Improving Multiprocessor Performance of a Large Telecommunication System by Replacing Interpretation with Compilation

Valdemar Mejstad, Karl-Johan Tångby and Lars Lundberg
Department of Software Engingeering and Computer Science
Blekinge Institute of Technology, SE-371 25 Ronneby, Sweden
E-mail: Lars.Lundberg@bth.se

Abstract

In this report we consider different techniques for increasing the multiprocessor performance of an interpreted processing language in a large real-time telecommunication system, called Billing Gateway. We have implemented a prototype in which we first translate the language into C++ code, and then compile it using a C++ compiler. In our prototype we experienced a more than fourfold increase in throughput, compared to the original system, when running on an SMP with eight CPUs. The prototype also showed better scalability than the original system, due to less use of dynamic memory.

1 Introduction

The exact usage of a software system cannot always be foreseen when it is delivered. Therefore, the system usually contains several ways to customize it for the user's needs. In some cases a scripting language might be needed. For example, Microsoft Word [12] contains a language that can be used to create macros to customize a document or form. The tasks performed by this language are usually rather simple and not especially processing intense. Run-time interpretation is usually enough to satisfy performance needs.

Currently, the need for customization after delivery is increasing in many performance demanding real-time systems, e.g. systems in the telecommunication domain. For such systems, the performance reduction caused by run-time interpretation may cause serious problems.

We have studied a large commercial multi-threaded telecommunication system, called the Billing Gateway (BGw). It uses an interpreted language called the DataUnit Processing language (DUP) to process billing data in a telecommunication network. Previous studies have shown that DUP, and in particular the excessive dynamic memory management caused by the interpretation of DUP, causes serious performance problems on multiprocessors [8]. With new services being introduced in the mobile networks, such as GPRS and 3G [2], there is a risk that DUP will be a serious performance bottleneck.

In this report we investigate how the performance and scalability of the DUP-language could be improved by using compilation instead of run-time interpretation. Our investigation is based on an evaluation of a prototype that uses compilation instead of interpretation.

We start by giving a description of the Billing Gateway and its processing language in Section 2. That section also discusses the current performance problems. Section 3 discusses possible solutions and the solution we selected. In Section 4 we talk about our prototype implementation, and in Section 5 we evaluate our prototype. Finally, Section 6 contains our conclusions.

2 The Billing Gateway

2.1 Overview

The Ericsson Billing Gateway (BGw) is a mediation device that connects networks elements (NEs) in telecommunications networks with post processing systems (PPSs) (see Figure 1). High throughput is very important in the BGw.

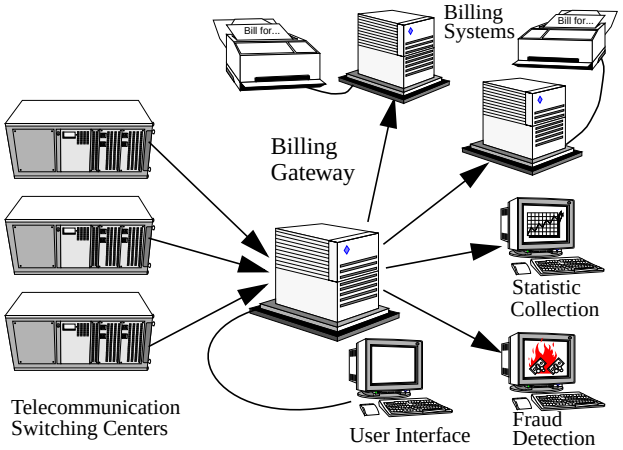

Figure 1. A billing configuration.

