AEP - Automatic Exchange of Embedded System Software Parameters.

Rita de Cássia Cazu Soldi and Antônio Augusto Medeiros Fröhlich
Federal University of Santa Catarina (UFSC)
Laboratory for Software and Hardware Integration (LISHA)
Florianópolis, SC, Brazil
rita@lisha.ufsc.br
guto@lisha.ufsc.br

***Abstract*—Embedded systems are present in most of humans' daily activities and they are so widespread that seem to be part of the natural process. Each of this system has a different task, but there are recent efforts to join them in a large and complex network in order to increase the welfare of the population.**

To ensure that there will not be suggested harmful actions to humans, it requires a good check before deploying this complex network. However, testing embedded systems is a marathon, since the developer performs the process repeatedly on different platforms, which vary according to the architecture, languages and vendors.

This paper presents AEP, a tool for Automatic exchange of embedded system software parameters. This tool can withdraw all configuration required to proceed test from a XML file, then it uses cross debug and emulation to automatic run tests and offer a full report of all tries to developers.

AEP was evaluated using a real world problem in terms of valid information in report and memory consumption. The obtained results indicates that even with no previous information this tool can produce helpful answers for developers to find and fix bugs.

I. Introduction

An embedded system can be presented as a combination of software and hardware designed to perform a specific task. These systems are widely attached to numerous electronic devices, and their activities are becoming more popular and intrinsic to daily life [6]. These systems are a key technology in the path toward the smart world since they can collect data and perform computations about the environment in which they are inserted.

Applications involving environmental monitoring and analysis, intelligent cities, and precision farming are only a sample of a set of possible applications. But with the advent of smart things and internet of things, it became even more interesting to increase communications and the computing power in these systems, enabling intelligent behavior to exchange data with other systems through a large network [11].

The motivation for build an intelligent world is to produce a greater comfort to human beings, however, the realization of this idea should be based on the certainty that each component of this network is performing its activities properly. When a software deviates from the expected/specified behaviour it can result in disasters involving loss of market share, client information, people time, etc. [20]. Therefore, it is essential that tests be thoroughly performed to ensure that these smart things are properly performing the activities assigned to them.

The debugging task consumes a considerable time and brings many challenges since it requires a thorough inspection of the source code making it a non-trivial process [13].Coding and testing embedded systems is very defiant, since developers need to find out how to optimize the use of the scarce resources and the platform depends on operating systems, architecture, vendors, debugging tool, etc [17]. This makes embedded systems more susceptible to errors as well as specification failures.

There are several approaches for debugging applications and each one has a different degree of automation, nevertheless, the more automated is the process more information about the application it needs.

Most of the debug tools is partially automated and requires some interaction with the developer to make decisions during testing [5], [21]. These tools save some development time, but not as much as total automation tools. The automation of the entire testing process without any human intervention is still a challenge to researchers, although there are some studies that can automate part of the process with data taken directly from the application [9], [23].

In this paper, we propose the automation of one part of the debugging process for embedded systems' application, the automatic exchange of configuration parameters (AEP). In this proposal, there is a shell script responsible for exchanging configuration parameters according to an XML file. Also, a solution to the problem of setting up a stable environment for testing embedded systems is also part of the contribution.

The rest of this paper is organized as follows. Section II presents the related work. Section III contain the details of the integrated GDB and QEMU environment for debugging embedded applications. Section IV presents the solution of exchange parameter's configuration applied in a study case. Section V shows the results and finally Section VI concludes the paper.

II. Related Work

The automated testing area has a vast literature that inspired this proposal. There are several approaches for debugging general purposing systems, and based on these works we created the parameters exchange script and the test environment for debugging embedded systems applications as a first step to achieve a fully automated tool.

The work presented by Seo et al. [18] proposes an interface technique that identify and classify interfaces between embedded system layers. They created a model based tool that generates and executes test cases to analyse these interface layers. Furthermore, they proposed the emulation test technique that integrates monitoring and debugging embedded systems. Despite the similarity, since we also emulate the target board environment and monitor the behaviour of environment variables, Seo et al. focus on testing interfaces and layers, while this proposal addresses the testing of components and their integration.

ATEMES [8] is a tool for automatic random tests, that includes coverage testing, unit testing, performance testing and race condition testing. ATEMES supports instrumentation of source code, generation of tests cases and generation of primitive input data for multi-core embedded software. This system is similar to ours since we also automatically run random tests under cross-testing environment to support embedded software testing. However, the idea of this work is to integrate the exchange parameters directly in the operating system, so it is possible not only test the application as well as optimize the choice of the configuration parameters.

