Using GDB and QEMU for cross debugging through automatic exchange of configuration parameters.

Rita de Cássia Cazu Soldi, Antônio Augusto Medeiros Fröhlich
Laboratory for Software and Hardware Integration (LISHA)
Computer Science Department (INE)
Federal University of Santa Catarina(UFSC)
Email: {rita,guto}@lisha.ufsc.br

***Abstract*—Debugging is one of the most time-consuming process in software development and it tends to be even more complicated as it adds complexity and constraints. The debugging task for embedded systems brings some additional problems in the process, since the developer should perform it in different platforms, which vary accordingly to the architecture and vendor. In order to facilitate embedded system development, this paper presents the automatic exchange of configuration parameters as one anatomic part of fully automated debug and also show how to construct an environment for cross debugging embedded applications based on specific hardware/software requirements.**

I. Introduction

Whenever a developer starts to write the source for a new software, there are some specifications and behaviors that it should contain. During this process, if any of these features were not working as specified, the developer must start to debug the code until find and correct this abnormal behavior. Debugging is one of the most challenging and time-consuming process of developing a software, since finding the reason for this unexpected behavior and fixing the problem is a non-trivial process.

Debug process becomes more challenging as long as systems are more constraints and complex. Unlike general purpose systems that are designed to be flexible and perform various applications for end-user needs, embedded systems are usually designed with minimum resources to perform a specific task. Besides a developer of applications for embedded systems should carefully use the scarce resources, different platforms must be used depending on debug tool, operating system, architecture and vendors, which makes both coding and testing of these systems more defiant.

There are several techniques for performing debugging of an application, such as: comparison between versions, analyze the trace of error, use state graphs to identify illegal operations, etc. It is important to note that, although each of these techniques work with a different degree of automation, the more automated debugger is also the one that need more initial information about the error to be resolved. Thereby, a fully debug automation requires a previous knowledge of all possible scenarios that could result in error.

Some activities such as checking if a value of a particular variable is the cause of the error are examples of automation tests whose performance requires a lot of information from the tested application. This exchange not only depends on the programming language used, but also on the type of this variable, target platform,etc. Although it seems to be trivial when a developer makes this changes, transmit this knowledge to a computer is a very laborious task and sometimes even more time-consuming than debugging itself.

There are two ways for building an automated debug process: (i) the developer must be able to transfer all his knowledge to the tool or (ii) the tool must be able to get information directly from the application. The former is the most used, since the development of this type of tool is less complex than a fully automated tool and still can help developers finding bugs in applications. Although the latter option is not widely used, it would be the real solution for faster debugging and also to allow the developer to carry out parallel activities while the code is tested.

The full test automation in applications is a very complex activity and to increase its practicability it can be splitted into smaller parts until get atomic operations. So its possible to see which parts of the process can be implemented to maximize the automation with the least possible complexity.

The main idea of this paper is to present the automation of one of these atomic parts from the process of debugging, the automatic exchange of parameters. By implementing this automation, we aim use computer to optimize the debugging software process, but without forgetting to supply logs for developers, since they can benefit for getting a full report of all the trials and results found.

In summary, by solving an anatomic operation of debug we make the following contributions:

- Development environment for embedded applications, configurable according to specific hardware/software requirements. We show how it is possible to create an environment for development and testing embedded applications using GDB to cross debug the code and Qemu to simulate its execution.
- Automate the operation of finding errors with a real world embedded system debug. In this case study

developers can run a script to automatic find errors and use a report to fix the code.

In this Section, we introduced the debugging challenges and why its an important area for research. The rest of this paper is organized as follows. Section II presents the related work in the area. Section III contain the details of the integrated GDB and QEMU environment for develop embedded applications. Section IV presents the anatomic solution of exchange parameter's configuration applied in a study case, Section V show some results and finally Section VI concludes the paper.

