Performance and Overhead Analysis in Runtime Code Modification

Jarosław Rudy

Wrocław University of Technology
Institute of Computer Engineering, Control and Robotics
Wybrzeże St. Wyspiańskiego 27, 50-370 Wrocław, Poland
jaroslaw.rudy@pwr.wroc.pl

Abstract. Availability is a key issue for a wide array of software systems and its importance in the era of cloud computing only continues to grow. For such systems every software update or system fault means decrease in availability or loss of the system’s state and both may involve heavy cost. In order to solve these problems, the system needs to either be able to quickly recover and retrieve its previous state or have the possibility of applying the needed changes at runtime, without the need to shut the system down. This paper proposes an experimental Versatile Code Generator tool, originally intended for simulating software, prototyping and programmable computer games, as a possible solution to above problems. The tool is capable of creating C programs subject to runtime code modification. The main focus of this paper are performance tests, researching overhead generated by the tool. To this end, static (i.e. original) and dynamic (i.e. created by the tool) applications were compared in a number of performance factors (including compilation and execution time, memory usage, source and executable code size). Different cases, including various functions number and size, various number and types of function arguments, have been considered.

Keywords: runtime software adaptation, code modification, performance analysis.
1. Introduction

Software systems are often designed using the client-server computing model. Nowadays this is even more true, as the concepts of distributed systems and cloud computing are used in wide array of software systems, reaching more and more aspects of life, including even everyday activities. Even now the users don’t really need a specific piece of software on their local machines. All they need, is access to the desired service at a remote site. Such solution allows the users to focus on the task at hand, without worrying about the details of the service itself. However, there is one important condition for such a system to work properly – the user needs access to the service whenever he wishes. In other words, in the case of a client-server and similar computing models, availability is a matter of a great importance. Moreover, achieving higher system reliability often requires use of considerable resources and effort. This is especially true for unusually large and complex software or computer systems. Consider, for example, the problem of transition from national currencies to the Euro that happened after 1995. Many European banks and insurance companies were faced with the challenge of updating their software (including the software used in automatic teller machines) while retaining all their normal services. According to Kniesel [1] not all of those institution succeeded, sometimes needing a few days to adapt to the new situation. Presumably, even those institutions that managed to apply the changes had to cope with heavy cost – that is the result of unanticipated changes. However, even predictable or scheduled changes can be difficult to deal with. For example, according to [2] at one point in time Visa Inc. managed its transactions with a system composed of roughly 50 million code lines, using over 20 mainframe computers. Such a system required frequent changes (estimated to take place 20,000 times each year). Since internal state of the system was crucial, all changes were applied by shutting down some of the computers, while others were used to retain their state. This solution resulted in 0.5% downtime on average.

There are many ways of increasing availability of a software system. One such idea is redundancy: having surplus resources (a set of HTTP server machines for example) allows the system to use them when the basic resources have been depleted (e.g. because too many users are requesting specific URL). Redundance is often desired or needed, but is not without its own drawbacks – additional server machines must be purchased and require maintenance (and both require additional time and money). Furthermore, the fact that the machines are surplus, implies that
they are not used part of the time or may be put on standby, therefore consuming resources while providing no services.

This paper focuses on a different approach. When the system is started, it usually meets the desired availability conditions (simply because there are no requests from clients yet). Then all threats to availability of the system during its lifetime stem from changes. Sources of these changes may be various: changes of clients (posting different number or size of requests than before), the environment (decrease in network throughput or increase in latency) or the system itself (due to software bugs, updates or security patches). In result, all that the system designers need to do, is to identify these changes and adjust the system accordingly. Such method has one crucial flaw: the alteration of a system is usually complex, expensive and risky process, involving the need to stop and restart the system, which causes another decrease in availability. Fortunately, there is another answer to this problem: software evolution (or SE). Software evolution is a general term meaning a series of changes applied to software system during its development life cycle (often both at design phase and runtime). This term is sometimes used interchangeably with the notion of runtime software adaptation, although the latter is used for changes made specifically during runtime – when system is deployed, active and in the maintenance phase. Moreover, it is often demanded that the change is applied during runtime, without the need to shut the system down, which alleviates the effects of availability loss. Our main interest lies in the alteration of software executable code (as opposed to the changes to the requirements specification for example), hence we will focus on the problem of runtime code modification. Many approaches were proposed to implement the concept of runtime code modification with varying success. They differ greatly in terms of: programming techniques and paradigms used to implement them, level of abstraction and restrictions on the target system. These approaches are usually accompanied by the analysis of their implementation details or possibilities. Often a case study is presented, showing practical use of discussed approach, along with a description of its positive effects on the target system (like reduced maintenance cost, increase in throughput or security etc.). However, there is very little research on the performance loss involved with the use of runtime code modification. While in many cases the resulting performance gain may exceed the overhead (generated by the modification or adaptation mechanism), still the extent of that overhead remains largely unknown. Thus, the primary purpose of this paper is to offer more detailed performance analysis of the runtime code modification, using the programming tool described further below. The secondary purpose is to present aforementioned
tool in context of its possible applications: high-availability systems, modelling, simulations and programmable video games.

