. The last section is about process issues, describing when and how new versions of the compiler were released during its development. This topic was of particular importance, since the new compiler and the firmware being compiled by it were developed in parallel.

zSeries firmware development and the role of PL8

The high-level programming language pl.8 was first used for developing the firmware of the 9377 Enterprise Server, an S/370* machine, in the early 1980s. At that time almost all firmware was written as horizontal microcode using special assemblers. The idea of using high-level languages and compilers for microcode resulted, as a matter of course, in many complaints about performance degradation.

Firmware performs the instruction interpretation of the very complex z/Architecture* instructions. While RISC-style instructions are directly implemented in processor hardware, the next level of firmware, called *millicode* [1], implements the medium-level instructions that occur frequently and are highly performance-critical. The internal 390 (i390) firmware layer performs the I/O redundant path management, I/O load balancing, recovery from hardware and firmware errors, and the system management functions in zSeries. These functions in other computer systems are typically implemented in operating-system layers. The interpretation of the control blocks associated with I/O instructions and system management functions requires precise control over the data layout in records and unions. Recovery functions provided by the i390 firmware layer perform many low-level hardware register accesses mainly in the hierarchical structure of the I/O hardware paths. They also require precise bit positioning in data structures, including arrays of bit-strings smaller than a byte.

The superior optimization and instruction scheduling of the Yorktown pl.8 compiler [2, 3] and the tremendous enhancements in productivity accompanying the use of a high-level language confirmed the validity of using a high-level language for firmware implementation. Two decades of using pl.8 to develop firmware demonstrated that this high-level language could successfully handle bit-precise data structure layout and provide reliability, both by detecting errors at compile time and by reporting defects such as array bounds violations at run time. Coupled with achieving excellent performance, this proved that choosing the pl.8 language back in the 1980s was the right decision then and remains so today.

The z900, predecessor of the z990, provides a true 64-bit z/Architecture for the hypervisor and the OS running in the logical partitions. The i390 firmware on the z900 uses the 32-bit instruction set, compatibility features built into the z/Architecture, and some special hardware support in the processing unit to provide the 64-bit z/Architecture.

For the z990, the greater number of CPUs, channels, and logical partitions required the introduction of *multiple channel subsystems* [4] and *multiple subchannel sets.* The demand for a hardware system area (HSA), a part of the system memory used by internal firmware, could increase to more than 2 GB. This forced the i390 firmware to extend into a true 64-bit implementation with fully unconstrained addressing in the HSA. All other proposals to implement multiple channel subsystems by some macros and special hardware support in the processor turned out to be either unacceptably complex or, because it would involve porting all i390 code to PL/I or rewriting all firmware code in C, not affordable. A total rewrite would have thrown the z990 completely off schedule or even caused a delivery pause of one system, while introducing a lot of additional instability and firmware-caused outages.

The no-longer-supported Yorktown compiler could neither use the new performance-boosting instructions of the 32-bit S/390* architecture nor the z/Architecture 64-bit instructions. This old compiler was also tied to the library and build environment on VM/CMS, an old mainframe operating system, as its only execution platform. Altogether, this was a decade-long unmaintained dead end. The quality of the Yorktown pl.8 compiler and the development library system allowed us to focus on adding many new functions to systems in the S/390 family and to leverage the original tools investment.

The new GNU PL8 compiler runs on any GCC-supported central processing unit (CPU) architecture and OS as well as on the library and build environment on VM/CMS, and generates code for any supported target machine. It also immediately benefits from the instruction-scheduling descriptions for performance optimization that are provided for Linux** for zSeries on the GNU C compiler, because it is based on the same target machine description. The GNU-based PL8 compiler lays the foundation for a true 64-bit i390 execution environment and opens migration paths to modern development library systems and build tools running under Linux for zSeries as the development platform for the PL8-based i390 firmware of the z990 machine.

PL8 programming language

The pl.8 language was an outgrowth of a subset of PL/I, which was itself developed by IBM during the 1960s as one of the first general-purpose languages [2, 5, 6]. Before that, the scientific community used Fortran, while commercial applications were mostly written in COBOL. PL/I combines elements from both languages to provide a unified solution for both purposes. This was particularly important because the distinction between business and

scientific applications was becoming less sharp: Business applications began including mathematical functionality, such as regression analysis, while scientists began collecting and organizing large files of data that required a kind of functionality previously used only in commercial applications.

Development and history of pl.8

In 1976, a project was started at the Thomas J. Watson Research Center to design and build the first RISC machine, code-named *801*. A significant part of the aim of the project was to show that a high-level programming language compiler could be built that would use the general algorithmic techniques espoused by John Cocke and others [7, 8] to generate very efficient code for the RISC architecture. It was so efficient, in fact, that the OS, running on general-purpose hardware, could be coded largely in that high-level language without compromising performance. That goal, also sought by several others at about that time, was actually attained. An additional aim was to investigate the extent to which linguistic means could be used to achieve safe code, e.g., protected accesses to data that eliminate illegal references, again without materially compromising performance.

To make a convincing demonstration, it was decided to use a high-level language, initially a restricted version of PL/I, to code not only the OS, but the compiler itself. The self-imposed restrictions were chosen to avoid those features of PL/I that are unsafe and/or would create difficulties for optimization: primarily aliasing and pointers. Pointers to variables and reference parameters are easy to abuse and create aliases in memory whose disentanglement requires a level of sophistication of program analysis that was not yet available.

Eventually, enough of the compiler had been programmed to enable the use of the compiler itself as the development tool. At this point, some of the linguistic constraints and ideas that had been brewing could be formally institutionalized in the language, since the team had achieved complete control over the compiler.

The language was named *pl.8*, and was allowed to evolve, but in a very controlled way. The ".8" originally indicated that it was 8/10 of PL/I, but it became something of a misnomer over time as features not present in PL/I were added. These included value parameters, first-class bit-field types, and an explicit storage model which allowed for declarative overlays that were well-defined (contrasted with the C union attribute) and that associated data types only with symbols (contrasted with the C casts of values). A limited form of data and function pointers was added.