II. Related Work

This section presents some related works with different techniques to automated debug, such as statistical debugging, program slicing, delta debug and capture/replay. Zhang et al [1] focus on reduce the search space with statical debugging. These techniques usually use features statistically related to failure to reduce the set of solutions that must be checked by the developer. It needs a large data set to accomplish this statistic, in order words, its necessary to have a big data storage to keep information of thousands times execution of the code. Despite the minimized search space, statical debug returns some locations spread in the code, which does not facilitate the developer's work.

In program slicing the main idea is to divide the code into different parts, until you can remove most of the paths that do not lead to the set of error checking [2]. This technique is interesting because it needs only one error path to simplify the set of inputs to be studied. However, even with this reduction, the final set can still contain many paths that do not lead to the error, so programmer must check all of them.

Delta debug and static slicing were combined for speed up the minimization of path leading to the error [3]. The main idea is to isolate all changes that cause the failure and simplify the input until get only paths that delivery to the error. Static slices reduce the speed of convergence to this simplified input by reducing the search space for a failure trace and, consequentially, make debug easier. One problem with this technique is the necessity of a original trace to identify the error, because is not always possible to have a bug-free trace.

In capture and replay the program must be executed until the end and all operations are stored in a log. Burger and Zeller developed a Jinsi tool that can capture and replay interactions between inter/intra-components [4]. So all relevant operations are observed and run step by step, considering all communications between two components, until find the bug. Besides being the most widely used, this kind of technique need to perform all possible paths from one object to another, making this technique time consuming.

III. Integrated Environment

This section presents details of debugging, simulation and how to integrate both in order to create a better environment do develop embedded applications.

Regardless of the technique to be used, debugging can be performed locally or remotely. Local debug is when the application runs in the same machine as debugger, so the process has lower latency, but big influence between both, for example, if a process causes a crash, debugger can only discovery what happened by halting or restarting.

This influence does not happen in remote debug, once application and debugger run in separate machines and the process occurs on an isolated box over a network connection. Despite of having some latency problem, from the debugging environment point of view, its like a local debug with two screens connected into only one system.

In order to provide the most number of possibilities to the developer, the emulator used to debug applications must provide both ways to perform this activity. QEMU is a generic and open source machine emulator and virtualizer [5]. When used as a machine emulator its possible to run applications made for one machine to another by using dynamic translation. The decision of use QEMU emulator was based on its active community, support of Linux as host machine, a native set of target machines and the possibility to integrate a new machine.

Thus, besides having QEMU to emulate applications, to perform a really useful debug, developer must think about others concepts involved in debugging, such as, how to configure the execution mode of the code, observe the outputs of the application, watch some environment's variables, log the tasks performed and others. This requires a good ally to see what steps the program was executing an moment before a crash or to specify anything that might affect its behavior. When debugging an application with GBD - *the GNU Project Debugger* it is possible to see inside the application while it executes [6].

One important characteristic of GBD is to enable remote debug because this way we can run the program on a given embedded platform and same time debug it with GDB. In remote debugging, GDB connects to a remote system over a network and then can control the execution of the program and retrieve information about its state.

The integration of both is particular for each machine host and target, therefore, maybe some steps presented here must be tailored depending on your target architecture. Figure 1 presents the activities required to perform remote debugging using IA-32 architecture and these steps and additional explanation of what techniques and tools used in this process are listed bellow:

1) **Compile with debug information** is the first and the most important step. As input is necessary the source code and the compiled application that has debug

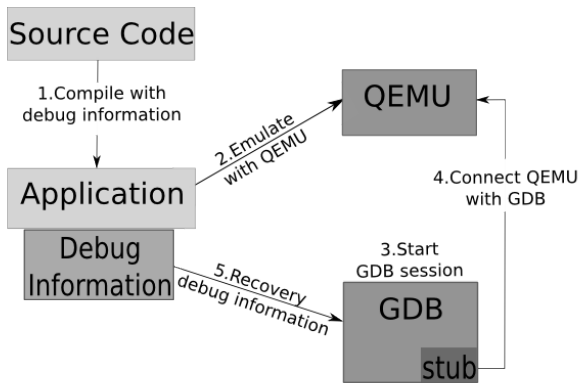


Figure 1. Steps to integrate QEMU and GDB

information its the expected output. The one who is using gcc (*GNU project C and C++ compiler*) can perform this activity simply by using *-g* option to compile.