The remainder of this paper is organized as follows. Section 2 presents short survey of previous attempts at performance analysis in runtime software adaptation. Section 3 briefly describes a Versatile Code Generator programming tool and its applications with particular emphasis on simulation software and programmable video games. Section 4 contains results of performance research of aforementioned programming tool. Finally, section 5 offers a summary of this paper along with research conclusions.

2. Previous work

As stated above, research on runtime software adaptation often disregards performance overhead analysis. Other issues concerning such systems are discussed instead. Those include (but are not limited too):

1. System consistency. Changing a part of code may interfere with existing data, classes and interfaces definitions, so it is important to ensure that the system remains functional and completely consistent after the modification has been applied. Approaches commonly address this point [3] and even consider it in greater detail [4].

2. Dealing with state of the system. Some approaches try to avoid the problem of state transfer (for example by adding or removing only those components or functions that contain only local state), but some identify, so called, global invariants (system elements that do not change during modification) and include special mechanisms for state transfer from old components to the new ones [5].

3. Performance increase analysis. This means studies of changes in performance of the target system. Usually only positive changes (like higher throughput or lower maintenance cost) are considered, however, while the overhead generated by the given approach is neglected.

The last of above points requires more elaboration. Consider a case study presented in [6], with use of KX and Worflakes systems in a multi-channel instant messaging service. While the main objective of this case study was to improve the overall quality of services, the results are much more extensive and include: (1)
drastic decrease in time needed for manual deployment procedure (from 2–3 days to \( \frac{1}{2} \)–1 day), (2) decrease in deployment script size from 500 Unix shell lines to 200 Java lines, (3) elimination of the need for an admin and technical team presence 24/7 in favour of automated alarm and procedures for handling well-known system faults, and (4) capability to start up another machine 40 seconds after the load threshold violation has occurred.

Another case study presented in [7] shows an example of a system capable of reacting to latency- and bandwidth-related problems by managing server groups and individual servers inside each group. This approach works by using architectural models and defining repair strategies. Research shows that the system is able to decrease and stabilize latency by detecting performance problems and deploying proper repair strategies.

Despite this, there are some approaches that research overhead introduced with their usage and here we will briefly discuss three of them. First example consists of framework capable of dynamic interface adaptation [8]. By utilizing the adapter design pattern, two components with incompatible interfaces can be connected properly. This allows for insertion of a new component when proper wrapping adapter is supplied. In this case adapters are the main source of trade-off, since every “dynamic” call needs to retrieve the appropriate adapter and invoke its method. The cost related to this was not directly expressed in numbers, but it was stated that it depends largely on the complexity of the adapter and the relationship between the components it connects.

Second approach is called POwerful Live Updating System (or POLUS) [9] and was designed with low overhead requirement as one of its main objectives. The system works by evolving the target system into newer version with a series of iterations. Presented research on servers (i.e. programs) of popular network protocols (HTTP, FTP and SSH) showed that impact on the target system is negligible in terms of update time (under 200 milliseconds) and performance loss per request (to 5%, around 1% most of the time).

The issue of performance overhead was also addressed in the case of DynamicRIO [10] – a tool used for code instrumentation and manipulation that works by monitoring and selectively executing target code. With regards to performance, the tool generally has little impact on the target system, but performance issues can arise – the reason being “not conforming to expected application patterns for which modern processors are heavily optimized” (e.g. using branching behaviour different from those commonly seen in modern compilers). Another important performance limitation of the DynamicRIO is high memory usage, which
becomes a serious problem when more than one application is being instrumented or manipulated on a single machine.

