Open-standard development environment for IBM System z9 host firmware

C. Axnix
T. Hendel
M. Mueller
A. Nuñez Mencias
H. Penner
S. Usenbinz

When the PL8 64-bit GNU compiler collection front end was introduced with the IBM z990 system, it laid the foundation to move toward an open-standard development environment for the i390 layer of IBM System z™ host firmware. However, when the z990 system was developed, the proprietary project development library system and the table of contents object file format for i390 code were still being used. With the IBM System z9™, we have moved to a fully open-standard development environment. This paper describes the steps we took to get there, to improve code performance, development efficiency, and regression testing, and to develop base functionality for important System z9 features such as enhanced driver maintenance. We also discuss plans to further enhance the development environment for future systems.

Overview

IBM System z* host firmware runs on the System z processor hardware. It provides the z/Architecture* interface (I/O processing instructions and complex processor instructions) and reliability, availability, and serviceability functions such as hardware reset and recovery, concurrent maintenance, and capacity on demand.

System z host firmware consists of two levels. The first is the lower-level millicode layer. This layer is written in assembly language and runs directly on the z9* processor hardware. It is used to implement performance-critical functions or functions that require direct control of the underlying hardware structures. The millicode layer has to be adjusted for each new System z hardware generation. The second firmware level, which runs on top of the millicode layer and can use functionality provided by the millicode, is the higher-level internal 390 (i390) code. It is written primarily in PL8 or C and implements functions that are less performance-critical or too complex to code in assembly language. An advantage of the i390 code layer is that most parts of it do not have to be adjusted for each new System z processor generation because the underlying millicode layer deals with most of the hardware-specific handling. (See [1] for a discussion of the firmware stack running on a System z.)

Since the advent of the z990, the i390 firmware layer has been compiled using the GNU compiler collection (GCC). Using GCC version 3.3 instead of version 2.95 for the System z9 enabled us to take advantage of the enhanced z990 z/Architecture instruction set. Also, code coverage measurements for i390 are now supported by using the standard GNU coverage support.

The library and build environments have been changed from the project development library (PDL) system under virtual machine (VM) to concurrent versions system (CVS) and software construction (SCons) under Linux**, which allows the use of standard open-source tools such as Red Hat** Package Manager (RPM) or Python**. We explain how this new development environment improves the development efficiency and the turnaround time to build code for each one of the more than 80 i390 developers.

To eliminate the last proprietary piece in the development environment, the i390 object file format was changed from the table of contents (TOC) format to the industry-standard executable and linkage format (ELF). The advantages and savings of moving to ELF are described.

0018-8646/07/$5.00 © 2007 IBM

The new object file format of an i390 load required a new i390 loader—the ELF loader. This new loader performs the i390 load process for loading initial microcode and applying concurrent microcode fix [a method to apply required microcode patches (e.g., for problem fixes) without requiring a system shutdown], also known as *concurrent patch*. Compared with previous machines, the functionality of this loader for concurrent patch was significantly enhanced to provide the base functionality for the enhanced driver maintenance feature for i390. For example, one requirement for enhanced driver maintenance was support for new global data variables and their initialization. Also, the loader now supports i390 function pointers, which can be used for registration and callback routines. This greatly improves decoupling of code and ease of maintenance for adding new procedures for future releases of the System z9. Finally, this paper provides a perspective on how this open-standard development environment for i390 can be further enhanced for future systems.

Continued development of the PL8 GCC compiler

With the IBM eServer* z990, we used the open-source GCC [2] for compiling our firmware written in high-level programming languages. We could take advantage of the ability of GCC to produce code for the System z processor architecture. This ability is implemented by the System z compiler back end provided by the IBM Linux-on-zSeries* project. Most of the high-level firmware code is written in PL8, an IBM proprietary programming language derived from PL/1. In order to translate PL8 code, a GCC front end for PL8 was developed. GCC with an integrated PL8 front end is called the GNU PL8 compiler, or *GPL8 compiler*.

The PL8 front end reads the PL8 code and builds up data structures understood by the core of the GCC, commonly called the *middle end*. The middle end carries into effect all optimizations and triggers platform-dependent code production done by the compiler back end. Both the System z back end and the PL8 front end have been described in former issues of this journal [3, 4]. This section focuses on how and why we continued to use the GCC for the IBM System z9.

The GCC is an ever-evolving project. The performance of the code it produces is continuously being improved by new methods of optimization implemented in the core compiler and by the exploitation of new hardware features or instructions by the back end. Fixes for compiler mistakes included with new compiler versions are as welcome as simplifications of the interface between the front end and the middle end. With the z990 we used GCC version 2.95. Staying with this version would have deprived us of the advantages of improved compiler versions, and it would have had some other drawbacks. The firmware compiler is part of a tool chain composed of preprocessor, compiler, assembler, linker, and other tools such as the GNU coverage measurement tool (GCOV) and tools to analyze and modify object files, such as objdump and objcopy. Newer versions of these tools will not be interoperable forever with GCC 2.95, and the wider the gap between the GPL8 compiler based on GCC 2.95 and current GCC development, the more troublesome will be an adaptation of the PL8 front end to the current GCC—an adaptation that will be unavoidable someday. It was therefore decided to invest the effort of continually adapting the PL8 front end to the current GCC version. For the System z9, we use a GPL8 compiler based on GCC 3.3.

