Guard:
A Tool for Migrating Scientific Applications to the .NET Framework

David Abramson, Greg Watson, Le Phu Dung

School of Computer Science & Software Engineering,
Monash University,
CLAYTON, VIC 3800,
Australia

Abstract. For many years, Unix has been the platform of choice for the development and execution of large scientific programs. The new Microsoft .NET Framework represents a major advance over previous runtime environments available in Windows platforms, and offers a number of architectural features that would be of value in scientific programs. However, there are such major differences between Unix and .NET under Windows, that the effort of migrating software is substantial. Accordingly, unless tools are developed for supporting this process, software migration is unlikely to occur. In this paper we discuss a 'relative debugger' called Guard, which provides powerful support for debugging programs as they are ported from one platform to another. We describe a prototype implementation developed for Microsoft's Visual Studio.NET, a rich interactive environment that supports code development for the .NET Framework. The paper discusses the overall architecture of Guard under VS.NET, and highlights some of the technical challenges that were encountered.

1 Introduction

The new Microsoft .NET Framework is a major initiative that provides a uniform multi-lingual platform for software development. It is based on a Common Language Specification (CLS) that supports a wide range of programming languages and run time environments. Further, it integrates web services in a way that facilitates the development of flexible and powerful distributed applications. Clearly, this has applicability in the commercial domain of e-commerce and P2P networks, which rely on distributed applications.

On the other hand, software development for computational science and engineering has traditionally been performed under the UNIX operating system on high performance workstations. There are good reasons for this. FORTRAN, which is available on all of these systems, is the defacto standard programming language for scientific software. Not only is it efficient and expressive, but also a large amount of software has already developed using the language. Unix has been the operating

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2330, pp. 834–843, 2002.
© Springer-Verlag Berlin Heidelberg 2002

system of choice for scientific research because it is available on a wide variety of hardware platforms including most high performance engineering workstations. In recent times high performance PCs have also become a viable option for supporting scientific computation. The availability of efficient FORTRAN compilers and the Linux operating system have meant that the process of porting code to these machines has been fairly easy. Consequently there has been a large increase in the range of scientific software available on PC platforms in recent years. The rise of Beowulf clusters consisting of networks of tightly coupled PCs running Linux has driven this trend even faster.

An analysis of the features available in .NET suggests that the new architecture is equally applicable to scientific computing as commercial applications. In particular .NET provides efficient implementations of a wide range of programming languages, including FORTRAN [11], because it makes use of just-in-time compilation strategies. Further, the Visual Studio development environment is a rich platform for performing software engineering as it supports integrated code development, testing and debugging within one tool.

Some of the more advanced features of .NET, such as Web Services, could also have interesting application in scientific code. For example it would be possible to source libraries dynamically from the Web in the same way that systems like NetSolve [10] and NEOS [9] provide scientific services remotely. This functionality could potentially offer dramatic productivity gains for scientists and engineers, because they can focus on the task at hand without the need to develop all of the support libraries.

Unfortunately the differences between UNIX and Windows are substantial and as a result there is a significant impediment to porting codes from one environment to another. Not only are the operating systems different functionally, but the libraries and machine architectures may differ as well. It is well established that different implementations of a programming language and its libraries can cause the same program to behave erroneously. Because of this the task of moving code from one environment to another can be error prone and expensive. Many of these applications may also be used in mission critical situations like nuclear safety, aircraft design or medicine, so the cost of incorrect software can potentially be enormous. Unless software tools are developed that specifically help users in migrating software from UNIX to the Windows based .NET Framework, it is likely that most scientists will continue to use UNIX systems for their software development.

In this paper we describe a debugging tool called Guard, which specifically supports the process of porting codes from one language, operating system or platform to another. Guard has been available under UNIX for some time now, and we have proven its applicability for assisting the porting of programs many times. We have recently implemented a version of Guard that is integrated into the Microsoft Visual Studio.NET development environment. In addition, we have also demonstrated the ability to support cross-platform debugging between a UNIX platform and a Windows platform from the .NET environment. This has shown that the tool is not only useful for supporting software development on one platform, but also supports the porting of

codes between Windows and UNIX. As discussed above, this is a critical issue if software is to be ported from UNIX to the new .NET Framework.

The paper begins with a discussion of the Guard debugger, followed by a description of the .NET Framework. We then describe the architecture of Guard as implemented under Visual Studio.NET, and illustrate its effectiveness in locating programming errors in this environment.