To achieve a measure of safe code, some semantic additions were explicitly incorporated into the language. Type checking was statically enforced. Array bounds and area references were always guaranteed safe: The compiler added code performing dynamic checks whenever it could not prove this to be unnecessary. A *by value* parameter attribute was added to the language to encourage safe parameter passing where possible. The linker checked function definition attributes (including by value) against function declaration attributes (entry declarations) providing link-time mismatch detection. The linkage conventions were designed to be as efficient as possible to encourage modular program development.

Parameter values were passed in registers whenever possible, as were return values. There was even a declaration-based in-lining capability. As an aside, since the conventions were system-wide, it was possible to compose programs from procedures written in different languages. Optimizations included constant propagation, common subexpression eliminations (CSEs), reassociation, elimination of partial redundancy, some loop optimizations, detection and elimination of dead code, test subsumption (which made it feasible to leave bounds checking always on at an estimated cost of less than 10% performance degradation), and register allocation based on graph coloring [3]. Later, more optimizations were included, some specialized to target architectures.

The experiment was regarded as successful in that the pl.8 language was entirely adequate to write most of the experimental OS and the compiler itself. The RISC architecture proved to be an excellent target for the optimizing compiler, resulting in code with outstanding performance. Many of the programmers exposed to pl.8 became quite productive and even fond of it, not least because the checking features were of enormous help in debugging (no more "wild stores"). This experience also made pl.8 a good choice as a language for firmware development. We provide more details about reliability issues in the next two sections.

While pl.8 was being reimplemented as a new GCC language, some additions and changes were made. All data types were extended to explicitly support 64 bit, and the rules for type conversion were changed accordingly. Also, a few changes were needed to meet certain constraints imposed by the GCC framework.

Control structures in PL8

PL8 is a procedural language that more or less supports the same control structures found in other programming languages. It provides a rich set of loop statements and several selection statements. Loops at any nesting level can be terminated. The elaborate set of PL8 control structures strongly reduces the programmer's need to use unstructured jump statements.

The so-called "`GOTO` density metric" is defined as the average number of lines of code between two `GOTO` statements. A comparison of Fortran programs written in

Table 1 Comparison of Fortran programs written in the 1970s and today's C, Ada, and zSeries firmware written in PL8, revealing `GOTO` densities that differ by several orders of magnitude.

	Fortran on punched cards	*C*	*Ada*	*PL8*
Files without `GOTO`	(none)	81.5%	99.4%	98.5%
Lines/`GOTO`	About 10 (8–13% of all statements are `GOTO`s)	386	13614	1310

the 1970s to today's C and Ada code revealed `GOTO` densities that differ by several orders of magnitude [9, 10]. Obviously, the more powerful the control structures of a language, the less frequently programmers use `GOTO`. We compared the reported values with those measured for zSeries firmware written in PL8 (**Table 1**).

Most of the PL8 files do not contain any `GOTO` at all. An (admittedly rough) analysis revealed that many of the `GOTO`s were used for error handling, which is generally considered acceptable. Since the `GOTO` density also correlates to software error rates [10], the above results suggest that the set of PL8 control structures contributes to firmware quality. Similar statements hold for data structuring, as is shown in the next section.

Low-level data structuring: PL8 compared with C

As was pointed out in the section on zSeries firmware development, there are significant differences to the high-level application programming. This also suggests differences in the level of support required by the programming language used. Here, PL8 has several advantages over C, which is sometimes considered a good choice for low-level programming. We discuss these advantages using two examples. The term *advantage* means that typical problems occurring in firmware development can be solved more easily and in a less error-prone manner in PL8.

Structure layout specification at bit level

Firmware must reference hardware registers and access data that is mapped to a precisely defined storage layout at the level of individual bits. These data areas introduce interfaces to routines which are written in Assembler language.

As a first example, consider this data structure declaration with redefinition in PL8:

```
DCL 1 DataRecord,
      2 FullWord BIT (64),
     .2 BitLayout,
        3 Flag1 BIT (1),
        3 *     BIT (1),
        3 Flag2 BIT (1);
        3 *      BIT (32),
        3 Subreg BIT (29);
```

Here, `DataRecord` is a structure with `FullWord` as its first field. The numbers indicate the level of structures and substructures. The type of `FullWord` is a bit string of length 64, which can also be viewed as an unsigned 64-bit integer, i.e., a machine word. Machine words are the unit of storage usually read and stored by elementary instructions. A pattern typically found in firmware development is that such a machine word can hold quite a large amount of differing information, usually made up of single bits or small bit strings.

In the example, the 64-bit field is redefined with a structure named `BitLayout`. The redefinition is indicated using a dot before the level number. Such a redefining field occupies the same storage as its redefinee. Thus, the first bit of `FullWord` is given the name `Flag1` and can be accessed this way. The next bit of `FullWord` has, in our example, no meaning and is thus explicitly not given a name, as the star indicates. Next comes another bit named `Flag2`, then 32 unused bits, and finally a bit string of length 29 having the name `Subreg`. By using stars as filler, the programmer can realize any storage layout. The PL8 compiler generates all necessary bit masking to access the required bits. In C, such a description of bit positions is not possible. While C provides so-called bit fields, it does not give the programmer control of their actual position in memory, but leaves this issue implementation-defined.

The PL8 concept of redefinitions is significantly different from the union data type found in C or the so-called variant records available in Pascal and Ada. These approaches all share the concept of storing differently typed fields at the same memory location, but the semantics when accessing the overlapping fields is different. Ada, Pascal, and C do not want programmers to store a value using one variant and then read the same data item using a differently typed field. C leaves the semantics of doing so undefined, and Ada has an elaborate mechanism which actually prevents this kind of type conversion. By contrast, the PL8 redefinition explicitly has the semantics of viewing the same storage using different types at the same time. It is perfectly legal, for example, to store a value in `SubReg` and then read `FullWord`, which will have the respective bits changed.

Conversely, assigning 0 to `FullWord` clears all fields belonging to `BitLayout`.