Most of the firmware code deals with hardware setup, recovery, and configuration tasks. This code is not performance-critical with respect to time, but there is performance-critical code that handles I/O requests and requires firmware support. One system performance parameter is the number of I/O requests initiated by a start-subchannel instruction that can be handled per second. Improvements in the System z9 start-subchannel capacity result primarily from three factors: the faster clock speed of the hardware; the System z9 firmware compiler, which supports the performance increase by its instruction scheduler that was optimized for the pipeline that was introduced with the z990 processors; and the exploitation of the long-displacement facility, an instruction set that allows the specification of relative address offsets of up to 1 MB and the addressing of data areas of up to 1 MB with the same base register [5]. After all, 2.1 percent of the System z9 firmware instructions use the long (20-bit) displacement, thus avoiding additional instructions dealing with address computation and the stalls related to them.

GCC 3.3 introduces thread local storage (TLS) support for the System z. TLS provides the ability to have unique static storage assigned to each thread, a concept that opens interesting prospects with respect to the firmware static variables that exist as local data variables of each processing unit (PU). Thus, keeping pace with the current compiler development allows a further approach to open standards in the future.

The interface between a front end and the GCC core compiler is quite complex. GCC 3.3 starts simplifying this interface by introducing so-called *language hooks*. A language hook is just a function pointer that is set by the front end and allows a callback from the middle end to the front end. This gives the front end an opportunity to influence the optimization process or to build up data structures when they are needed by the middle end. The introduction of language hooks clearly shows a trend that is welcomed by compiler front-end developers. GCC

development focuses more and more on simplifying the introduction of compiler front-end extensions, and GCC 4.0 consequently continues this development. This means that adaptations to new GCC versions for the PL8 front end can be done more safely and simply.

GPL8 compiler development also involves maintenance of the PL8 language. The System z must steadily meet higher demands with respect to hot-plugging new hardware and offering continuous availability. Firmware code structures must comply with these demands. For the first time, the System z9 firmware exploits function pointers, enabling dynamic registration of firmware components. The PL8 language provided basic function-pointer support, but it was not implemented by GPL8 for the z990 system. For the System z9 system, the language definition was extended in order to add strong type checking with function pointer assignments and to make the use of function pointers as easy as the use of any variable.

Moving toward an open-source build environment

A cornerstone of the modernization of the firmware development environment was to replace the proprietary source-control and build systems with standard open-source tools. The challenges of migrating project source files from the old to the new source-control system and the motivation to choose SCons as the build system are discussed in the next sections.

From PDL to CVS

The VM-based PDL used in System z systems prior to System z9 used a tree structure in which the nodes of the tree represented different release levels. The node names had a length of four characters so that information such as the name of the project to which the library level belonged had to be encoded with a single character. The PDL tree structure with four-character node names encoding the release-level information can be shown as follows:

```
      DG2R
     /    \
DG3R       DG2I
```

where `D` is the development library, `G` is the abbreviation for the project name, the numerals are the specification of project rollout (code for the second and third project rollout), and `R` is the normal release level.

During the build process, files not found in a lower level were automatically searched for in higher levels. Therefore, the lower levels stored only changed files; unchanged files were inherited from higher levels.

All sources were stored on a conversational monitor system (CMS) disk on a flat file system in which the last four characters of the filename extension represented the library level to which the file belonged. Together with a separator character (`$`), only three characters remained for the real filename extension because CMS has an 8.8-character limitation for filenames. Longer extensions were abbreviated (`.pl8inc` → `.p8i`), and shorter ones extended (`.c` → `.c__`).

When importing the sources from PDL to CVS using a network file system (NFS) mount of the CMS disks, a shell script sorted all files into directories that corresponded to their library level, taking care of the PDL inheritance scheme and translating the filename extensions back from the PDL encoding scheme.

The timestamps of the PDL files were not a reliable mechanism to determine whether a file had changed since the last shadowing operation because, by removing a file from a lower level, a potentially older file from a higher level could become the current one for the lower level. Thus, all files always had to be copied starting from the highest level, and files from lower levels had to overwrite the previously copied files where necessary. Copying sources from a CMS NFS mount to a CVS working copy is done as follows:

```
Step 1: bbihrrun.p8i$dg2r → dg2r/bbihrrun.pl8inc
        bbihrrun.p8i$dg2r → dg3r/bbihrrun.pl8inc
Step 2: bbihrrun.p8i$dg3r → dg3r/bbihrrun.pl8inc
```

Because of the reverse delta storage mechanism of CVS, only files whose content had changed from the last shadowing operation created a new revision in CVS.

This CVS shadow of the PDL library was the base for the migration to a Linux-based development environment for System z9. Owing to the flat CMS file system, the component to which a source file belonged in PDL could be distinguished only by a naming convention, such as using the first four characters of the filename as an indication for the component (e.g., `BBIGxxxx` for all files belonging to the i390 reset component). For the Linux-based development environment, each component was given its own subdirectory (e.g., `src/reset`). A Perl script was used to sort all source files into their target component subdirectories on the basis of regular expressions for the filenames (e.g., by default all files starting with BBIG were placed in the `src/reset` subdirectory).

The script also replaced dollar characters in filenames with underscores, because a dollar sign in a UNIX** shell refers to an environment variable, and handling filenames containing dollar signs in a UNIX shell is cumbersome at best. A simple shell script using `sed`, the stream editor program, fixed the include statements in the source files accordingly.