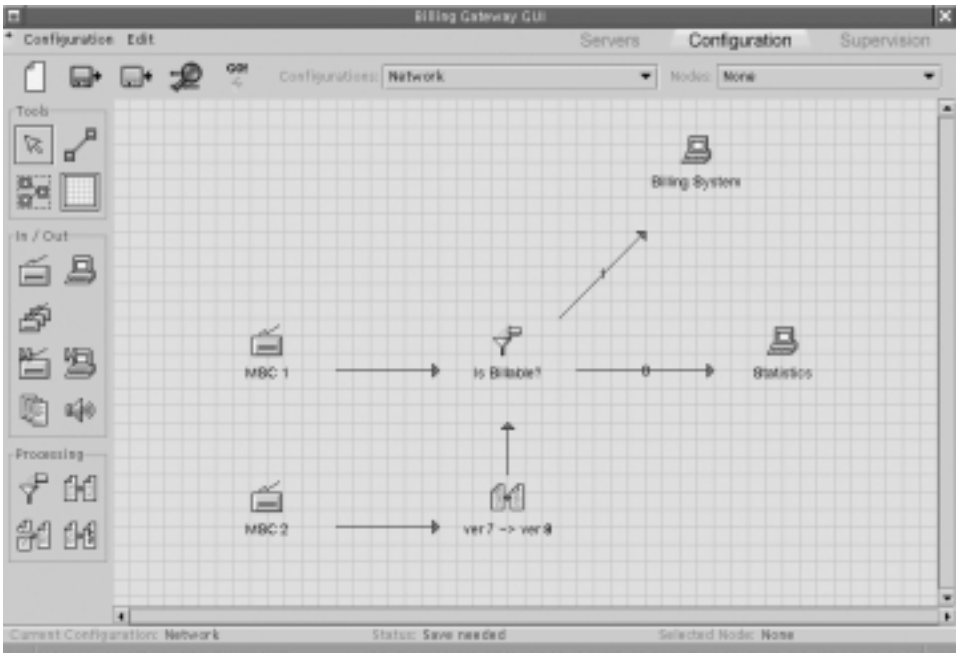

Figure 2. A billing configuration.

All information about calls in a network are handled in call data records (CDRs). The CDRs are generated by NEs, such as telecom switches, and can contain information about call duration, originating telephone number, terminating telephone number, etc. The CDRs are sent to post processing systems in order to perform, for example, billing and fraud detection.

One reason for using BGw is the flexibility it introduces. The BGw can handle several different protocols and data formats. It can perform filtering, formatting and other processing of the call data. The data flow in the BGw is configured using a graphical user interface (see Figure 2), icons are used to represent external systems, and the processing of the call data is accomplished using an internal processing language called DUP that we will examine further in this paper.

Call data is usually coded with Basic Encoding Rules according to a specification in Abstract Syntax Notation One (ASN.1/BER) [9]. When the data arrives to the BGw it is first decoded into a format that can be processed internally. All data is re-encoded before it is sent to PPSs.

In Figure 2 files are retrieved from two Mobile Switching Centers (MSCs) of two different versions. CDRs from MSC 2 are formatted to conform to the newer version, and all CDRs are checked to see if the calls are billable. Billable CDRs are sent to the billing system and others are saved for statistical purposes. For more information about the BGw architecture see [10].

2.2 Components

There are four components that use the DUP language: filter-, formatter-, matching- and rating nodes. A component in this case is represented in the GUI by an icon. Each node contains one DUP-script that is executed for every CDR passing through the BGw.

2.2.1 Filter

A filter node is used to filter out CDRs (e.g. IsBillable? in Figure 2). A filter node can, for example, filter out all roaming calls (a call made in a net other than the home net, e.g. when travelling in another country). The typical filter is rather simple and contains no more than around 10 lines of DUP code.

2.2.2 Formatter

Sometimes it is necessary to convert a CDR from one format to another before sending it on to post processing systems (e.g. ver 7 -> ver 8 in Figure 2). The size, in lines of code, differs very much from one formatter to another. In its simplest form it might only change a couple of fields, whereas large formatters can contain several thousand lines of code.

2.2.3 Matching

CDR matching makes it possible to combine a number of CDRs into one CDR. It is possible to collect data produced in different network elements or at different points in time and combine them into one CDR. Matching nodes usually contain a lot of code. From a couple of hundred lines up to several thousand lines of code.

2.2.4 Rating

Rating makes it possible to put price tags or tariffs on CDRs. It can be divided into charging analysis and price setting. The main purpose of charging analysis is to define

which tariff class that should be used for a CDR. The tariff class can depend on subscriber type, traffic activity, and so on. The price is then calculated based on the given tariff class, call duration and time for start of charge.

2.3 The Billing Gateway processing language

All data that will be processed in the BGw must be defined in ASN.1. The BGw builds up internal object structures of the data called DataUnits. One of the Billing Gateway's strengths lies in the fact that the user can use the DUP language to operate on those internal structures. That gives the BGw flexibility to suit a wide range of applications without the need for re-compilation, since the functionality can be changed on-site by the customer.

The DUP language is a mixture of C and ASN.1. It is a functional languages that borrows its structure from C, while the way of using variables comes from ASN.1.

2.3.1 Functions

A function is declared as follows:

```
<return type> <function name> ( <argument list> )
{<function body>}
```

The `<return type>` is one of the ASN.1 types supported by the BGw (see Section 2.3.2). The `<argument list>` is a list of arguments. Each argument is declared as follows:

```
<argument name> <argument type>
```

The `<argument type>` is an ASN.1 data type. All return values and arguments are passed as reference.