To sum it up, most approaches lack performance overhead research. Even when approaches that contain that kind of research are taken into account, considerable gaps in our knowledge regarding the performance overhead still exist. Firstly, the above research focus mostly on execution time and memory usage, ignoring some performance factors important for the user and the designer like compilation time and executable code size. Secondly, there is little research about the changes in the source code, which are important since many runtime software adaptation approaches rely on alterations to the original source code (such alterations are commonly referred to as “hooks”). Thirdly, the performance analysis rarely includes the resources consumed by the tool itself (as opposed to the resources used by the target system created with that tool) like memory usage or size and complexity. Finally, case studies usually involve only one or two independent real-life applications. However, a few similar applications should be compared in each case to determine more general dependencies involving performance overhead in relation to specific features of the target systems.

3. Versatile Code Generator

Research presented in next section were performed using a prototype programming tool called Versatile Code Generator [11] (or VCG for short). However, the primary purpose of this paper is overhead and performance research and not the detailed presentation of the tool itself. Therefore, here we will only briefly describe the tool as well as its most important properties and possible applications. The article cited above can be consulted for more details about the VCG tool and its inner workings.

3.1. The basics

The VCG tool is a set of simple C++ programs, that enable the software designer to transform slightly modified C/C++ “static” source code into “dynamic” executable code. Only change needed in the source code is addition of special preprocessor directives (#versatile) to denote chosen non-member functions as dynamic. Such functions are implemented as a separate dynamic libraries and can change their code and maybe even purpose (hence term “versatile code”) during runtime. The most important features of the VCG tool are listed below:
1. Ability to create applications which are capable of runtime code modification (via function hotswapping) for testing purposes. Changeable parts of the program take the form of functions which are called from external dynamic libraries, loaded and unloaded on demand. Whole process is supervised by Versatile Code Manager module and transparent to the user.

2. Support for versatile code versioning: many versions of the same function can exist at the same time with only one of them considered the current one. Applications created with VCG are capable of reacting to certain system faults and replace the faulty function version with an older (or default) one, but more stable. A process, called monitor, exists to perform this replacement, by detecting the abnormal termination (e.g. due to a signal) of the original application.

3. Low resource consumption and easiness of use – the VCG is a lightweight tool and is used similar to a compiler and requires little alteration to the source code (i.e. #versatile directives).

4. Possibility of applying the VCG tool to already designed or existing applications and minimization of effort required to do so. All that is needed is the C/C++ source code of the original program, regardless of software engineering techniques used to create it.

5. Possibility of application in any C/C++ software with need for frequent changes (e.g. alpha, beta and release candidate phases of software release cycle, software with frequent updates). Also possibility of use in software related to modelling, simulations and programmable video games (explained further below).

The research presented in the next section were conducted using slightly enhanced version of the VCG tool. Compared to the original tool, some performance issues were fixed, aiming at alleviating the main problem of the tool – high execution time overhead. Those efforts were partially successful and reduced some of the aforementioned overhead while affecting (and possible worsening) other performance factors like executable code size. Nonetheless, the VCG remains lightweight and simple tool, yet it is currently still in prototype stage aimed at performance research and is unsuited for large-scale or commercial use due to its restricted capabilities. Another important limitation is support for modification of non-member functions only. In order to remove this restriction in the future, two
possible enhancements are considered: 1) support for treating a set of bundled versatile functions as an interface of a single module (package, component), while retaining the low-level capabilities of the tool and 2) support for modification of entire classes. While the former option can be implemented relatively easily, the latter requires different approach than what the VCG tool uses in its current form.

3.2. Automated function restoration

One of the features mentioned above includes the concept of code versioning. Each new recompilation creates a new dynamic library, which is a subsequent version of the same versatile function and becomes the current (valid) one. Old function versions remain, however. Original reason for this was as follows: in some usages (described in next subsection) new software version can bring new software faults and, in result, crash an application. If the error occurred in one of the versatile functions, the application could mark current code version as faulty (invalid) and replace it with one of the old ones (presumably not marked as faulty). With this, the application could possibly continue running despite of the previous fault.