The include directives in PL8 source files do not contain a filename extension and, because the CMS file system—unlike a Linux file system—is case-insensitive, sources could contain mixed-case include directives (such as `%MACINCL BBIeerms;`). The PL8 preprocessor has been modified to always translate filenames to lowercase and append `.pl8inc` as an extension before looking for a file in the file system. The consequence is that PL8 files, unlike C files, must now always be lowercase.

SCons

For the initial CVS shadow of the PDL library, a makefile (a file that describes how a system is built) was written that, together with Linux versions of the build tools, allowed firmware code to be built on Linux. Nevertheless, experience with this makefile led to the conclusion that `make` would not be a good choice for a production build system, and after some evaluation SCons [6] was finally chosen. The primary differences between `make` and SCons are discussed in the following sections.

- *Dependency handling*
  A build target is dependent not only on a primary source file, but also on the include files directly and indirectly included by the primary source file. Whenever one of these input files changes, the build target has to be rebuilt. In addition, the dependency tree for the build target has to be updated when include files are being added or removed. `Make` can include dependency information generated by an external dependency generator (e.g., a compiler) and can therefore take care of rebuilding a target when the timestamp of an include file changes. However, trying to automatically keep the dependency information itself up to date can become quite challenging, especially when dependencies are removed and the former input files no longer exist. Therefore, `make` is typically used only with static dependency information generated by an explicit dependency generation step (typically invoked with a `make dep` command). Such static dependency information can rapidly become outdated, potentially resulting in inconsistent build results.

  SCons not only takes care of such include file dependencies automatically, but it also includes changes in the build commands (e.g., using different compiler flags) or other arbitrary information in the dependency tree. Also, while `make` relies on timestamps to decide whether a target has to be rebuilt, SCons can instead use MD5 message-digest algorithm hashes of the file contents so that build targets are rebuilt whenever the content of a source file has changed since the last build, even when the target seems to be newer than the source. This is especially important with backing builds, as discussed next.
- *Backing build support*
  In this project, all of the more than 80 i390 code developers have access to a central server. It would be a waste of resources if everyone had to build all currently active maintenance releases from scratch, especially when the number of parallel maintenance streams is high. Therefore, the sources and build results of all currently relevant releases are kept below a central directory, and the active maintenance and development streams are automatically updated and rebuilt periodically. These central directories are called *backing builds*. A developer keeps only modified sources locally and sets a pointer to the desired backing build. The build system then takes all sources and build results that do not exist locally from the backing build.

  `Make` has only limited support for backing builds and, in combination with the requirement for subdirectories and automatic updates of dependency information, the situation gets even worse. SCons, on the other hand, has full built-in support for backing builds and, because of its use of MD5 hashes, can handle a scenario in which a locally modified source is removed after a local build, so that the source from the backing build becomes the relevant source. Because it is likely that the source in the backing build has an older timestamp than the local build result, `make` would consider the build target up to date. SCons, on the other hand, detects that the content of the source file has changed from the last build and rebuilds the target again with the source from the backing build.
- *Maintainability and extensibility*
  Complex makefiles tend to become difficult to read and maintain. SCons, however, is written in Python [7] and uses Python for the build rule specifications. The control files are therefore easier to understand and, with the full power of an object-oriented scripting language at hand, even complex build rules or complete build procedures are relatively easy to implement.

A major difference from the PDL environment is the turnaround time that can be achieved in the Linux-based development environment. In the PDL environment, a developer had to start all compile and link tasks manually, one after the other. If an include file was changed that had been used by more than a few

sources, recompiling all dependent files manually became impractical. Therefore, the file had to be released more or less untested to a special library level, and a librarian then had to be asked to process that level. The resulting build failures could then be analyzed by the developer, and an updated file had to be re-released. This step had to be repeated until the library build was clean, whereupon a librarian could promote the file to its intended destination library level.

In the Linux-based development environment, a developer has to specify only the final build target, and SCons automatically determines which build steps, such as compile and link, have to be executed, on the basis of rules defined in a configuration file called SConscript. The time required to build a complete code load is proportional to the impact of a source change. Optimizations in the SConscript keep the startup time short when only a single source compile is requested.

Development tools

All build tools (the PL8 compiler, for example) are available as standard RPMs. By installing SCons and CVS plus three RPMs containing i390-specific tools, each Linux system can be used for i390 firmware development. While CVS requires a network connection to the CVS server, the build system also works disconnected, thereby allowing development on a standalone workstation, such as a Lenovo Thinkpad**, without a permanent connection to a central server.

All i390 RPMs are relocatable and, with a script such as that in **Figure 1,** multiple versions of the same RPM can be kept on a system. This makes it possible, for example, to phase in a new compiler version with the current development stream while all maintenance streams stay with the old compiler version for reasons of stability, and both streams can be built on the same system. The SConscript automatically detects and uses the required RPM versions.

The move to Linux as the development platform also allows the use of tools such as the Ctags source tagging system or the Doxygen documentation generator.

