Assembler Restructuring in FermaT

Martin Ward
Software Technology Research Lab
De Montfort University
Bede Island Building,
Leicester LE1 9BH, UK
martin@gkc.org.uk

***Abstract*—The FermaT transformation system has proved to be a very successful tool for migrating from assembler to high level languages, including C and COBOL. One of the more challenging aspects facing automated migration, specifically when the aim is to produce maintainable code from unstructured "spaghetti" code, is to restructure assembler subroutines into semantically equivalent high level language procedures. In this paper we describe some of the many varieties of assembler subroutine structures and the techniques used by the migration engine to transform these into structured code. These transformations require a deep analysis of both control flow and data flow in order to guarantee the correctness of the result.**

Two separate case studies, involving over 10,000 assembler modules from commercial systems, demonstrate that these techniques are able to restructure over 99% of hand-written assembler, with no human intervention required.

I. Introduction

According to IDC research, much of the world's information still resides on mainframe systems [1] with some estimates claiming that over 70% of all business critical software runs on mainframes [5]. In industries such as banking, transportation, finance, insurance, government, and utilities, mainframe systems continue to run critical business processes. The most commonly used language in these systems is COBOL, but a significant proportion of systems are implemented in Assembler: amounting to a total of around 140 – 220 billion lines of assembler code [3]. The percentage of assembler varies in different countries, for example, in Germany it is estimated that about half of all data processing organizations uses information systems written in Assembler [6]. A typical large organisation will have several million lines of assembler code in operation: for example, the US Inland Revenue Service has over ten million lines of assembler currently in use.

In a recent survey of 520 CIOs internationally [2], more than half (56%) said that mainframe developers were struggling to meet the needs of the business and 78% stated that the mainframe will remain a key business asset over the next decade. 71% of CIOs are concerned that the looming mainframe skills shortage will hurt their business.

Analysing assembler code is significantly more difficult than analysing high level language code. With a typical well-written high level language program it is fairly easy to see the top-level structure of a section of code at a glance: conditional statements and loops are clearly indicated, and the conditions are visible. A programmer can glance at a line of code and see at once that it is, say, within a double-nested loop in the **else** clause of a conditional statement.

Assembler code, on the other hand, is simply a list of instructions with labels and conditional or unconditional branches. A branch to a label does not indicate whether it is a forwards or backwards branch: and a backwards branch does not necessarily imply a loop. Simply finding all the branch instructions which lead to a particular label involves scanning the whole program: and this does not take into account the possibility of "relative branch" instructions where the target of a branch is an offset from a label, or from the current location. An unlabelled instruction can therefore still be the target of a branch: so simply determining the "basic blocks" of assembler code is far from trivial.

As well as being more difficult to analyse, for a given functionality there is more code to analyse. A single function point, which requires 220 lines of C or COBOL to implement, will need about 400 lines of macro assembler or 575 lines of basic assembler to implement. On the other hand, a higher level language such as perl will require only 50 lines on average to implement one function point [4].

Assembler systems are also more expensive to maintain than equivalent systems written in high level languages. Capers Jones Research computed the annual cost per function point as follows:

Assembler	£48.00
PL/1	£39.00
C	£21.00
COBOL	£17.00

Many of these systems were originally implemented in Assembler due to the need to maximise the limited memory, CPU and disk capacity of the systems available at that time. Today, the greatest need is for flexibility: the ability to update systems to meet new business challenges. There is therefore a great need to migrate from assembler to more modern high-level languages, and to move from the mainframe to more cost-effective hardware platforms.

In previous papers [7,8,9,10,11] we have described the application of program transformation technology to automated migration from assembler to a high level language. The basic approach follows three stages:

1) Translate the assembler into our internal Wide Spectrum Language (called WSL);
2) Apply correctness-preserving WSL to WSL transformations to the code to restructure, simplify, raise the

978-1-4673-5739-5/13/$31.00

abstraction level, etc. These may include syntactic and/or semantic code slicing;
3) Translate the high-level WSL directly into the target language (currently either C or COBOL).

Since these papers were published there have been many improvements made to FermaT including:

- Improved detection and translation of self-modifying code;
- Extensive jump table detection;
- Improved dataflow analysis;
- Array detection and analysis (including detection of arrays of structures);
- Implementation of program slicing for WSL (our internal Wide Spectrum Language) and assembler;
- Static Single Assignment computation;
- Improvements in subroutine restructuring.

In this paper we focus on the last of these improvements, which tackles one of the more challenging aspects of assembler restructuring: extracting self-contained procedures from a mass of spaghetti code containing subroutine calls and returns.

First, a note on the terminology we use:

Module A module is assembled from a single source file plus associated copybooks and macros and generates a single listing and object file. FermaT processes each module in an assembler program separately and generates a target language file for each assembler module;

Assembler Program An assembler program consists of one or more modules which are assembled separately into object files which in turn are linked together to form an executable file;

Subroutine A subroutine is a group of assembler instructions which can be called from within the same module via an instruction which saves the return address (the address of the instruction following the call) in a register and branches to the start of the subroutine. A subroutine returns by branching to the saved return address;

Return Register The register in the subroutine call instruction in which the return address is stored;

Procedure A procedure is a WSL programming construct which can be called and which always returns to the statement after the call. The aim of assembler restructuring is to convert subroutines to procedures, and thereby eliminate the use of code addresses in the program.

