Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Automatic testing environment for multi-core embedded software—ATEMES

Chorng-Shiuh Koong[a,*], Chihhsiong Shih[b], Pao-Ann Hsiung[c], Hung-Jui Lai[a], Chih-Hung Chang[d], William C. Chu[b], Nien-Lin Hsueh[e], Chao-Tung Yang[b]

[a] *Dept. of Computer and Information Science, National Taichung University, Taichung, Taiwan*
[b] *Dept. of Computer Science and Information Engineering, Tunghai University, Taichung, Taiwan*
[c] *Dept. of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan*
[d] *Dept. of Information Management, Hsiuping Institute of Technology, Taichung, Taiwan*
[e] *Dept. of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan*

A R T I C L E   I N F O

Article history:
Received 2 April 2010
Received in revised form 7 July 2011
Accepted 29 August 2011
Available online 10 September 2011

Keywords:
Automatic testing
Embedded software testing
Coverage testing
Unit testing
Cross-testing
Testing tool
Test case generation
Object testing
Multi-core embedded software testing
Parallelism degree testing
TBB testing

A B S T R A C T

Software testing during the development process of embedded software is not only complex, but also the heart of quality control. Multi-core embedded software testing faces even more challenges. Major issues include: (1) how demanding efforts and repetitive tedious actions can be reduced; (2) how resource restraints of embedded system platform such as temporal and memory capacity can be tackled; (3) how embedded software parallelism degree can be controlled to empower multi-core CPU computing capacity; (4) how analysis is exercised to ensure sufficient coverage test of embedded software; (5) how to do data synchronization to address issues such as race conditions in the interrupt driven multi-core embedded system; (6) high level reliability testing to ensure customer satisfaction. To address these issues, this study develops an automatic testing environment for multi-core embedded software (ATEMES). Based on the automatic mechanism, the system can parse source code, instrument source code, generate testing programs for test case and test driver, support generating primitive, structure and object types of test input data, multi-round cross-testing, and visualize testing results. To both reduce test engineer's burden and enhance his efficiency when embedded software testing is in process, this system developed automatic testing functions including unit testing, coverage testing, multi-core performance monitoring. Moreover, ATEMES can perform automatic multi-round cross-testing benchmark testing on multi-core embedded platform for parallel programs adopting Intel TBB library to recommend optimized parallel parameters such as pipeline tokens. Using ATEMES on the ARM11 multi-core platform to conduct testing experiments, the results show that our constructed testing environment is effective, and can reduce burdens of test engineer, and can enhance efficiency of testing task.

1. Introduction

As computer software technology evolves to more sophistication, large scale of embedded software application and growing customized demands highlight the importance of embedded software quality control (Yin and Liu, 2009). Among the most discussed issues of software engineering, extensive attention is on improving software quality (Tasse, 2002). Software testing is one of the leading approaches (Hailpern and Santhanam, 2002) to ensure software quality since it enhances software quality and reliability (Kaner et al., 1999; Bertolino, 2007). However, software testing consumes massive manpower and efforts. Automatic or semi-automatic approaches are regarded as useful tools to economize testing time span and efforts.

Traditional unit testing tools mainly focus on workstation platform. Test case and test input data are generated from manual or automatic input. Current method for automatic test case generation still needs to be improved (Michael et al., 2002). Most unit testing tools are only capable of either automatically generating test case program framework, or merely supporting primitive type. Test engineer is obliged to manually write testing program segment and input test data under the generated program framework, or to generate test case manually (Sen et al., 2005). Meanwhile, enormous unwanted workloads are accompanied by repetitive actions during testing. Further, testing conducted by manually input test data is neither efficient nor capable of increasing test coverage (King, 1976).

Generally, embedded system confronted with more hardware and software resource constraints than desktop computer does (Broekman and Notenboom, 2002). Test engineer has to exhaust more efforts to test embedded software before improving quality of embedded software (Myers, 2004). For this reason, embedded

* Corresponding author. Fax: +886 4 22183580.
E-mail addresses: csko@mail.ntcu.edu.tw, shihc@go.thu.edu.tw (C.-S. Koong).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.08.030

software testing differs from conventional testing in that target program can first execute complicated operations on PC side for the required resource necessary to testing, and then execute testing on embedded hardware. This allows software to reduce testing efforts on embedded platform (Delamaro et al., 2006).

The use of multi-core embedded system has been prevalently considered to be better options than other alternatives. This also makes multi-thread program indispensable in improving performance (Hung et al., 2008). However, synchronizing defect such as race condition is an inevitable issue in parallel program. The factor of data synchronization, thus, demands special attention when executing parallel program. This factor coupled with the resource constraints problem faced with testing an embedded system as mentioned before makes the testing of multi-core embedded system especially challenging.

Parallel programs running on multi-core embedded systems usually adopt express coding library such as Intel TBB library (TBB) for scalability and performance. Intel TBB pipeline allows user to decide the extent of parallelism. Parameters such as pipeline token numbers and pipeline stage numbers can be modified as required. Experiences are essential in deciding the better pipeline token numbers and pipeline stage numbers. Parallelism degree can be limited if the token numbers selected are too small. On the contrary, resources can be consumed unnecessarily. For example, more buffer spaces may be needed if token numbers selected are too big. Traditionally, numerous manual modifications are required from programmer before better pipeline token number parameter can be found. To address the issue, this study enhances performance testing by expanding functionalities of automatic multi-round testing. To find better pipeline token number for target program, raw data returned from target-side is calculated automatically and analyzed during runtime. This technology not only frees programmer from consuming extra energy to find parallel pipeline parameter, but also elevates parallel program performance in more precision.

Summing up from the discussion, issues waiting for current multi-core embedded software testing to address include: (1) how demanding efforts and repetitive tedious actions can be reduced; (2) how resource restraints of embedded system platform such as temporal and memory capacity can be tackled; (3) how embedded software parallelism degree can be controlled to empower multi-core CPU computing capacity; (4) how analysis is exercised to ensure sufficient coverage test of embedded software; (5) how to do data synchronization to address issues such as race conditions in the interrupt driven multi-core embedded system; (6) high level reliability testing to ensure customer satisfaction.

This study developed an automatic testing tool to support cross-testing to both reduce target program overhead from performing testing functionality on embedded platform and decrease efforts for test engineer. The ATEMES can not only automatically generate test data with primitive type, structure type, object type and array type but also generate CppUnit-based test case and test driver. This addresses the first issue.

Moreover, ATEMES can execute automatic multi-round performance testing over multi-core embedded software adopting Intel TBB library. The system can support locating recommended value for better parallel parameter token number, which not only allows embedded software parallelism but also facilitates computing capacity of multi-core CPU to operate more efficiently. This feature addresses the issues of (2) and (3).

With the automatic multi-round mechanism, unit testing and coverage testing (Lyu et al., 1994) can be implemented to save test engineer from massive repetitive tasks. With the cross-testing technology between host-side (workstation) and target-side (embedded system), factors resulted from embedded system resource restraints can be reduced for testing. With the cross-testing technology, test case, test driver and target program can be cross-compiled automatically and uploaded to target-side for automatic implementation. Target-side test log data including runtime data, output result, and data of each core utilization during runtime from target-side CPU can be passed to host-side for runtime analysis, results of which can also be visually presented. This helps tackling the issues of (1) and (4).

To address issue (5), we instrument proper lock mechanism, such as mutex of C++, to source code segment of target program pertinent to performance testing. This way the share data can be shielded, and accurate testing results can be ensured. As for multi-core system performance monitor, to analyze overhead of each CPU core in the embedded system, ARM Linux system call is also requisite to effectively monitor CPU core number allocated by the task. Repetitive automatic multi-round testing, similarly, provides a vital scenario to analyze more precisely what performance the task is executing in embedded system, and to locate bottleneck to be tackled. We measure the response time of target program(parallel program) with different interrupt intervals using different core processor numbers on the ARM11 multi-core platform. The test data show that the response time generally decreases as number of cores increases. This provides evidences that our constructed testing environment can not only reduce burdens of test engineer, but also enhance efficiency of multi-core embedded testing task. This helps relieve both issues (3) and (5). Finally, a set of usability testing has been arranged to evaluate testing reliability for issue (6).

2. Related work

Delamaro et al. (2006) developed a coverage testing tool for mobile device software. The tool, named JaBUTi/ME, mainly supports java source code and can solve restrictions of mobile device performance and storage. Testing conducted on mobile device is difficult since issues such as memory limitations, persistent storage, and network connection availability have to be taken into consideration.

The tool not only can be implemented on emulators, but also can help testing on mobile device. With a desktop computer, test engineer can implement testing, instrument class, and generate test session on mobile devices. Communication between desktop computer and mobile device is exchanged through test server. Execution of test case can be monitored through the transfer of trace data when instrumented code is being executed on mobile device. This approach can reduce workloads from testing mobile device. Visualized interface of test result allows test engineer to be informed of what programs are to be executed. However, the tool does not support automatic testing. Failure of automatically generating test input data and test case results in more tasks from test engineer to edit test source code and manually generating test input data.

Ki et al. (2008) proposed an automated scheme of embedded software interface test based on the emulated target board called “Justitia”. The setting of breakpoints allows test engineer to debug. However, the tool is efficient only to experienced test engineer who is skilled in the embedded system architecture.

The merit of Justitia is in its automatically detecting errors capability on program interface. Embedded software testing is engaged by combining the existing monitoring and debugging technology of emulator. By defining embedded software interface pattern, an automated scheme is created for locating source code interface. The tool can automatically generate test case, namely, interface test feature, location of interface, symbol to be monitored at the interface, input data, and expected output, and execute test case using emulation testing technology. Further, the system also supports memory test and interrupt test. After testing is finished, result of test coverage and interface error is presented on visualized interface. The system mainly supports single/unit testing rather on multi-round automatic testing. More time is required to increase test coverage.