Move from TOC to ELF

The original toolset of the i390 environment that was used up to and including System z990 was based on the TOC format. This format is the base of the file format used for IBM AIX* on the PowerPC* architecture; it is called the common object file format (see [8], pp. 47–92 for an explanation of object file formats). In this format a set of related source files are compiled and linked together into modules. Each data segment of these modules contains a TOC section which contains the pointer list of the module that points to the global variables inside and

```
#!/bin/sh
PKGNAME=`rpm -qp ${1?}`
if [ "x$PKGNAME" != "x" ]
then
  PREFIX=`rpm -q --info -p ${1} | awk '/Relocations:/ { print $5
}'`
  UMASK=`umask`
  PERM=`expr 777 - ${UMASK}`
  install -d --mode=${PERM} ${PREFIX}/${PKGNAME}/rpm
  rpm --verbose --initdb --dbpath ${PREFIX}/${PKGNAME}/rpm
  rpm -U -F --oldpackage --test ${1}
  if [ $? -eq 0 ]
  then
    rpm --verbose -i --prefix ${PREFIX}/${PKGNAME} --dbpath
${PREFIX}/${PKGNAME}/rpm ${1} -nodeps
  fi
fi
```

Figure 1

Simple script to install different versions of the same RPM in different directories.

outside the module. This TOC also contains pointers to the TOC and addresses of external procedures. At load time during system startup, the TOC loader of the system is responsible for relocating the unresolved symbols. Missing symbols therefore cannot be detected before the TOC loader runs; instead, they result in a late link error during initial microcode load (IML), and system startup will fail. In addition to the name of an external symbol that must be resolved at load time, this format also provides a hash as a numerical value of the interface to check whether an external symbol has the right type.

Because each module has its own TOC, the TOC pointer must be reloaded for intermodule calls. For a call from one module to another, the TOC binder in the library creates a specific, very efficient, TOC-unique glue code to be called instead of the external function. This glue code is responsible for reloading the TOC pointer when calling in to and when returning from the external function. This adds a small overhead for external calls compared with internal calls, forcing the system design to place related functions in the same module. Levine [8] gives a good overview of the loader process and load-time relocation.

In i390, the split into multiple modules was required basically to overcome the 4-KB TOC limitation. The limitation was in effect up to and including Enterprise Systems Architecture/390*, which allowed addressing only 4 KB of data with the same base register because it supported only 12-bit displacement fields [5].

These i390 TOC modules were not true modules in the sense of a modular code structure with loadable modules that can be plugged into the system or unplugged to introduce or remove functionality. The split into modules

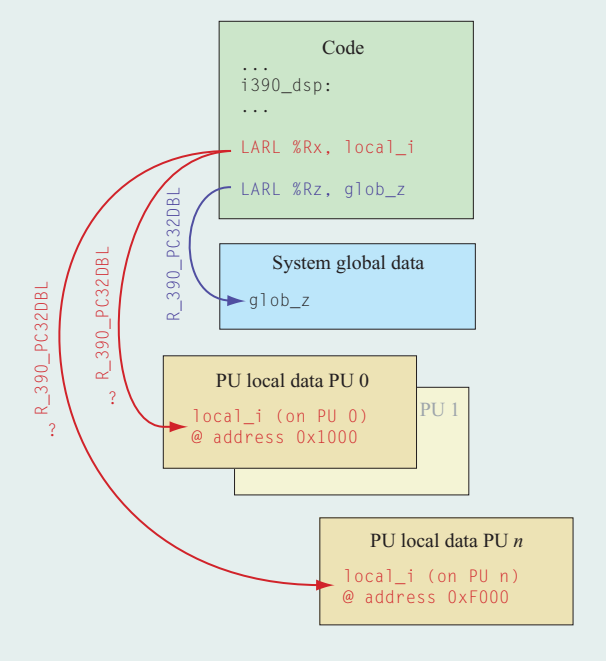

Figure 2

Relocations and multiple data area handling in non-PIC mode.

was required only because of the 4-KB TOC limitation. For a functional i390 system, it was not possible to load only a subset of the modules; all of the modules always had to be loaded to resolve all external references.

The 4-KB TOC limitation no longer applies in the z/Architecture because of the introduction of the long-displacement facility [5]. Thus, there was no longer any reason to split the code into multiple modules for System z9. Being able to place all i390 code in one module makes it possible to perform the final link step within the library rather than on the machine at runtime. This eliminates the requirement for the TOC-unique glue code for intermodule calls and significantly lowers the dependency on the TOC format. Instead, a more standard link format could be considered—the widely used ELF format—which is predominant on Linux. Moving to ELF eliminated the need to port the existing TOC binary utilities (binutils) support, high-level assembler, and proprietary TOC binder from the VM-based development tool chain to a Linux-based development environment. It also eliminated the need to continue TOC format support in the GCC compiler back end.

The ELF format has two modes: the position-independent code (PIC) mode and non-PIC mode (see [8], pp. 169–176 for a discussion of PIC mode). The non-PIC code directly implements accesses from the code to the data section by having relative relocations in the code (read only) section. This is shown in **Figure 2**: To access the variable `glob_z`, its address is loaded by using a Load Address Relative Long (`LARL %Rz, glob_z`) instruction.

In the PIC case, a global offset table (GOT) is generated by the linker for all symbols and put into the executable. A GOT is basically a list of symbol addresses, and data accesses from the code in the PIC mode are done indirectly by looking up the data addresses in the GOT. **Figure 3** illustrates this. For example, to access variable `glob_z`, the pointer to it is first loaded from the GOT into the general-purpose register `Rz` using a `Load LG %Rz, glob_z@GOT` instruction. Subsequent accesses to `glob_z` are done using `Rz` as a base register. The standard GNU compiler generates code to load the GOT pointer in the prolog of each function that accesses static (i.e., non-automatic) variables. The code to reload the GOT pointer has a somewhat higher performance impact than the glue code in the TOC format, but in the ELF format it is normally used only for shared libraries, which are currently not used in the System z i390 firmware environment.