II. Assembler to WSL Translation

A WSL *action system* consist of a collection of parameterless procedures with a starting action name and takes the form:

actions A_1 :
$A_1 \equiv$
  $\mathbf{S}_1$ **end**
...
$A_n \equiv$
  $\mathbf{S}_n$ **end endactions**

Within each action body $\mathbf{S}_i$, an action call of the form **call** A_j acts as a procedure call which results in the execution of the body $\mathbf{S}_j$. If execution reaches the end of $\mathbf{S}_j$ then it continues with the statement after the **call**. There may also be a special action Z such that **call** Z causes termination of the whole action system: in this case control flow is passed directly to the statement following the action system. If every execution of an action leads to an action call (i.e. control can never directly reach the end of the action body), then the action is called *regular*. If every action in the action system is regular then no action call will return and the action system can only be terminated by a **call** Z. In this case, actions are similar to labels and action calls are similar to **goto** statements.

The FermaT assembler to WSL translator translates each assembler module to a regular action system: so an unconditional branch translates directly to an action call, and a conditional branch translates to an **if** statement containing an action call. Any instruction or macro which causes the module to return to the caller, or to terminate abnormally, is translated using a **call** Z.

The translator works from the listing produced by the IBM assembler, rather than the collection of source files. This has the disadvantage that the translator has to be able to recognise and parse the many different listing formats produced by different versions of the assembler and different option settings. To do this, the translator is table driven: a text file `listings.tab` contains information about all the recognised listing formats (including the Tachyon assembler, Siemens assembler, Amdahl Personal Assembler, CA-Realia 370 Macro Assembler, MicroFocus Assembler, and z390 assembler). In all, there are a total of 38 different listing formats currently recognised.

Despite this disadvantage, there are several advantages to working from the assembler listing:

1) With a fully expanded listing, all information including copybooks and macro expansions is known to be present;
2) Object code data, instruction addresses and branch target addresses are available;
3) The cross reference listing gives the length, type, location and value of each assembler symbol.

The above information is, of course, implicitly present in the source code, but deriving it requires duplicating much of the functionality of the assembler. In addition, the translator would need to know exactly which options were used to assemble the production module: and the most accurate source of *this* piece of information is the listing itself.

The 16 general purpose registers are translated to the special variables r_0 to r_{15} and the condition code is translated to the special variable cc which can take one of the three values 0, 1, 2 or 3. Each instruction or macro is translated to a single action whose body consists of WSL code which implements all the behaviour of the instruction (including side-effects such as setting the condition code). The action name is either the label on the instruction or macro or a name of the form A_hexloc where hexloc is the six or eight digit hex representation of the current location (as provided in the listing).

As well as IBM mainframe assembler, WSL translators for other languages including Intel x86 assembler and a proprietary 16 bit embedded systems processor have been developed.

A. Subroutines in IBM Assembler

The IBM 370 and z/OS mainframes do not have a hardware stack. Inter-module calls are handled via a linked list of save areas. Each save area stores copies of all register values together with pointers to the next and previous save areas, and it is standard for every module to save and restore all registers in its savearea. So a caller can assume that register values are preserved over calls to an external module. Note that some non-standard code will pass parameters in non-standard registers, and even return results in registers: the migration engine can detect and process this non-standard code. Within a single module (which may consist of many thousands of lines of code) a different mechanism is used for subroutine call and return:

- A subroutine call is implemented as a BAL (Branch And Link) or BAS (Branch And Save) instruction. This stores the return address (the address of the next instruction in the sequence) in the indicated register and then branches to the indicated label. The subroutine body is responsible for storing the return address elsewhere, if the register is needed within the subroutine (or any of the subroutines it calls) for some other purpose.
- A subroutine return is implemented by reloading a register with the return address (if necessary) and then executing a BR (Branch to Register) instruction which branches to the address in the register.

Return addresses may be saved and restored in various places, loaded into a different register, overwritten, or simply ignored. Also, a return address may be incremented, or the return instruction may branch to an offset of the given return address. This feature might be used to branch over parameter data which appears after the BAL instruction, or to branch to one of several different return points. Merely determining which instructions form the body of the subroutine can be a major analysis task: there is nothing to stop the programmer from branching from the middle of one subroutine to the middle of another routine, or sharing code between subroutines, or branching directly out of a subroutine instead of doing a normal return to the stored address.

B. WSL Translation of BAL and BR

The WSL language does not have the concept of a "code address": so the assembler to WSL translator has to use some mechanism to emulate the BAL and BR instructions. A return address is represented as the integer value of the location of the instruction (this is the offset of the instruction from the start of the module).

A branch to register (BR) instruction is translated into WSL code to copy the register value to the special variable destination, and then call the dispatch action. The dispatch action tests destination against all the possible return addresses (these are all the possible return addresses which are encountered during the translation of the module). If the value matches, then we call the corresponding action. These return addresses are called "dispatch codes".

A branch and link instruction (BAL, BALR, BAS, BASR etc.) is translated to code which stores the dispatch code of the return point in the register indicated in the instruction, and then calls the action specified by the label in the instruction. The translator ensures that this dispatch code will be included in the dispatch action.