While such behaviour is desired, the VCG tool doesn’t support it in that way. There are two main reasons for that. First, when the error occurs, the application is midway during function call and it might be difficult to determine where exactly the execution stopped. Next, even if we manage to replace the faulty function with the old one, we are faced with the problem of calling a function that had its another version started already. This means that the concerned function should be reentrant, which greatly limits what such function can do – it shouldn’t hold any static or global data for example. Responsibility for making the versatile functions suitable for reentrancy would be placed on the designer most of the time, however approaches (see for example [12]) exist that are capable of transforming programs into their reentrant versions. Some programming paradigms and concepts also aim at avoiding non-local state and side effects in order to help solve above problems. These include, but are not limited to functional programming (not to be confused with procedural programming) and referential transparency. More on these concepts can be found in [13, 14]. Even when both function versions – current (faulty) and old – are reentrant, we still need to determine how to convert the local state of one function to the local state of the other. Most likely this would be defined by the designer and would require additional effort (which contradicts with the tool’s original aim).
Second main reason for not supporting original online faulty function replacement is because most of the system faults (like memory access violation) will result in termination of the process, meaning that the state of the application is lost. This effect is difficult to avoid without interfering with the operating system and language runtime libraries or putting additional conditions on the application (like handling all possible signals on Unix or storing the application state manually and restoring it later). We decided on a simpler approach: we deploy additional process (called monitor) as a parent process to the target application. Once the system fault occurs, the monitor will be woken up as a result of its child process terminating. Monitor can then use information left by the terminated process and perform function replacement. However, the main application needs to be restarted and its state is lost.

3.3. Possible applications

The aforementioned properties of the VCG tool can be used to address the issue of system availability described in the beginning of this paper. With versatile functions defined in proper places, it would be possible to perform system updates, apply certain fixes or patches without the need to restart the system. The change would require the reload of system dynamic libraries, but it would suspend the system for only a short time (few milliseconds in the case of smaller updates, though large-scale update would probably take up to few seconds). Moreover, only the versatile code execution will be suspended – all other functions can be called normally (meaning many threads can run simultaneously during update as long as they do not use the versatile code). The automated function restoration feature usefulness is limited in this case, but still certain system functions can have backup versions. These emergency versions can be designed with decreased performance, but with increased dependability or debugging capabilities for example. Therefore in the case of a system fault it would be possible to restart the system using less effective, but working emergency functions.

Aside from availability issues, two more possible applications were proposed when the VCG was first designed. The first is to use the tool in software related to simulation and modelling. Simulation systems commonly have some sort of function used to transform the state of the target model and this function is applied in a loop to simulate subsequent steps in the evolution of the model. Such function could be declared as a versatile function and would allow to change not only the basic parameters of the model, but its whole behaviour. Sometimes the
model may reach a state that would be an interesting starting point for simulation of another model. In this case there is no need to save the system state and use it in different simulation. All we need to do, is pause the simulation (but not terminate or restart it) to preserve the state, recompile the software to implement new behaviour, deploy the new code version and unpause the software to let the model evolve according to new rules. Moreover, the model is often changed in order to study new possibilities. If new model version is faulty, it would be possible to quickly restore old (and presumably correct) behaviour without the need to recompile the software.

Second possible application concerns online video games. We can imagine a multi-player network game where the players, being highly demanding type of user in general, are given the possibility to modify some parts of the game itself (for example by using code snippets in high-level and easy-to-use programming language designed specifically for this purpose). However, this approaches introduces two problems. Firstly, the changes made by the players can be applied quite often and should be performed in such a way, as not to hinder the game and the other players currently logged in. Second, players are not programmers, so their changes may contain many faults or be mutually exclusive. Game admins can resolve some of those cases, but the probability of a fault or undesired change making its way into the game still remains. Both problems can be avoided or alleviated by the use of VCG tool (assuming the tool gets past its prototype stage): the code snippet supplied by the players can be translated into C++ code which, in turn, can be supplied to the tool in order to generate new dynamic libraries. The libraries can then be deployed and used in the game. When errors are encountered, the game servers can be quickly restarted with previous (or default) version of the code. The whole process would require high level of management, however, in order to monitor which changes are considered as permissible, according to the current state of the game.

4. Research

In order to research the performance overhead involved with the use of Versatile Code Generator tool mentioned above we perform a series of tests. In most of those tests we build two C applications – one created by standard gcc/g++ compiler and the other generated by VCG. For clarity let’s call applications created by gcc/g++ and VCG “static” and “dynamic” respectively. Research results pre-
presented in this section are always results of comparison of: a) static and dynamic version of the same application or b) a few dynamic versions of different, but similar applications (in order to study how the overhead changes with the changes of application parameters). In both cases we compare the applications in terms of several performance factors namely:

1. **Source code.** This means both the source code ($C_S$) supplied by the programmer and the source code supplied to the gcc / VCG tool (called intermediate code, $C_I$). For static programs both source and intermediate code are one and the same. For dynamic programs source code is the input for VCG code analyser and intermediate code is its output. All values are in kilobytes (with 1 kB equal to 1024 bytes).