2) **Emulate with QEMU** is a necessary step to execute the application in the correct target architecture. To perform this step, developer must initiate QEMU with *-s -S* options. The first option enables the GDB stub, in order to open communication between QEMU and GDB. The *-S* option its used to force QEMU to wait GDB to connect after the system restart.
   Fox example, if the one has compiled an application with debug information (*app.img*), that prints something in screen (*stdio*), QEMU call should look like
   ```
   qemu -fda app.img -serial stdio -s
   -S
   ```
3) **Connect with GDB** starts with a GDB session, that must be initialized in an separate window. Then, to connect GDB in QEMU the developer must explicitate that the target to be examined is remote and inform the host address and port of this target (in this case, QEMU). When host is in the same machine as GDB, its possible inform only the port, but the complete line must be something like
   ```
   target remote [host]:[port]
   ```
4) **Recovery debug information** is an important step to help developers to find errors, once its possible to use autocomplete to recovery the all name contained in the symbols table. The file used to keep debug information (as the path) must be informed to GDB using the command
   ```
   file [path_to_the_file]
   ```
5) **Finding errors** is an activity that depends on the program to be debugged. From this step, the developer can set breakpoints, watchpoints, control the execution of the program and even enable logs. More information about command set can be found in GDB's page [6].

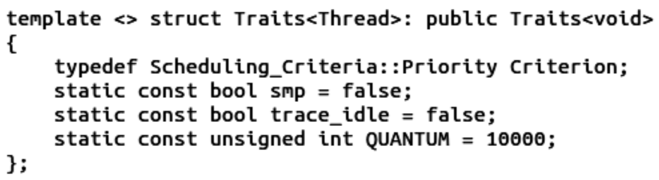
IV. The Automatic Exchange of Configuration Parameters

The automatic exchange of configuration parameters is part of an effort to a punctual implementation of each party involved in an atomic tool for complete automation of testing. By using a real-world application its possible to emphasize that even though a small contribution, if compared with the ultimate goal, this solution its already a useful tool to help the debugging process.

The real-world application is based on the development of software in embedded systems using EPOS (*Embedded Parallel Operating System*) [7], a component-based framework that provides all traditional abstractions of operating systems and services like memory management, communication and time management.

A. Exchange of Configuration Parameters in EPOS

EPOS uses generic programming techniques so each abstraction can be configured as desired. Traits are parameterized classes that describe the properties of a given object/algorithm. Figure 2 shows a piece of traits classes used by EPOS, where a set of static members describe some definitions used by abstractions.



```
template <> struct Traits<Thread>: public Traits<void>
{
    typedef Scheduling_Criteria::Priority Criterion;
    static const bool smp = false;
    static const bool trace_idle = false;
    static const unsigned int QUANTUM = 10000;
};
```

Figure 2. Set of definitions from a traits class in EPOS

By definition, EPOS is instantiated only with the support needed for their dedicated application, it is important to remember that an individual member of a trait is a characteristic of the system and all features of a component must be set appropriately for a better performance of the system. In this context, the automated exchange of these parameters can be used both to discovery a failure in the program by an wrong characterization of components, or to improve the performance for the application by selecting a better configuration.