As an example, the following fragment of an assembler listing starts at location `0x0002F6`

```
CONVNX3  DS    0H
* LINK TO CONVERSION RTN
         BAL   R14,MM2MTH
* MOVE MMM TO OUTPUT
         MVC   DDO2M,WRKMTH
```

It translates to the following three WSL actions:

```
CONVNX3 ≡
  call A_0002F6 end
A_0002F6 ≡
  C : LINK TO CONVERSION RTN;
  r_14 := 762;
  call MM2MTH end
A_0002FA ≡
  C : MOVE MMM TO OUTPUT;
  DDO2M := WRKMTH;
  call A_000300 end
```

Note that each action ends with a **call** to the next action in the sequence. The value 762 is the decimal equivalent of the hex location `0x0002FA` for the return address. Code to test detination against the value 762 is also added to the dispatch action:

```
dispatch ≡
  if destination = 0 then call Z
  ...
  elsif destination = 762 then call A_0002FA
  ...
  fi end
```

The body of subroutine `MM2MTH` ends with a BR instruction to return from the subroutine:

```
MM2MTH   EQU   *
* CONVERT MM TO MMM (ALPHA)
...
MM2MTHX  EQU   *
* RETURN FROM SUB ROUTINE
         BR    R14
```

This translates to the WSL code:

```
MM2MTH ≡
  C : CONVERT MM TO MMM (ALPHA);
  call A_000374 end
...
MM2MTHX ≡
  call A_0003D0 end
A_0003D0 ≡
```

C : RETURN FROM SUB ROUTINE;
destination := r_{14};
call dispatch **end**

So, the code at A_0002F6 sets r_{14} to the value 762 and branches to MM2MTH. When action MM2MTHX is called, the value in r_{14} is copied to destination and dispatch is called. Since destination has the value 762, the dispatch action will call A_0002FA and the code following the subroutine call is executed.

III. WSL TO WSL TRANSFORMATION

Stage two in the migration process is the automated application of WSL to WSL program transformations. Space precludes a full discussion of all the transformations applied in the migration process: since are 156 different transformations currently implemented in the engine, of which around 50 are used in a typical assembler to COBOL migration. The process is controlled by the transformation Fix_Assembler which analyses the module at each stage and determines the next transformation to apply. Many transformations are applied repeatedly: typically a module has several thousand transformations applied before translation to the target language, although large and complex modules can require more than one million transformations!

In this paper we will focus on the transformations which convert assembler subroutine code to inline code or WSL procedures. These transformations come under the general heading of "dispatch removal" since the aim is to eliminate calls to the dispatch action, and ultimately eliminate the action itself.

A. Single Call

The simplest case is where a subroutine is only called from one place in the program. In this situation, the value of the return address at the start of the subroutine is a known constant: provided the start of the subroutine can only be reached from the single call. (In other words, there are no direct branches to the subroutine entry point or to labels inside the subroutine body, and control flow cannot "fall through" into the body of the subroutine). If this is the case, then the Constant Propagation transformation will replace references to the return register, which appear in the body of the subroutine, by the actual dispatch code. For example, if the call to MM2MTHX at A_0002F6 in our example were the only call to that subroutine (and there was no other way to reach the subroutine body), then Constant Propagation would replace references to r_{14} by the dispatch code 762. When Constant Propagation encounters a call to dispatch it checks if destination contains a known dispatch code: which is true in this case. If so, then the call to dispatch is expanded (the call is replaced by the body of the action) and simplified. The result is that the **call** dispatch statement is replaced by **call** A_0002FA.

A dispatch call has been eliminated and further restructuring of the action system will have the effect of "inlining" the procedure body. In the case of a simple subroutine, there is an option in FermaT which will allow the subroutine body to be recovered and converted to a WSL procedure just before translation to the target language. It is still important to inline subroutines wherever possible since this can allow FermaT to unscramble some unstructured code around the module. For example, a branch out of the middle of the subroutine does not cause problems if the subroutine has been inlined.

Constant Propagation can also determine when a branch to register is actually a return from the module itself to the calling module. Registers are initialised with a special dispatch code which indicates a return from the module when branched to (i.e. the WSL program should terminate). If dataflow analysis shows that this dispatch code reaches a branch to register, then the **call** dispatch is replaced by a **call** Z. Similarly, a branch to register which is applied to the address of an external module is converted to a call to the module. If there is a valid dispatch code in another register, then the called module is assumed to return to this address.

B. Multiple Calls to a Simple Subroutine

A simple subroutine is one with the following characteristics:

1) It has a single entry point;
2) It only returns directly to the caller: e.g. it does not branch to the middle of another subroutine;
3) It contains no calls to other subroutines.

This last restriction might appear to be rather severe: but note that once a subroutine has been inlined or transformed into a WSL procedure, there are no longer any *subroutine* calls to that code. Once all the subroutines called by a subroutine have been processed, the transformed subroutine body will no longer contain *subroutine* calls, and requirement (3) is satisfied.

The FermaT transformation engine therefore processes subroutines in a "bottom up" order as they appear in the subroutine call graph: "leaf" nodes which contain no subroutine calls are processed first, followed by the next higher "layer" in the call graph, and so on.

Therefore, provided all subroutines satisfy requirements (1) and (2), and there are no recursive calls, then there will be subroutines which satisfy requirement (3), and by processing the call graph in a "bottom up" order, *all* subroutines can be handled by this method.