Cho and Choi (2008) proposed a Multi-paradigm views embedded software testing tool. The tool is based on client/server model of host–target architecture. It is a performance testing tool independent from extra hardware support. On host-side, the tool provides test engineer a handy graphic user interface. On target-side, software testing including memory test, code coverage test, and function performance test are being executed on embedded system platform. Cho and Choi designed a xml-based DTDs (document type definitions) to increase test script usability and reusability. On host-side, test engineer can generate test script, test suite, and test driver by manually inputting data with the aid of test script wizard and test driver wizard. Test result can be viewed through different visualized options. Target-side is the embedded software test platform where target program is being executed on the platform of embedded Linux HRP-SC2410 (Ami). The tool facilitates test engineer to generate test script, test suit, and test driver. However, test script and test driver cannot be generated automatically and it lacks multi-round automatic testing. More time is demanded from test engineer to write test script and test driver.

3. Automatic testing mechanism

This section presents automatic testing approaches adopted by the system.

Section 3.1 introduces automatic approaches assisting generating test data with C\C++ complex types, and supporting generating programs for test case and test driver. The tool allows input parameter of unit testing to support C\C++ complex data type.

Section 3.2 aims at proposing automatic multi-round testing method so as to reduce test engineer's burden from repetitive works for executing target program.

Section 3.3 focuses on the Intel TBB (TBB) parallel program executing on multi-core embedded platform. The section prescribes a multi-round testing approach with varied token numbers for test engineer to locate better parallel parameter and improve the capacity of multi-core CPU computing power.

3.1. Unit testing

ATEMES can support automatic and semi-automatic testing. The system can detect execution exception or result error occurred from each program function/method. Before testing, test data, test case, and test driver have to be generated. ATEMES supports automatically generating C\C++ complex test data. Types of generated test input data include primitive type, structure type, object type, and array type. Based on the generated input data, the system would generate test case and test driver program.

For automatic testing, firstly, test input data is generated automatically from ATEMES. However, test result data is not included in test data since it cannot be generated automatically from ATEMES. Secondly, ATEMES generates programs of test case and test driver automatically. Finally, the system would perform multi-round testing. Exception or segment fault occurred from target program is recorded when testing is in process. Output is not checked by the system.

For semi-automatic testing, firstly, test input data is either generated automatically from ATEMES or from manual input by test engineer. Secondly, test engineer has to input test result data. Thirdly, ATEMES generates programs of test case and test driver automatically. Finally, the system would perform multi-round testing. Exception or segment fault occurred from target program is recorded when testing is in process. Likewise, output is checked according to test result data.

3.1.1. Algorithm for automatically generating test data

ATEMES can generate complex data type for automatic testing. Fig. 1 displays algorithm for automatic generating test data. Following are the term descriptions pertinent to the algorithm. The automatic generation of test data makes sure the data format is

```
Algorithm : test input data generation function

Function-List:FL
Object-List:OL
Parameter-List:PL

Procedure TestInputDataGenerationMainFunction()
begin
  for_each method in FL
    get Parameter-List PL from FL
    get Return-List PL from FL
    call TestInputDataGenerator (PL)
  end_for each method
end

Procedure TestInputDataGenerator(Parameter-List PL)
begin
  for_each parameter p in PL
    switch (type of p)
      case int_type:
        if p = single element    // p= single element type
          then generate int test data
        else // p= array type
          get dimension i from p
          repeat i times to generate input data
        end if
      break;
      case float_type :
      if p = single element    // p= single element type
        then generate float test data
      else // p= array type
        get dimension i from p
        repeat i times to generate input data
      end_if
      break;

      //case char , short, double, ...

      //case object_type
      case object_type :
      if p = single element    // p= single element type
        then
        get Object Attribute Set OAS from Object-List
        let Object Parameter-List OPL = OAS;
        call TestInputDataGenerator (OPL)
      else // p= array type
        get Object Attribute Set OAS from Object-List
        let Object Parameter-List OPL = OAS;
        get dimension i from p
        repeat i times
              to call TestInputDataGenerator (OPL)
      end_if
      break;

      //case structure_type: be similar to object type

      default:
    end_switch
  end_for
end
```

Fig. 1. Algorithm for automatically generating input data.

consistent with the testing requirements of ATEMES environments. In addition, the generated test data allows input parameter of unit testing to support C\C++ complex data type. This is a unique feature of the ATEMES, none seen in many existing test systems except for Putrycz's system. This is evidenced by Table 7.

Function-List: The term means saving all function/method information from target program, including name, parameter type, and return type. Function-List stands for data parsed by Code Analyzer module.

Struct-List: The term means saving all structure type attributes from target program.

Object-List: The term means saving all object type attributes from target program.

Parameter-List: The term means saving all parameter types of function/method.

TestInputDataGenerationMainFunction: This is the main function for generating test input data. It controls the flow for getting all parameter type and return type of function/method information from target program, and generate input data.

TestInputDataGenerationGenerator: This function is mainly responsible for generating test data.

Steps of the algorithm are described as follows:

Step 1: TestInputDataGenerationMainFunction procedure gets method name, parameter type, and return type from Function-List. Parameter type and return type are then passed to TestInputDataGenerator procedure. Repeat previous actions till data of Function-List is empty.
Step 2: TestInputDataGenerator creates corresponding test input data based on parameter type.
  Step 2-1: If the type is primitive, the system will create corresponding test data such as integer, float, and char.
  Step 2-2: If the type is array, the system will get array dimension and calculate the intended data volume, and creates corresponding test data.
  Step 2-3: If the type is structure type, the system will recognize all primitive type information from the structure recursively to create corresponding test data.
  Step 2-4: If the type is object type, the system will recognize all primitive type data from the object recursively to create corresponding test data.

3.1.2. Algorithm for automatically generating test case

After test data are created, all test cases of target function/method also have to be created. By reading each test data from the file, ATEMES would generate test cases based on each test data type, and then would generate test drivers. When testing is in process, the generated test case would check if any exception or segment fault occurs from target program. The system then verifies executed result according to the expected result. This feature is unique for ATEMES and not seen in other similar works as seen in Table 7.

Fig. 2 displays algorithm for automatic generated test case. Following are the term descriptions pertinent to the algorithm.

Function-List, **Struct-List**, **Object-List and Parameter-List**: Descriptions of these teams are the same with previous section.

TestCaseGenerationFunction: This function is mainly responsible for generating test input data based on the parameter types of function/method.

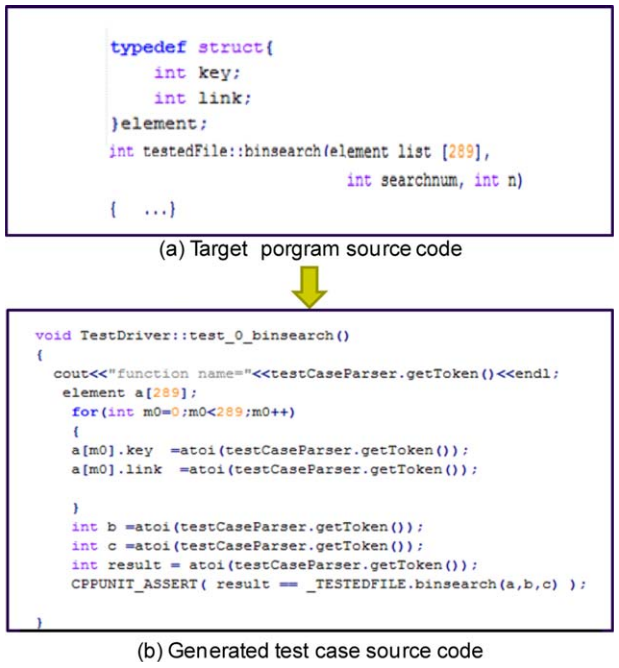
Steps of the algorithm are described as follows:

Step 1: Get method name, parameter type, and return type from Function-List.
Step 2: Generate function signature source code segment for test case.
Step 3: Determine parameter types as primitive type, structure type, or object type.
Step 4: Instrument a source code segment for variable declaration based on parameter type.
  Step 4-1: If parameter type is primitive, instrument source code segment to read a test input data which is assigned to the variable.
  Step 4-2: If parameter type is object or structure, get all primitive type attributes of object or structure, and instrument source code segment allowing all attributes to read test data.
  Step 4-3: If parameter type is array, calculate numbers of test input data need to be read and create source code segment which can read all test data using loop instruction.

```
Algorithm : test case generation function

Function-List:FL
Object-List:OL
Sturct-List: SL
Struct-List:SL
Parameter-List:PL

Procedure TestCaseGenerationFunction()
begin
  read test input data file

  while has test data set
    get function name from FL
    generate test-case-function-name code
    get Parameter-List PL from FL

// Get test input data of primitive type
    for_each parameter p in PL
     switch (type of p)
     case int_type :
       if p = single element
        then
         generate code for int variable declaration
         generate code for getting input data for int
       else // p= array type
         generate code for int array variable declaration
         get int array dimension from p
         generate code for for-loop to get input data for int
       end_if
     break;

//Case:char_type , short_type, double_type ...

// Get test input data of struct type
     case struct_type:
       if p = single element
        then
         generate code for struct variable declaration
         get struct attribute set SAS from SL
         generate code for getting input data for SAS's
element
        else // p= array type
         generate code for sturct array variable declaration
         get sturct array dimension from p
         get sturct attribute set SAS from SL
         generate code for for-loop to get input data for
SAS's element
        end_if
        break;

// Get test input data of object type
     case object_type :
      if p = single element
       then
         generate code for object variable declaration
         get object attribute set OAS from OL

       else // p= array type
         generate code for object array variable declaration
         get object array dimension from p
         get object attribute set OAS from OL
         generate code for for-loop to get expected result for
OAS's element from FL
         generate code for function-call to get return value
         generate code for CppUnit Assertion
       end_if
      break;

     case void_type :
      if p = single element
        then generate code for function-call
      break;
     end_switch
  end_while
end
```

Fig. 2. Algorithm for automatically generating test case.