An example of DUP code can be seen below:

```
CONST INTEGER add(a CONST INTEGER)
{ declare result INTEGER;
  result ::= 10;
  result += a;
  return result;}
```

2.3.2 Variables and data types

Local variables can be declared at the beginning of a scope. A scope is enclosed by a '`{`' and a '`}`'. A variable is declared with the keyword `declare`.

```
declare <variable name> <variable type>
```

The variable type can be any of the standard ASN.1 types that DUP supports. The following types are supported.

```
1  BOOLEAN
2  INTEGER
3  OCTET STRING
4  NULL
5  ENUMERATED
6  SEQUENCE
7  SEQUENCE OF <type>
8  SET
9  SET OF <type>
10 CHOICE
11 IA5String
12 GraphicString
13 TBCD STRING
```

2.3.3 Assignments and comparisons

The following syntax is used for assignments:

- Assign: `<lvalue> ::= <rvalue>`
- Add assign: `<lvalue> += <rvalue>`
- Sub assign: `<lvalue> -= <rvalue>`
- Or assign: `<lvalue> |= <rvalue>`

The `<lvalue>` can be any expression returning a non constant data type (ex. a variable or a function returning a variable). The `<rvalue>` can be any expression returning a constant or non constant data type.

The following syntax is used for comparison:

- Equal: `<lvalue> == <rvalue>`
- Not equal: `<lvalue> != <rvalue>`
- Less than: `<lvalue> < <rvalue>`
- Greater than: `<lvalue> > <rvalue>`
- Less than or equal: `<lvalue> <= <rvalue>`
- Greater than or equal: `<lvalue> >= <rvalue>`

The `<lvalue>` and `<rvalue>` can be any expression returning a constant or non constant data type.

2.3.4 Control structures

DUP contains `if`-, `switch`-, `for`- and `while`-statements. All of these statements work in the same way as their C++ counterparts, except for the `switch` statement. In DUP the switch statement can use the ASN.1 `CHOICE`-data type. The condition statements can be any expression returning a `BOOLEAN`.

2.3.5 Jump statements

There are three different jump statements in DUP: `return`, `break` and `continue`. All these statements work as in C++. Goto and labels are not supported in DUP.

2.3.6 IN statement

DUP has a built in operator for checking if the value of a variable is equal to one value of a set of values.

```
<expression> IN [<constant list>]
```

The `<expression>` can be any type of expression returning a constant or non constant data type. The `<constant list>` is a comma separated list of constants.

2.3.7 Built in functions

DUP also contains a set of built in functions that can be used to perform on DataUnits. These functions are implemented in C++ to increase performance. Examples of built in functions are `mid`(), which is used to extract parts of a string, and different functions for handling dates.

2.4 Interpretation of DUP

The entire DUP implementation is accessed through three interface classes (see Figure 3): DUPBuilder,

DUPRouter, and DUPFormatter. A fourth class called DUProcessing is used by these classes to interpret and execute a DUP-script.

2.4.1 DUPBuilder

The DUPBuilder uses a PCCTS [15] generated parser to build a syntax tree [1] of the DUP-script. This means that a tree structure of C++ objects is created that represents the DUP source code. The DUPBuilder returns a DUProcessing object, which is the root node in the syntax tree.

2.4.2 DUProcessing

In order to execute a DUP-script, the syntax tree created by the DUPBuilder is traversed in a recursive-descendent manner. As each node in the syntax tree is traversed, code that is associated with that node is executed. When the entire syntax tree has been traversed the execution of the DUP-script is complete.

The syntax tree is traversed by using the `()`-operator on a DUProcessing object, which in turn uses the `()`-operator on each of its directly underlying node-objects. This results in a large number of function calls, which is one reason why the interpretation of DUP-scripts is slow.

Parameters to a DUP-script are stored in a `DUPDUVector` that represents a stack. This object is then passed to the `()`-operator. This stack is passed to every node in the syntax tree as the tree is being traversed. It is used to pass arguments and return values between function calls during the execution of the DUP-script.

2.4.3 DUPRouter and DUPFormatter

The DUPRouter is the DUP interface towards filter components in the BGw. The DUPFormatter is likewise the DUP interface towards formatter-, matching-, and rating components. This means that filter-, formatter-, matching-, and rating components have DUP-scripts, which are represented internally by either DUPRouter or DUPFormatter objects.

Both classes receive the parameters to the DUP-script as DataUnits and store them on a stack that is passed to the `()`-operator of their DUProcessing object, which executes the DUP-script.

2.5 Performance problems

Slow processing is not the only problem in the BGw. An earlier paper on the performance of the BGw [8] has identified sever multiprocessor scaling problems. Being a multi-threaded application, one expects that performance would improve when adding CPUs. This was, however, not the case.

At the time for the investigation [8], BGw 3 was the latest release. The version had some new features and the most significant one was the DUP language. It was concluded in [8] that DUP was the main reason for the poor scaling and the reason is its heavy use of dynamic memory, and thereby the shared heap. This results in threads being locked on mutexes as multiple threads tries to allocate/deallocate dynamic memory simultaneously.