Unfortunately, none of these modes could be used right away for i390. To understand this, one additional requirement must be mentioned: i390 supports two types of static global variables. One type is called *PU local data*. It is accessible by only one processor of the symmetric multiprocessor system and is duplicated for every processor in the system. The other type, called *system global data*, is accessible by all processors as shared data and exists only once in the system. Until the advent of the z990 system, this PU local data variables setup was achieved by the loader of the system by duplicating the PU local data section and TOC for each processor in the system. Each TOC pointer was initially set up to point to the TOC of the specific processor. Because the glue code replaces the TOC when doing a transition from one module to another, each processor is able to access only its own PU local variables and the shared variables; it cannot access the PU local data areas of other processors. In that respect, the concept of using a GOT to address the PU local data variables is quite similar to the old TOC concept.

Using the thread local storage (TLS) concept introduced with GCC 3.3 would have been another option to handle the PU local data requirements of i390, but when i390 development for System z9 began, the GCC 3.3 TLS support was still under development. The decision was made not to use it for System z9, but rather to use the GOT concept as a staged approach to open-source standards for i390 code.

From this description of the PU local data requirements for i390, it can be seen that the non-PIC code does not work in this environment because it is not possible to access PU local data variables, which reside at

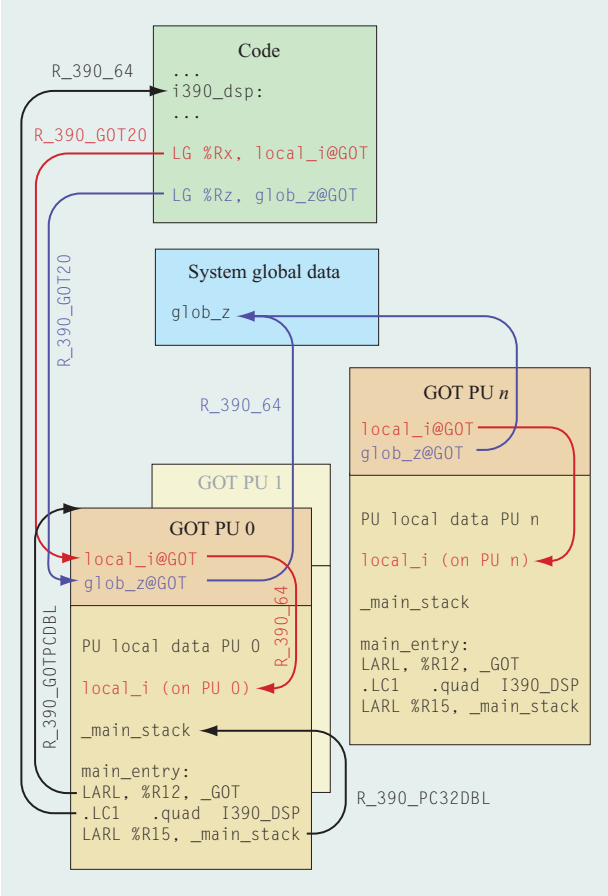

Figure 3

Relocations and multiple data area handling in i390 code.

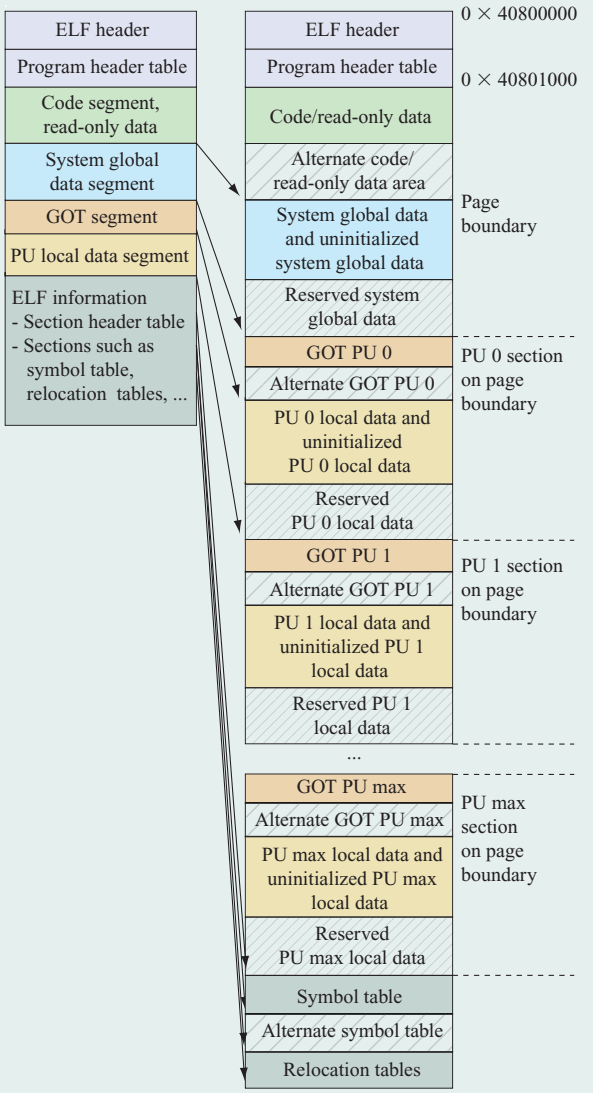

Figure 4

i390 ELF file layout and HSA storage layout.

different addresses for each PU. A relative relocation from the code (which exists only once in the system) to multiple data areas (a 1-to-*n* relocation) is not possible. As can be seen in Figure 2, one cannot directly (i.e., without table lookup) refer to variables `local_i on PU 0` and `local_i on PU n` using the same LARL instruction because these variables are stored at different machine addresses.