Figure 3 shows the overview of how the automatic exchange of configurations parameters is performed. Basically a member of traits is selected, its definition is changed according to the specification and then application is recompiled. Traces generated by each version of application are compared and reported to developer. Before a new cycle is complete, the integrated test environment is used to verify the application execution. The performance is analyzed and

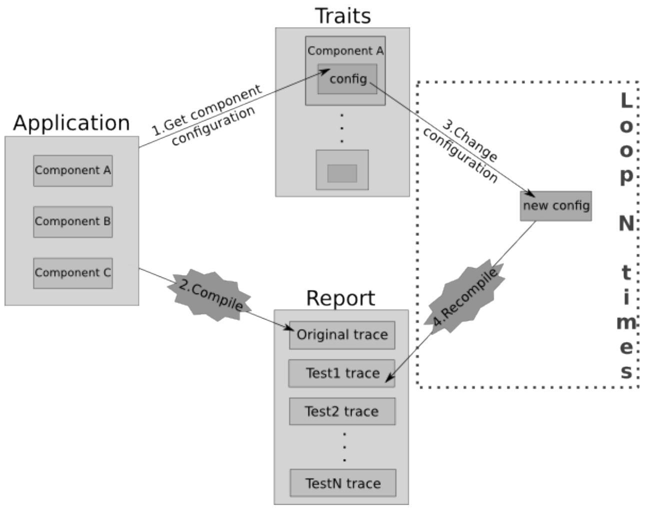


Figure 3. Overview of automatic exchange of configuration parameters

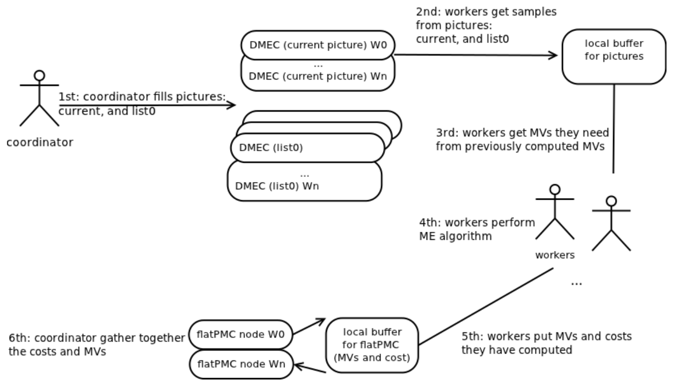

coordinator
1st: coordinator fills pictures: current, and list0
DMEC (current picture) W0
DMEC (current picture) Wn
2nd: workers get samples from pictures: current, and list0
local buffer for pictures
DMEC (list0)
DMEC (list0) Wn
3rd: workers get MVs they need from previously computed MVs
4th: workers perform ME algorithm
workers
5th: workers put MVs and costs they have computed
local buffer for flatPMC (MVs and cost)
flatPMC node W0
flatPMC node Wn
6th: coordinator gather together the costs and MVs

Figure 4. Interaction between `Coordinator` and `Workers` threads [11]

compared with other versions of the application, which also generates a report of the execution.

In the current version of the script, a configuration is selected and its parameter is modified randomly, but can also be supplemented with an artificial intelligence tool or some Application-Oriented System Design tool to provide all information for the script. There is already an tool [8] being used with EPOS that can generate all necessary computational support and configuration from the source code of the application. However, the tool is not yet attached to the script because the main idea of this paper is to solve this first atomic part of the automated debug process with the minimum complexity and maximum flexibility possible.

B. Real-World Application

The automation script exchange parameters was used to test the Distributed Motion Estimation Component (DMEC). This component performs a motion estimation that exploits the similarity between adjacent images in a video sequence, which allows images to be coded differentially, increasing the compression ratio of the generated bitstream [9]. Motion Estimation is an significant stage for H.264 encoding, since it consumes around 90% of the total time of the encoding process [10].

DMEC's test check the performance of motion estimation using a data partitioning strategy. This estimate is made by `Workers` threads and the result is processed by the `Coordinator` thread [11].

Figure 4 presents the interaction between the threads. The `Coordinator` is responsible for defining the partitioning of picture, provide the image to be processed and return results generated to encoder, while `Workers` must calculate motion cost and motion vectors.

The Distributed Motion Estimation Component was tested using the integrated environment demonstrated in the section III. Despite the first part of the script generates multiple configurations, only compile the code does not guarantee that the application is bug free. Figures 5 and 6 show the DMEC execution using values 6 and 60 for `NUM_WORKERS` configuration.