The solution at that time was to introduce a product called SmartHeap [13] that handles the heap more efficiently. Although that resulted in improved performance, the root of the problem still remains. The design of DUP has not changed since BGw 3 which means that dynamic memory is still heavily used. It would be desirable to reduce the dynamic memory allocations since this could increase multiprocessor scalability.

3 Possible solutions

Since DUP is an interpreted language, we looked at how the performance of other interpreted languages has been improved. Romer, et al [17], showed that the performance of an interpreted language is primarily a function of the interpreter itself and relatively independent of the application being interpreted. This implies that in order to improve the performance of DUP we should focus on the interpreter. One solution for doing so is to use specialized hardware to support specific language environments. However, the paper [17] states that there is significant potential for improvement through software means. One technique incorporated by many of the discussed interpreters is some form of compilation.

One of the most popular interpreted languages that has used compilation to gain performance is Java. Java is not interpreted in the same form as DUP. Instead, Java is “compiled” to an intermediate language, called byte-code, that is then interpreted by a Java Virtual Machine. Much has been written about the performance problems of Java and Sun has in different ways tried to improve it. In version 1.1 of the Java Development Kit a just-in-time compiler (JIT) was introduced. It performed byte-code to machine code compilation on the fly when running a Java program. Later, in version 1.3.1 of the Java 2 Platform Standard Edition, Java HotSpot [21] was introduced. It uses adaptive optimization to boost performance more than the JIT could do. Similar to a JIT, Java HotSpot uses byte-code to machine-code compilation. If compiling to machine-code is the best way to improve performance, why not compile the complete language? Java cannot do that because of its platform independence. However, we are not constrained by this in the BGw and we can make DUP a compiled language.

Our suggestion is to compile the processing language when the BGw is still running, and then load the code dynamically at run-time. We believe that this will increase throughput and multiprocessor scalability.

3.1 Executing compiled scripts

Before investigating different compilation techniques we will look at how we can execute the compiled scripts from the BGw. There are several ways to add new

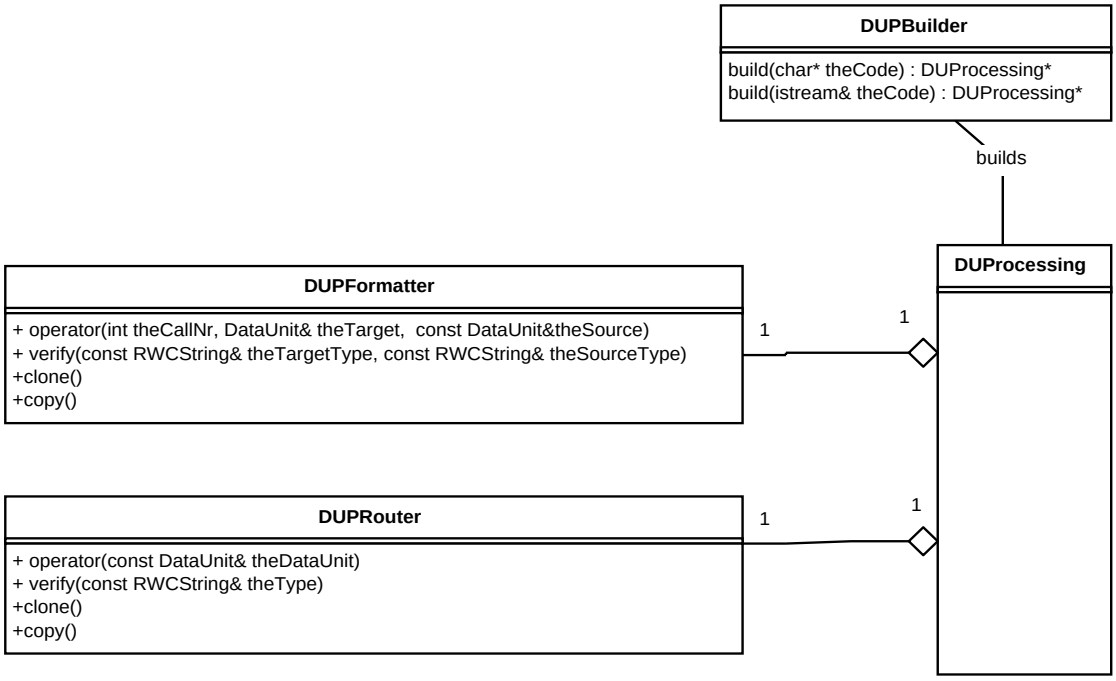

Figure 3. DUP interface classes.

executable code in a running program. The easiest way is to compile all DUP-code into executables and then run them from the BGw. The problem with this approach is that it would require a `fork()` [18], which is a system call. This means that the processor has to switch to kernel mode [22], which takes time. This solution would be very time consuming in the BGw since `fork()` would be called once for every CDR in every component. This would result in thousands of calls to `fork()` every second, which would reduce performance.