2. **Executable code.** For dynamic programs the executable code ($C_E$) includes all versatile code dynamic libraries, main application and the monitor program. For static programs it is just single gcc output executable. All values are in kilobytes.

3. **Memory usage.** Two values are considered. The higher one is total virtual memory (total memory, $M_T$) used by the process. The lower one is the size of data, stack, text (program) and shared memory segments of the process (called partial memory, $M_P$). Both values are retrieved using the Linux /proc psuedo-file system as a number of memory pages and are then multiplied by page size (4 kB in our case) and presented in kilobytes.

4. **Compilation time.** Time needed to build the application from the source code, including VCG code analysis and code generation phases in the case of dynamic applications ($T_C$). All values are in seconds.

5. **Execution time.** Time from the start of the application to the moment it terminates ($T_E$), measured in seconds.

During the tests a few situations were considered. First, several applications were tested with varying number of versatile functions and their share in total percentage of functions. Next, functions with different body size and arguments (meaning arguments type, number and size) were tested. After that the resource consumption of the tool itself was considered, including constant overhead it imposes on target applications. Finally, exemplary application was prepared in order
to assess the performance overhead in more typical real-life situations\textsuperscript{1}. Each presented value is an arithmetic mean of 8–10 separate results. All tests were conducted on a Intel\textsuperscript{©} Core\textsuperscript{TM} 2 Duo (P8400, 2.26 GHz) machine with 2 GB of RAM. Operating system was Linux with kernel 2.6.35-32-generic and included gcc/g++ with version 4.4.5.

4.1. Function number and size

We define 8 identical functions, each executed 1 000 000 times, and we declare different number of them as versatile (dynamic) each time. To reduce the influence of other factors, we use functions that are as simple as possible (single return statement computed from one argument), while ensuring that the compiler doesn’t take the optimization process too far. Since that approach is close to the worst-case scenario, we repeat the research with more complex functions (3 arguments, body containing some trigonometrical functions and a loop) to show more natural usage. Results for worst-case scenario are shown in Table 1 ($D_N$ means dynamic version of application with N functions out of 8 declared as versatile).

<table>
<thead>
<tr>
<th>Factor</th>
<th>Static $C_S$ [kB]</th>
<th>$D_0$ $C_S$ [kB]</th>
<th>$D_1$ $C_S$ [kB]</th>
<th>$D_2$ $C_S$ [kB]</th>
<th>$D_4$ $C_S$ [kB]</th>
<th>$D_8$ $C_S$ [kB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_T$</td>
<td>1.153</td>
<td>1.153</td>
<td>1.162</td>
<td>1.172</td>
<td>1.194</td>
<td>1.237</td>
</tr>
<tr>
<td>$C_E$</td>
<td>7.583</td>
<td>64.181</td>
<td>122.195</td>
<td>149.875</td>
<td>209.235</td>
<td>323.553</td>
</tr>
<tr>
<td>$T_E$</td>
<td>0.0115</td>
<td>0.693</td>
<td>1.480</td>
<td>1.719</td>
<td>2.220</td>
<td>3.181</td>
</tr>
</tbody>
</table>

Table 1. Performance factors in relation to number of versatile functions in worst-case scenario

Size of the source code remains almost the same all the time (because each function needs only a single keyword to declare it as versatile). Each function declared as versatile costs us (on average) 0.8 kB of intermediate code, 32 kB of executable code and 25 to 32 kB of memory. Increase in compilation time is mostly because the VCG tool is working outside of the original compiler, wasting time to process the code that will be processed again during compilation phase anyway.

\textsuperscript{1}With so many different performance factors and research cases we decided to present them in form of tables most of the time.
Figure 1. Execution time increase in relation to the function body size

The worst performance drop affects the execution time, which is probably most important factor. In the worst-case scenario the process of invoking versatile function through Versatile Code Module is very time-consuming compared to greatly optimized and simple function calls, slowing down the application hundreds or thousand of times. However, in the case of more complex functions (described above) the performance drop is reduced from over 140 to 14 (0 versatile functions) and from nearly 20 000 to 36 (all 8 functions defined as versatile).