Statistical Debugging techniques [25], [24], [14] are capable of isolating a bug by automatic running an application several times and using generated statistical data to analyse these executions information. As a result of this analysis it is possible to pinpointing a suspiciousness ranking, that reduces the bugs' search area. This technique could be only incorporated into an embedded system by using cross-debug since it needs large data set to accomplish this statistic and the necessity to save information of all executions. Although the ranking pointing out possible errors is a breakthrough in the developer's work, its incorporation on a debugging environment requires a machine with more storage and processing than is not available on an embedded system.

In program slicing [16], [22], [2] the main idea is to divide the code into different parts, testing and removing paths that do not lead to errors. This technique has two approaches for reducing the path that lead to error: static slicing and dynamic slicing. Static slicing has faster reduction of application path. However, it does not consider the initial entry of the program and then the final set of paths leading to the error are an approximation of the real set. In dynamic slicing, the initial entry has a great influence on how the slices are performed, allowing a greater precision for final errors path. This technique is interesting because it needs only one error path to simplify the set of inputs to be examined. This present proposal was designed to support both types of slicing through configuration files (traits) that can address the whole system or just a part of the application.

In capture and replay [4], [15], [12] the program is executed until it reaches the end and all operations performed are stored in a log. Burger and Zeller developed a JINSI tool that can capture and replay interactions between inter/intracomponent. 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 technique is time consuming when it needs to perform all possible paths from one object to another. From this technique, our work absorbs the idea of debug focusing on the components that compose the application. We use a highly configurable operating system, on which can be plugged in a single component with different implementations, so a developer can verify the difference between them.

III. Embedded Systems Debugging Environment

This section presents details of the debugging process, simulation and how to integrate both in order to create a better environment for developing and testing embedded applications.

Regardless of the technique to be used, debugging can be achieved locally or remotely. Local debugging is when the application runs on the same machine as the debugger. As a result, application and debugger have a lower communication latency. However, the application interferes in the debug process, e.g. if the application under test crashes, the debugger will need to halt or restart to seek the cause.

This influence does not happen in remote debug, once application and debugger run in separate machines. The tests are performed into an isolated box over a network connection. Despite of having some latency issues, from the debugging point of view, the rest of the process can be viewed as a local debug with two screens connected in only one system.

In order to provide the most number of possibilities for the developer, the emulator used to debug applications must provide both ways to perform this activity. Also, for a useful debug, developers must consider others concepts involved in debugging, such as, how to configure the code execution mode, to observe the application outputs, watch some environment's variables, log the tasks performed and others configurations. This requires a good ally to follow program steps and analyse executing state a moment before a crash, or even to specify anything that might affect its behaviour.

A. *Debugging with QEMU and GDB*

QEMU is a generic and open source machine emulator and virtualizer. When used as a machine emulator, it is possible to run applications made for one machine to another via dynamic translation. The decision to use QEMU emulator was based on active community, support of Linux as the host machine, a native set of target machines and the possibility to integrate a new machine.

Thus, besides having QEMU to emulate applications, we still need to examine the state and variables of the application. Using GDB - *the GNU Project Debugger* - it is possible to see inside the application while it executes [1]. One important characteristic of GDB is to enable remote debug. Therefore, it is possible to run the program on a given embedded platform while we debug it with GDB running in separated machine. In remote debugging, GDB connects to a remote system over a network and then control the execution of the program and retrieve information about its state.

The integration of both is particular for each host/target machine; thus, 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. These steps and additional explanation of which techniques and tools are used in this process are listed bellow:

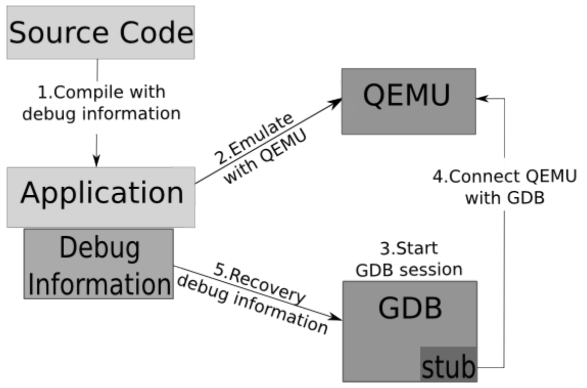


Figure 1: Steps to integrate QEMU and GDB

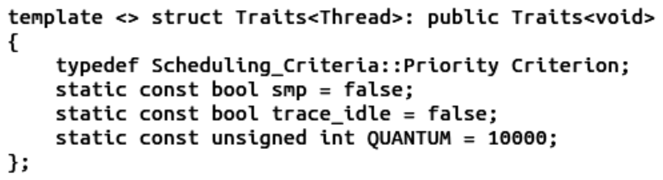
1) **Compile with debug information** is the first and the most important step. The source code is the input, and the output is the compiled application that has debug information. Using GCC (*GNU project C and C++ compiler*) it can be performed by using `-g` option to compile.
2) **Emulate with QEMU** is a necessary step to execute the application in the correct target architecture. To perform this step, the 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 forces QEMU to wait GDB to connect after the system restart, e.g., if we compile an application with debug information (*app.img*) that prints information in the 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 a separate window. Then, to connect GDB in QEMU the developer must explicitly specify that the target to be examined is remote and inform the host address and port of the target (in this case, QEMU). When host is in the same machine as GDB, it is possible inform only the port, but the complete line must be similar to:
   `target remote [host]:[port]`
4) **Recovery debug information** is an important step to help developers to find errors, once it is possible to autocomplete to recovery 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, watch points, control the execution of the program and even enable logs. More information about command set can be found in GDB's page [1].

IV. Automatic Exchange of Configuration Parameters

The automatic exchange of configuration parameters (AEP) is part of an effort to integrate a new debugging tool into operating system methods in order to reduce software development efforts.

The present work uses on the Embedded Parallel Operating System (EPOS) [7] since it adds a great deal of configurability of the system, which is very suitable for evaluating an exchange configuration script. Also, EPOS is a component-based framework that provides all traditional abstractions of operating systems and services like memory management, communication and time management. Furthermore, it was referenced in several [2] academic and industrial projects.

Once EPOS uses generic programming techniques, each abstraction can be configured as desired using traits template parameter [19]. Traits are parametrized classes that describe the properties of a given object/algorithm.



```
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

Figure 2 shows a piece of traits classes used in EPOS to configure the main thread as a set of static members that describes some definitions used by this abstraction.

This operating system instantiates only with the basic support for its dedicated application. It is important to highlight 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 wrong characterization of components, or to improve the performance for the application by selecting a better configuration.

Figure 3 is an overview of how the exchange of configurations parameters occurs. Basically, a trait information is selected, and its definition is changed according to the specification, then the application is recompiled. Traces generated by each version of application is compared and reported to the developer. Before a new cycle is complete, the application execution is verified by the integrated test environment. The performance is analysed and compared with other versions of the application, which also generates an execution report.

[1]http://www.gnu.org/s/gdb/
[2]http://www.lisha.ufsc.br/pub/index.php?key=EPOS

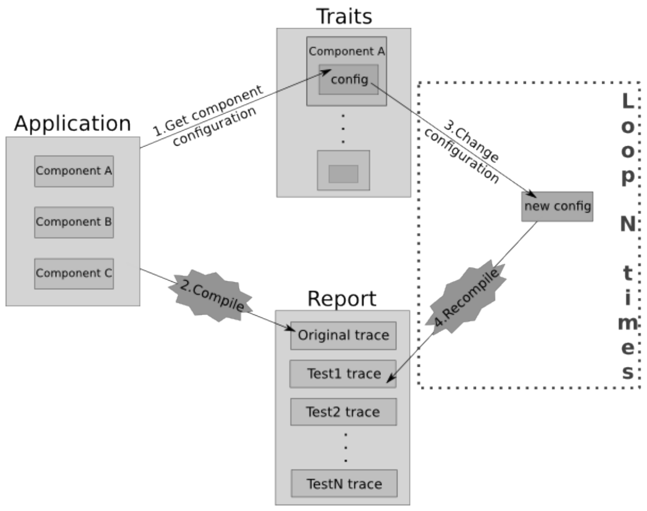


Figure 3: Overview of automatic exchange of configuration parameters

In the current version of the AEP shell script, the selection of configuration and its parameter modification are random, but can also be supplemented with an artificial intelligence tool or some application-oriented system design tool to provide all information for the script.

A. Configuration file

To run the script with a specific configuration it was necessary to manually fill the information before each new round of tests. But to improve the usability of the script now it is possible to define a configuration file with the information needed to run the test. We chose XML to set test settings because it can set all necessary rules to run the script in a way human-readable and furthermore is also easily interpreted by the computer.

Extensible Markup Language (XML) [3] describes the behavior of computer programs and it documents are made up of parsed (characters) or unparsed data (markup encodes). Figure 4 brings one example of the AEP configuration file for DMEC application.