Local pointers

Explicitly dealing with pointers and addresses is error-prone. The C language, for example, is rather liberal with pointers, thus making pointers a frequent source of hard-to-find problems. In contrast, the pointers of PL8 are variables declared as `OFFSET` types. An offset variable is bound to a particular `AREA`, which is simply a contiguous sequence of bytes. The value of an offset variable is used as a displacement local to that area and specifies the storage location of a variable declared with storage class `BASED`. Based variables have no memory of their own, but are associated with a storage location using an offset expression. A based variable is essentially a template, much like a data type, describing how the data it references is to be viewed. Hence, the absolute address of a based reference is the address of the area plus the value of the offset variable, conceptually similar to an array reference. The following is an example of a `BASED/OFFSET` declaration:

```
DCL IntVar INTEGER BASED (OffsVar),
    OffsVar OFFSET (AreaVar),
    AreaVar AREA (1024);

OffsVar = 512;
IntVar = 99;
```

Areas always have a certain size, and the PL8 compiler guarantees that an access to a based variable accesses only memory belonging to the appropriate area. When possible, an error message is issued at compile time; otherwise, the compiler generates a run-time check. Thus, the potential risk of unwanted data access through pointers is limited. For example, the PL8 source code shown in the above example uses an area `AreaVar` consisting of 1024 bytes. The declarations associate `IntVar` with `OffsVar`, and `OffsVar` with `AreaVar`. The PL8 checking mechanism would cause the above code to terminate with a run-time error at the reference to `IntVar` if `OffsVar` had the value 1022, since the variable `IntVar` requires four bytes and must fit completely into `AreaVar`.

The general form of pointers that contain real addresses rather than displacements is also available by means of a special, predefined area variable named `$MEMORY`, which equals the system main storage. Pointer variables as used in C are thus a special case of the more elaborate PL8 concept. The explicit syntax used in PL8 makes it easy to check all cases in which real addresses are used—for example, as part of a formal code review.

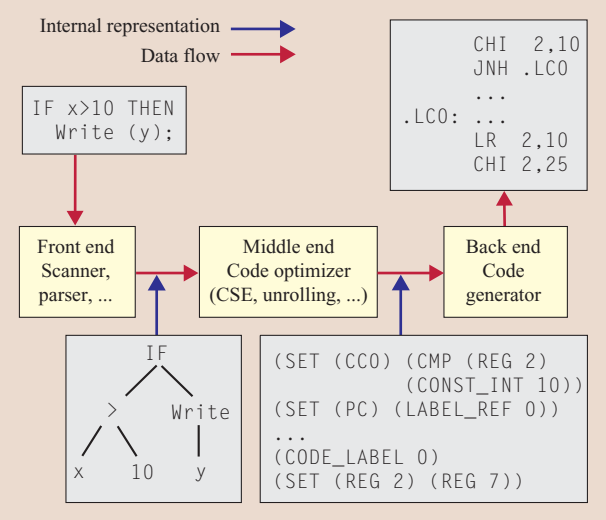

Figure 1

General structure of GCC and its intermediate program representations.

GNU compiler collection (GCC)

Compiling a PL8 file, `prog.p18`, with GNU PL8 consists of the following steps:

1. The PL8 source file is preprocessed to expand all macro calls and include files. The preprocessor output is named `prog.ipl`. Source code not containing any preprocessor directives can omit this step.
2. The GNU PL8 compiler translates the PL8 source code into assembler source code. The example file is named `prog.s`.
3. This assembler source code is assembled by the GNU assembler, resulting in a file called `prog.o`.
4. All `.o` files belonging to the current project are linked together, yielding an executable file.

GCC structure

The input to a compiler is program source code written in a higher-level programming language that is more reflective of the way a human being thinks. The purpose of a compiler is to translate such source code into machine-executable form.

Usually this translation process is organized into three phases (**Figure 1**). The first phase, called the compiler *front end*, analyzes the source code and checks whether it conforms to the programming language specification. Most of the compiler error messages and warnings are generated by this phase. The source code analysis results in an internal program representation. GCC uses attributed syntax tree fragments for this purpose.

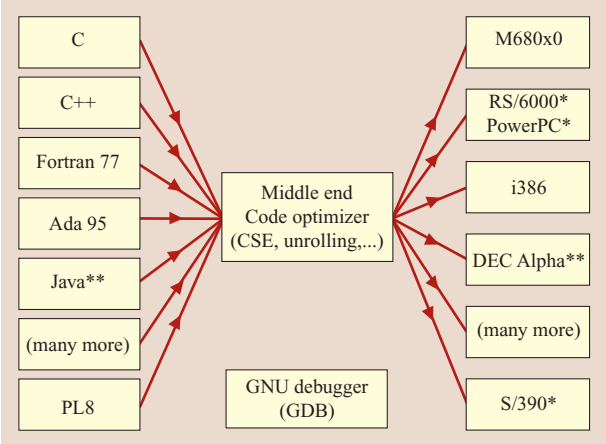

Figure 2

Modular structure of the GNU compiler collection (GCC).

The second phase, called the *middle end* (ME), an unfortunate oxymoron, translates the internal program representation from tree format into a second internal format, called *register transfer language* (RTL). The most important building blocks are RTL instructions (called *INSNs* in GCC terminology), which describe side effects on registers. In particular, this means that the highly recursive syntax-tree-based representation is replaced by a purely sequential stream of INSNs.

At the beginning, a stream of INSNs is generated while assuming an infinite number of registers. Each optimizer pass transforms that INSN sequence into a semantically equivalent sequence, which is supposed to be less expensive with respect to a given cost function (usually execution time, but GCC can also optimize for space). Available optimizations include common subexpression elimination (CSE), loop optimizations, jump optimizations, and elimination of unreachable code. Most optimizations are machine-independent, so very little platform-specific information is needed.

In addition, a set of highly platform-dependent optimizations exists. The instruction combiner attempts to combine a group of dependent INSNs into fewer, more efficient INSNs. This pass is capable of exploiting even the most bizarre instructions available in z/Architecture [11, 12]. The instruction scheduler depends not only on the target architecture, but also on the given implementation of this architecture; e.g., for a z990, the optimal instruction stream may differ from that for the z900.

So far, the instruction stream still references an infinite number of (virtual) registers and must be transformed to an instruction stream using only the platform registers. This register allocation also depends on the application binary interface (ABI). Since the register allocator decides which virtual registers are kept in memory, the process has a profound influence on the efficiency of the generated code. Finally, the third phase, named *back end* or *code generator*, translates the INSN stream into assembler source code.