2 Guard – a Relative debugger

Relative debugging was first proposed by Abramson and Sosic in 1994. It is a powerful paradigm that enables a programmer to locate errors in programs by observing the divergence in key data structures as the programs are executing [2], [3], [4], [5], [6], [7], [8]. The relative debugging technique allows the programmer to make comparisons of a *suspect* program against a *reference* code. It is particularly valuable when a program is ported to, or rewritten for, another computer platform. Relative debugging is effective because the user can concentrate on *where* two related codes are producing different results, rather than being concerned with the actual values in the data structures. Various case studies reporting the results of using relative debugging have been published [2] [3], [5], [7], and these have demonstrated the efficiency and effectiveness of the technique. The concept of relative debugging is both language and machine independent. It allows a user to compare data structures without concern for the implementation, and thus attention can be focussed on the *cause* of the errors rather than implementation details.

To the user, a relative debugger appears as a traditional debugger, but also provides additional commands that allow data from different processes to be compared. The debugger is able to control more than one process at a time so that, once the processes are halted at breakpoints, data comparison can be performed. There are a number of methods of comparing data but the most powerful of these is facilitated by a user-supplied *declarative assertion*. Such an assertion consists of a combination of data structure names, process identifiers and breakpoint locations. Assertions are processed by the debugger before program execution commences and an internal graph [8] is built which describes when the two programs must pause, and which data structures are to be compared. In the following example:

```
assert $reference::Var1@1000 = $suspect::Var2@2000
```

the `assert` statement compares data from `Var1` in `$reference` at line 1000 with `Var2` in `$suspect` at line 2000. A user can formulate as many assertions as necessary and can refine them after the programs have begun execution. This makes it possible to locate an error by placing new assertions iteratively until the suspect region of code is small enough to inspect manually. This process is incredibly efficient. Even if the programs contain millions of lines of code, because the

debugging process refines the suspect region in a binary fashion, it only takes a few iterations to reduce the region to a few lines of code.

Our implementation of relative debugging is embodied in a tool called Guard. We have produced implementations of Guard for many varieties of UNIX, in particular Linux, Solaris and AIX. A parallel variant is available for debugging applications on shared memory machines, distributed memory machines and clusters. Currently this is supported with UNIX System V shared memory primitives, the MPICH library, as well as the experimental data parallel language ZPL [12].

The UNIX versions of Guard are controlled by a command line interface that is similar in appearance to debuggers like GDB [16]. In this environment an assert statement such as the one above is typed into the debug interpreter, and must include the actual line numbers in the source as well as the correct spelling of the variables. As discussed later in the paper Guard is now integrated into the Microsoft Visual Studio environment and so is able to use the interactive nature of the user interface to make the process of defining assertions easier.

3 Success Stories

Over the last few years we have used Guard to debug a number of scientific codes that have been migrated from one platform to another or from one language to another (or both). In one case study we used Guard to isolate some discrepancies that occurred when a global climate model was ported from a vector architecture to a parallel machine [7]. This study illustrated that it is possible to locate subtle errors that are introduced when programs are parallelised. In this case both models were written in the same language, but the target architecture was so different that many changes were required in order to produce an efficient solution. Specifically, the mathematical formulation needed to be altered to reduce the amount of message passing in the parallel implementation, and other changes such as the order of the indexes on key array data structures needed to be made to account for a RISC architecture as opposed to a vector one.

In another case study we isolated errors that occurred when a photo-chemical pollution model was ported from one sequential workstation to another [3]. In this case the code was identical but the two machines produced different answers. The errors were finally attributed to the different behaviour of a key library function, which returned slightly divergent results on the two platforms.

In a more recent case study we isolated errors that occurred when a program was rewritten from C into another language, ZPL, for execution on a parallel platform [5]. This case study was interesting because even though the two codes were producing slightly different answers, the divergence was attributed to different floating point precision. However by using Guard it was possible to show that there were actually

four independent coding errors – one in the original C code, and three in the new ZPL program.

All of these case studies have highlighted the power of relative debugging in the process of developing scientific code. We believe that many of the same issues will arise when migrating scientific software from UNIX to Windows under the new .NET Framework and that Guard will be able to play an important role in assisting this process.

4 The .NET Framework

The Microsoft .NET Framework represents a significant change to the underlying platform on which Windows applications run [1]. The .NET Framework defines a runtime environment that is common across all languages. This means that it is possible to write applications in a range of languages, from experimental research ones to standard production ones, with the expectation that similar levels of performance and efficiency will be achieved. An individual program can also be composed of modules that are written in different languages, but that interoperate seamlessly. All compilers that target the .NET environment generate code in an Intermediate Language (IL) that conforms to a Common Language Specification (CLS). The IL is in turn compiled into native code using a just-in-time compilation strategy. These features mean that the .NET Framework should provide an efficient platform for developing computational models.