Step 5: Determine method return type.
  Step 5-1: If return types are primitive, structure, and object, instrument source code segment which can read all test data, call target function, and assertion function. Assertion function can be used to judge if method return value fulfills the expected value.
  Step 5-2: If return type is void, generate source code segment which can call target function.



```
typedef struct{
    int key;
    int link;
}element;
int testedFile::binsearch(element list [289],
                          int searchnum, int n)
{   ...}
```

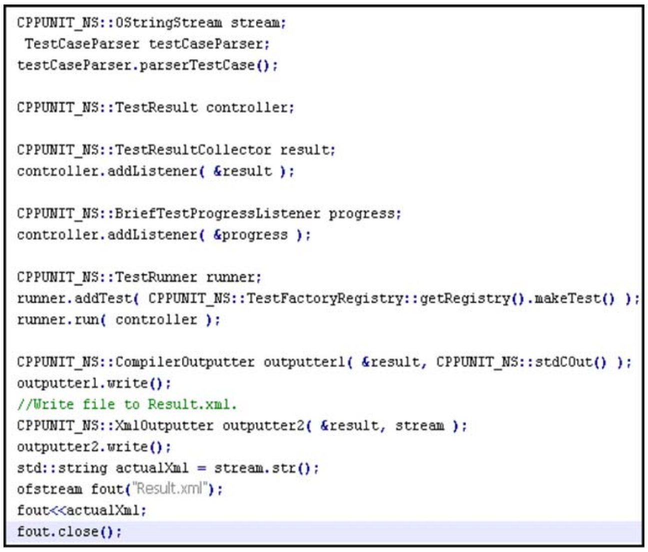
```
void TestDriver::test_0_binsearch()
{
  cout<<"function name="<<testCaseParser.getToken()<<endl;
   element a[289];
    for(int m0=0;m0<289;m0++)
    {
    a[m0].key  =atoi(testCaseParser.getToken());
    a[m0].link  =atoi(testCaseParser.getToken());

    }
    int b =atoi(testCaseParser.getToken());
    int c =atoi(testCaseParser.getToken());
    int result = atoi(testCaseParser.getToken());
    CPPUNIT_ASSERT( result == _TESTEDFILE.binsearch(a,b,c) );

}
```

Fig. 3. Example for generated test case.

Following is an example for the actual executing flow from generating test data to performing testing. According to the target program, the system can generate testing program of test case and test driver using CppUnit library.

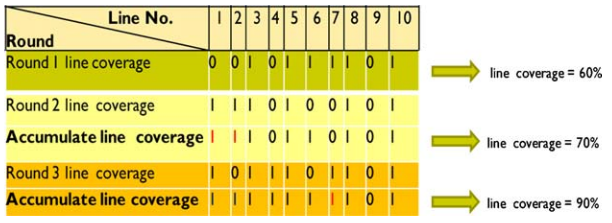
Fig. 3(a) shows the target function binsearch() with three parameter types, including structure array type, integer type, and integer type, respectively. Return value type is integer. Fig. 3(b) shows the generated test case program. The program can read test data corresponding to parameter types. Through CPPUNIT_ASSERT function, testing function is implemented to evaluate whether output result fulfills test engineer's expected value or not. If not, test case will signal failed message, such as assertion failed.

3.1.2.1. Generating test driver automatically. To control testing flow, ATEMES in the last phase would generate test driver after test case is generated. Fig. 4 illustrates the automatically generated



```
CPPUNIT_NS::OStringStream stream;
 TestCaseParser testCaseParser;
testCaseParser.parserTestCase();

CPPUNIT_NS::TestResult controller;

CPPUNIT_NS::TestResultCollector result;
controller.addListener( &result );

CPPUNIT_NS::BriefTestProgressListener progress;
controller.addListener( &progress );

CPPUNIT_NS::TestRunner runner;
runner.addTest( CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest() );
runner.run( controller );

CPPUNIT_NS::CompilerOutputter outputter1( &result, CPPUNIT_NS::stdCOut() );
outputter1.write();
//Write file to Result.xml.
CPPUNIT_NS::XmlOutputter outputter2( &result, stream );
outputter2.write();
std::string actualXml = stream.str();
ofstream fout("Result.xml");
fout<<actualXml;
fout.close();
```

Fig. 4. Example for generated test driver.



Line No. / Round	1	2	3	4	5	6	7	8	9	10	
Round 1 line coverage	0	0	1	0	1	1	1	1	0	1	line coverage = 60%
Round 2 line coverage	1	1	1	0	1	0	0	1	0	1	
Accumulate line coverage	1	1	1	0	1	1	0	1	0	1	line coverage = 70%
Round 3 line coverage	1	0	1	1	1	0	1	1	0	1	
Accumulate line coverage	1	1	1	1	1	1	1	1	0	1	line coverage = 90%

Fig. 5. Coverage calculation example.

test driver example program based on CppUnit. Test driver governs flows of all test cases, and can extract target program errors during runtime. Once testing is completed, the system would write the output result into XML file. The goal is to relieve users' burden and enhance testing quality. The design of test driver automatic generation can guarantee tight interaction among automatic test case/data generation to support multi-round coverage testing, a special and unique feature of ATEMES. This feature is essential to the testing of multi-core embedded system and to be evidenced through out the experiments section especially in experiment 6. None of other research has such function.

3.2. Coverage testing

3.2.1. Coverage testing

The operation of coverage testing is based on the functions of GNOME gcov (Gcov). Multi-round coverage testing incorporates the automatic testing mechanisms of generating data, generating test case and test driver program, multi-round testing, and cross-testing.

During executing coverage testing functionality, GCov flag would be automatically added when HSATM module of ATEMES is compiling target source code. Executable file and *.gcno file of target program are then generated. The two files are later passed to the embedded platform for testing. After each round of testing, *.gcda and .gcov files are automatically generated and are transferred back to host-side. POPM module of ATEMES simultaneously analyzes coverage testing log data (*.gcda and *.gcov files) and calculates line coverage and branch coverage.

The increase of total target program coverage is done by summing up each round's testing coverage result. Visualized testing coverage is presented dynamically on host-side during runtime. Example of calculation is shown in Fig. 5. Fig. 6 illustrates the workflow of multi-round coverage testing. Algorithm of multi-round coverage testing is introduced as follows:

```
Step 1: Read target program.
Step 2: Parse target program and extract function information including
  parameter type.
Step 3: Generate test data, test case, and test driver automatically.
Step 4: Include an extra gcov parameter while cross-compiling target program
  and test driver into target execution files.
Step 5: Upload executable target program, executable test case, and *.gcon
  generated by gcov to target-side for execution.
Step 6: When testing task is completed, test log generated by gcov is transferred
  back to host-side.
Step 7: Analyze test log and deciding if the results fulfill coverage threshold
  value.
  Step 7-1: If the results fulfill coverage threshold value, the system stops testing
  and demonstrate visualized test log of each round to test engineer.
  Step 7-2: If the result does not reach coverage threshold value, the system
  generates test case automatically and go back to step 5 to continue next round
  of testing. The system simultaneously demonstrates visualized test log of each
  round to test engineer.
```

3.2.2. Algorithm for multi-round testing

Intel TBB pipeline allows user to decide the extent of parallelism. Parameters such as pipeline token numbers and pipeline stage numbers can be modified as required. Experiences are essential

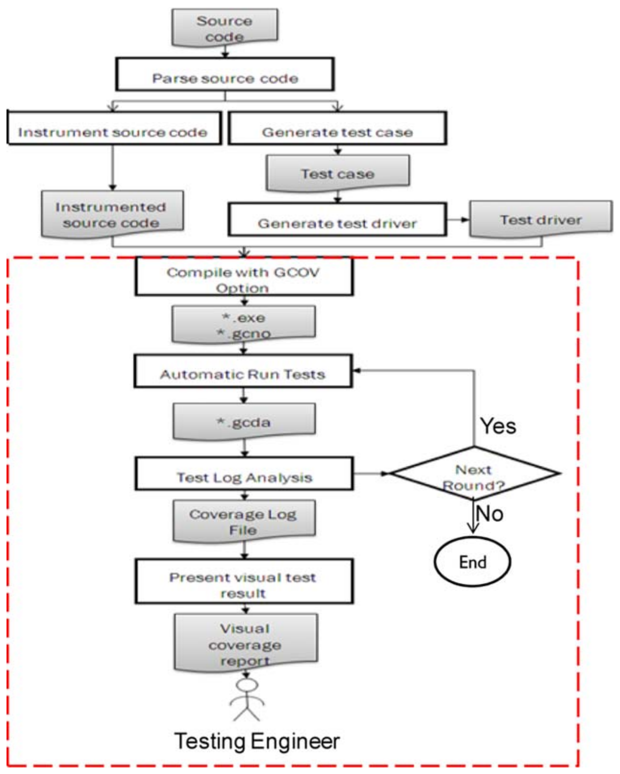


Fig. 6. Coverage testing workflow.

```
1. input: String s, where s read from GCOV log
          file
2. input: String t, where t read from
          totalCoverge log file
3. output: coverage
4. set flag={}
5. set s'=s tokenize with line
6. set t'=t tokenize with line
7. while s' not null
8.     if s' equal to 1
9.         add 1 to flag
10.    else
11.        if t' equal to 1
12.            add 1 to flag
13.        else
14.            add 0 to flag
15.        end_if
16.    end_if
17. set s'=s tokenize with line
18. set t'=t tokenize with line
19. end_while
20. write flag to totalCoverage log file
21. coverage=count(flag equl to 1)/size of flag
22. Exit
```

Fig. 7. Algorithm for coverage testing.

in deciding the better pipeline token numbers and pipeline stage numbers. Parallelism degree can be limited if the token numbers selected are too small. On the contrary, resources can be consumed unnecessarily. For example, more buffers may be needed if token numbers selected are too big. Traditionally, numerous manual modifications are required from programmer before better pipeline token number parameter can be found. To address the issue, this study enhances performance testing by expanding functionalities of automatic multi-round testing. To find better pipeline token number for target program, raw data returned from target-side is calculated automatically and analyzed during runtime.