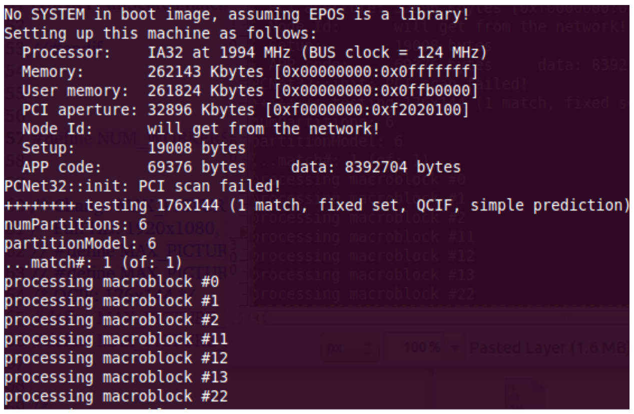

```
No SYSTEM in boot image, assuming EPOS is a library!
Setting up this machine as follows:
  Processor:    IA32 at 1994 MHz (BUS clock = 124 MHz)
  Memory:       262143 Kbytes [0x00000000:0x0fffffff]
  User memory:  261824 Kbytes [0x00000000:0x0ffb0000]
  PCI aperture: 32896 Kbytes [0xf0000000:0xf2020100]
  Node Id:      will get from the network!
  Setup:        19008 bytes
  APP code:     69376 bytes     data: 8392704 bytes
PCNet32::init: PCI scan failed!
++++++++ testing 176x144 (1 match, fixed set, QCIF, simple prediction)
numPartitions: 6
partitionModel: 6
...match#: 1 (of: 1)
processing macroblock #0
processing macroblock #1
processing macroblock #2
processing macroblock #11
processing macroblock #12
processing macroblock #13
processing macroblock #22
```

Figure 5. DMEC emulated execution - `NUM_WORKERS` = 6

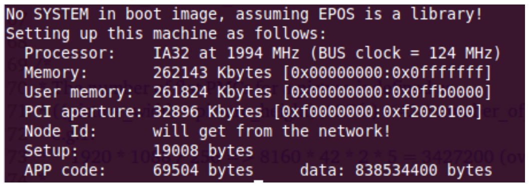
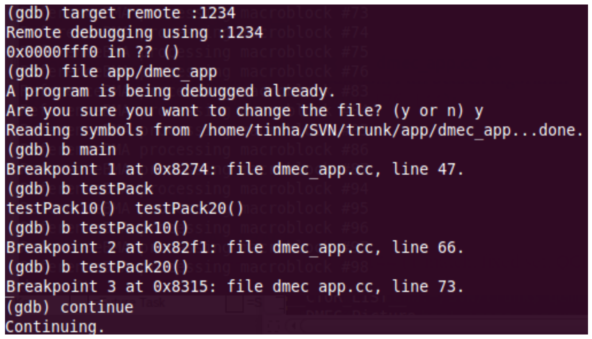

```
No SYSTEM in boot image, assuming EPOS is a library!
Setting up this machine as follows:
  Processor:    IA32 at 1994 MHz (BUS clock = 124 MHz)
  Memory:       262143 Kbytes [0x00000000:0x0fffffff]
  User memory:  261824 Kbytes [0x00000000:0x0ffb0000]
  PCI aperture: 32896 Kbytes [0xf0000000:0xf2020100]
  Node Id:      will get from the network!
  Setup:        19008 bytes
  APP code:     69504 bytes     data: 838534400 bytes
```

Figure 6. DMEC emulated execution - `NUM_WORKERS` = 60

The biggest difference between the two figures is that after retrieving the information from the application, QEMU has a response only for the six `workers` configuration.