Note that the three phases of the compiler communicate via two data structures for internal program representation. These data structures are almost independent of both the programming language used as input and the assembler source code to be output. This modular approach allows front ends for many different programming languages and code generators for many different target architectures to be combined in a way shown in **Figure 2**. It now becomes obvious why this package is called the GNU compiler *collection*. Major advantages of the highly modular GCC structure are that adding a new language requires writing only a new front end, while the code optimization and all existing back ends can be reused, thus making the new language immediately available on a large variety of machines. Conversely, developing a new code generator makes all languages supported by GCC available on the new architecture. The next two sections describe our PL8 front end and the zSeries code generator.

PL8 front end

In contrast to most other GCC front ends, the PL8 front end is itself organized in two passes. The reason is that PL8 allows forward references to declarations. The two-pass approach also simplifies certain other translations. The first pass performs lexical and syntactic analysis, which we implemented using the open-source compiler-generating tools Flex and Bison, respectively. Flex generates a program for the lexical analysis from a regular specification describing how keywords, special characters, identifiers, or numbers are defined in PL8. Bison converts a specification containing the PL8 context-free grammar rules into C source code. Flex and Bison specifications allow definition of the actions to be taken if PL8 constructs are recognized (primarily storing and checking). The PL8 front end is approximately 50K lines of code (LOC) in size.

The output of pass 1 is a front-end internal representation of the input program, which is a kind of attributed syntax tree. Trees are implemented as records with fields containing data or pointers to other tree nodes. The most important kinds of trees are declarations, types, expressions, and constants. Whenever possible, GCC predefined tree nodes are used to represent PL8 constructs; for example, this is done for `IF`, `DO WHILE`, and

DO UNTIL statements. More elaborate statements, such as SELECT and counting loops, are first translated into PL8-specific nodes, as are most declarations concerning PL8 features such as BASED, OFFSET, and REDEFINES.

Pass 2 starts working on the data structures generated by pass 1 and performs a couple of semantic checks. These include type-compatibility checks to verify that two variables are assignable. Implicit type conversions are inserted if the PL8 language definition allows the assignment of variables with different types. Range checks are generated for array accesses, and for all accesses to BASED variables via OFFSETs. Pass 2 also does some optimizations, e.g., constant folding and elimination of array bounds checks if it can determine at compile time that an index will never be out of the valid range. Finally, the PL8-specific nodes are translated into GCC-defined tree nodes and are passed to the GCC middle end.

The example shown in the local pointers section demonstrates how the PL8 concept of BASED variables is handled by the compiler. When accessing IntVar, one cannot simply build a normal declaration node and pass it to the ME, because the concept of BASED variable is unknown there. Rather, the access requires an expression consisting only of tree nodes: Take the address of AreaVar, add the value of OffsVar, do the bounds check, and finally access the memory at the computed location, treating the content as an integer value.

PL8 has its own rules describing how to pack arrays of small bit strings. For example, in

```
DCL cpu(64) BIT(1);
    cpu(17) = 1;
```

the array cpu is 64 bits long. There is no expression tree known to the middle end with the semantic "take bit 17 from the left of a variable with a size of 64 bits and set it to 1." However, the middle end knows array references, shift, and bitwise OR expressions. Thus, the PL8 front end builds the following expression (simplified):

- Regard cpu as an array of eight bytes and build an array reference that accesses the third byte from the left.
- Build a shift expression that shifts the constant "1" seven bits to the left.
- Build a bitwise OR expression with the array reference of the third byte of cpu and the shift expression as operands and assign the result to the third byte of cpu.

This example also touches upon efficiency. A simple assignment creates a number of expressions. It is not sufficient merely to find a way to implement PL8 specialties; the resulting code must also be efficient. Integrating PL8 as an official GCC front end may generate heretofore unseen code sequences that engender the development of new middle-end optimizations to boost the performance of the generated code.

In some cases, we were faced with subtle semantic differences between PL8 and the C-based GCC. Recall from the previous section that C leaves the semantics of storing into one variant of a union type and then reading a different variant undefined, while PL8 explicitly describes how both variants of a redefinition are to be mapped (overlaid). The problem was that the data-dependence analysis exploits the C rules, assuming that certain code movements were correct. Our first approach to map PL8 redefinitions onto tree nodes designed for union types failed for this reason.

Yielding the proper warning or error messages is also a task of the front end. Precise line-number information is sometimes difficult to obtain.

The quality of compiler warnings and error messages, and in particular the precision of their references into the faulty source code locations, have a significant impact on customer satisfaction and on the productivity of the teams using the compiler. Because of the two-pass structure of the front end, generating line number information had to be supported by the internal program representation. Several new warnings were suggested by PL8 programmers.

GCC code generator for zSeries

The task of writing a target code generator for GCC is to provide all information needed to generate INSNs, optimize these INSNs, and generate assembler code for the target platform. This is done for GCC by providing two files: the machine description and the target macros.

The machine description is a file consisting of textual RTL code fragments describing the target architecture. This is done to enable the ME to extract the set of available instructions, the explicit semantics, and the assembler syntax of these instructions. The example shown in **Figure 3** demonstrates this more clearly.

The "addsi3" tells the ME that an add instruction for "si" (single integer) is available. The terms in square brackets describe the semantics. In this case, a register is added to a second operand, which can be a register, an immediate value, or a memory access. The result is stored in the register given as first operand. As a second side effect, the condition code is changed in an arbitrary way. The term following the @ is the assembler code for the different operand types, where the %i are replaced by the specific operand i.

The target macros describe the ABI and architecture-specific parameters to the ME. In particular, the available registers, stack layout, calling conventions, and many more issues are described as a set of C macros.