We wanted to dynamically load executable code into the running BGw. Today the BGw uses shared libraries in some places to accomplish this, but there are many other solutions [14]. Piculell and Myrén [14] concludes that Dynamic C++ classes is the best way to dynamically load new code into the BGw. However, Dynamic C++ [7] classes are built upon shared libraries and the extra features that it provides introduces some overhead compared to using pure shared libraries. Since DUP is not an object oriented language we do not need the extra features of Dynamic C++ classes and we came to the conclusion that shared libraries was the best choice.

3.2 Compiling the code

With compiling we mean translating the DUP source into machine language for the target machine. In this section we evaluate different techniques for doing this.

3.2.1 Making a complete compiler

One solution is to create a compiler for the DUP language from scratch. We would then do the complete translation from DUP to machine code.

By creating a new compiler we have complete control over compilation and linking. We can also design the compiler to fit the rather small DUP language, and by doing so we would get a small and fast compiler. The structure of DUP is simple. Therefore, it is straightforward to construct a compiler for the language.

A problem if doing a compiler of our own, is that we will have to implement the dynamic loading of code ourselves. In order to integrate easily with the rest of the BGw we would preferably use standard shared libraries as discussed earlier. Implementing this seems unnecessary since it has already been done by others.

Another problem is connected to the fact that the BGw supports two platforms: Sun Solaris and HP/UX. This means that we have to produce two compilers, one that produces code for Sun hardware and one that can support HP. Two implementations means two code-bases to maintain.

We believe that the biggest technical problem is the fact that the DUP code operates on DataUnits, which are C++ objects. Making our compiled code able to interact with C++ objects would probably be difficult. The built in DUP functions with which lots of interaction are done, are also written in C++. We can only speculate about the complexity of this last problem, but the other problems alone are enough to consider alternative solutions.

3.2.2 Producing middle code

To overcome the problems of creating shared libraries ourselves and multiple platforms, another solution is to produce middle code for an existing back-end. The GNU C++ Compiler [5] is modularly built so that the same back end can be used with different front-ends.

This approach solves the problem with supporting two platforms, since the compiler exists for both Solaris and HP/UX.

The problem with shared libraries can also be solved with this approach. With the GNU compiler it is possible to create UNIX shared libraries that can be loaded and unloaded at run-time.

We think that this approach is better than the previous one but it still has drawbacks. Producing middle code is an easier task than producing machine code since we only have to learn one target language, instead of two. However, we still have to learn the structure of the middle code. Although machine language might be more complex than middle code, the most work when implementing a compiler is the front-end [1]; middle code has a structure similar to assembler and the translation from middle code to machine code is rather straightforward. This means that we would still be implementing the hardest part of the compiler.

3.2.3 Translating DUP to C++

An alternative to using proprietary middle-code is to use C as an intermediate language. This approach is today used in for example Eiffel [11]. Ertl and Maierhofer [3] have done a study in which they make a compiler for the Forth language by first translating the Fort source into C. The initial reason for doing so was that they wanted to take advantage of the optimization facilities in modern C compilers. In their paper they show that their solution is several times faster than interpreting Forth code, and that it is even faster than BigForth, which is a native Forth compiler. We believe that we could use this approach for achieving similar results for DUP.

Using a intermediate language solves the problems discussed above. Because we interact with DataUnits, which are C++ objects, we choose to translate the DUP-scripts into C++ instead of C. This simplifies the interaction with the DataUnit. Calls to the DataUnit-objects are inserted at appropriate places directly when doing the code translation. A drawback is that we have to include header files for the DataUnit and some other BGw-classes when doing the compilation. This means that certain header files must be shipped together with the BGw.

Using the C++ compiler means that we loose control over the time it takes to compile. The translation we do from DUP to C++ code is fast. However, a complete C++ compiler is a big application and it will take some time to complete the compilation process. Since C++ is more complex than DUP, a C++ compiler contains a lot of functions that are unnecessary when compiling DUP. Right now it is not a big problem if it takes a few minutes to activate a new configuration. However, in the near future requirements of availability will rise to a point where it may not be possible to stop the system to do any kind of maintenance.

3.3 Selected solution

After looking at our possible solutions we have come to the conclusion that the best way is to translate the DUP code to C++. The control and flexibility that is given by creating a complete compiler for DUP is outweighed by the technical problems it imposes. By using C++ as an intermediate language we also benefit from the advanced optimization facilities of a C++ compiler.

4 Prototype implementation

As described earlier, the entire DUP implementation is a separate package accessed only through a couple of interface classes. We decided to rewrite these interface classes in order to test our approach.

The solution we selected resulted in four major steps:

1. Translating DUP source code to C++ code.
2. Compiling and linking the translated C++ code to a shared library.
3. Loading the shared library into the running BGw.
4. Executing the compiled DUP-script.