To further research this relationship between performance and function complexity we performed another series of tests with varying function size, while their number remained constant. The basic “size of one” function consists of single \( \sin() \) call, while “size of 60” function repeats that same call 59 more times. Results for execution time are shown in Figure 1. We notice that both the static and dynamic line in this graph have the same slope, meaning that the tool induces no “absolute” overhead dependant on the size of the function. However, we can define a “relative” overhead \( O_R \) as the result of dividing function overhead time (difference between execution time of dynamic and static version) \( T_O \) by total function body execution time\(^1\) \( T_B \) and express it as a percentage:

\[ O_R = \frac{T_O}{T_B} \]

\(^1\)Execution times of considered functions are too short, so for each function we call it 10 000 times in a loop and measure total time of all calls.
\[ O_R = 100\% \cdot \frac{T_O}{T_B} \]  

(1)

In this case the initial “size of one” and final “size of 60” relative overheads are:

\[ O_{R_1} \approx 100\% \cdot \frac{0.028}{0.002} \approx 1400\% \quad O_{R_{60}} \approx 100\% \cdot \frac{0.028}{0.03} \approx 93\% \]  

(2)

These calculations show that the relative overhead was reduced 15 times, while the function size increased 60 times. This means that the overhead can be alleviated to acceptable values assuming the versatile functions are complex enough. While it is difficult to expect such large function body, there are other ways of increasing function complexity: use of loops, I/O operations or calling other functions. For brevity’s sake only execution time research was presented here, but other performance factors show similar characteristics.

### 4.2. Function arguments

While the overhead has no relation to the function size, it should depend on the function type signature, since the tool internally works by examining the function arguments and wrapping them in a single dynamic array. We consider two cases: (1) variable number of function arguments, while their type remains constant and (2) single function argument with variable size. Results of the research for the most important performance factors in both cases are presented in Tables 2a and 2b. All values in Table 2a are differences between static and dynamic versions.

It turns out that each additional argument of `double` type causes increase of about 0.17 kB in intermediate code and 0.35 kB in executable code, though the values vary according to optimization performed during the compilation process, which makes the analysis more difficult. Memory usage remained mostly constant, only once rising by one page (4 kB), meaning that each additional argument required less (presumably much less) than 0.33 kB of memory. Additional compilation time per argument is just few milliseconds and can be further reduced by making VCG tool a part of an existing compiler, eliminating the need to perform the syntax analysis twice and need for two separate processes. Not surprisingly, the most noticeable performance drop concerns the execution time, which rises by about 1.18 second with addition of each argument. However, considering that the
Table 2. Performance analysis results according to number and size of function arguments

(a) Different number of arguments

<table>
<thead>
<tr>
<th>Arguments Factor</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>9</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_E$ [kB]</td>
<td>114.61</td>
<td>114.61</td>
<td>114.61</td>
<td>118.60</td>
<td>118.60</td>
<td>118.60</td>
</tr>
<tr>
<td>$T_C$ [s]</td>
<td>1.411</td>
<td>1.415</td>
<td>1.416</td>
<td>1.420</td>
<td>1.452</td>
<td>1.437</td>
</tr>
<tr>
<td>$T_E$ [s]</td>
<td>24.89</td>
<td>25.90</td>
<td>28.29</td>
<td>30.62</td>
<td>34.13</td>
<td>37.85</td>
</tr>
</tbody>
</table>

(b) Execution time with different argument size

<table>
<thead>
<tr>
<th>Byte size</th>
<th>Compilation</th>
<th>4</th>
<th>16</th>
<th>48</th>
<th>72</th>
<th>128</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static [s]</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.174</td>
<td>0.235</td>
<td>0.423</td>
</tr>
</tbody>
</table>

application called that one function a total of 10 000 000 times, we can conclude that each call generated an overhead below 1 $\mu$s per argument.

In the case of different argument size we used a single C-style structure with different size (by adding int fields to the structure) and passed it to function by value. All other performance factors remained largely constant during the tests, so we focus on the execution time alone. We observe that execution time rises faster in the case of dynamic application compared to the static one. Once again, the optimization performed by the compiler makes things harder, but the execution time rise is still below one second, which is negligible when we take into account fact, that the function was called more than million times and that passing such large arguments by value is rare in practice. We conclude that argument size has little influence on the target system’s performance.