Transforming a simple subroutine into a WSL procedure takes these stages:

1) **Control Flow Analysis**: Starting with the subroutine entry point, FermaT traces forwards through the action system call graph to find the actions which compose the body of the subroutine. The analysis stops at any call to dispatch. When the analysis is complete, each of the actions in the proposed procedure body is checked to see if it is reachable from outside the subroutine without going through the entry action. This may be due to a branch into the middle of the subroutine body, or a branch out of the subroutine body (for example, into a generic error handler). Since there is no way of determining, in advance of the analysis, which instructions comprise the subroutine body, a branch out of the body will initially cause the external code to be included in the

proposed body. See below for how these situations are dealt with.

2) **Data Flow Analysis**: Once a suitable procedure body has been determined, FermaT carries out a data flow analysis on the proposed procedure body. This analysis checks that the value assigned to the return register on entry to the subroutine will be propagated to the destination variable for every call to dispatch. The analysis needs to track the return address through any assignments which save and restore the return register. Note that some subroutines may increment the return address before returning (see below);
3) **Create a New Procedure**: If the above tests are successful then the set of actions composing the body of the subroutine are extracted from the main action system and composed into a new (sub) action system with the subroutine entry point as the entry action. Within this new action system calls to dispatch are replaced by **call** Z. This action system forms the body of a new WSL procedure. Calls to the original subroutine in the main action system are replaced by a call to the WSL procedure followed by **call** dispatch;
4) **Constant Propagation**: The dispatch calls introduced in step (3) can now be eliminated via constant propagation. Since the subroutine call has been converted to a procedure call, the return address can be propagated over the procedure body and used to eliminate the call to dispatch. Note that if control flow falls through into the subroutine body from the body of another subroutine, or if there is a branch to the subroutine entry point from another subroutine, then there may be dispatch calls which cannot be removed at this stage.

Each time a simple subroutine is successfully converted to a procedure, one or more calls to dispatch (the return points of the original subroutine) are eliminated from the program.

In our example, it turns out that the subroutine MM2MTHX is a simple subroutine which does not call any other subroutines, so it can be converted to a WSL procedure. The result is:

```
CONVNX3 ≡
  call A_0002F6 end
A_0002F6 ≡
  C : LINK TO CONVERSION RTN;
  r14 := 762;
  MM2MTH();
  call dispatch end
A_0002FA ≡
  C : MOVE MMM TO OUTPUT;
  DDO2M := WRKMTH;
  call A_000300 end
```

where:

```
proc MM2MTH() ≡
  actions MM2MTH :
  MM2MTH ≡
    C : CONVERT MM TO MMM (ALPHA);
    call A_000374 end
  ...
  MM2MTHX ≡
    call A_0003D0 end
A_0003D0 ≡
  C : RETURN FROM SUB ROUTINE;
  destination := r14;
  call Z end
```

We have removed the single **call** dispatch at the end of the subroutine and inserted calls to dispatch at each of the original subroutine calls (which are now procedure calls). However, constant propagation can remove these calls. For example, the **call** dispatch above transforms to **call** A_0002FA. The assignment $r_{14} := 762$ can also be deleted, since we know that this dispatch code has been accounted for. If there are no other references to this dispatch code, then we know that a **call** dispatch can no longer lead to **call** A_0002FA, so the call to A_0002FA can be removed from dispatch. In turn, this allows the call to be restructured: for example, if the call in A_0002F6 is the only call to A_0002FA, then it can be expanded and the action deleted from the action system.

This algorithm will handle any simple subroutine. Once all the subroutine calls in a subroutine body have been converted to procedures, then the subroutine itself can be converted. So, if an assembler module consist entirely of simple subroutines, it can be fully restructured using these techniques: regardless of the degree of subroutine call nesting present. However, in practice, there are many modules which do not keep to the constraints of a simple subroutines. The exceptions include:

- Subroutines which exit abnormally from the middle (for example, branching directly to common error handling code). This is extremely common in assembler programs;
- Falling through from one subroutine into the start of another;
- Branching from the middle of one subroutine into the middle of another subroutine;
- Returning directly to the caller's caller (instead of via the immediate caller);
- Multiple entry points to a subroutine, or equivalently, having a section of common code shared by several subroutines;
- Multiple return points: either returning to the given return address or to an offset on the return address;
- Passing parameters as inline data after the subroutine call. Here the subroutine uses the return register to address data, then increments it to get the actual return address;
- Sometimes saving the return address and sometimes not;

All the above exceptional cases appear so regularly that any automated assembler migration solution needs to be able to handle them. Each of these will be discussed in more detail in subsequent subsections.

In addition to the above, there are also frequently "bugs" in the assembler code which can prevent the module from restructuring. The code might be a genuine bug in the sense

that the module would crash or give incorrect results under certain circumstances, or it might be a highly convoluted way of coding something which happens to give the correct results but is very difficult to analyse and understand. Such highly convoluted code may only work "by accident" in the sense that a small and apparently innocuous change to the program may cause it to stop working correctly.

The most common bugs which prevent a module from restructuring are:

- Recursive subroutines, or mutually recursive sets of subroutines. Since assembler calls do not use a stack (unless explicitly programmed to do so), but store the return address in a fixed memory location, a direct or indirect recursive call will cause the original return address to be overwritten. A common example of this is when an error handler needs to write to a file or write a message to the operator, and the file operation or message handler itself checks for errors and calls the error handling routine. This may not be discovered in testing if it is unlikely for an error to occur while displaying a message during error handling. However, the fact that this control flow path appears may well prevent the module from restructuring. It is also regarded as a bad programming practice.