On the other hand, the PIC code reloads the GOT pointer in the prolog of each function. This method does not support the concept of multiple GOTs because it is not possible to load a different GOT pointer for each processor from the function prolog. The function prolog exists only once in the system, that is to say, in the code area, which would again make this a l-to-*n* relocation, which is not possible.

To overcome this problem and still use ELF, the GCC was changed so that it does not set the GOT pointer in the function prolog of the compiled firmware. Instead, the system sets up the GOT pointer once at system startup. Having this compiler in place, the i390 code could be restructured to consist of only a single image instead of multiple modules.

New i390 ELF loader

The new i390 code format in ELF object file mode, which consists of only a single image, required a new loader program for the IML and concurrent i390 code update process. **Figure 4** shows the layout of the i390 code ELF object file [9] and how it is mapped into the hardware system area (HSA) of a System z9. The color code is used to show which parts of the file are copied into which

locations in memory (e.g., the orange shows the location of the GOT section in the file, and how it is copied multiple times into memory).

IML loader operation

The ELF file header and program header table are left unchanged in HSA. The code segment is copied to the target HSA location, and room for another code segment is allocated for later use as alternate code area for the concurrent i390 code update process. Then the system global data area is copied from the i390 code load file to HSA and extended by as many zero bytes as needed to hold the uninitialized portion of system global data, which is not contained in the ELF code load file. At the end of the system global data area, some extra space is reserved for new system global data variables that are introduced with a new i390 code load during the concurrent application of i390 code. Following this area, the GOT section is copied, and room for an alternate GOT section is reserved. The PU local data area section is also copied and extended as needed to hold the uninitialized PU local data variables, with space reserved at the end to support new PU local data variables. The primary and alternate GOT and PU local data areas are then duplicated for every processor in the system. Following these data areas, ELF control structures needed for i390 loader operation, such as the symbol table and relocation tables, are copied.

The access from the code to the data is accomplished by looking up the address of a variable in the GOT. The System z ELF application binary interface [10] defines general-purpose register (GPR) 12 as the GOT pointer. This pointer is initially loaded into GPR 12 by the i390 runtime environment, which resides in the PU local data sections. A System z processor enters i390 mode by means of a special restart interruption using an i390 restart new program status word that is set up by the i390 loader to point to the runtime environment of that processor. Since the runtime environment resides in the PU local data section (`main_entry` in Figure 3), it exists once per processor and can load the correct GOT pointer for this processor into GPR 12 using a LARL instruction. To allow each processor to access its own PU local data variables, the addresses of data variables in the GOTs for each processor have to be updated by the i390 loader to reflect the correct PU local data address for each processor. Figure 3 shows the most important relocations in i390 code that illustrate the IML loader operation and i390 code structure, as described above.

Concurrent i390 code update

Concurrent microcode update is an important characteristic of System z systems. It means that code fixes and updates can be installed and activated concurrently with normal system operation while operating systems and customer applications continue to run. Concurrent i390 code update is accomplished by copying the new i390 code section in the alternate code section (reserved during IML) while preserving the old i390 static data variables that reflect the state of the system. To make this work, the concurrent i390 loader has to update the references from the new code to correctly point to the old static data variables. In the PIC mode, there are no direct references between code and data; all references are done indirectly via the GOT. Thus, the loader has only to copy the new GOT that comes with the new code into the alternate GOT areas for each processor and then update the data addresses in the new GOT to point to the corresponding old data variables. All required updates and relocations are performed in a background process concurrently with normal system operation. When this process is finished, the system is synchronized and switched to the new code load.

Support for adding static data variables

A considerable restriction for concurrent i390 code update prior to the advent of System z9 was that it was not possible to add new data variables during this process. The layout of the static data areas could not be changed, and the data areas could not be extended. This prohibited the concurrent addition of new functionality, which typically requires new control data.

The System z9 concurrent loader introduced support for adding new data variables concurrently, as follows. The loader compares the symbol table of the new code load with the existing symbol table to identify new data variables that are introduced with the new code load. When a new data variable is found, it is copied into the space that was reserved during IML at the end of the existing data areas. The GOT entry holding the address of the new variable is already contained in the new GOT that came with the new code load. Only the content of the GOT entry has to be corrected to point to the new location of the variable. There is also initialization routine support for new variables: Whenever the loader finds a new variable, it searches for a corresponding initialization routine, which is identified by a special naming convention. All identified initialization routines are called with the highest priority after the switch to the new code load so that they can perform runtime initialization of the added variables before the new code accesses them for the first time. (See [11] for examples of the ways in which this feature enabled enhanced driver maintenance and is exploited there.) Similarly, it is now possible to delete existing static variables that are no longer needed by the new code load. (See [12] for a more detailed description of the process of adding and removing static variables.)

Function pointer support for i390

System z9 now fully supports the use of function pointers in i390. Function pointers are widely used in high-level programming languages instead of a procedure name literal string to refer to a specific procedure. They allow decoupling and modularization of code and ease the addition of new functionality into already existing computer systems. They do this, for example, by allowing registration of a new function at a base service provider such as a function recovery manager or communication handler service, or by providing support for procedure callback.