The Web Services features of .NET also offer significant scope for scientific applications. At present most computational models are built as single monolithic codes that call library modules using a local procedure call. More recent developments such as the NetSolve and NEOS application servers have provided an exception to this strategy. These services provide complex functions such as matrix algebra and optimisation algorithms using calls to external servers. When an application uses NetSolve, it calls a local "stub" module that communicates with the NetSolve server to perform some computation. Parameters are sent via messages to the server, and results are returned the same way. The advantage of this approach is that application programmers can benefit by using 'state of the art' algorithms on external high performance computers, without the need to run the codes locally. Further, the load balancing features of the systems are able to allocate the work to servers that are most lightly loaded. The major drawback of external services like this is that the application must be able to access the required server and so network connectivity becomes a central point of failure. Also, building new server libraries is not easy and requires the construction of complex web hosted applications. The .NET Framework has simplified the task of building such servers using its Web Services technology. Application of Web Services to science and engineering programs is an area of interest that requires further examination.

Visual Studio.NET (VS.NET) is the preferred code development environment for the .NET Framework. The VS.NET environment represents a substantial change to

previous versions of Visual Studio. Older versions of Visual Studio behaved differently depending on the language being supported – thus Visual Basic used a different set of technologies for building applications to Visual C++. The new VS.NET platform has been substantially re-engineered and as a consequence languages are now supported in a much more consistent manner.

VS.NET also differs from previous versions by exposing many key functions via a set of remote APIs known as 'automation'. This means that it is possible to write a third party package that interacts with VS.NET. For example, an external application can set breakpoints in a program and start the execution without user interaction. A separate Software Development Kit (SDK) called VSIP makes it possible to embed new functions directly into the environment. This allows a programmer to augment VS.NET with new functionality that is consistent with other functions that are already available. This feature has allowed us to integrate a version of Guard with Visual Studio as discussed in the next section.

5 Architecture of Guard

Fig. 1 shows a simplified schematic view of the architecture of Guard under Visual Studio.NET. VS.NET is built around a core 'shell' with functionality being provided by commands that are implemented by a set of 'packages' These packages are conventional COM objects that are activated as a result of user interaction (such as menu selection) within VS.NET, and also when various asynchronous events occur. This component architecture makes it possible to integrate new functionality into the environment by loading additional packages.

Debugging within the VS.NET environment is supported by three main components. The Debugger package provides the traditional user interface commands such as 'Go', 'Step', 'Set Breakpoint', etc. that appear in the user interface. This module communicates with the Session Debug Manager, which in turn provides a multiplexed interface into one or more per-process Debug Engines. The Debug Engines implement low-level debug functions such as starting and stopping a process and setting breakpoints, and providing access to the state of the process. Debug Engines can cause events to occur in response to conditions such as a breakpoint being reached, and these are passed back through the Session Debug Manager to registered event handlers. Each Debug Engine is responsible for controlling the execution of a single process. However the VS.NET architecture supports the concept of remote debugging, so this process may be running on a remote Windows system.

The VS.NET implementation of Guard consists of three main components. A package is loaded into the VS.NET shell that incorporates logic to respond to specific menu selections and handle debugger events. This package executes in the main thread of the shell, and therefore has had to be designed to avoid blocking for any extended time period. The main relative debugging logic is built into a local COM component called the Guard Controller. This is a separate process that provides a user interface

for managing assertions and a dataflow interpreter that is necessary to implement the relative debugging process. Because the Guard Controller runs as a separate process it does not affect the response of the main VS.NET thread. The Guard Controller controls the programs being debugged using the VS.NET Automation Interface, a public API that provides functions like set-breakpoint, evaluate expression, etc. We have also built a Debug Engine that is able to control a process running on an external UNIX platform. This works by communicating with the remote debug server developed for the original UNIX version of Guard. The UNIX debug server is based on the GNU GDB debugger and is available for most variants of UNIX. We have modified GDB to provide support for an Architecture Independent Form (AIF) [5] for data structures, which means it is possible to move data between machines with different architectural characteristics, such as word size, endian'ness, etc. AIF also facilitates machine independent comparison of data structures. It is the addition of this Debug Engine that allows us to compare programs executing on Windows and UNIX platforms.