```
<Test>
   <Application>dmec_app<Application>
      <Configuration>
         <Trait>
            <id>NUM_WORKERS</id>
            <type>Integer</type>
            <Interval>
               <begin>0</begin>
               <end>10</end>
            </Interval>
         </Trait>

         <Breakpoint>
            <path>/home/breakpoint_dmec.txt<path>
         </Breakpoint>
   </Configuration>
<Test>
```

Figure 4: Example of AEP configuration file

The configuration will be set up only once, but even if it requires some manual changes it will not add difficulties, since this file can be read almost as a text: there is a <Test> of the "dmec_app" <Application> with two <Configuration>, one <Trait> and one <Breakpoint>. The <Trait> identified as "NUM_WORKERS" is an "integer" <Type> and it could be in <Interval> that starts at "0" and ends at "10". The <Breakpoint> can be found in "/home/breakpoint_dmec.txt" path.

B. Real-World Application

The automatic exchange of parameters script 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. Motion Estimation is a 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 while `Workers` threads estimate and the `Coordinator` processes results [10].

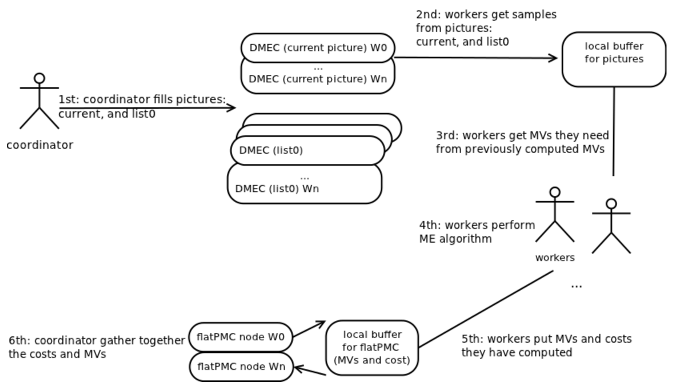

2nd: workers get samples from pictures: current, and list0
DMEC (current picture) W0
...
DMEC (current picture) Wn
local buffer for pictures
1st: coordinator fills pictures: current, and list0
coordinator
DMEC (list0)
...
DMEC (list0) Wn
3rd: workers get MVs they need from previously computed MVs
4th: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

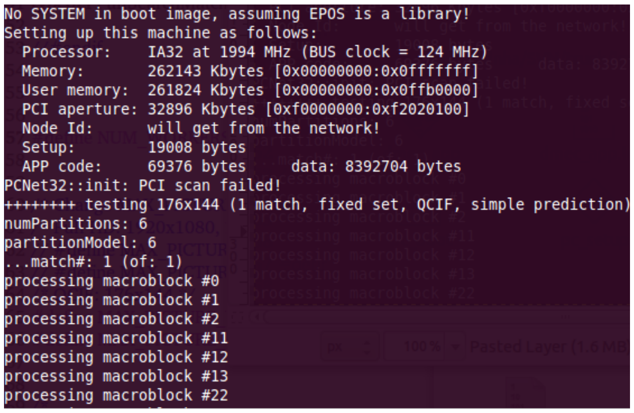
Figure 5: Interaction between `Coordinator` and `Workers` threads [10]

Figure 5 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 6 and 7 show the DMEC execution using values 6 and 60 for `NUM_WORKERS` configuration.

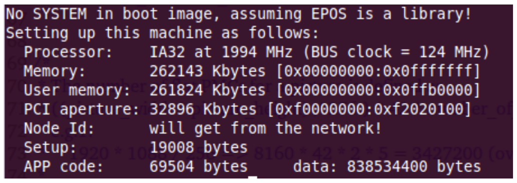
The difference between the two scenarios is that after retrieving information from the application, QEMU has a response only for the six `workers` configuration.

Part of the configurations changed by the script do not even compile. The AEP script uses GDB for debugging all configurations that could be compiled. This process was crucial