However, prior to the advent of System z9, there was no real concurrent i390 code update support for function pointers in i390. The problem was that function pointers are basically just addresses that can be stored anywhere in the i390 data areas. The concurrent loader had no knowledge of the locations of these function pointers, nor did it know how many of them existed. There were no relocation table entries that showed the relationship of a function pointer to the corresponding target procedure. After a concurrent i390 code update, the addresses of target procedures of function pointers changed, but the concurrent loader could not correct the function pointers because it was not able to find them. **Figure 5(a)** illustrates this problem.

To overcome this restriction, the concept of *smart function pointers* was introduced [13] [**Figure 5(b)**]. Smart function pointers are not just data structures that contain the target procedure address, like plain function pointers; instead, they are generated by the GPL8 compiler as a data structure that contains a relative branch instruction to the target procedure. It is important that the smart function pointers are part of the data section (not the code or the GOT section of the i390 code load) because they must be preserved across concurrent i390 code updates. The fact that smart function pointers are located in the data section differentiates them from the procedure linkage table stubs that are used by Linux. In Linux, the procedure linkage table stubs exist in the GOT section, which is replaced during concurrent patch. By using a relative branch instruction for the smart function pointers, a relocation entry is generated in the ELF relocation tables for each of the pointers, which can be used by the concurrent loader to identify all locations in the data area that have to be updated when the address of a target procedure of a smart function pointer changes. Since this update is part of the normal concurrent loader operation, no special code had to be added in the loader to support concurrent i390 code update for i390 function pointers.

Outlook

The work described in this paper is being continued for future generations of the zSeries. The agenda includes such topics as the following:

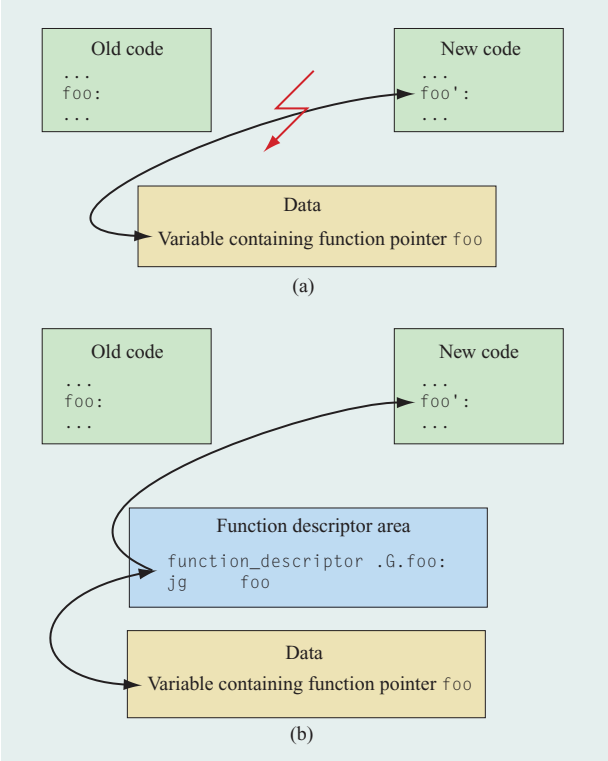

Figure 5

Concurrent i390 code update: (a) plain function pointers; (b) smart function pointers.

- *Updating the GPL8 compiler to be based on GCC 4.1:* The new version of the GCC family includes support for the new instructions introduced with the System z9, such as the add fullword immediate instruction, which improves the performance of thread local storage accesses. Using this compiler, one can consider using the thread local storage instead of the GOT concept for the PU local storage accesses. It is also quite interesting that, since the introduction of GCC 4.0, a new brand of compiler optimizations are being added at the middle end. Before, all of the optimizations were performed at the level of machine instructions (Register Transfer Language trees) where much of the information about the original code structure was no longer available. These so-called *general optimizer improvements* should help to make i390 code faster and smaller without extra work at the PL8 front end.
- *Supporting C++ for i390 development:* With the latest versions of GCC, the compiler has fewer dependencies on external libraries. With GCC 4.0, it is already possible to write C++ programs that do not have any

call to `Glibc`, thus allowing the use of C++ for i390 development. Using C++ offers multiple advantages: Among others, it is an object-oriented language that allows the use of new programming paradigms, it can be used as a more robust C compiler with array boundary checks and linker-type checks, and it supports templates.

- *Integrated Development Environment (IDE) for i390 development:* The current development requires the use of separate tools for the different tasks: SCons to build the targets, CVS to store the source code, dump-extractor to analyze the memory dumps, and so on. As a result, each developer must first be aware of the existence of the different tools, then understand the possibilities each one offers, and then learn how to use them (which generally implies writing text commands in a Linux console). This could be made easier by providing an IDE for microcode development. The open-source platform-independent Eclipse framework is being considered. Eclipse might be extended to add support for writing in PL8, compiling code, CVS inspection, and debugging.
- *Code measurements:* Many code-measurement tools are available, but in the past not all of them have been supported for the PL8 language. Using GCC allowed the introduction of code coverage measurements for PL8 with GCOV. The next GPL8 compiler will provide even more support for software metrics. Apart from McCabe's Cyclomatic Complexity [14, 15] and Halstead's Metrics [16], we will survey metrics that are PL8-specific or indicate code figures regarded as error-prone within firmware development.