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "register_operand" "=d,d,d,d")
        (plus:SI (match_operand:SI 1 "nonimmediate_operand" "%0,0,0,0")
                 (match_operand:SI 2 "general_operand" "d,K,R,T")))
   (clobber (reg:CC 33))]
 ""
 "@
  ar\\t%0,%2
  ahi\\t%0,%h2
  a\\t%0,%2
  ay\\t%0,%2"
 [(set_attr "op_type" "RR,RI,RX,RXY")])
```

Figure 3

Example of an INSN as used in the GCC machine description.

Altogether, a back end consists of about 15K lines of code, which is fairly small compared with the rest of the compiler. This demonstrates again the effectiveness of the modular structure of GCC, especially considering that GCC on zSeries produces code with performance close to that of commercial compilers specific to this platform.

Validation of GNU PL8

The term *validation* subsumes all activities to ensure that the compiler behaves according to the PL8 language reference manual. The first part of our validation effort covers component and unit test and is performed by the development team. It is called the *development regression test*. The second step is part of the integration and final product validation process. It covers complete product validation in user scenarios, is performed by an independent validation team, and is referred to as the *firmware focus test*. Both steps are described in detail in the following sections. As a third step, certain parts of the compiler were checked by formal code reviews. (Note that the actual hardware of the new machine was developed in parallel with the firmware and was thus not available for testing.)

Development regression test

The first step for the validation of GNU PL8 (GPL8) was to develop a large set of small and specific test cases that covered the complete language. These tests were developed for component and unit test; later, the complete set of test cases served as a regression test package that was used to verify new compiler versions. Based on the PL8 language reference manual (LRM), parts of the language were assigned to experienced PL8 programmers who systematically wrote test cases for their assigned language subset. This also included error test cases to verify that the compiler detects erroneous PL8 code constructs and properly reports them to the programmer. The test coverage was later checked with reviews.

A shell script, later replaced by a Perl program, was used to compile all test cases, execute them, and check the observed behavior and the compiler warning and error messages against expected results. The results are stored in a report text file. Thus, previous test reports can be compared with others that result from a modified compiler. A Perl program analyzes any differences within the reports.

With respect to test strategies, our approach is a combination of input coverage and equivalence class testing. Although this may sound much like a black-box strategy, taking care of only the compiler input–output behavior, it also includes a glass-box-like strategy, since compiler authors were involved in the test-case-coverage reviews and made suggestions for useful additional tests based on their knowledge of the compiler internals.

The regression test package was later extended in several ways. As already mentioned, the original pl.8 compiler was developed at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York [2]. Another pl.8 compiler was later written at the IBM Toronto Laboratory. We asked our colleagues for help, and they kindly provided us all the pl.8 test cases they had used to verify their pl.8 compilers. We had, of course, to rewrite return code handling and related issues to make those test cases fit into our automated evaluation environment. Parts of that work were done by students who, by the way, had no problem becoming familiar with PL8. Also, for each compiler problem that was detected in its production environment and during the i390 focus test, a specific test case was developed and added to the

regression test package. At the end of the GNU PL8 project, the package consisted of roughly 3500 test programs. This approach to testing proved to be extremely valuable in verifying compiler modifications.

As a last step of the build validation test, a compile of the complete PL8 i390 source code was performed. The result had to achieve INITIAL MICROCODE LOAD COMPLETE in the z/CECSIM simulator [13].

Code reviews

In addition to the test activities, formal code reviews were conducted. This validation technique was developed three decades ago by Michael Fagan, at that time an IBM Senior Technical Staff Member. His original work has recently been republished [14], together with the progress achieved since then. Code reviews tend to reveal problems that are difficult to find with testing, and vice versa, so both are useful as a combination.

Because of the effort involved with code reviews, we did not check the whole source code of GPL8, but only those parts the team considered to be critical. The number of defects found was limited, since we began code reviews rather late.

Firmware focus test

As mentioned earlier, the firmware focus test was the main part of an independent test activity to verify correctness and completeness of the GPL8 compiler deliverables for the zSeries firmware development teams. This part of the validation activity is strictly a black-box test strategy, as it focuses only on the validation of the correct functionality as required for firmware development.

The validation effort used the usual set of tools required for zSeries firmware code generation [5, 13]. The major (and fundamentally new) part is the GPL8 front end to the GCC compiler, but it also covers the whole tool chain (i.e., assembler and linker) needed for processing PL8. A key element for testing was a special VM/CMS-based simulation and debugging environment called z/CECSIM [13]. For normal code development and test purposes, about 1300 functional test cases are available.

Goals and approach

Given the specific focus of the GPL8 compiler effort to provide 64-bit support for the zSeries firmware execution environment, the primary goal of the validation effort was to ensure that correct object code is generated, specifically for the zSeries firmware code, assuming that the source code correctly implements the desired function. A secondary goal of the validation effort was to ensure that performance and throughput of the generated object code is sufficient. Consequently, we did not plan to run a full engineering system test (EST) with the generated code, since EST is typically designed to verify correct function implementation and requires significant effort and time.

The given firmware code on zSeries uses only a limited subset of the GPL8 language constructs and syntax, and so the focus of the independent GPL8 product validation was to ensure that the compiler handles these constructs correctly. While it is an important goal of a compiler to provide extensive and strict error handling and reporting for source code bugs, this was not a focus item of the validation efforts. In particular, we did not plan any systematic validation of this. We took a two-pronged approach to achieve the above goals:

1. We defined a 32-bit reference environment to verify the new GPL8 compiler in and for this reference environment. We picked the 32-bit firmware implementation of the z900 system as the reference environment, both for functional and for performance testing. For all practical purposes, this environment is assumed to provide correct source code.
2. For validation of the 64-bit front end, we used z/CECSIM simulation and a standard regression test package of the firmware code under development. Validation of compiler release criteria was always performed on a stable driver level of 64-bit firmware code.

As is not unusual for tool projects, the GPL8 compiler project had to be run with a certain time overlap to the actual z990 firmware code development project. This implied that special consideration was required to ensure a staged delivery of compiler functions and capabilities with a high quality level in order to avoid hampering the progress of the actual firmware code development.

Setup

The original pl.8 compiler is VM/CMS-based, supports only a 32-bit environment, and is used to compile z900 32-bit firmware. The new GPL8 compiler is Linux-based, supports both a 32-bit and a 64-bit environment, and is needed to compile 64-bit i390 code. At the time this compiler was developed, the 64-bit i390 code was not complete, and it had not been verified in any environment. Therefore, a special setup was required to verify the new GPL8 compiler not only with the target 64-bit i390 code, but with the existing and well-tested 32-bit i390 code as well.

VM/CMS-based testing

After a new version of the GPL8 compiler had passed development tests, two VM/CMS versions of the compiler were built: a 32-bit version and a 64-bit version. The 32-bit version is used with the existing and well-tested 32-bit i390 code, while the 64-bit version is used with the newly

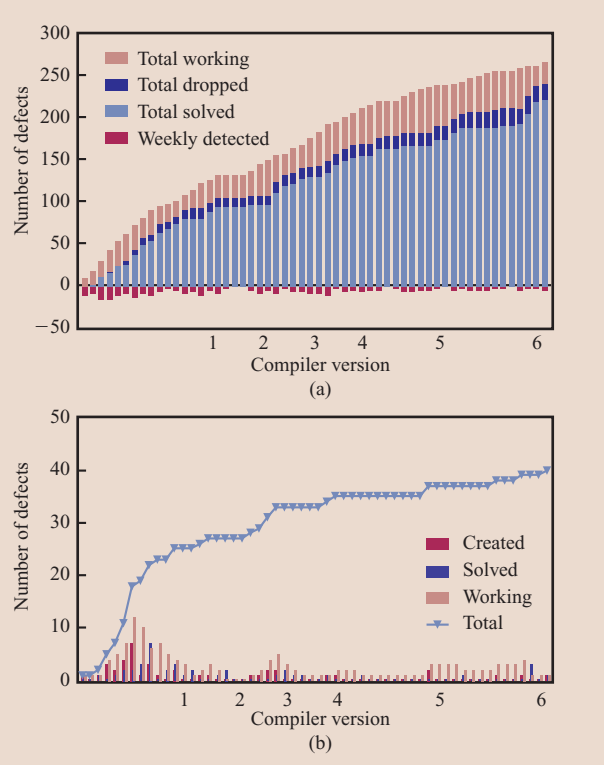

Figure 4

Functional tests performed on the GPL8 compiler: (a) Defect summary statistics. Problems detected weekly are shown as negative numbers. (b) Critical defect statistics.

generated 64-bit i390 code. These two test environments were used in parallel until the 64-bit i390 code achieved a level of quality that allowed unique use of this environment for tests. At this point, the 32-bit CMS environment was dropped from the test cycle.

32-bit environment

The GPL8 compiler enhances code and data type checking compared with the original pl.8 compiler. Therefore, the 32-bit i390 code could not be used as is, but, with a very limited rewrite, the z900 32-bit i390 code could be used as a validation aid for the GPL8 compiler. The following steps were executed:

- Compile the 32-bit i390 code.
- Achieve power-on reset (POR, initial microcode load) with this code complete in the VM/CMS-based 32-bit z/CECSIM simulation environment.
- Run the complete 32-bit-based function regression package error-free.
- Activate the 32-bit i390 code on a z900 system and achieve a complete POR there.

64-bit environment

When the quality level of the 64-bit i390 code became high enough, this code could also be used for compiler validation. The same validation steps as with the 32-bit code were then executed with the 64-bit code:

- Compile the 64-bit i390 code.
- Achieve POR complete with this code in the VM/CMS-based 64-bit z/CECSIM simulation environment.
- Run the complete 64-bit function-based regression package error-free.
- Activate the 64-bit i390 code on a zSeries machine and achieve a complete POR there.

Results

Functional tests

From project start to end over a period of 12 months, we released six versions of the GPL8 compiler to the development process of the z990 system. The overall stability was excellent, and the releases were smooth phase-ins without impact to the integration and test efforts for the program. The problem finding-and-solving rate shows the usual asymptotic behavior, as can be seen in **Figure 4(a)**.

To have an indication of the potential impact on the development process, we defined the term *critical defect* in the compiler: *A compiler defect was considered critical when it generated incorrect object code for correct source code.* The critical defect statistics in the GPL8 compiler are shown in **Figure 4(b)**. In both Figures 4(a) and 4(b), the slope of the envelope (the *Total* line) indicates that the GPL8 compiler version 6 was mature and well prepared to support z990 system shipment.

Performance tests

The performance considerations and goals for the GPL8 compiler are primarily related to the performance and efficiency of the generated object code, again focusing on the i390 firmware code in zSeries.

32-bit environment, z900

Within the 32-bit reference environment of the z900, the goal was to achieve object code performance equal to that delivered by the original pl.8 compiler. Since the performance-relevant i390 code is channel subsystem code executing on the system assist processors (SAPs), we chose the SAP capacity as the metric for the i390 object code performance. SAP capacity is measured in *start subchannel operations per second* (SSCH/s) on a product-level zSeries system, and for each production-level compiler release, we have performed such SAP capacity measurements. **Table 2** shows the relative achieved SAP capacity, per GPL8 production level release, compared with the

reference SAP capacity measured on a z900 product, which runs code generated with the original pl.8 compiler.

64-bit environment, z990

For the 64-bit target environment of the z990, the goal was to reach the projected SAP capacities. This was based on a projected increase in path length of 10%, mainly because of architectural differences between a z900 and a z990 system. Using this path length, a prediction of the SAP capacity on a z990 system compared with a z900 system showed an increase of 38%. The final arbiter of whether or not the anticipated SAP capacity had been achieved was the SSCH/s measurement of the 64-bit i390 code on a z990 system. The first results were not satisfactory, and path-length analysis of the executed i390 code in z/CECSIM was used to identify the root cause for this mismatch. Among other things, it turned out that the 64-bit compiler showed some performance weaknesses of the generated code which were not seen with the 32-bit compiler. On the basis of this analysis, some moderate source code changes reduced the initial increase in path length of one i390 I/O task from 23% to the expected value of 11%. The final measurements on a z990 system yielded an increase of 45%, which shows that the path length of an I/O task is just one contributor to the SAP capacity.

Path coverage analysis

The procedures described so far did not verify the GPL8 compiler directly, but permitted its correctness to be inferred from the correctness of some functional code that was compiled by it. One side effect of changing the environment from VM/CMS to the open source world was that many additional tools and utilities from the GNU tools chain became available. It was therefore possible to obtain additional evidence in the form of path-coverage measurements for the GPL8 compiler source code by using the test-coverage checking tool GCOV [15, 16], while GPL8 was compiling the following code buckets:

- The PL8 test-case suite under Linux.
- The complete i390 code under VM/CMS.
- Only those i390 procedures that are actually executed during the functional test runs in z/CECSIM.

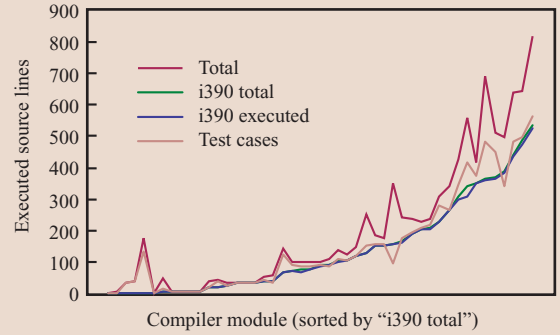

Figure 5

Path coverage for GPL8 compiler source code.

Figure 5 represents a snapshot taken from Release 4 of the GPL8 compiler. Whereas the absolute figures might have changed in the meantime, the relationship between the different scenarios and, in turn, the significance of the following statements should not depend on a special version of the compiler. The number of lines of code that were executed in the individual modules is shown in Figure 5.

Referring to the figure, compiling the test cases showed coverage patterns very similar to those of the GPL8 compiler in total, which suggests that the test-case suite covers the PL8 language very well. An analysis of the coverage measurements for weak spots (significant deviation between the test-case line and the total line for certain modules) was initiated, and the results were used to improve the test-case suite.

In addition, it can be seen that there is only a small difference between the i390 total line and the i390 executed line coverage pattern. In other words, when the part of the i390 code that is not executed during the z/CECSIM regression runs is compiled, it does not use compiler source code that is significantly different from that used to compile the i390 code that is "proved-to-be-correct" by the z/CECSIM functional regression runs. This is the main justification for using the validation procedure described above rather than running a complete test cycle on the raised floor with GPL8-compiled i390 code, and it illustrates the efficiency of our validation procedures.

Table 2 Performance of the code generated by different GNU PL8 compiler versions, relative to the code generated by the original pl.8 compiler.

GPL8 Rel. 1	*GPL8 Rel. 2*	*GPL8 Rel. 3*	*GPL8 Rel. 4*	*GPL8 Rel. 5*	*GPL8 Rel. 6*	*Reference (z900 product)*
87.4%	93.3%	96.2%	98.5%	99.2%	100%	100%

Process considerations

Process issues were of particular importance, since part of the compiler development and the firmware development had to be done in parallel. From a process point of view, the development of the GPL8 compiler was handled like other code development projects in zSeries development. In particular, the zSeries code development process was applied, including design reviews, component and unit test, source code repositories with version control, defect tracking, and build tests.

In addition to these standard processes, special consideration was necessary to satisfy two conflicting objectives: on one hand, the capability for frequent releases with rapid validation for the compiler development team, and on the other, the requirement for stability and high-quality functionality for the i390 code development community.

Version control and beta releases

Compiler version releases to the user community have been strictly tagged in the source-code repository. The concept of beta releases was used for rapid delivery of required fixes and functional enhancements. Only after a beta release had seen substantial test coverage and user experience was it released to the user community as a new production-level version of the compiler.

Typically, the timing of beta releases throughout the GPL8 compiler project was of the order of days, whereas new production-level compilers were released only every four to six weeks, maintaining the desired level of stability for the i390 code development teams.

Acceptance procedures

After the GPL8 compiler development team had successfully executed its regression test package for a new compiler version, it was offered as a new beta version to the independent validation process. Before this beta version was released to the general user community, it had to pass an acceptance filter that was defined on the basis of i390 focus test activities. Similarly, for new releases of production-level compilers, an extended and more stringent acceptance filter was applied.

Typically, the acceptance procedures for a new beta release involved extensive tests of the generated i390 code in the z/CECSIM simulation environment [13], 32-bit as well as 64-bit, whereas the acceptance procedures for production-level compilers also involved functional tests of the generated i390 code on the 32-bit environment of the z900 target machine.

Observations and experience

Quality and stability of the GPL8 compiler has been outstanding, especially in the new 64-bit environment. The production version of the compiler, which was intended to support i390 code generation for z990, was actually released nine months prior to z990 shipment. We had projected at that time less than ten test escapes, i.e., critical compiler defects that would be found during z990 product system test activities. The actual number turned out to be five.

Though the error-handling and reporting capabilities of the GPL8 compiler were not a focus of the validation activities, a couple of observations made during the course of the validation efforts throughout the project are worth noting:

- The z/CECSIM environment proved again to be very efficient, particularly for the rather special efforts for the GPL8 validation. Most of the critical compiler defects (defined above) were encountered in the z/CECSIM environment, and even the remaining ones, which were initially seen on the machine, could be recreated in z/CECSIM, where they could be analyzed with much higher productivity than on the target system.
- When the checking of array bounds was implemented and enabled in the GPL8 compiler, 14 source-code bugs were uncovered within the i390 source code under development for z990. Four errors were detected at compile time and ten errors at run time.

In summary, our validation methodology with the i390 focus test and parallel development of the GPL8 compiler and the z990 product-level i390 code has posed its challenges, but it has proved very successful and saved a full EST validation cycle for the introduction of the new compiler.

Summary and concluding remarks

The programming language pl.8 was originally developed around 1980 for the then newly designed RISC architecture. In this environment, the language was used for the implementation of an operating system and for the compiler itself, successfully avoiding a number of problems that emerge when certain other languages are used.

In particular, pl.8 supports low-level programming with certain elaborate concepts that reduce the risk of typical errors. These advantages are also evident when the language is compared with C. Therefore, pl.8 was chosen for the implementation of zSeries firmware about a decade ago. Unfortunately, there was no further development for pl.8, and its original compiler was ultimately no longer maintained. This, and the need for a 64-bit PL8 compiler, led to the development of GNU PL8.