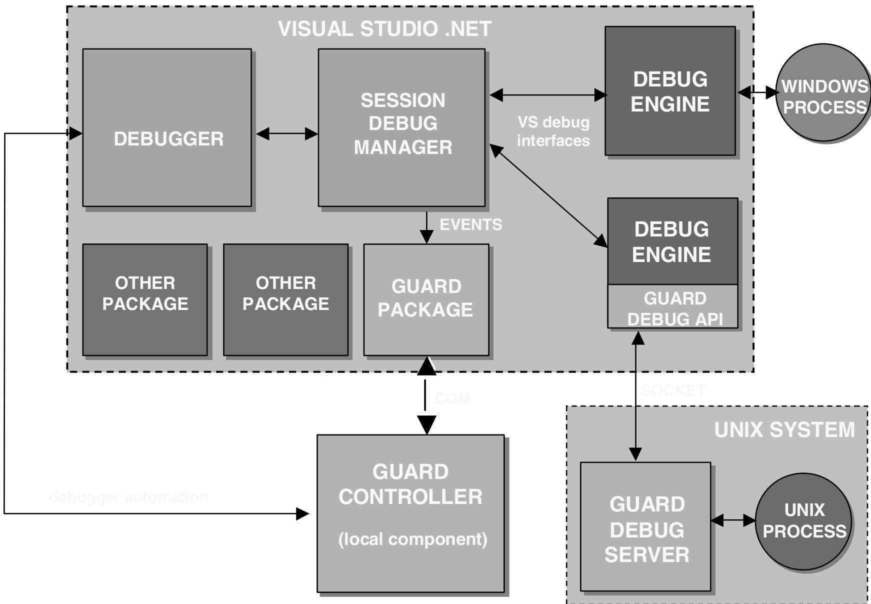

Fig. 1. Architecture of Guard for VS.NET

Fig. 2 shows the Guard control panel when running under VS.NET. When a user wishes to compare two running programs they must first be loaded into a VS.NET "solution" as separate "projects". The solution is then configured to start both programs running at the same time under the control of individual Debug Engines. The source windows of each project can then be tiled to allow both to be displayed at once.

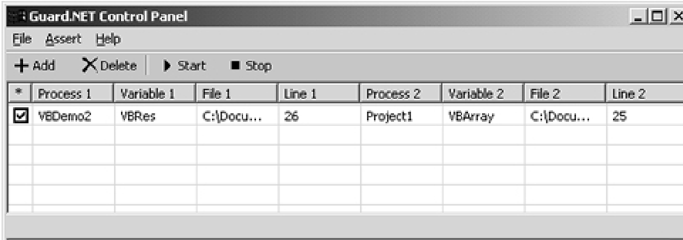

Fig. 2. Guard control panel

A user creates an assertion between the two programs using the Guard Controller, which is started by selecting the “GUARD” item from the “Tools” menu. The Guard Controller has a separate window as shown. An assertion is created in a few simple steps. A new, empty, assertion is created by selecting the “Add” button. Guard displays the dialog box shown in Fig. 3, which allows the user to enter the information necessary to create an assertion. The left hand side of the assertion can be automatically populated with the variable name, line number, source file and program information by selecting the required variable in the appropriate source window and then using a single right-mouse click. The right hand side of the assertion can be filled in using the same technique in the other source window. Finally the user is able to specify properties about the assertion such as the error value at which output is generated, when the debugger should be stopped and the type of output to display. The user can create any number of assertions by repeating this process and then launch the programs using the “Start” button.

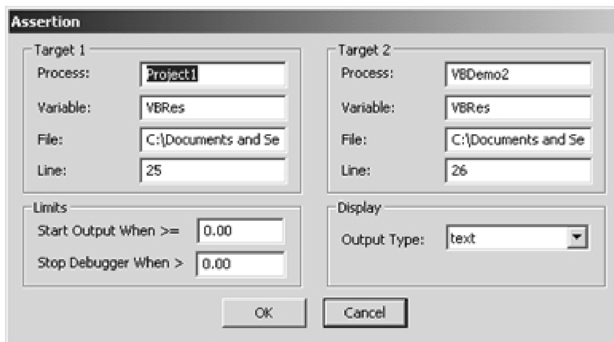

Fig. 3. New assertion dialog

Before commencing execution Guard automatically sets breakpoints at the locations in the source files specified by the assertions. During execution Guard will extract the contents of a variable when its corresponding breakpoint is reached and then perform a comparison once data from each half of the assertion has been obtained. Once the appropriate error threshold has been reached (as specified in each assertion), Guard will either display the results in a separate window or stop the debugger to allow interactive examination of the programs’ state. Guard currently supports a number of

display types include text, bitmaps and the ability to export data into a visualisation package.