- Returning from a subroutine before it has been called. This can occur when there is a control flow path from a module entry point to the code which loads a saved return address and then executes a BR instruction to return from a subroutine, and where there is no call to the subroutine along the path. If this path is taken, then the program will either crash (if nothing has been saved in the return address) or branch to the return point taken in the last call to the subroutine (which may have occurred in a previous call to this module);

- Restoring the return address for a different subroutine: for example, subroutine A stores the return address in `ASAVE`, subroutine B stores the return address in `BSAVE` but then when B returns it loads the address in `ASAVE` and branches to it.

- Not restoring the return register on every path through the subroutine: there may be a path on which the return register's value is corrupted (e.g. by being used as the return register for another subroutine call), but is not restored. If this path can be taken, then it is clearly a bug, but if the path is not taken in normal processing its presence will still impede the restructuring process;

Less common bugs include calling a subroutine and passing the return address in the wrong register, eg calling via `BAL R15,SUBR` when `SUBR` expects a return address in `R14`! Another example is a branch back to the instruction which saves the return address: if this branch is taken after the return address register has been modified, then the corrupted value will overwrite the correct value in the save area.

All the above bugs (and many others!) have been found in production code.

C. Subroutine With Exit

The most common way in which a subroutine fails to be a simple subroutine is for there to be a branch out of the middle of the subroutine: typically this branch will be to error handling code. The subroutine has detected an error: so it is no longer interested in returning but branches directly to an appropriate error handling routine.

If all simple subroutines in a module have been processed, but there are still subroutines remaining, then another level of analysis is triggered:

- Any call to Z or to a label which is also reachable from outside the subroutine body is treated as an "abnormal exit" from the subroutine. The labelled action is not included in the procedure body, instead code is generated to store a value in the special variable exit_flag and the subroutine then returns to the caller. This flag is set to 0 for a normal return, a value of 1 means that the subroutine terminated by a **call** Z, each higher value (if any) indicates that the subroutine terminated by calling a distinct label outside the subroutine body;

- A dataflow analysis is then carried out on this modified subroutine body. If the analysis succeeds, then the subroutine is converted to a procedure.

- Subroutine calls are replaced by the following WSL code:
  SUBR();
  if exit_flag $= 0$ **then call** dispatch
  elsif exit_flag $= 1$ **then call** Z
  elsif exit_flag $= 2$ **then call** A
  ... **fi**
  where SUBR is the name of the new procedure.

- Constant propagation will now eliminate the dispatch call, as in Section III-B

If the subroutine body executes code which causes an abnormal exit or a return from the whole module, then this code will be translated to WSL statements ending in a **call** Z. This can be handled as an exit from the subroutine, as above. In our case studies, over 21% of modules needed exit_flag before they could be restructured (see Section V).

D. Returning to the Caller's Caller

Suppose subroutine `SUB1`, which has a return address in `R1` calls subroutine `SUB2` which has a return address in `R2`. The WSL code for a return from `SUB2` will therefore be:

destination $:= r_2$; **call** dispatch

However, `SUB2` might also decide to return to the caller's caller: i.e. to the address in `R1` which is the return address for `SUB1`:

destination $:= r_1$; **call** dispatch

If this is the case, then the dataflow analysis will fail: since the value of destination on this call to dispatch is not the value in r_2 on entry to `SUB2`.

To handle this situation, if the dataflow analysis fails then FermaT will check that:

- There are calls to dispatch where destination is loaded from the return register; and
- There are other calls to dispatch where destination is loaded from a *different* register.

If this is the case, then the second set of calls are treated as subroutine exits (as in Section III-C) and the dataflow analysis is re-computed. Note that this rule only applies when the original dataflow analysis failed: since it is possible for a subroutine to save the return register and reload it into a different register.

E. Fall Through into a Subroutine

Another common case is where a subroutine consists of some initial code followed by the execution of another subroutine, which uses the same return register. Instead of implementing a call to the second subroutine, a parsimonious programmer might just branch directly to the start of the second subroutine, or even arrange the code so that execution "falls through" into the top of the second subroutine. For example:

```
SUB1 ...
     body of SUB1
SUB2 ...
     body of SUB2
     BR    R14
```

where both SUB1 and SUB2 take a return address in R14. If SUB2 can be converted to a WSL procedure, then the resulting WSL code is:

SUB1 $\equiv$
... body of SUB1...; SUB2(); **call** dispatch **end**

Now it should be possible to process SUB1.

Note that if the initialisation code in SUB1 includes a subroutine return, then the branch (or fall through) to SUB2 can be mistakenly identified as an exit from SUB1. This does not necessarily prevent restructuring, but may cause problems later.

F. Multiple Entry Points

Section III-E is an example of a more general problem: a subroutine which has multiple entry points, or, equivalently, multiple subroutines which share a section of code. (These issues illustrate our comment in Section II-A that it can be difficult to determine which instructions form the body of a subroutine).

If the common code can be restructured into a single action which calls dispatch (or dispatch and Z only), then FermaT can create a WSL procedure out of the common code. In this case, the two subroutines can be "disentangled" since each includes a call to the common code.