Is part of the automation script to use GDB for debugging all configurations that could be compiled. This process was crucial to determine the error in DMEC case. In this sense, some breakpoints were added to all functions, specially main, according to Figure 7. Its possible to check that "continuing" is the last line that appears in the execution, that fails because a high value is defined for the number of threads.



```
(gdb) target remote :1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb) file app/dmec_app
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from /home/tinha/SVN/trunk/app/dmec_app...done.
(gdb) b main
Breakpoint 1 at 0x8274: file dmec_app.cc, line 47.
(gdb) b testPack
testPack10()  testPack20()
(gdb) b testPack10()
Breakpoint 2 at 0x82f1: file dmec_app.cc, line 66.
(gdb) b testPack20()
Breakpoint 3 at 0x8315: file dmec app.cc, line 73.
(gdb) continue
Continuing.
```

Figure 7. DMEC debug with GDB execution - `NUM_WORKERS` = 60

Through the second part of the script, the debug, was possible to verify that the program can not even reach the main function, which means that now the script must change configurations before the main call.

V. Evaluation

Tests were performed with the chosen application running under EPOS 1.1 and compiled with GNU 4.5.2 for IA32 architecture. The integrated environment is composed by GDB 7.2 and QEMU 0.14.0. For evaluation were collected data from totally random and partially random tests.

Totally random test is the one that has no prior information on the application. That is, any configuration within `traits` can change, including parameters that not influence the application.

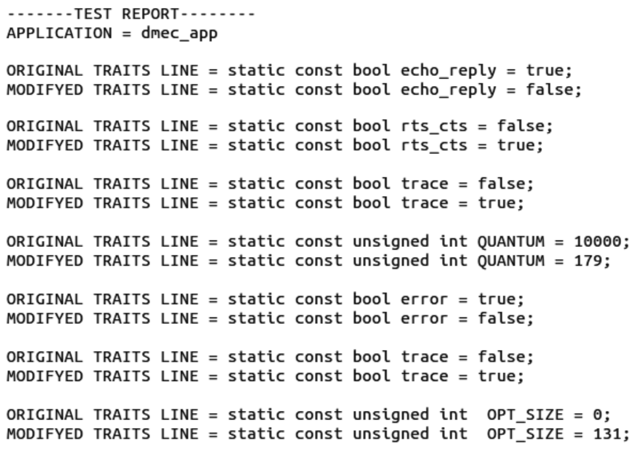
In this case, test was run hundred times, generating 85 different configurations in which 23% of them could be correctly compiled, but less than 5% of configurations were relevant to DMEC. Figure 8 presents a piece of report with some generated configuration.

The execution of this versions showed that application did not change significantly, since most part of exchanged parameters not influence the application.

On the other hand, a partially random test has some tips about application, such as relevant settings and valid configurations. In other words, the script changes only parameters that directly influences the application.

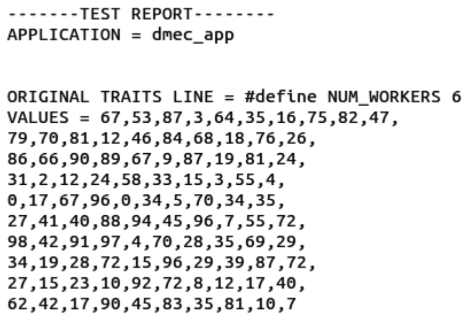
This second test was concentrated in only one configuration. Thus it was possible to focus on just one parameter to try to find the best option for the application. Figure 9 presents a report with all tries.

In this case, the test got 69 different configurations (same number of tries) and all of them could be correctly compiled



```
-------TEST REPORT--------
APPLICATION = dmec_app

ORIGINAL TRAITS LINE = static const bool echo_reply = true;
MODIFYED TRAITS LINE = static const bool echo_reply = false;

ORIGINAL TRAITS LINE = static const bool rts_cts = false;
MODIFYED TRAITS LINE = static const bool rts_cts = true;

ORIGINAL TRAITS LINE = static const bool trace = false;
MODIFYED TRAITS LINE = static const bool trace = true;

ORIGINAL TRAITS LINE = static const unsigned int QUANTUM = 10000;
MODIFYED TRAITS LINE = static const unsigned int QUANTUM = 179;

ORIGINAL TRAITS LINE = static const bool error = true;
MODIFYED TRAITS LINE = static const bool error = false;

ORIGINAL TRAITS LINE = static const bool trace = false;
MODIFYED TRAITS LINE = static const bool trace = true;

ORIGINAL TRAITS LINE = static const unsigned int  OPT_SIZE = 0;
MODIFYED TRAITS LINE = static const unsigned int  OPT_SIZE = 131;
```