Further more, multi-round testing functionality allows measuring of the response time of target program(parallel program) with different interrupt intervals under embedded platform using different core processor numbers. ATEMES is employed to automatically generate test driver and different test data, namely, core processor number, interrupt time interval, and interrupt times. With the multi-round testing functionality, the testing tool executes testing automatically and repetitively.

Following is the design for calculating multi-round coverage testing. Fig. 7 shows the algorithm for multi-round coverage testing. The algorithm has 2 inputs, which are test logs generated by GCOV function, and log saved from the summing records. Flag set is used in the system to record the execution of each line/branch. If the line/branch is executed, its record is set as 1. If not, its record is set as 0. Hence, the size of flag set is the total line/branch numbers of program. Single-round of coverage is calculated as follows.

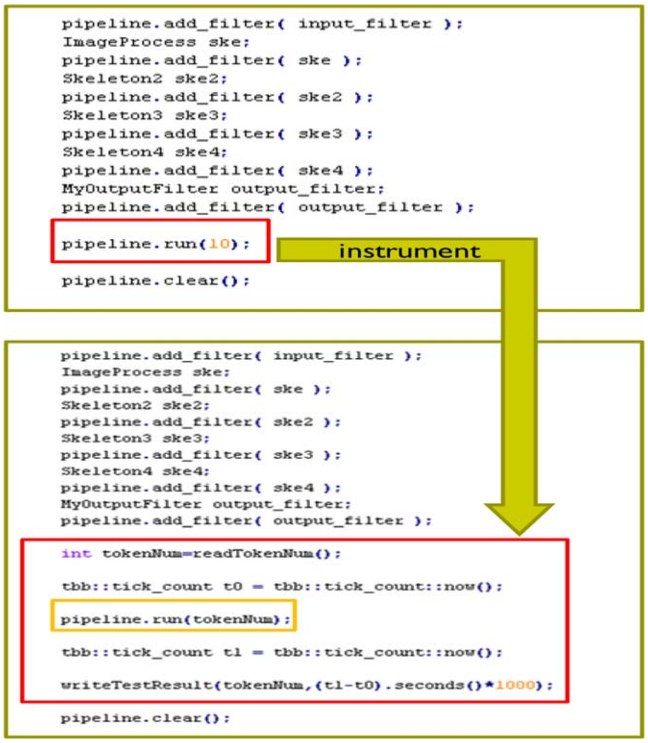
The system is checking whether or not the ith line/branch of the program has been executed. If it has been executed, flag[i] is set as 1. If not, totalCoverage log would be checked to confirm if that specific line/branch has been executed in previous round. If it has been executed, flag[i] is set as 1. If not, flag[i] is set as 0. After all lines/branches of the program are checked and signposted flags, flag set is written into totalCoverage log.

3.3. Parallel program performance measurement

This section focuses on parallel program performance measurement approach supporting parallel program adopting Intel TBB library (TBB). This technology not only frees programmer from consuming extra energy to find parallel pipeline parameter, but also elevates parallel program performance in more precision. For pipeline stage numbers, due to task specificity, division of task unit and selection of different stage numbers still relies mainly on programmer's expertise. After selecting each set of stage numbers, automatic multi-round testing can analyze and find the better combination of token numbers and stage numbers.

Firstly, test engineer must select target program containing TBB pipeline function. The system then automatically instruments testing code segment into target program to collect performance data, as shown in Fig. 8. Within the testing code segment, timer provided by TBB library is for collecting pipeline program execution time and replacing the execution parameter (ppline.run(int)) of the original pipeline program. This allows testing code segment to read the automatically generated pipeline token numbers. Finally, target-host mechanism begins performing automatic multi-round testing. Distribution chart of execution time and utilization of each CPU core is presented on host-side when target program execution is completed. The system can analyze the collected testing data automatically. For pipeline parallel program to achieve better performance, test engineer is provided by the system with a recommended range of parallel token numbers.

According to our experience, our analysis method concludes that grouping 5 token numbers as a unit is likely to get the best mean value. Fig. 9 exemplifies the execution time of the measured token numbers. Mean value is calculated before a suggested range of token numbers is found. As shown in Fig. 10, token numbers being measured are 1–10. Mean value of execution time for token number 1–5 is 17.6 ms, for 2–6 is 12.4 ms, for 3–7 is 11.8 ms, for 4–8 is 12.4 ms.



```
pipeline.add_filter( input_filter );
ImageProcess ske;
pipeline.add_filter( ske );
Skeleton2 ske2;
pipeline.add_filter( ske2 );
Skeleton3 ske3;
pipeline.add_filter( ske3 );
Skeleton4 ske4;
pipeline.add_filter( ske4 );
MyOutputFilter output_filter;
pipeline.add_filter( output_filter );

pipeline.run(10);

pipeline.clear();
```

```
pipeline.add_filter( input_filter );
ImageProcess ske;
pipeline.add_filter( ske );
Skeleton2 ske2;
pipeline.add_filter( ske2 );
Skeleton3 ske3;
pipeline.add_filter( ske3 );
Skeleton4 ske4;
pipeline.add_filter( ske4 );
MyOutputFilter output_filter;
pipeline.add_filter( output_filter );

int tokenNum=readTokenNum();

tbb::tick_count t0 = tbb::tick_count::now();

pipeline.run(tokenNum);

tbb::tick_count t1 = tbb::tick_count::now();

writeTestResult(tokenNum,(t1-t0).seconds()*1000);

pipeline.clear();
```

Fig. 8. Example for instrumented code to TBB parallel program.

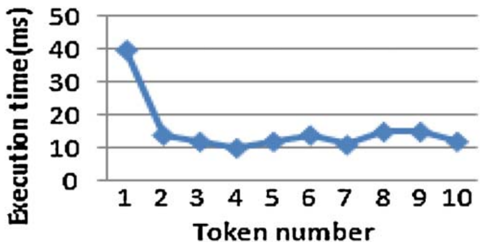
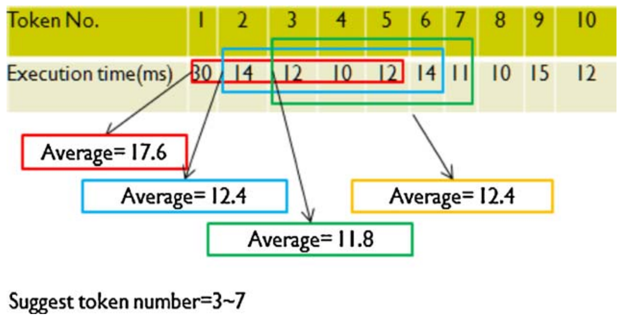


Fig. 9. Time distribution for performance measurement.

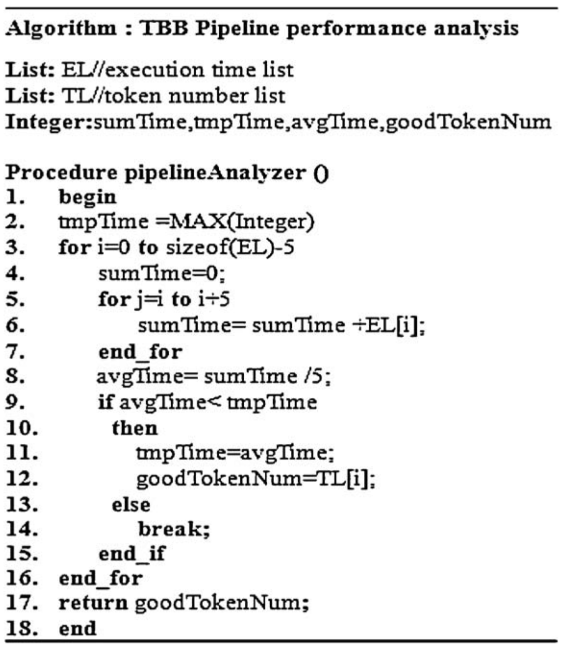
Inferring from this, the next mean value can be expected to be greater than the current one. That is, execution time for token number 5–9 would be greater than those of token number 4–8. Thus, it is suggested that token number should be selected from the range of mean value. Take Fig. 10 for example, the suggested token numbers are 3–7. Algorithm is shown in Fig. 11.



Token No.	1	2	3	4	5	6	7	8	9	10
Execution time(ms)	30	14	12	10	12	14	11	10	15	12

Average= 17.6
Average= 12.4
Average= 12.4
Average= 11.8
Suggest token number=3~7

Fig. 10. Mean value of token numbers' performance for TBB pipeline.



```
Algorithm : TBB Pipeline performance analysis

List: EL//execution time list
List: TL//token number list
Integer:sumTime,tmpTime,avgTime,goodTokenNum

Procedure pipelineAnalyzer ()
1.   begin
2.   tmpTime =MAX(Integer)
3.   for i=0 to sizeof(EL)-5
4.        sumTime=0;
5.        for j=i to i+5
6.             sumTime= sumTime +EL[i];
7.        end_for
8.        avgTime= sumTime /5;
9.        if avgTime< tmpTime
10.         then
11.            tmpTime=avgTime;
12.            goodTokenNum=TL[i];
13.         else
14.            break;
15.       end_if
16.  end_for
17.  return goodTokenNum;
18.  end
```

Fig. 11. Algorithm for TBB pipeline performance analysis.