4.3. Tool performance and constant overhead

Next research concerns the constant overhead induced by the VCG tool itself. Every application designed with the tool has additional module responsible for versatile code management attached to it. This module needs to be initialized and
creates its own thread of execution (which blocks, waiting for the update events nearly all the time). To calculate the constant overhead we prepared an “empty” (i.e. containing only necessary measuring code and – in the case of dynamic application – dynamic module inclusion and initialization) application and compiled it in both static and dynamic way. We then repeated the process, but each time we added identical “blocks” of code to the application – each block consisting of several allocation and trigonometrical instructions. Since the researched overhead is presumably constant, it should be independent of the size of the entire application. The results are shown in Table 3, all values are differences between static and dynamic version of the same application.

Table 3. Results of constant overhead research

<table>
<thead>
<tr>
<th>Blocks Factor</th>
<th>0</th>
<th>5</th>
<th>25</th>
<th>125</th>
<th>625</th>
<th>2500</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$C_S$ [kB]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$C_T$ [kB]</td>
<td>5.617</td>
<td>5.617</td>
<td>5.617</td>
<td>5.618</td>
<td>5.617</td>
<td>5.617</td>
</tr>
<tr>
<td>$C_E$ [kB]</td>
<td>52.599</td>
<td>56.598</td>
<td>52.598</td>
<td>52.598</td>
<td>52.598</td>
<td>56.511</td>
</tr>
<tr>
<td>$M_T$ [kB]</td>
<td>8 364</td>
<td>8 360</td>
<td>8 360</td>
<td>8 360</td>
<td>8 356</td>
<td>8 360</td>
</tr>
<tr>
<td>$T_C$ [s]</td>
<td>0.530</td>
<td>0.529</td>
<td>0.529</td>
<td>0.528</td>
<td>0.527</td>
<td>0.586</td>
</tr>
<tr>
<td>$T_E$ [s]</td>
<td>0.0016</td>
<td>0.0018</td>
<td>0.0016</td>
<td>0.0020</td>
<td>0.0018</td>
<td>0.0033</td>
</tr>
</tbody>
</table>

Results show that despite the application growing considerably in size (from using 3 MB of RAM and 30 lines of source code in the beginning to over 60 MB and 40,000 code lines in the end), the difference remains constant. Source code is completely unchanged, since we don’t define any versatile functions. Intermediate and executable code size needed to attach the dynamic module is around 5.6 kB and 55 kB respectively. Compilation time differs by approximately half a second and can sometimes get lower, as the VCG tool is less complex than standard gcc compiler and may parse files at faster rate (provided there are no versatile functions defined). Time needed to initialize the dynamic module is few milliseconds at most. Most considerable performance issue is the rise in memory usage of about 8.3 MB, which can prove to be a problem in small programs.

Having constant overhead researched, we study the resource consumption of the tool itself. As mentioned earlier, the VCG tool was meant as a lightweight so-

---

1The memory measurement includes shared memory, which can be used by many programs, but should be counted only once. Because of that, the actual memory usage may be lower than what was measured here.
olution. The total size of all programs it consists of is about 130 kB (95 kB for code analyser and 35 kB for library generator, monitor and dynamic code module). We also researched memory usage of the code analyser while it works and compared it to memory used by gcc when processing the same source code. Results are shown in Table 4. “2500 blocks” means the largest application used with constant overhead research above, while “Exemplary” means application that is described in next subsection.

Table 4. Comparison of memory usage of the tool and gcc during work

<table>
<thead>
<tr>
<th>Application</th>
<th>Tool</th>
<th>Total memory usage ($M_T$) [kB]</th>
<th>Partial memory usage ($M_P$) [kB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500 blocks</td>
<td>gcc</td>
<td>5232</td>
<td>1060</td>
</tr>
<tr>
<td>2500 blocks</td>
<td>VCG</td>
<td>4196</td>
<td>2452</td>
</tr>
<tr>
<td>Exemplary</td>
<td>gcc</td>
<td>5232</td>
<td>1060</td>
</tr>
<tr>
<td>Exemplary</td>
<td>VCG</td>
<td>3100</td>
<td>1324</td>
</tr>
</tbody>
</table>

Results are different depending on the memory usage we consider. In total memory usage the VCG tool is considerably less memory-consuming than gcc, but when we look at data, stack, text and shared memory (“partial memory”), VCG may need over 2 times the memory required by the gcc. More complex applications should be tested to get more clear understanding of this matter, but we will stop here and conclude that VCG memory usage is comparable to gcc on average.