4.1 Translation process

The translation was by far the most time-consuming and complex step. The BGw uses PCCTS [15] to describe the grammar of the DUP language and generate a parser for it. A source to source translation using PCCTS can be done in at least two ways. The first approach is to add actions to a PCCTS grammar that generate a complete translation on the fly. The second approach is to add actions to a PCCTS grammar that build an intermediate abstract syntax tree of the input. This abstract syntax tree can then be traversed as many times as needed in order to perform a correct source-to-source translation.

The ability of the first approach to translate between two languages is limited. It is best suited when the two languages are similar. However, since DUP is very similar to C++, we decided to implement the source-to-source translator using the first approach.

4.1.1 Translation optimizations

DUP-scripts are executed over and over again and it is therefore very rewarding to optimize the C++ code. When doing optimizations we studied several configurations to see what was most frequently used. We saw that all components, especially formatters and matching points, relied very much on DataUnit access. Thus, we chose to

focus most of our optimizations efforts on speeding up DataUnit access and, when possible, avoiding it.

Creating a DataUnit is very expensive since it involves creation of a lot of internal objects and thus also allocation of dynamic memory. The first optimization that we did was to avoid generating C++ code that created DataUnit objects for local variables every time the DUP-script was executed.

Further, all constants in a DUP-script are translated into DataUnits in C++ code. Here we saw another possibility to optimize. By creating global static DataUnits for each constant we could avoid generating code that created DataUnits for the constants each time the DUP-script was executed.

Accessing data fields that are located deeper in the hierarchy of a DataUnit involves a lot of string matching operations, which is not very performance effective. A DUName can be used to speed up the access to fields in a DataUnit. It works like a pointer directly to the wanted field. We generated global static DUNames in the C++ code to speed up the DataUnit accesses.

4.1.2 DataUnit operations

As previously stated, all operations that can be done on a DataUnit from the DUP language can be done from C++ using the existing classes in the BGw. To find out exactly how operations on a DataUnit should be done from C++ we analyzed how this was done in the existing BGw code when interpreting a DUP-script.

The DataUnit class overloads almost all C++ operators, but the operators only work between two DataUnit objects. This turned out to be a problem when translating logical expressions in the DUP language to C++ since we needed to be able to compare a DataUnit object to an integer that represented a boolean value in the C++ code.

To overcome this problem we added functionality to the DataUnit class to compare DataUnit objects with integers using the comparison operators ==, `!=`, <, >, <=, and >=. We also overloaded the operators &&, `||`, and `!` between two DataUnit objects and between DataUnit objects and integers.

4.1.3 Using existing language features

The built in DUP functions, that were mentioned in Section 2, were originally compiled into shared libraries and loaded at start-up by the BGw. We wanted to use these functions directly from the generated C++ code to avoid the extra overhead of loading and using shared libraries in the generated C++ code.

The built in functions were implemented as static methods in three separate C++ classes. This made it easy for us to call them directly from the generated C++ code.

A DUPDUVector object, which represents the stack in the interpreted version, was originally passed as the only parameter to these static methods and the actual parameters to the DUP function were stored on this stack. In the DUP language, parameters to the built-in DUP functions are passed as separate arguments, and we wanted to do the same thing in C++. We therefore changed the interfaces of the static methods to take as their parameters the DataUnit objects that were earlier stored on the stack.

The translation process simply replaces a call to a built in DUP functions with a call to the corresponding static method of one these three C++ classes.

4.2 Compiling and linking the C++ code

The PCCTS generated translator outputs C++ code to a file that is ready for compilation and linking. The DUPCompiler then invokes an external C++ compiler through a `system()` call to compile and link the translated C++ code to a shared library. We used Sun Compiler 4.2, which is a part of Sun WorkShop Pro [20].

4.3 Loading the shared library

The shared library that was created by the compiler is loaded into memory with `dlopen()`, which is a member of a family of routines that gives direct access to the dynamic linking facilities provided by the standard C library. The `dlopen()` routine returns a handle to the loaded library. This handle is returned by the `compile()` method in DUPCompiler.

4.4 Integrating with the Billing Gateway

In order to integrate our solution with the BGw we needed to change the implementation of the four classes that makes up the interface to the existing DUP implementation (see Figure 3). We kept the interfaces of the DUPBuilder, DUPFormatter, and DUPRouter towards the rest of the BGw as they were, only changing their internal behaviors. The interface of DUProcessing was slightly modified, but since it is only used from the DUPFormatter and DUPRouter there were no problems with this.

4.4.1 DUPBuilder

The `build()` method in DUPBuilder was changed to use the DUPCompiler to compile and load a DUP-script. The handle returned by `compile()` in DUPCompiler is passed to the constructed DUProcessing object, which is the C++ interface to the compiled DUP-script.