4. Automatic testing tool for embedded software

4.1. ATEMES system module description

The automatic testing environment for multi-core embedded software (ATEMES) is composed of four parts: Pre-Processing Module (PRPM), Host-Side Auto-Testing Module (HSATM), Target-Side Auto-Testing Module (TSATM), and Post-Processing Module (POPM). Cross-testing for embedded software can be realized with these 4 modules. Testing functions supported by ATEMES system include coverage testing, unit testing, multi-core performance testing, and race condition testing. This paper focuses mainly on coverage testing, unit testing, and multi-core performance testing. Fig. 12 illustrates system module. Fig. 13 presents system architecture layers. Functions and implementation details of respective modules are detailed in the following sections.

4.2. System architecture layers

ATEMES system is divided into 5 layers as shown in Fig. 13. The bottom is hardware platform layer. X86 Platforms is adopted

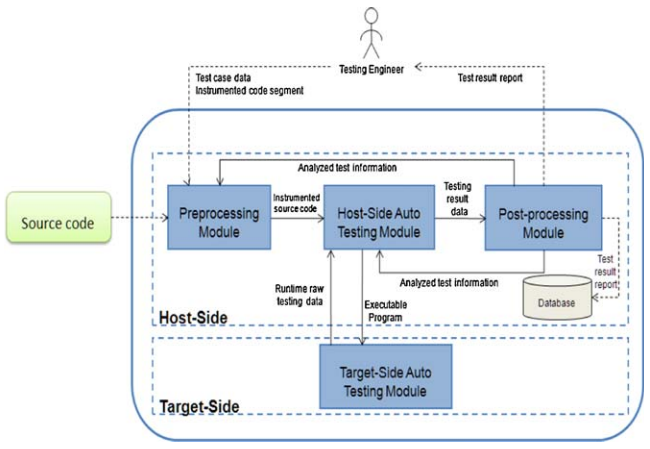


Fig. 12. System modules.

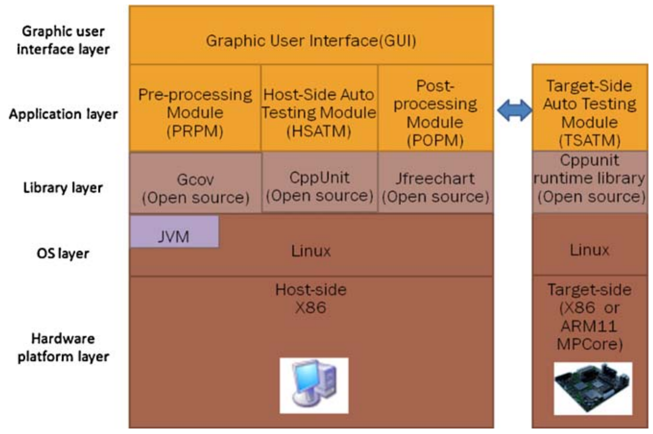

Graphic user interface layer
Application layer
Library layer
OS layer
Hardware platform layer
Graphic User Interface(GUI)
Pre-processing Module (PRPM)
Host-Side Auto Testing Module (HSATM)
Post-processing Module (POPM)
Target-Side Auto Testing Module (TSATM)
Gcov (Open source)
CppUnit (Open source)
Jfreechart (Open source)
Cppunit runtime library (Open source)
JVM
Linux
Linux
Host-side X86
Target-side (X86 or ARM11 MPCore)

Fig. 13. System architecture layers.

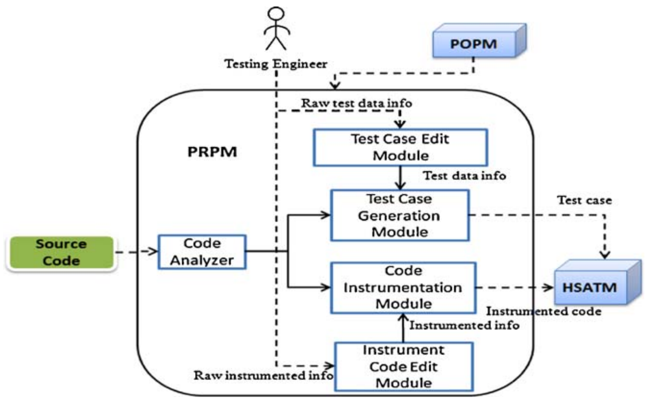


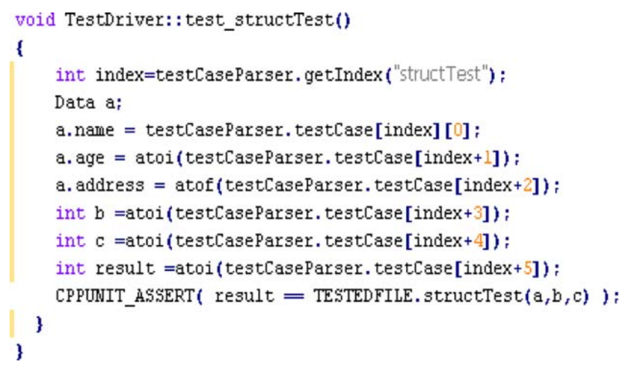
Fig. 15. PRPM module.

on host-side. Target-side platforms can be either X86 or ARM11 MPCore. The 2nd is OS layer. Linux operation system is used on both sides. The 3rd is the library layer, on which GCOV, CppUnit, and Jfreechart (JFreeChart) are adopted. The 4th which is implemented by C/C++ and Java is application layer constructed by ATEMES system modules. The 5th which is implemented by Java is the host-side graphic user interface layer for presenting test result and inputting test data.

4.3. Pre-processing testing module

PRPM manages the pre-processing testing. The module undertakes tasks including parsing source code, and automatically generating test case and test input data. The module also can automatically instrument testing code segment to target program so that raw data can be collected during the execution of target program and is reported back to host-side during runtime. The instrumented testing code segment can support functionalities of multi-round testing, collecting multi-core utilization, and controlling parallelism degree for multi-core program. Fig. 14 displays example of automatically generated test case framework.

PRPM is responsible for receiving target program. Source code of target program is parsed and tokenized through Code Manager. CodeParser extracts program information such as function name, parameter, and keyword to UnitTestCaseGenerator for generating test case automatically. Further, CodeParser locates the source code segment to be instrumented for InstrumentCodeEditor to generate the intended instrumented code such as the collection of multi-core utilization code segment. Through a user interface, test



```
void TestDriver::test_structTest()
{
    int index=testCaseParser.getIndex("structTest");
    Data a;
    a.name = testCaseParser.testCase[index][0];
    a.age = atoi(testCaseParser.testCase[index+1]);
    a.address = atof(testCaseParser.testCase[index+2]);
    int b =atoi(testCaseParser.testCase[index+3]);
    int c =atoi(testCaseParser.testCase[index+4]);
    int result =atoi(testCaseParser.testCase[index+5]);
    CPPUNIT_ASSERT( result == TESTEDFILE.structTest(a,b,c) );
  }
}
```

Fig. 14. Generated test case framework example.

engineer can also input test input data, test result data, and specially requested instrument code segment semi-automatically. Fig. 15 shows the module structure. Fig. 16 displays class diagram.

Fig. 17 illustrates PRPM sequence diagram. Purpose and steps are detailed as follows:

Scenario 1:
Purpose: generating test case automatically
Step 1: Code Manager reads target program code.
Step 2: CodeParser parses and extracts all function information from target program, including function name, parameter type, internal structure flow of program, and other relevant data.
Step 3: UnitTestCaseGeneratror analyzes information extracted by step 2.
Step 4: UnitTestCaseGeneratror generates test data after completing analyzing target program.
Step 5: UnitTestCaseGeneratror generates test case based on class method and output to files.

4.4. Host-side testing module

Automatic testing is completed by two modules, the HSATM on the host-side (workstation), and the TSATM on the target-side (embedded hardware platform). Modules are shown in Fig. 12.

HSATM is functioned to automatically generating test driver based on CppUnit library. Test driver and instrumented source code are automatically cross-compiled into target-side executable files under the module. The executable files and test input data are then passed to target-side for execution. Fig. 18 shows the module structure. Fig. 19 displays class diagram.

Figs. 20 and 21 illustrate HSATM sequence diagram. Purpose and steps are detailed as follows:

Scenario 1:
Purpose: generating test driver automatically
Step 1: TestDriverGenerator reads test case information generated by PRPM.
Step 2: TestDriverGenerator generates test driver based on the parsed test case information.
Step 3: AutomaticTestHost reads instrumented source code generated by PRPM.
Step 4: AutomaticTestHost compiles instrumented source code and test driver into executable files of target-side.
Step 5: AutomaticTestHost passes executable file to TSATM for performing testing.
Step 6: HSATM receives test log when testing is completed.
Step 7: Test log is passed to POPM.

Scenario 2:
Purpose: automatically measuring CPU utilization of target program during execution and presenting result dynamically.
Step 1: AutomaticTestHost generates test driver program to automatically measure CPU utilization
Step 2: AutomaticTestHost reades instrumented source code generated by PRPM.
Step 3: AutomaticTestHost compiles instrumented source code and test driver into executable files for target-side and passes to TSATM for testing.
Step 4: AutomaticTestHost receives CPU utilization returned from TSATM.
Step 5: AutomaticTestModule of Host-side runtime passes CPU utilization to POPM for presentation of testing result.
Step 6: Repeat step 4 and step 5 until testing is completed.