G. Branch to the Middle of a Subroutine

A subroutine SUBX may include in its body a direct branch (conditional or unconditional) into the middle of another subroutine SUBY. This includes two common cases (among others):

1) SUBY may be the subroutine which called SUBX, or may be the caller's caller. In this case, instead of returning normally, we have an abnormal exit (Section III-C). If it is possible to call SUBX without having previously called SUBY, then there is a bug in the program: since when SUBY tries to return, there will be no valid return address.
2) SUBY may be using the same return register as SUBX. In this case, we have code which is common to both SUBX and SUBY. Instead of creating a new subroutine to share this common code, the programmer has made use of the fact that both subroutine use the same return register.

For correct restructuring, these cases may need to be handled differently. Case (1) is an exit from the middle of a subroutine (Section III-C), so the branch should be translated into code which sets a flag and returns. Case (2) is an example of shared code: the code we branch to needs to be converted to a procedure which can be called from both SUBX and SUBY.

FermaT can usually distinguish between these two situations due to a careful ordering of the application of restructuring heuristics.

H. Multiple Return Points

If a subroutine is carrying out a test, whose result needs to be returned to the caller, then the usual way to handle this is either:

1) Set a flag in the subroutine which is tested in the caller; or
2) Execute a test (eg a compare instruction) in the subroutine just before returning. The test will set the condition code, and the condition code is not modified by the Branch to Register instruction, so the condition code can be tested by the caller.

However, some programmers eschew these methods and instead make use of the fact that an unconditional branch instruction is exactly four bytes long. If an unconditional branch is inserted immediately after the call (BAL) instruction, then the called subroutine can choose to return to the instruction immediately *after* the branch, by incrementing the return address by four. The code to call the subroutine is:

```
         BAL   SUBR,R14
         B     SUBERR
NORM     ... normal return
```

In this case, the code labelled NORM may appear to be unreachable (typically, it is not even labelled): but it can be reached if SUBR increments the value in R14 by four before returning. Note that it is quite common for a BAL to be followed by an unconditional branch (which may in turn be followed by unreachable code) when the subroutine does *not* increment its

return address: so the two situations must be distinguished by the migration engine.

The body of SUBR may contain code like this:

```
SUBR     ...
         CLC   WRKMM,=CL2'12'
         BH    SUBRERR
         B     4(R14)
SUBRERR  BR    R14
```

If the month number (in WRKMM) is greater than 12, then we flag the error by returning to the return address passed by the caller (the caller will then execute the branch to SUBERR). Otherwise, we return to the address four bytes on from the given return address (which leads to the code labelled NORM).

The instruction B 4(R14) translates to WSL as:

destination $:= r_{14} + 4$; **call** dispatch

The same effect could be achieved via LR R14,4(R14) followed by BR R14, which will translate as:

$r_{14} := r_{14} + 4$; destination $:= r_{14}$; **call** dispatch

More generally, the subroutine call can be followed by two or more unconditional branches:

```
         BAL   SUBR,R14
         B     RET1
         B     RET2
         ... more branches
         B     RETn
FINAL    ... final return point is here
```

To select the return point, SUBR can increment R14 by the appropriate multiple of 4 (0, 4, 8, 12 etc.).

This will have the effect of preventing the dataflow analysis from succeeding: we cannot prove that the original value in the return register ends up in the variable destination, if the value has been incremented by 4 (or more) in between. To handle this case, the dataflow analysis needs to be more subtle:

- Any increment of the return address (by a multiple of 4) is noted, by setting the flag Incremented_Return;
- An increment of the return address by a multiple of 4 is treated as a copy, as far as the dataflow analysis is concerned.

With these modifications, the dataflow analysis will succeed but will note that the return address may be incremented. since the increment may be inside a loop, it may not be possible (via static analysis of the subroutine body) to determine the exact set of possible increments for the return address: and therefore the number of return points. Instead, the migration engine looks at the set of calls and checks for a sequence of one or more unconditional branch instructions after the call to determine the set of return points.

Note that there may also be abnormal exits from the subroutine: so we choose not to use the variable exit_flag to determine the required return point. Instead, we use the return register to indicate which return point the subroutine selected. The migration engine generates code which sets the return register to zero, then calls the procedure (generated from the subroutine body), then tests the return register to see which return point is required:

$r_{14} := 0$;
SUBR();
if $r_{14} = 0$ **then call** RET1
elsif $r_{14} = 4$ **then call** RET2
$\ldots$
elsif $r_{14} = 4 * (n-1)$ **then call** RETn
else call FINAL **fi**

I. Inline Parameters Passed to Subroutine

Usually, parameters are passed to a subroutine in named data areas, or in registers or via a pointer in a register. However, some programmers have noticed that the return register can serve two purposes: if we include inline data immediately after the subroutine call, then this data can be accessed via the return register.

This situation is particularly tricky because a single register is, in effect, being used to store two pieces of information: (a) a pointer to the parameters; and (b) an offset from the return address. In the assembler, the code is arranged so that these two values are the same, but in the WSL translation the values are distinct. Therefore, the assembler to WSL translator has to detect when parameters are being passed as inline data and generate specific code to handle this. Just because a subroutine call is followed by data does not necessarily mean that this data is being used as parameters: for example an error routine might be called via BAL, even though it does not return. In this case, the data could be the start of a data area used by the module. The assembler to WSL translator therefore checks that the following three conditions all hold:

1) The subroutine is called via a BAL or BALR which is followed by data declarations, which in turn are followed by more executable code;
2) The subroutine body uses the return register to address data, eg via a Load or MVC (Move Characters) instruction;
3) The subroutine return is to an offset on the return register.