```
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
```

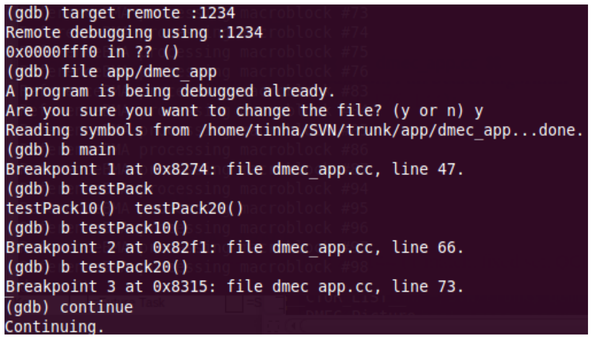
Figure 6: DMEC emulated execution with `NUM_WORKERS` = 6



```
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 7: DMEC emulated execution with `NUM_WORKERS` = 60

to determine the error in DMEC's case. In this sense, some breakpoints were added to all functions, specially the main function, see Figure 8. It is possible to check "continuing" is the last line that appears in the execution. The execution failed because a "too high" value was 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.
```

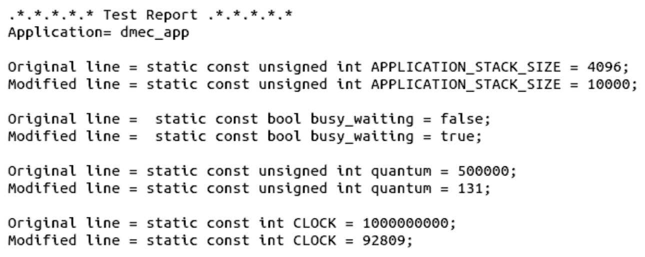
Figure 8: DMEC debug with GDB execution with `NUM_WORKERS` = 60

Through the second part of the script, i.e., the debugging process, it was possible to verify that the program not even reach the main function, which means that now the script must change configurations before calling the main function again.

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. Evaluation considered data from totally random and partially random tests usingof DMEC application.

Totally random test is the one that has no prior information on the application. In other words, any configuration within `traits` can change, including parameters that not influence the application. Figure 9 presents a piece of the report with some generated configuration, e.g., the size of the application stack if a thread should be busy waiting, the value of a quantum, how much cycles clock should consider, etc.



```
.*.*.*.*.* Test Report .*.*.*.*.*
Application= dmec_app

Original line = static const unsigned int APPLICATION_STACK_SIZE = 4096;
Modified line = static const unsigned int APPLICATION_STACK_SIZE = 10000;

Original line =  static const bool busy_waiting = false;
Modified line =  static const bool busy_waiting = true;

Original line = static const unsigned int quantum = 500000;
Modified line = static const unsigned int quantum = 131;

Original line = static const int CLOCK = 1000000000;
Modified line = static const int CLOCK = 92809;
```

Figure 9: Totally random generated configurations

In total random execution case, test performed hundred tries, generating 85 different configurations in which only 23% of them could be correctly compiled, but less than 5% of configurations were relevant to DMEC. Figure 10 presents the ratio of the different configurations generated, those that have been compiled and those that were actually relevant.

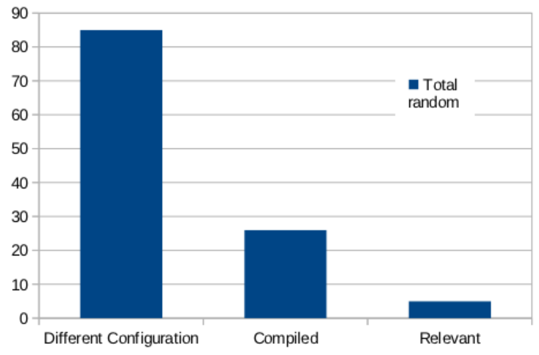
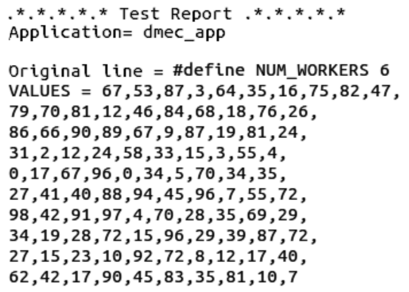


Figure 10: Totally random configurations versus their relevance

The execution of these versions showed that the 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, the number of `Workers`. By focusing in only one parameter, it became possible to try finding the best option for the application. Figure 11 presents a report with all tries.

The test got 69 different configurations (same number of tries) and all of them could be correctly compiled, and only 8 could be executed, in other words, less than 10% of tries



```
.*.*.*.*.* Test Report .*.*.*.*.*
Application= dmec_app

Original 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 11: Partially random generated configurations

could be used as configuration. Figure 12 presents the ratio of the different configurations versus its relevance.