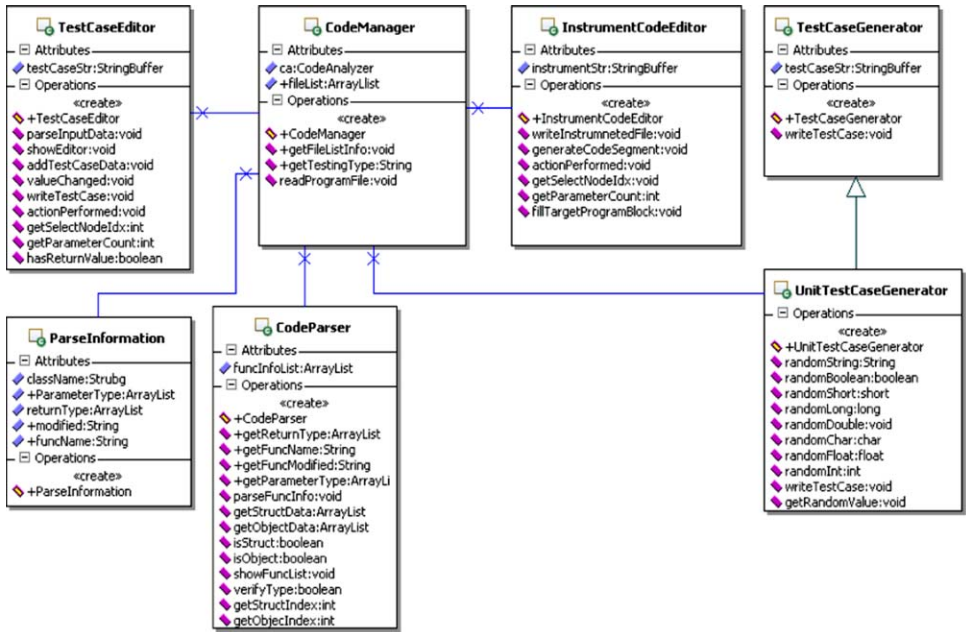


Fig. 16. PRPM class diagram.

4.5. Target-side testing module

The major task of TSATM module is to trigger test driver to execute or terminate testing tasks, to monitor profiler to collect testing data, and to return collected testing raw data to host-side during runtime. This module executes on the embedded hardware platform. Target-side Test Controller is used to trigger test driver, perform Profiler Manager, and exchange runtime message with HSATM. Test driver dynamically executes testing task based on test input data from input files or from automatically generated by host-side during runtime. When target program is under testing, Profiler Manager would trigger Performance Profiler and Test Coverage Profiler to collect runtime data. Output result and data collected during testing would be recorded into test log memory or files. Test log is finally returned to Test Log Manager of POPM for subsequent analysis and result presentation. Fig. 22 shows the module structure. Fig. 23 displays class diagram.

Fig. 24 illustrate TSATM sequence diagram. Purpose and steps are detailed as follows:

Scenario 1:

Purpose: automatic multi-round testing

Step 1: TargetSideTestController receives related testing program (instrumented executable target program, test driver) from HSATM.

Step 2: TargetSideTestController receives related testing data (test input data) from HSATM.

Step 3: TargetSideTestController triggers test driver to perform testing.

Step 4: TargetSideTestController triggers test driver to call instrumented executable code to perform testing, and repeat testing based on test case.

Step 5: TargetSideTestController triggers Profile Manager to collect testing data.

Step 6: Profiler continues collecting testing result.

Step 7: Profiler writes the collected testing data and test result data into test log file.

Step 8: TargetSideTestController passes test log to HSATM.

Step 9: Repeating step 4 to step 8 until test engineer stops testing.

4.6. Post-processing testing module

POPM manages analyzing and processing testing result. This module is composed of 3 parts, test result presentation module,

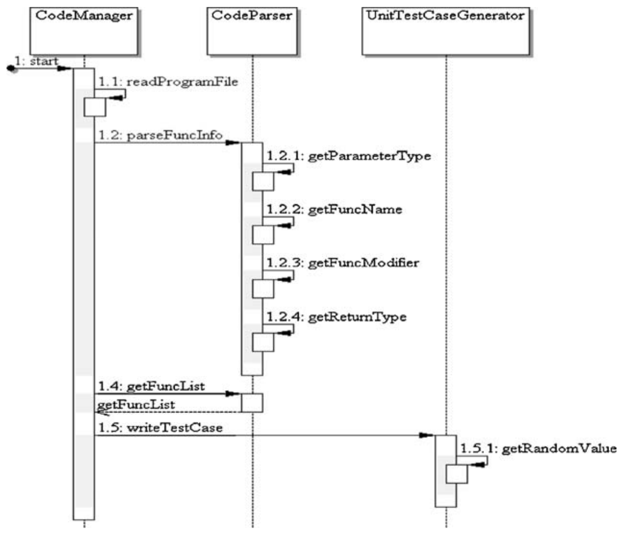


Fig. 17. Sequence diagram for automatically generating test case.

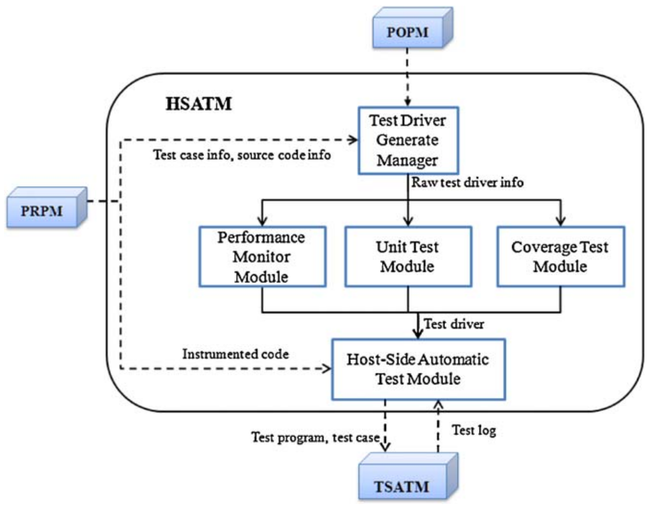


Fig. 18. HSATM module.

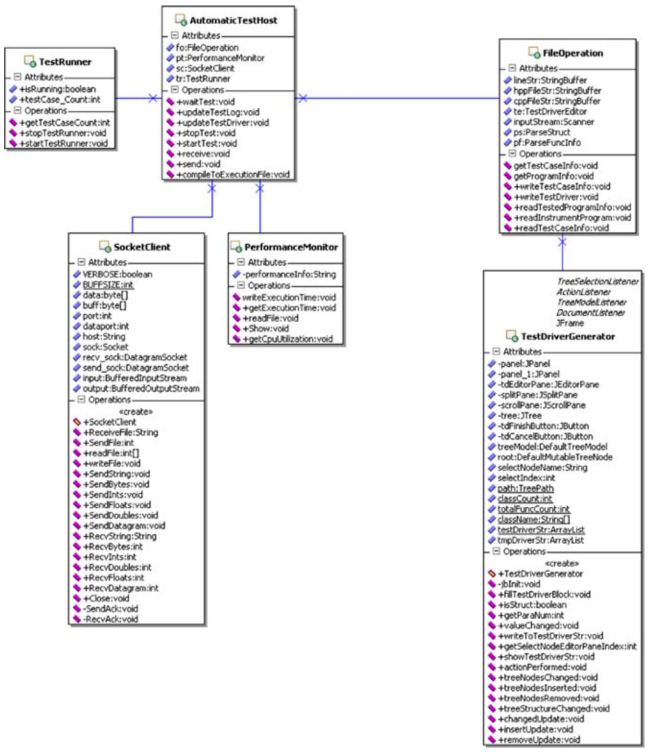


Fig. 19. HSATM class diagram.

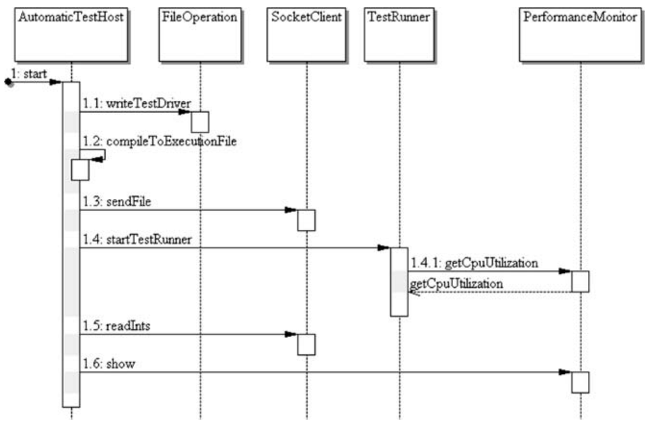


Fig. 21. Sequence diagram for automatically measuring CPU utilization.

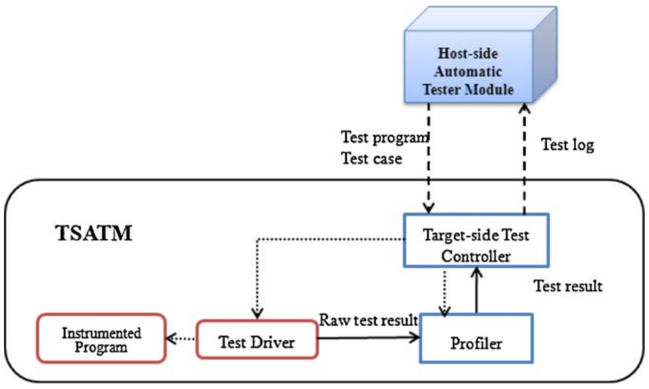


Fig. 22. TSATM module.

test log analyzer and test log manager. The collected testing result is categorized into sub-items and presented in textual or graphical report. Based on testing demand, Test log manager would perform various parsers to runtime analyze test log data such as coverage test log parser, unit test log parser, and performance monitor log parser. Feedback of the analyzed result would be passed dynamically to PRPM and HSATM as reference for next testing. During testing, the module allows test engineer to runtime observe result of coverage test or performance monitor. Test result presentation module would present testing result in textual description or with graphical interface. Fig. 25 shows the module structure. Fig. 26 displays class diagram. Fig. 27 shows user interface of test result visualization.

Fig. 28 illustrate POPM sequence diagram. Purpose and steps are detailed as follows:

Scenario 1:
Purpose: presenting graphic test result dynamically
Step 1: TestLogParser reads test log from TSATM.
Step 2: TestLogParser gets the catalogue of test result and writes the log into files based on different catalogues including testResultInfo, testLogInfo, and TestDriverLog.
Step 3: PaintChart reads testResultInfo when testing is completed.
Step 4: PaintChart presents different visualized testing report according to testing demands.