If all these conditions hold, then the subroutine call is translated as follows:

1) Ensure that the first inline parameter (the first data declaration after the call) has a label;
2) Generate the following code for the call:
   $r_n :=$!XF inline_par(code, ADDRESS_OF(par));
   call SUBR
   where code is the dispatch code for the return address (the address of the code following the inline data), and par is the name of the first data area;
3) Change any branch to the return address plus an offset, where the offset equals the length of the inline data, to a direct return.

The transformations which scan WSL code looking for subroutine calls are modified to check for WSL code of the form

$r_n :=$!XF inline_par(code, ADDRESS_OF(par));
call SUBR

as well as $r_n :=$ code; **call** SUBR.

Once a subroutine has been detected and the control flow and dataflow analysis confirmed that it can be converted to a procedure, the statement

$r_n :=$!XF inline_par(code, ADDRESS_OF(par))

is converted to $r_n :=$ ADDRESS_OF(par). The converted subroutine body (which is now a WSL procedure) can now treat r_n as a simple data pointer from which the parameters can be addressed: in other words, the code in the subroutine which accesses the parameters requires no special handling.

This approach can also handle cases where a subroutine has both inline parameters *and* multiple return points.

J. Inline Code Converted to a Subroutine

One day a programmer wanted to re-use a section of inline code which appeared elsewhere in the program. Instead of going to all the trouble of extracting this section of code and turning it into a subroutine, he or she realised that by inserting a suitable Load Address instruction just before the block of code, and a Branch to Register instruction just after it, the block of code could be used as a subroutine without moving it out of place. The modified code looks like this:

```
        LA    R14,RETLAB
SUBR    . . .
        block of code is here
        . . .
        BR    R14
RETLAB  . . .
```

Elsewhere, the block of code can now be called as a subroutine via: `BAL SUBR,R14`

This form of subroutine causes no difficulty to FermaT because the Load Address instruction, followed by falling through to the `SUBR` label generates the following WSL:

$r14 := 1234;$ **call** SUBR

where 1234 is the offset of the label RETLAB. The dispatch code 1234 is also added to the dispatch action with corresponding label RETLAB. This is identical to the code generated by a normal subroutine call. The code between SUBR and RETLAB is converted to a WSL procedure, called as follows:

SUBR(); **call** RETLAB

and the RETLAB action is then restructured in the usual way.

IV. WSL to Target Language Translation

Once the automated WSL to WSL transformation stage is complete, the final stage in the migration process is translation from WSL to the target language. However, this is preceded by a further transformation stage in which the WSL code is manipulated to bring it closer to the target language. This stage is particularly important for migration to COBOL since the COBOL programming language has many restrictions and limitations which must be accommodated in order to generate compilable and executable COBOL.

A major limitation with some COBOL compilers is a lack of bit manipulation functions. Although bit fields and bit operations were introduced to the language in the ISO/IEC 1989:2002 standard, which was published in 2002, with a CS (Committee Draft) available in 1997, current mainframe compilers do not have native support for these operations. The WSL to COBOL translator can generate code for several targets including the following:

1) If the target is for Microfocus COBOL, then calls are generated to the built-in bit operations in Microfocus;
2) If the target is for IBM mainframe COBOL, then calls are generated to assembler support functions which implement the bit operations. These can process the bit operation at full speed, albeit with the overhead of a call.

Pack and unpack instructions (which convert string data to packed decimal and vice versa) can usually be implemented as a COBOL MOVE. For example, moving from a decimal field to a packed field will convert the string of decimal digits to a packed decimal value. However, the assembler instructions do not check the validity of the data, so a pack or unpack from a one byte source to a one byte target simply reverses the nybbles in the source field. This operation is frequently applied to general hex data, so has to be translated as a call to a support function.

Pointers are available in COBOL via the `SET ADDRESS OF ...` and `SET ... TO ADDRESS OF` statements but many COBOL programmers are unfamiliar with pointers, so the transformations attempt to eliminate as many pointer operations as possible via dataflow analysis and converting pointers to array indices.

The actual translation step is then a simple line-by-line translation of the "COBOL-like" WSL into a COBOL source file. This is followed by a further conversion of the COBOL source which handles formatting details such as indentation levels, spacing, and coping with the COBOL file format. A COBOL source line has a fixed format, dating back to the punched card era. The on-line card readers for the IBM 704, 709, 7090 and 7094 computers (introduced between 1954 and 1964) operated only in 'row binary' format: reading cards row-by-row into 12 pairs of 36-bit words ($2 \times 36 = 72$). The reader was not capable of reading more than 72 of the 80 columns of a card, so early compilers and assemblers could only 'see' those 72 columns. All COBOL compilers, including the most recent versions, therefore ignore columns 73–80 in order to be compatible with existing source code.

V. Case Studies

To test the effectiveness of the FermaT migration engine at restructuring commercial assembler modules, we took a copy of an assembler system currently in production in a large American insurance company. The system consists of over 3,000 programs and 8,991 assembler modules (many of which are used in more than one program). The FermaT migration engine was able to restructure all but 84 out of the 8,991

modules for a success rate of 99.07%. A significant number of the failures turned out, on analysis, to be due to bugs in the code or to code which would only work "by accident" (in the sense that an apparently innocuous change to the code would cause an error elsewhere in the program). Of the modules which were able to be restructured, 1,953 (21.9%) had subroutines with unstructured exits: these needed the exit_flag variable to be added before they could be restructured (see Section III-C).