4.4. Exemplary application

Until now, the overhead we researched was considerable, especially in the case of execution time. To prove that this overhead would be greatly reduced in typical situation, we conducted one more research using exemplary application designed specifically for this test. The application works by reading supplied file and performing some computations according to instructions found in the file. We implemented functions for calculating variance and Fibonacci numbers, finding shortest path in a graph (using Dijkstra algorithm) and sorting numbers (using simple bubble sort implementation). Moreover, a logging function was added to write results to the output log file. Aside from this, some additional functions were defined (calculating arithmetical mean, parsing input file, operations on strings, etc.). Three functions, namely logging, sorting and pathfinding, were defined as versatile, since they are most likely the ones to change (for example: using different output format,
changing bubble sort to quick sort or Dijsktra to A*). To measure the overhead, the application was once more built and executed with both standard gcc compiler and VCG tool. Results are shown in Table 5.

Table 5. Performance overhead in the case of the exemplary application

<table>
<thead>
<tr>
<th>Factor</th>
<th>Static (gcc)</th>
<th>Dynamic (VCG)</th>
<th>Total overhead [%]</th>
<th>Non-constant overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_S$ [kB]</td>
<td>5.208</td>
<td>5.240</td>
<td>0.6%</td>
<td>0.6%</td>
</tr>
<tr>
<td>$C_I$ [kB]</td>
<td>5.208</td>
<td>15.101</td>
<td>190%</td>
<td>82%</td>
</tr>
<tr>
<td>$C_E$ [kB]</td>
<td>41.129</td>
<td>335.459</td>
<td>716%</td>
<td>586%</td>
</tr>
<tr>
<td>$M_T$ [kB]</td>
<td>3056</td>
<td>11612</td>
<td>280%</td>
<td>6%</td>
</tr>
<tr>
<td>$M_P$ [kB]</td>
<td>1368</td>
<td>9832</td>
<td>620%</td>
<td>19%</td>
</tr>
<tr>
<td>$T_C$ [s]</td>
<td>1.237</td>
<td>4.515</td>
<td>265%</td>
<td>222%</td>
</tr>
<tr>
<td>$T_E$ [s]</td>
<td>0.866</td>
<td>0.958</td>
<td>10.6%</td>
<td>10.6%</td>
</tr>
</tbody>
</table>

Source code overhead is negligible. The rise in the intermediate code is considerable, but this code is only temporary. Moreover, half of the intermediate code increase is due to the constant overhead. Execution time increase, which was the source of the greatest performance drop so far, has been reduced to only slightly over 10%. This is due to the fact that versatile functions are used in more favouring conditions, though they are still called several thousand times. Memory usage overhead seems to be very high, but it is once again result of the constant overhead researched before. Actual memory rise due to the presence of few versatile functions is below 200 kB.

Other performance factors can cause problems however. Execution code size increase is huge, even if we consider the constant overhead, and will only continue to grow with higher number of versatile functions. Lastly, it takes much more time to compile the application than it was taking before. This is not much of a problem to the end-user, but can prove to be a serious issue for the system developers.

5. Conclusions

In this paper we discussed a few applications of runtime code modification, namely: (1) high-availability systems, (2) simulation and modelling software, and (3) programmable online video games. We also proposed the Versatile Code Generator prototype lightweight programming tool as a possible solution. The tool is able to swap versatile code during runtime and supports automated return to older
function versions in case of system crash. However, the main purpose of this paper was to research and evaluate the negative performance overhead imposed on the target system by the runtime code modification mechanisms in different situations and considering several performance factors.

Presented research show that the tool itself consumes little resources and only few small changes to the source code are required, emphasizing the tool’s easiness of use. Different function size as well as number and size of arguments were considered, showing that main drawback of the tool is execution time overhead which can be huge in some cases. However, the research also prove that this overhead is easily controlled and greatly alleviated when used in more typical situations. The constant overhead imposed on every application regardless of its size can be an issue in smaller programs, but becomes negligible in the case of larger applications.

To sum it up, presented research show that despite the variety of approaches to runtime software adaptation our understanding of its effects on the target systems is still incomplete, as most of these solutions only research the positive effects without much consideration to the performance overhead. In the end, more research must be conducted in order to improve our understanding of risks and trade-offs involved with use of runtime software adaptation.

References