4.4.2 DUProcessing

This is were the compiled DUP-script is actually executed. The DUP-scripts of the formatter-, matching-, and rating components have start functions with identical interfaces, while the filter component's DUP-script have a different interface. This means that we only had to define two types of start function pointers:

```
DataUnit (*myRouterStartFunction)(DataUnit&);
DataUnit (*myFormatterStartFunction)(DataUnit&,
DataUnit&, DataUnit&);
```

These function pointers are initialized by using `dlsym()` with the loaded library's handle and the symbol name of the actual start function. The initialization is done by the constructor of the DUProcessing class.

On a higher lever, a compiled DUP-script is invoked by using the `()`-operator on its DUProcessing object. This operator originally took a DUPDUVector object as its parameter. On this stack the actual parameters to the DUP-script's start function were stored as DUType objects. DUType objects are used internally in a DataUnit to store its data.

Since we had to pass DataUnit objects rather than their internal DUType objects to the start function we changed the interface of the `()`-operator. Its parameter is a linked list of DataUnit objects containing the parameters to the start function.

The start function is invoked by calling the function pointer returned by the call to `dlsym()`. In the case of a filter component it looks like this:

```
DataUnit *theDataUnit = theParams.removeLast();
(*myRouterStartFunction)(*theDataUnit);
```

In this example `theParams` is the linked list of DataUnit objects passed to the `()`-operator and `theDataUnit` is the parameter to the DUP-script's start function.

4.4.3 DUPFormatter and DUPRouter

As described in Section 2.4, the DUPFormatter is used by formatter-, matching-, and rating components to execute their DUP-scripts. The DUPRouter is used by the filter component to do the same thing. The `()`-operator of these classes receive the parameters that should be passed to the DUP-script as DataUnit objects.

We had to change very little in these two classes. A linked list containing the parameters to the DUP-script is created and passed to the `()`-operator of the DUProcessing object that is associated with the actual DUPFormatter or DUPRouter.

5 Evaluation of solution

Since the BGw is a performance critical application, it is important to evaluate the performance increase that we accomplished with our changes.

One difference between how DUP is executed when interpreted and when compiled is the number of function calls that are made. As described in Section 2.4, the syntax tree that is created when interpreting a DUP script is executed by making function calls to each node recursively. All these function calls are avoided when compiling a DUP script and we believe that this is the primary reason for increased performance.

Another difference is the use of dynamic memory. In our solution we have removed all dynamic memory allocations and deallocations that were previously done during processing. We have accomplished this by using static DataUnits and DUNames. By doing this we hoped to address the scalability problem identified by Häggander [8].

When doing the evaluation we wanted to test the overall performance impact that our changes had on the system. Because of this we have measured throughput of data on an existing BGw configuration. For our tests we used a large test configuration that uses up all available processing resources. This configuration has been used by Ericsson to conduct performance tests. The configuration has ten equal processing flows with seven processing nodes each, which means a total of 70 processing nodes. The configuration reads data from disk and uses filters and formatters to process the data before it is finally written back to disk. Each flow processed about 55 Mb of data which means that the whole configuration processed approximately 550 Mb of data.

All tests were done on a Sun Enterprise 10000 [19] with eight CPUs. We ran our test configuration on 1, 2, 4 and 8 CPUs, respectively. The configuration is processing intense enough to keep all processors under full load during the whole test, even when running on eight CPUs. We measured how long the test took to complete and then calculated the throughput. For commercial reasons, Figure 4 does not show absolute throughput figures.

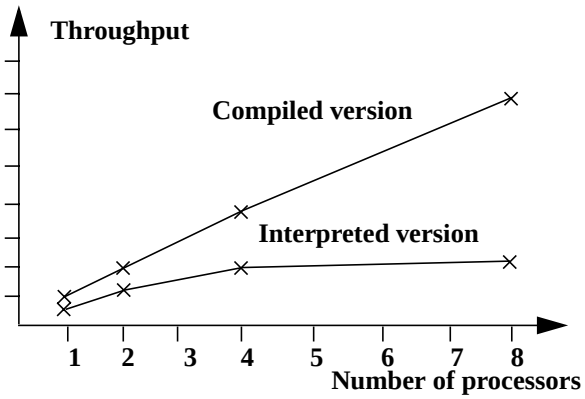

Figure 4. Throughput

The throughput performance is shown in Figure 4. As can be seen from the figure we managed to double the throughput already on one processor, but the most interesting results occur when we add processors. While the speedup of the current version of BGw starts to degrade considerably already with four processors, our prototype shows a nearly linear speedup curve up to eight processors. This means that with eight processors we manage to increase throughput with more than a factor of four, compared to the interpreted version.