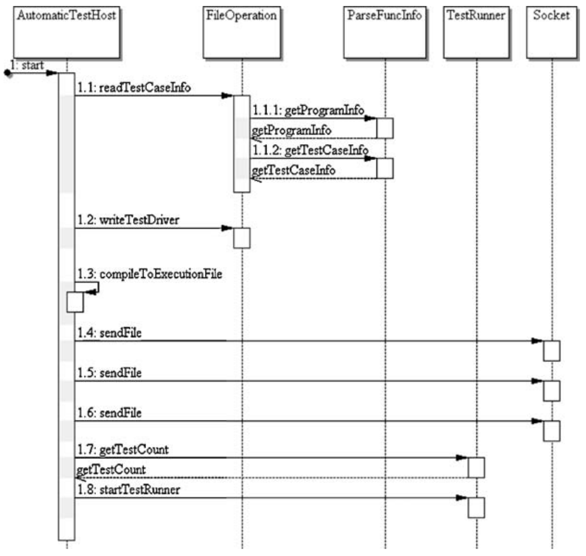


Fig. 20. Sequence diagram for automatically generating test driver.

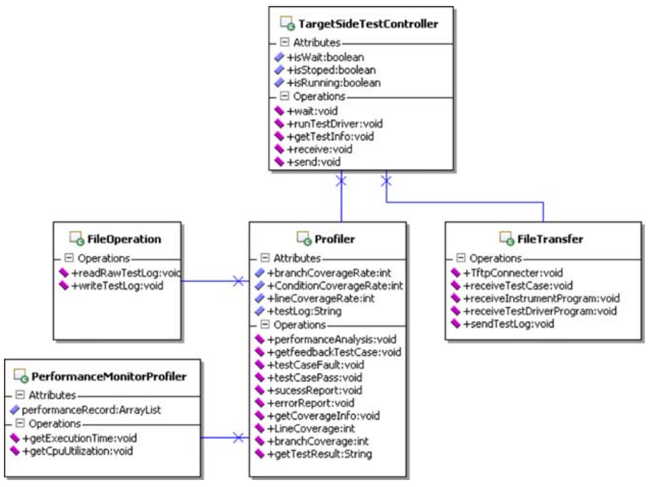


Fig. 23. TSATM class diagram.

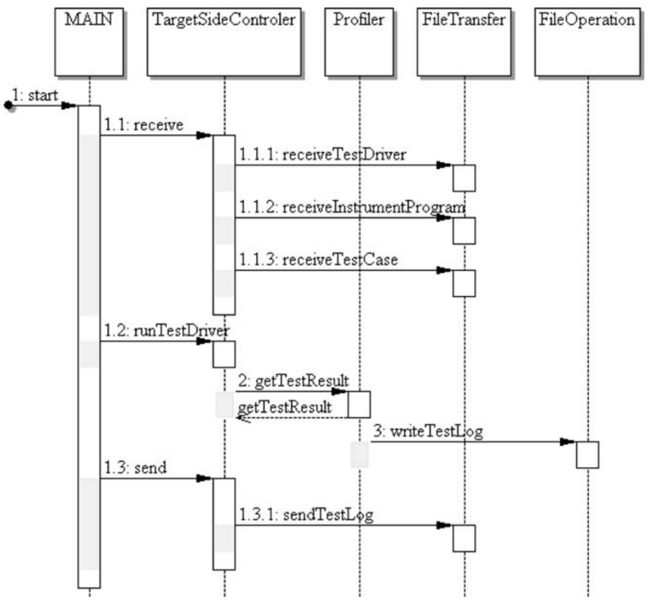


Fig. 24. Sequence diagram for automatic multi-round testing.

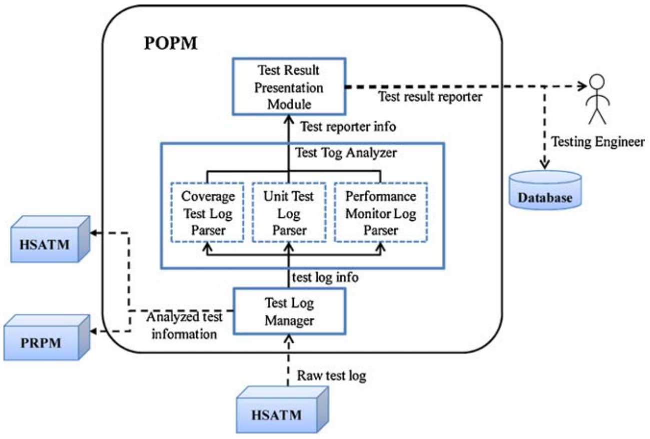


Fig. 25. POPM module.

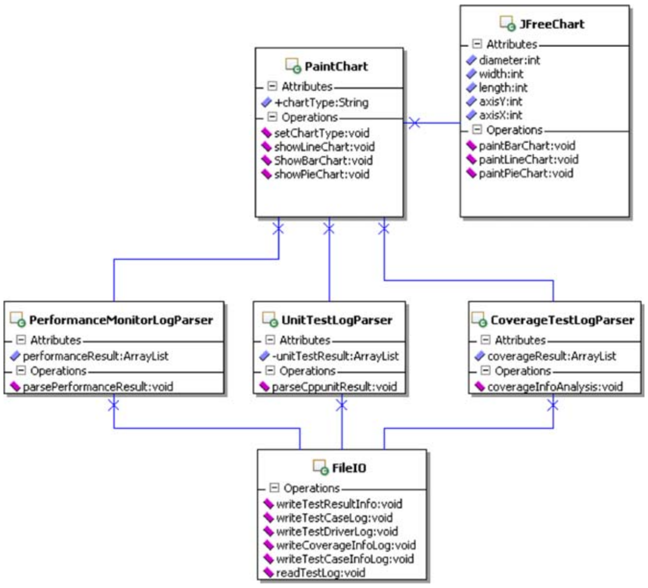
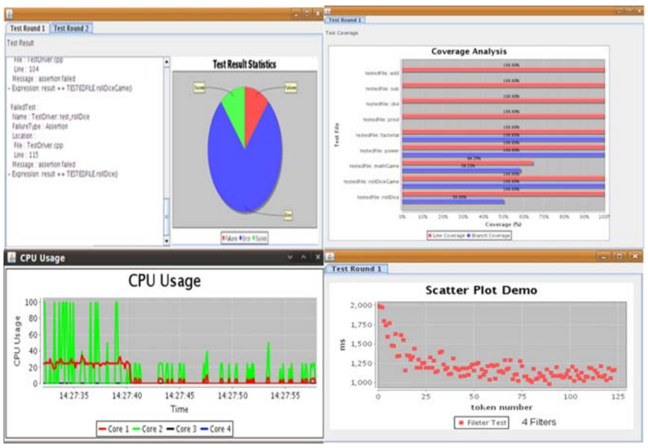


Fig. 26. POPM class diagram.



File : TestDriver.cpp
Line : 104
Message : assertion failed
- Expression: result == TESTEDFILE.rollDiceGame()

FailedTest :
Name : TestDriver::test_rollDice
FailureType : Assertion
Location :
File : TestDriver.cpp
Line : 115
Message : assertion failed
- Expression: result == TESTEDFILE.rollDice()

Fig. 27. Test result visualization.

4.7. Multi-round testing scenario

This section demonstrates multi-round automatic testing scenario which is illustrated in UML sequence diagram as shown in Fig. 29.

The scenario includes a series of tasks: parsing source code, generating intended data such as test input data, instrument code segment, test case, and test driver, executing testing automatically, collecting and parsing test log file, and executing automatically the next round testing based on the parsed result till testing conditions are met. Actions are detailed as follows:

Step 1: PRPM reads source code.
Step 2: Based on demand script file, PRPM parses the read source code, and proceeds getting program function name, parameter, program internal structure, and related data.
Step 3: PRPM analyzes the extracted data gathered from step 2. Test case is either generated automatically after analysis, or new test case (test input data) is generated based on the test result provided by POPM during testing.
Step 4: PRPM processes the extracted gathered from step 3. Analysis is conducted according to the intended testing items. Instrumented source code is automatically generated after analysis.

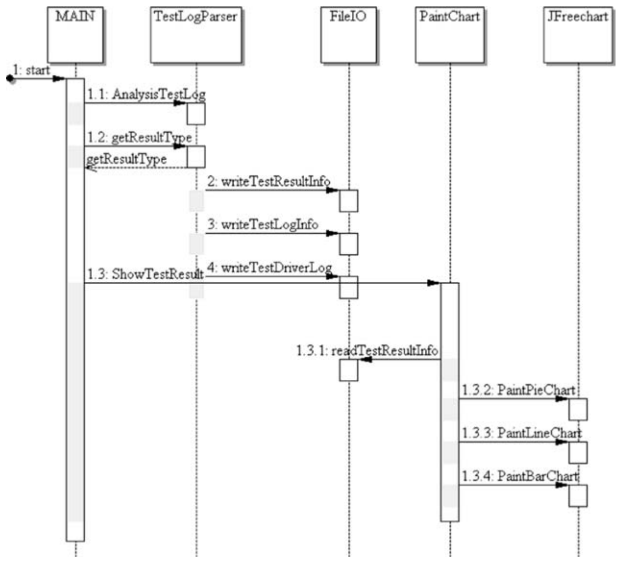


Fig. 28. Sequence diagram for presenting graphic test result.