6 Future Work and Conclusions

It is far too early to claim that .NET is a suitable platform for scientific computation since .NET is only currently available in Beta form and there are few commercially oriented codes available now, let alone scientific ones. As discussed in the introduction, we believe that .NET offers a number of potential benefits for large numeric models. However, the execution environment is very different from a traditional UNIX platform and so it is critical that as many tools as possible are available to facilitate the transition of existing legacy software. Guard is one such tool because it allows a user to compare two executing programs on two different platforms.

Whilst the implementation of Guard under UNIX is mature and has been used on many case studies, the current version under Visual Studio.NET is still a research prototype. We are planning a number of extensions that will be required if Guard is to be of practical use in supporting migration to .NET. The current user interface is fairly simple and must be made more powerful if it is to be applied to large programs. At present only simple data types and arrays are supported. We need to extend this to encompass the range of types found in scientific codes, such as structures and complex numbers. Assertions need to be able to be saved and restored when the environment is restarted, and line numbers should be replaced by symbolic markers which are independent of the actual numeric line number. We are also planning to integrate Guard into Source Safe [13], Microsoft's equivalent of SCCS [14] or RCS [15] making it possible to compare one version of a program with previous versions automatically. Finally, we plan to enhance the support for multi-process programs to make it feasible to debug programs running on a cluster of Windows machines.

In spite of these shortcomings we have shown that the prototype implementation works and that the technique of relative debugging is feasible in the .NET environment. We have tested this by debugging a number of small .NET programs using Guard.

Acknowledgments

This work has been funded by grants from the Australian Research Council and Microsoft Corporation. We wish to acknowledge the support of a number of individuals at Microsoft for their assistance on various issues related to Visual Studio and .NET. Particular thanks go to Todd Needham, Dan Fay and Frank Gocinski. We also wish to acknowledge our colleagues, A/Professor Christine Mingins, Professor Bertrand Meyer and Dr Damien Watkins for many helpful discussions.

References

1. Meyer, B. ".NET is coming", IEEE Computer, Vol. 34, No. 8; AUGUST 2001, pp. 92-97.
2. Abramson, D.A. and Sosic, R. "A Debugging and Testing Tool for Supporting Software Evolution", Journal of Automated Software Engineering, 3 (1996), pp 369 - 390.
3. Abramson D., Foster, I., Michalakes, J. and Sosic R., "Relative Debugging: A new paradigm for debugging scientific applications", the Communications of the Association for Computing Machinery (CACM), Vol 39, No 11, pp 67 - 77, Nov 1996.
4. Sosic, R. and Abramson, D. A. "Guard: A Relative Debugger", Software Practice and Experience, Vol 27(2), pp 185 – 206 (Feb 1997).
5. Watson, G. and Abramson, D. "Relative Debugging For Data Parallel Programs: A ZPL Case Study", IEEE Concurrency, Vol 8, No 4, October 2000, pp 42 – 52.
6. Abramson, D.A. and Sosic, R. "A Debugging Tool for Software Evolution", CASE-95, 7th International Workshop on Computer-Aided Software Engineering, Toronto, Ontario, Canada, July 1995, pp 206 - 214. Also appeared in proceedings of 2nd Working Conference on Reverse Engineering, Toronto, Ontario, Canada, July 1995.
7. Abramson D., Foster, I., Michalakes, J. and Sosic R., "Relative Debugging and its Application to the Development of Large Numerical Models", Proceedings of IEEE Supercomputing 1995, San Diego, December 95.
8. Abramson, D.A., Sosic, R. and Watson, G. "Implementation Techniques for a Parallel Relative Debugger ", International Conference on Parallel Architectures and Compilation Techniques - PACT '96, October 20-23, 1996, Boston, Massachusetts, USA
9. Czyzyk, J, Owen, J. and Wright, S. "Optimization on the Internet", OR/MS Today, October 1997.
10. Casanova, H. and Dongarra, J. "NetSolve: A Network Server for Solving Computational Science Problems", The International Journal of Supercomputing Applications and High Performance Computing, Vol 11, Number 3, pp 212-223, 1997.
11. http://www.lahey.com/netwtpr1.htm
12. L. Snyder, A Programmer's Guide to ZPL, MIT Press, Cambridge, Mass., 1999.
13. http://msdn.microsoft.com/ssafe/
14. Programming Utilities and Libraries', Sun Release 4.1, Sun Microsystems, 1988.
15. Walter F. Tichy, "RCS-A System for Version Control", Software-Practice & Experience 15, 7 (July 1985), 637-654.
16. Stallman, R. *Debugging with GDB – The GNU Source Level Debugger*, Edition 4.12, Free Software Foundation, January 1994.