Figure 8. Totally random generated configurations



```
-------TEST REPORT--------
APPLICATION = dmec_app


ORIGINAL TRAITS LINE = #define NUM_WORKERS 6
VALUES = 67,53,87,3,64,35,16,75,82,47,
79,70,81,12,46,84,68,18,76,26,
86,66,90,89,67,9,87,19,81,24,
31,2,12,24,58,33,15,3,55,4,
0,17,67,96,0,34,5,70,34,35,
27,41,40,88,94,45,96,7,55,72,
98,42,91,97,4,70,28,35,69,29,
34,19,28,72,15,96,29,39,87,72,
27,15,23,10,92,72,8,12,17,40,
62,42,17,90,45,83,35,81,10,7
```

Figure 9. Partially random generated configurations

and only 8 could be executed, in other words, less than 10% of tries could be used as configuration.

Also, to use the integrated debug environment it was necessary to build the code with a special setup, in which it is possible to generate information about the application to be tested.

Without this information the original DMEC image consumes more than 50kB, but with the generation of debug symbols the new image consumes about 70 kB, in other words, it increases in 80% the cost in terms of memory to debug DMEC application.

VI. Conclusion

In this paper, we show how to construct a development environment for embedded applications based on specific hardware/software requirements and introduce the automatic exchange of configuration parameters as one anatomic part of fully automated debug.

The integrated development environment provides independence of the physical target platform for development and test. Its an important step, since some embedded systems may not be able to store the extra data needed to support debug. The impact of enable debug information in code size and in the execution time of the real-world application was more than 80%. Also, developers no longer need to spend time understanding a new development platform whenever some characteristic of the embedded system changes.

The automatic exchange was evaluated using two kinds of test. The fully automated test works with no prior information of the application, but it was possible to generate valid configurations, that could be tested as alternative solutions. In partial automated test all generated configurations were valid and the report was useful to discovery that some parameter values were better then others.

In this sense, was possible to realize that even a small part of the complete automated solution produce answers to help developers find and fix a bug. With only a hundred tries was possible to find error/restriction in the code. Thus, as future work we can integrate the automatic exchange script with a tool that has artificial intelligence, in order to achieve a conscious exchange of type parameters.

References

[1] Z. Zhang, W. Chan, T. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 43–52.

[2] N. Sasirekha, A. Robert, and D. Hemalatha, "Program slicing techniques and its applications," *Arxiv preprint arXiv:1108.1352*, 2011.

[3] Y. Lei and J. Andrews, "Minimization of randomized unit test cases," in *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*. IEEE, 2005, pp. 10–pp.

[4] M. Burger and A. Zeller, "Replaying and isolating failing multi-object interactions," in *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*. ACM, 2008, pp. 71–77.

[5] (2011, Nov.) About qemu. [Online]. Available: http://www.qemu.org

[6] (2011, Nov.) Gdb: The gnu project debugger. [Online]. Available: http://www.gnu.org/s/gdb/

[7] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[8] R. Cancian, M. Stemmer, A. Schulter, and A. Frölich, "Ferramenta de suporte ao projeto automatizado de sistemas computacionais embarcados."

[9] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the h.264/avc video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, no. 7, pp. 560 –576, july 2003.

[10] X. Li, E. Li, and Y.-K. Chen, "Fast multi-frame motion estimation algorithm with adaptive search strategies in h.264," in *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, vol. 3, may 2004, pp. iii – 369–72 vol.3.

[11] M. Ludwich and A. Frohlich, "Interfacing hardware devices to embedded java," in *Computing System Engineering (SBESC), 2011 Brazilian Symposium on*, nov. 2011, pp. 176 –181.