A second case study involved 1,822 modules from an employee management system. After fixing all bugs uncovered by the migration process, all but seven modules could be restructured automatically, for a success rate of 99.62%. Of the modules which were restructured, 368 (21.3%) needed the exit_flag variable.

A major requirement is for the migrated COBOL to be maintainable. All modules which restructure will be converted into a hierarchy of single-entry single-exit procedures consisting of structured code with no GO TO statements. In addition, the McCabe cyclomatic complexity is typically reduced by at least 25%.

An example of a bug uncovered by the failure to restructure is the following code:

```
          BAS   R04,S00100
          ...
S00100    do some processing
          ...
          L     R04,S00R04
          BR    R04
```

Elsewhere in the module, S00R04 is used to save and restore the return address in R4, but in this subroutine the register is "restored" without having been saved. The return address will be overwritten by the contents of S00R04, which might be zero, or the return address left over from a previous call to a different subroutine.

Another example:

```
          LA    R15,4           IT CAME FROM VIM
          BAL   R10,SUBR020     GO DECIDE WHICH ONE
          LTR   R5,R5           DID WE FIND ANYTHING
          BZ    ERROR040        NO, SO ERROR CONDITION
          ...
ERROR040  EQU   *       ERROR CONDITION IF WE GET HERE
          ST    R15,ERRPECD2    SAVE ERROR CODE
          LA    R15,252         MAJOR ERROR CODE
          ICM   R15,12,ERRMODID INDICATE THE MODULE
          BR    R10             AND RETURN
```

Here, this code was called with a return address in R10. The error handler at ERROR040 stores some information and then attempts to return. But when we branch to ERROR040 after calling SUBR020, R10 now contains the return address that was passed in the call to SUBR020 so BR R10 will branch back to the LTR instruction and loop endlessly.

A. Performance

Performance of the migrated code depends somewhat on the type of code and also depends on the target platform. Typically, there is little degradation in performance when the COBOL is running in the mainframe environment: one test reported a 4% decrease in performance. In some cases, the COBOL can perform better than the original assembler: for example, the COBOL compiler can make use of newer and faster instructions than were available to the original assembler programmers. Also, self-modifying code can cause a severe performance hit on modern machines since the whole instruction cache is flushed when any instruction is modified.

VI. Conclusion

Despite the enormous technical and theoretical challenges presented by the analysis of assembler code: totally automated migration of assembler to a high-level language such as C or COBOL is feasible with complete restructuring achieved for over 99% of assembler modules.

References

[1] Robert Amatruda, "The Critical Need to Protect Mainframe Business-Critical Applications," IDC, White Paper, Jan., 2012.

[2] Compuware Corporation, "Mainframe Succession: Long Live the Mainframe," Compuware, White Paper, 2012.

[3] Capers Jones, *The Year 2000 Software Problem — Quantifying the Costs and Assessing the Consequences.*, Addison Wesley, Reading, MA, 1998.

[4] Capers Jones, "Backfiring: Converting Lines of Code to Function Points," *IEEE Computer* 28 #11 (Nov., 1995), 87–88.

[5] J. Scott, "The e-Business Hat Trick — Adaptive Enterprises, Adaptable Software, Agile IT Professionals," *Cutter IT Journal* 13 #4 (Apr. 2000), 7–12.

[6] Harry Sneed & Chris Verhoef, "Reengineering the Corporation–A Manifesto for IT Evolution," ⟨http://www.cs.vu.nl/∼x/br/br.html⟩.

[7] M. Ward, "Assembler to C Migration using the FermaT Transformation System," *International Conference on Software Maintenance, 30th Aug–3rd Sept 1999, Oxford, England* (1999).

[8] Martin Ward, "Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations," *Science of Computer Programming, Special Issue on Program Transformation* 52 #1–3 (2004), 213–255, ⟨http://www. cse. dmu. ac. uk/ ∼mward/ martin/ papers/ migration-t.ps.gz⟩ doi:dx.doi.org/10.1016/j.scico.2004.03.007.

[9] Martin Ward & Hussein Zedan, "Combining Dynamic and Static Slicing for Analysing Assembler," *Science of Computer Programming* 75 #3 (Mar., 2010), 134–175, ⟨http://www. cse. dmu. ac. uk/ ∼mward/ martin/ papers/ combined-slicing-t.pdf⟩ doi:10.1016/j.scico.2009.11.001.

[10] Martin Ward, Hussein Zedan & Tim Hardcastle, "Legacy Assembler Reengineering and Migration," *20th IEEE International Conference on Software Maintenance, 11th-17th Sept Chicago Illinois, USA.* (2004).

[11] Martin Ward, Hussein Zedan, Matthias Ladkau & Stefan Natelberg, "Conditioned Semantic Slicing for Abstraction; Industrial Experiment," *Software Practice and Experience* 38 #12 (Oct., 2008), 1273–1304, ⟨http://www. cse. dmu. ac. uk/ ∼mward/ martin/ papers/ slicing-paper-final.pdf⟩ doi:doi.wiley.com/10.1002/spe.869.