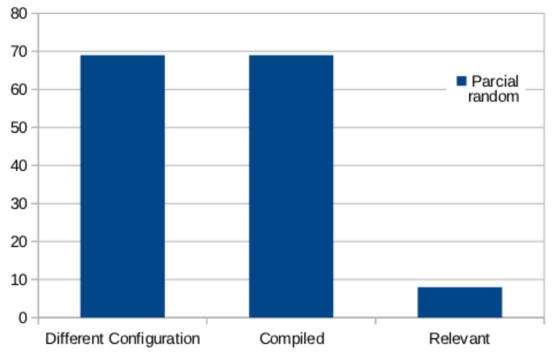


Figure 12: Partially random configurations versus their relevance

Also, to use the integrated debug environment it was necessary to build the code with a special set-up, 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 and increases in 80% the cost in terms of memory to debug DMEC application.

VI. Conclusion

In this paper, we introduce the automatic exchange of configuration parameters ans show how to set up a development environment for embedded applications based on specific hardware/software requirements.

The integrated development environment provides independence of the physical target platform for development and test. It is 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 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 than others.

In this sense, it was possible to realize that even a small automation solution produce answers to help developers finding and fixing bugs. With only a hundred tries were possible to find error/restriction in the code.

References

[1] Gdb:the gnu project debugger, Nov. 2012.

[2] C. Artho. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(3):223–246, 2011.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[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)*, pages 71–77. ACM, 2008.

[5] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381. ACM, 2012.

[6] L. Carro and F. R. Wagner. Sistemas computacionais embarcados. *Jornadas de atualização em informática. Campinas: UNICAMP*, 2003.

[7] A. A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug. 2001.

[8] C.-S. Koong, C. Shih, P.-A. Hsiung, H.-J. Lai, C.-H. Chang, W. C. Chu, N.-L. Hsueh, and C.-T. Yang. Automatic testing environment for multi-core embedded software-atemes. *J. Syst. Softw.*, 85(1):43–60, Jan. 2012.

[9] E. Larson and R. Palting. Mdat: a multithreading debugging and testing tool. In *Proceeding of the 44th ACM technical symposium on Computer science education*, SIGCSE '13, pages 403–408, New York, NY, USA, 2013. ACM.

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

[11] J. Ma, L. T. Yang, B. O. Apduhan, R. Huang, L. Barolli, and M. Takizawa. Towards a smart world and ubiquitous intelligence: a walkthrough from smart things to smart hyperspaces and ubickids. *International Journal of Pervasive Computing and Communications*, 1(1):53–68, 2005.

[12] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[13] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 199–209. ACM, 2011.

[14] S. Parsa, M. Asadi-Aghbolaghi, and M. Vahidi-Asl. Statistical debugging using a hierarchical model of correlated predicates. *Artificial Intelligence and Computational Intelligence*, pages 251–256, 2011.

[15] D. Qi, M. Ngo, T. Sun, and A. Roychoudhury. Locating failure-inducing environment changes. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 29–36. ACM, 2011.

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

[17] S. Schneider and L. Fraleigh. The ten secrets of embedded debugging. *Embedded Systems Programming*, 17:21–32, 2004.

[18] J. Seo, A. Sung, B. Choi, and S. Kang. Automating embedded software testing on an emulated target board. In H. Zhu, W. E. Wong, and A. M. Paradkar, editors, *AST*, pages 44–50. IEEE, 2007.

[19] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[20] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, pages 02–3, 2002.

[21] D. Toupin. Using tracing to diagnose or monitor systems. *Software, IEEE*, 28(1):87–91, 2011.

[22] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.

[23] C. Zhang, J. Yang, D. Yan, S. Yang, and Y. Chen. Automated breakpoint generation for debugging. *Journal of Software*, 8(3), 2013.

[24] 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*, pages 43–52. ACM, 2009.

[25] A. Zheng, M. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112. ACM, 2006.