Conclusion

It has been shown that compiler development within a firmware department allows quick responses to changed requirements and the ability to quickly implement improvements and new features needed for high-quality firmware development. Moving to a Linux-based development environment eliminates the need for proprietary tools and makes it possible to utilize industry-standard toolsets and open-source software verified by a large user community. This also makes the i390 development environment much more attractive, especially to new colleagues when they begin working at IBM. Introducing the new ELF loader eliminated severe restrictions in the area of concurrent i390 code update and allowed us to significantly enhance continuous availability for our customers by creating features such as enhanced driver maintenance. On the basis of the agenda described in the Outlook section, we can conclude that there will be many more improvements to come.

References

1. 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 (2002).
2. GCC Online Documentation; see *http://gcc.gnu.org/onlinedocs/*.
3. D. Edelsohn, W. Gellerich, M. Hagog, D. Naishlos, M. Namolaru, E. Pasch, H. Penner, U. Weigand, and A. Zaks, "Contributions to the GNU Compiler Collection," *IBM Syst. J.* **44**, No. 2, 259–278 (2005).
4. W. Gellerich, T. Hendel, R. Land, H. Lehmann, M. Mueller, P. H. Oden, and H. Penner, "The GNU 64-Bit PL8 Compiler: Toward an Open Standard Environment for Firmware Development," *IBM J. Res. & Dev.* **48**, No. 3/4, 543–556 (2004).
5. IBM Corporation, *zArchitecture Principles of Operation* (SA22-7832-04); see *http://www-03.ibm.com/servers/eserver/zseries/zos/bkserv/r7pdf/zarchpops.html*.
6. The SCons Foundation; see *www.scons.org*.
7. Python Software Foundation, The Python Programming Language; see *www.python.org*.
8. J. R. Levin, *Linkers and Loaders*, Morgan Kaufmann Publishers, San Francisco, CA, 2000; ISBN 1-55860-496–0.
9. ELF-64 Object File Format, Version 1.5, Draft 2, May 27, 1998; see *http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf*.
10. IBM Corporation, *zSeries ELF Application Binary Interface Supplement*; see *http://www.freestandards.org/spec/ELF/zSeries/lzsabi0_zSeries.html*.
11. A. Muehlbach, B. D. Valentine, D. Immel, M. S. Bomar, and T. V. Bolan, "Concurrent Driver Upgrade: Method to Eliminate Scheduled System Outages for New Function Releases," *IBM J. Res. & Dev.* **51**, No. 1/2, 185–193 (2007, this issue).
12. C. Axnix, H. Penner, and M. Mueller, "Method and System for Applying Patches to a Computer Program Concurrently with Its Execution," U.S. Patent Application No. DE920040070US1, March 10, 2006.
13. C. Axnix, H. Penner, and M. Mueller, "Method and System for Generating and Applying Patches to a Computer Program Concurrently with Its Execution," U.S. Patent Application No. DE920050025US1, June 21, 2006.
14. T. J. McCabe and A. H. Watson, "Software Complexity," *Crosstalk* **7**, No. 12, 5–9 (1994).
15. T. J. McCabe and C. W. Butler, "Design Complexity Measurement and Testing," *Commun. ACM* **32**, No. 12, 1415–1425 (1989).
16. M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*, Elsevier Science, Inc., New York, 1977; ISBN 0444002057.

Received March 10, 2006; accepted for publication May 1, 2006; Internet publication December 6, 2006

Christine Axnix *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (caxnix@de.ibm.com)*. Ms. Axnix is a Senior Engineer in zSeries processor firmware development. She received the Dipl. Ing. degree in electrical engineering from the Berufsakademie, Stuttgart, Germany. In 1989 she joined the IBM development laboratories in Boeblingen, where she initially worked on S/390* processor and coupling architecture verification test programs. She has also worked on various S/390 and zSeries projects in the i390 code area, including hardware initialization and reset, the communication interface between support element and CEC, capacity on demand, and initial firmware load and concurrent firmware maintenance for System z9. Ms. Axnix is currently leading the design for the capacity-on-demand functions for the IBM System z9 follow-on project.

Torsten Hendel *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (hendelt@de.ibm.com)*. Mr. Hendel studied software engineering at the University of Stuttgart and graduated with an M.A. degree in computer science. He works in zSeries processor firmware development in the areas of i390 kernel functions, virtualization, and SCLP. Mr. Hendel is currently the leader of GPL8 development and has responsibility for the PL8 front end and firmware-specific back end.

Michael Mueller *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (mulm@de.ibm.com)*. Mr. Mueller is a Distinguished Engineer 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 that same year, working in the S/370* Product Assurance Test Laboratory at Boeblingen. Mr. Mueller has held various positions in S/390 microcode development and system design.

Angel Nuñez Mencias *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (anunez@de.ibm.com)*. Mr. Nuñez Mencias received an Ingenieur degree in telecommunications from the Polytechnic University of Madrid and an M.S. degree in electrical engineering and computer science from the University of Stuttgart in 2004. He completed an M.B.A. degree from the National University of Distance Education in 2005. He joined IBM in 2002 and has worked on i390-related projects, including the ELF loader and the GPL8 compiler.

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.S. degree, joining IBM that same year. He worked on the development of the zSeries back end for GCC and is working on the Linux port for zSeries. Mr. Penner is currently working on a firmware stack for high-volume power systems.

Stefan Usenbinz *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (usenbinz@de.ibm.com)*. Mr. Usenbinz is an Advisory Engineer in zSeries Processor Firmware Development. After studying at the University of Cooperative Education Stuttgart, he became a graduate engineer and joined the IBM development laboratories in Boeblingen in 1997. He has worked in the area of RAS and is currently leading the design for the service element–CEC communication interface for the next-generation zSeries systems.