Future improvements to PL8 could be certain new language features—in particular, features that support modularity via more strictly defined interfaces. Although the GNU PL8 compiler produces quite efficient code,

some additional optimizations could be done. There also is an ongoing discussion as to whether the compiler should be released into the open-source community.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of The Open Group, Linus Torvalds, Hewlett-Packard Company, or Sun Microsystems, Inc.

References

1. L. C. Heller and M S. Farrell, "Millicode in an IBM zSeries Processor," *IBM J. Res. & Dev.* **48,** No. 3/4, 425–434 (May/July 2004, this issue).
2. M. A. Auslander and M. E. Hopkins, "An Overview of the PL.8 Compiler," *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* Boston, MA, June 1982, pp. 22–31.
3. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation Via Coloring," *Computer Languages* **6,** No. 1, 47–57 (January 1981).
4. L. W. Wyman, H. M. Yudenfriend, J. S. Trotter, and K. J. Oakes, "Multiple-Logical-Channel Subsystems: Increasing zSeries I/O Scalability and Connectivity," *IBM J. Res. & Dev.* **48,** No. 3/4, 489–505 (May/July 2004, this issue).
5. J. Maergner and H. R. Schwermer, "High Level Microprogramming in i370," *The Design of a Microprocessor*, W. G. Spruth, Ed., Springer-Verlag, New York, 1989, pp. 303–316; ISBN 0387513957.
6. R. W. Sebesta, *Concepts of Programming Languages*, Second Edition, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993; ISBN 0201752956.
7. F. E. Allen, "Program Optimization," *Annual Review in Automatic Programming*, Volume 5, Pergamon Press, New York, 1969, pp. 239–307.
8. J. Cocke and J. T. Schwartz, "Programming Languages and Their Compilers: Preliminary Notes," *Technical Report*, Courant Institute of Mathematical Sciences, New York University, 1970.
9. W. Gellerich, M. Kosiol, and E. Plödereder, *Where Does GOTO Go To?, Reliable Software Technologies—Ada-Europe '96*, Alfred Strohmeier, Ed., Springer-Verlag, Berlin, 1996, pp. 385–395.
10. W. Gellerich and E. Plödereder, "The Evolution of GOTO Usage and Its Effects on Software Quality," *Informatik '99*, K. Beiersdörfer, G. Engels, and W. Schäfer, Eds., Springer-Verlag, Berlin, 1999, pp. 380–389.
11. H. Penner and U. Weigand, "Porting GCC to the IBM S/390 Platform," *Proceedings of the GCC Developer's Summit,* Ottawa, May 2003; see *http://www.gccsummit.org/2003/*.
12. IBM Corporation, *z/Architecture Principles of Operation* (SA22-7832); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
13. J. von Buttlar, H. Böhm, R. Ernst, A. Horsch, A. Kohler, H. Schein, M. Stetter, and K. Theurich, "z/CECSIM: An Efficient and Comprehensive Microcode Simulator for the IBM eServer z900," *IBM J. Res. & Dev.* **46,** No. 4/5, 607–615 (July/September 2002).
14. Manfred Broy and Ernst Denert, Eds., *Software Pioneers*, Springer-Verlag, New York, 2002.
15. R. M. Stallmann, "Using and Porting the GNU Compiler Collection," gcc-2.95 Edition, Free Software Foundation, Boston, MA, July 1999.
16. Richard M. Stallmann, "GNU Compiler Collection Internals," Free Software Foundation, Boston, MA.

Received September 22, 2003; accepted for publication November 24, 2003; Internet publication April 6, 2004

Wolfgang Gellerich *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (gellerich@de.ibm.com).* Dr. Gellerich studied computer science and chemistry at the University of Erlangen-Nuernberg and graduated in 1993 with an M.A. degree in computer science. Until 1999, he was with the programming languages group of Stuttgart University, where he received a Ph.D. degree. Dr. Gellerich joined the IBM Development Laboratories at Boeblingen in 2000. He is with the Firmware Development Group, where his main responsibility currently is the development of GNU PL8.

Torsten Hendel *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (hendel@de.ibm.com).* Mr. Hendel studied software engineering at the University of Stuttgart and graduated in 2001 with an M.A. degree in computer science. In 2001 he joined IBM, where he works mainly in firmware development.

Rudolf Land *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (land@de.ibm.com).* Mr. Land is a Senior Technical Staff Member in the Hardware Development Group, responsible for platform architecture and system design of IBM eServers. He studied mathematics at the universities of Aachen, Columbus, and Cologne, where he graduated with a thesis on computational algebra in 1979. He joined IBM in 1981 in the Server Hardware Development organization, where he held various positions in S/390 microcode development, system design, and system integration.

Helge Lehmann *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (hhlehmann@de.ibm.com).* Dr. Lehmann is an Advisory Engineer in hardware development, where he is working on future strategies in the I/O area for zSeries systems. He studied mathematics and physics at the University of Cologne, where he received a Ph.D. degree for numerical solutions of partial differential equations. In 1985, he joined IBM and since that time has worked primarily with channel subsystem design and I/O configuration definition for S/370, S/390, and zSeries systems.

Michael Mueller *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaichestrasse 220, 71032 Boeblingen, Germany (mulm@de.ibm.com).* Mr. Mueller is a Senior Technical Staff Member responsible for the platform architecture and design of the reliability, availability, and serviceability of IBM iSeries, pSeries, and zSeries eServers. He studied electrical engineering at the University of Stuttgart, receiving his Dipl. Ing. degree in 1985. He joined IBM in 1985, working in the S/370 Product Assurance Test Laboratory at Boeblingen. He has held various positions in S/390 microcode development and system design.

Peter H. Oden *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (oden@us.ibm.com).* Dr. Oden is a Research Staff Member in the Systems Department at the Thomas J. Watson Research Center. He received an A.B. degree from Columbia College in 1955, and M.S. and Ph.D. degrees in electrical engineering from Columbia University in 1958 and 1966, respectively. In 1963 he joined IBM at the Thomas J. Watson Research Center, where he has worked on design automation, programming languages and compilers, and computer microarchitecture. In 1968, he received an IBM Outstanding Contribution Award for his work on design automation, and in 1981 an IBM Research Outstanding Technical Achievement Award for work on compilers. Dr. Oden is an author or co-author of several patents and technical papers.

Hartmut Penner *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (hpenner@de.ibm.com).* Mr. Penner studied computer science at the University of Kaiserslautern and graduated in 1996 with an M.A. degree. He joined IBM in 1996 and worked in firmware development, commencing shortly after the zSeries back end for GCC and working on the Linux port for zSeries. He is currently responsible for the zSeries back end within and outside IBM.