Earlier reports [6][8] have connected the scalability problems of the BGw to the high number of spins on mutexes. A high spin-lock figure means that there are currently many threads waiting to lock a mutex, and are thus not executing. With multiple threads running on multiple CPUs this can lead to poor use of the hardware since the thread holding the mutex can prevent other

threads from running. To be able to scale well it was important for us to make sure that this figure was as low as possible. The mutex spin results are presented in Table 1.

Table 1: Mutex spins

Spins on mutexes (spins/second)	Interpreted	Compiled
1 CPU	0	0
2 CPUs	15-20	0-2
4 CPUs	280-400	5-13
8 CPUs	2000-3500	10-15

As can be seen from Table 1, BGw had 2000-3500 mutex spins per CPU every second when running on eight processors. With our changes we manages to lower the number of mutex spins to virtually zero. We contribute the better scalability of our prototype to this reduction of mutex spins. The table also indicates that we have succeeded with our goal to reduce dynamic memory managemen.

6 Conclusions

Flexibility and performance are often hard to combine. Today there are many applications that use an interpreted processing language to increase flexibility. This processing language can become a severe performance bottleneck. We have looked at this problem in a large and performance demanding telecommunication product called Billing Gateway. The product uses an interpreted processing language to process data in telecommunication networks.

After evaluating several solutions we decided to compile the language by first translating it to C++ and then using an off-the-shelf C++ compiler to produce executable code in the form of shared libraries. The translation, compilation and loading of libraries are performed on the fly when activating a configuration in the Billing Gateway.

We experienced an increase in throughput performance that exceeded a factor of four when running a large configuration on an eight CPU SMP [16]. We also experienced almost linear scalability when running the same configuration on 1, 2, 4 and 8 processors. This is an improvement over the current version of the BGw and we contribute the better scalability to the fact that we reduced dynamic memory management and thereby reduced the number of threads waiting on mutexes.

This case study shows that modern compilation and run-time linking techniques makes it possible to combine high flexibility, using a language like DUP, with high real-time performance. We expect that this approach will be useful for a number of large and complex real-time applications that needs to be customized by the user.

References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1986.

[2] Ericsson, *Technologies*, http://www.ericsson.com/technology/technologies_az.shtml

[3] M. Anton Ertl, Martin Maierhofer, *Translating Forth to Efficient C*, Proceedings of EuroForth '95 Conference, 1995.

[4] Robert A. Gingell, Meng Lee, Xuong T. Dang, Mary S. Weeks. *Shared Libraries in SunOS*, Sun Microsystems, Inc., 1987

[5] GNU, *GNU C++ Compiler*, http://gcc.gnu.org/

[6] Henrik Hermansson and Mattias Johansson. *Evolving into a distributed component architecture*, in Proceedings of the IEEE Seventh Asia-Pacific Software Engineering Conference, December, 2000, Singapore, pp. 188-195.

[7] Gísli Hjálmtysson and Robert Gray, *Dynamic C++ classes - A lightweight mechanism to update code in a running program*, Proceedings of the USENIX Annual Technical Conference, pages 65-76, June, 1998.

[8] Daniel Häggander and Lars Lundberg. *Optimizing Dynamic Memory Management in a Multi threaded application Executing on a Multiprocessor*, in Proceedings of International Conference on Parallel Processing, August, 1998, pp. 262-269.

[9] ITU-T. X.209, *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, 1993.

[10] Lars Lundberg and Daniel Häggander, *Multiprocessor Performance Evaluation of Billing Gateway Systems for Telecommunication Applications*, in Proceedings of ISCA, the 9th International Conference on Computer Applications in Industry and Engineering, Orlando, USA, December 1996

[11] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall, 1992.

[12] Microsoft, *Microsoft Word*, http://www.microsoft.com/office/word/

[13] MicroQuill, *Smart heap for SMP*, http://www.microquill.com/

[14] Henrik Myrén and Johan Piculell. *Run-Time Upgradable Software*, in Proceedings of the Seventh IEEE Real-Time Technology and Applications Symposium, May, 2001, Taiwan, pp. 226-235.

[15] Terence John Parr, *Language Translation Using PCCTS and C++ - A Reference Guide.* Automata Publishing Company, 1993.

[16] Gregory F. Pfister, *In search of Clusters (Second Edition)*, Prentice-Hall, 1998.

[17] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, et al, *The Structure and Performance of Interpreters*, ASPLOS VII, 1996.

[18] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison Wesley, 1992.

[19] Sun, *Sun Enterprise 10000*, http://www.sun.com/servers/highend/10000/

[20] Sun, *Sun Forte C++* (formerly WorkShop), http://www.sun.com/forte/cplusplus/

[21] Sun, *The Java HotSpot Virtual Machine*, Technical White Paper, http://java.sun.com/docs/white/index.html

[22] Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 2001.