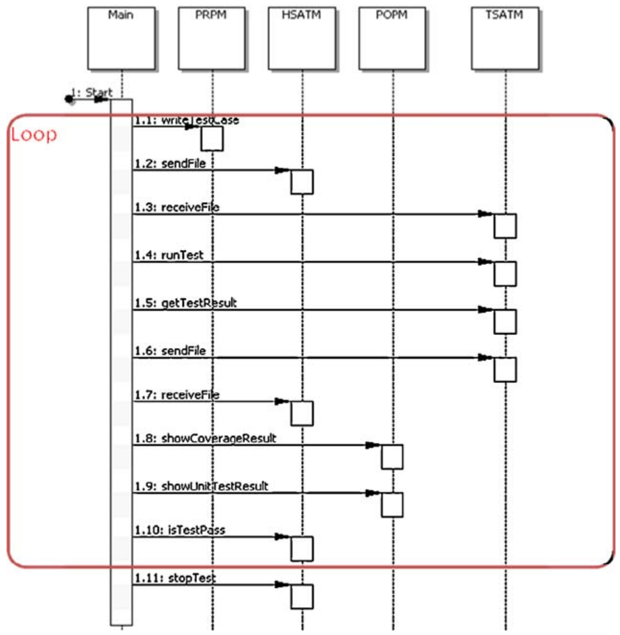


Fig. 29. Sequence diagram for multi-round testing.

Step 5: HSATM reads test case data generated from PRPM.
Step 6: HSATM automatically generates test driver based on the data collected from PRPM.
Step 7: HSATM reads the instrumented source code generated from PRPM.
Step 8: HSATM compiles the instrumented source code and test driver into executable files on target-side and uploads to TSATM.
Step 9: TSATM is triggered to execute automatic testing. It collects the test result logs and returns to HSATM.
Step 10: HSATM receives the test result logs executed by TSATM and passes to POPM module for analysis.
Step 11: Repeat step 3 to step 10 until testing conditions are met.
Step 12: POPM presents test result to test engineer.

5. Experiment

To demonstrate the functionalities of ATEMES, data structure, array multiplication and image processing programs are selected as the target programs. ATEMES is executed on the ARM11 multi-core platform. Automatic multi-round testing mechanism is incorporated to conduct coverage testing experiment, unit testing experiment, multi-core utilization monitoring experiment, parallelism benchmark experiment, and usability assessment experiment. Software and hardware platforms for the experiments are detailed as follows:

a. Host-side hardware platform: Intel® Core^{TM} 2 Duo CPU P8400.
b. Host-side OS platform: Linux UBUNTU 9.10.
c. Target-side hardware platform: The platform baseboard for ARM 11 MPCore with 4 ARM11MPCore CPUs.
d. Target-side OS platform: Linux Kernel 2.6.24.

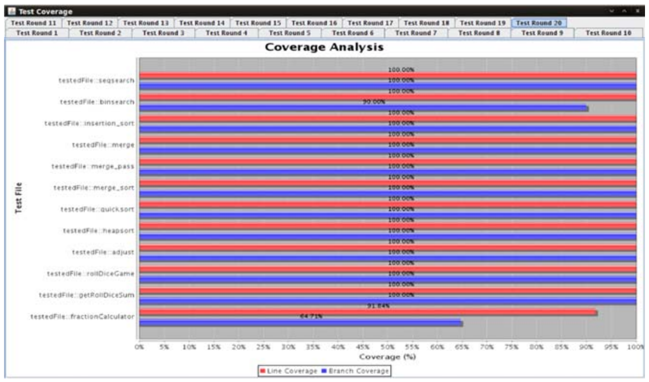


Fig. 30. Line and branch coverage testing result.

5.1. *Coverage testing experiment*

Thirteen programs of data structures were selected as target program. Most of the input data types were array structure type. Random testing method was adopted for the testing environment where test input data was generated. For each function, each round generated 3 sets of input data, and each function was arranged to execute 20 rounds of testing. During runtime, test engineer can observe each round result and final result of coverage testing (Fig. 30). The system can record history data for each round, including line coverage, branch coverage, executing time, and failed test case data.

5.1.1. *Experiment result*

As shown in Tables 1 and 2, and Fig. 30, for insertion_sort(), 100% line coverage is covered from the 1st round to the 5th round. 100% branch coverage is covered from the 1st round to the 5th round. For other functions, similar coverage is likely to reach up to 100%.

Table 1
Line coverage.

Function	Round										
	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	20th
seqsearch	92%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
binsearch	79%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
insertion_sort	91%	91%	91%	91%	100%	100%	100%	100%	100%	100%	100%
merge	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
merge_pass	92%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
merge_sort	92%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
quicksort	26%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
heapsort	92%	92%	92%	92%	92%	92%	92%	92%	92%	100%	100%
adjust	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
rollDiceGame	72%	83%	83%	83%	83%	94%	94%	94%	100%	100%	100%
getRollDiceSum	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
fractionCalculator	55%	63%	80%	92%	92%	92%	92%	92%	92%	92%	92%
Execute time (ms)	958	907	939	889	980	891	994	926	1006	977	909

Table 2
Branch coverage.

Function	Round										
	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	20th
seqsearch	83%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
binsearch	60%	90%	90%	90%	90%	90%	90%	90%	90%	90%	90%
insertion_sort	90%	90%	90%	90%	100%	100%	100%	100%	100%	100%	100%
merge	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
merge_pass	88%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
merge_sort	75%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
quicksort	17%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
heapsort	83%	83%	83%	83%	83%	83%	83%	83%	83%	100%	100%
adjust	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
rollDiceGame	64%	82%	82%	82%	82%	91%	91%	91%	100%	100%	100%
getRollDiceSum	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
fractionCalculator	29%	41%	53%	65%	65%	65%	65%	65%	65%	65%	65%
Execute time (ms)	958	907	939	889	980	891	994	926	1006	977	909

Table 3
Unit test result information.

Function name	Test case				
	0	1	2	3	4
getSubString	Success	Exception out of range	Success	Success	Exception out of range
electricityTraiff	Success	Assertion failed	Success	Assertion failed	Success
factorial	Success	Success	Assertion failed	Assertion failed	Success
insertionsort	Success	Success	Segment fault	Segment fault	Segment fault
seqsearch	Success	Success	Success	Success	Success
fractionCalculator	Success	Assertion failed	Success	Success	Success

Success: result corresponds to expected value; Failure: result does not correspond to expected value, for example, assertion failed; Error: exception occurred, for example, segment fault and exception out of range.

However, coverage of some functions could hardly reach to 100%. For fractionCalculator(), line coverage and branch coverage in the 4th round reached to 92% and 65%. Coverage of which could hardly increase beyond that. Possible interpretation could be that the algorithm adopted is random testing method (Bird and Munoz, 1983; Chen et al., 2009; Claessen and Hughes, 2000). Higher coverage could be foreseen if other algorithms were adopted, for example, adaptive random testing (ART) (Chen et al., 2004b; Chan et al., 2006; Pacheco et al., 2007; Ciupa et al., 2008; Chen et al., 2010) .

5.2. Unit testing experiment

For unit testing, ATEMES can support automatic and semi-automatic testing. This experiment adopted semi-automatic unit testing method. Six homework programs of data structures course were used as the target programs. After analyzing source code done by ATEMES testing environment, target functions were identified and located automatically, and the corresponding input dialog of test data was shown. When test engineers input test data and test result, the ATEMES can automatically generate test case programs and test driver programs, conduct automatic cross-compilation, upload test package to ARM 11 MPCore platform, and undertake testing. Test result logs can be transferred back to host-side for simultaneous analysis during runtime.

5.2.1. Experiment result

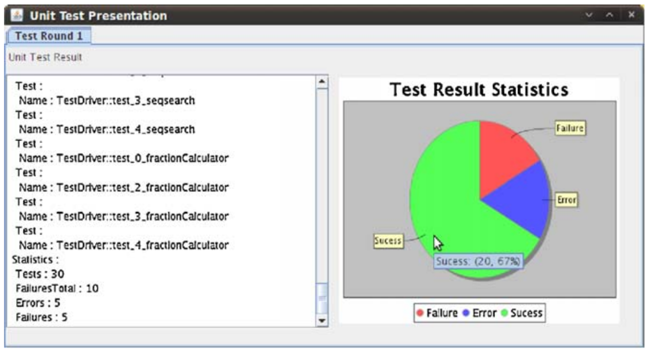
As shown in Fig. 31 and Table 3, left side of Fig. 31 presents test result in text, including data of success, failure, and error test case. On the right side of Fig. 31, unit test result is shown in pie chart. Summing up statistics from the experiment, total success test cases are 67% (20 test cases), failure test cases are 16.5% (5 test cases), and error test cases are 16.5% (5 test cases). *Success* in the experiment means executed result of target function corresponds to expected value. *Failure* means executed result of target function does not correspond to expected value, for example, assertion failed. *Error* means exception occurred during execution, for example, segment fault and exception out of range.

5.3. Multi-core utilization monitoring experiment

This experiment adopted a multi-thread program with the multiplication of 2 dimensions array, which are 256×128 and 128×256. After analyzing source code done by ATEMES, target functions were identified and located automatically. The ATEMES can automatically instrument code to monitor target program execution time, conduct automatic cross-compiling, upload test package to ARM 11 MPCore platform, and perform testing. During program runtime on target-side, the utilization of each core can be monitored. Test result logs can be transferred back to host-side for simultaneous analysis during runtime.

5.3.1. Experiment result

The result of experiment is shown in Fig. 32 and Table 4. Fig. 32(a)–(d) shows performances at different timing during executing of 2 dimensions array multiplication. Performance of Core



```
Test :
 Name : TestDriver::test_3_seqsearch
Test :
 Name : TestDriver::test_4_seqsearch
Test :
 Name : TestDriver::test_0_fractionCalculator
Test :
 Name : TestDriver::test_2_fractionCalculator
Test :
 Name : TestDriver::test_3_fractionCalculator
Test :
 Name : TestDriver::test_4_fractionCalculator
Statistics :
 Tests : 30
 FailuresTotal : 10
 Errors : 5
 Failures : 5
```

Fig. 31. Pie chart of unit test result.

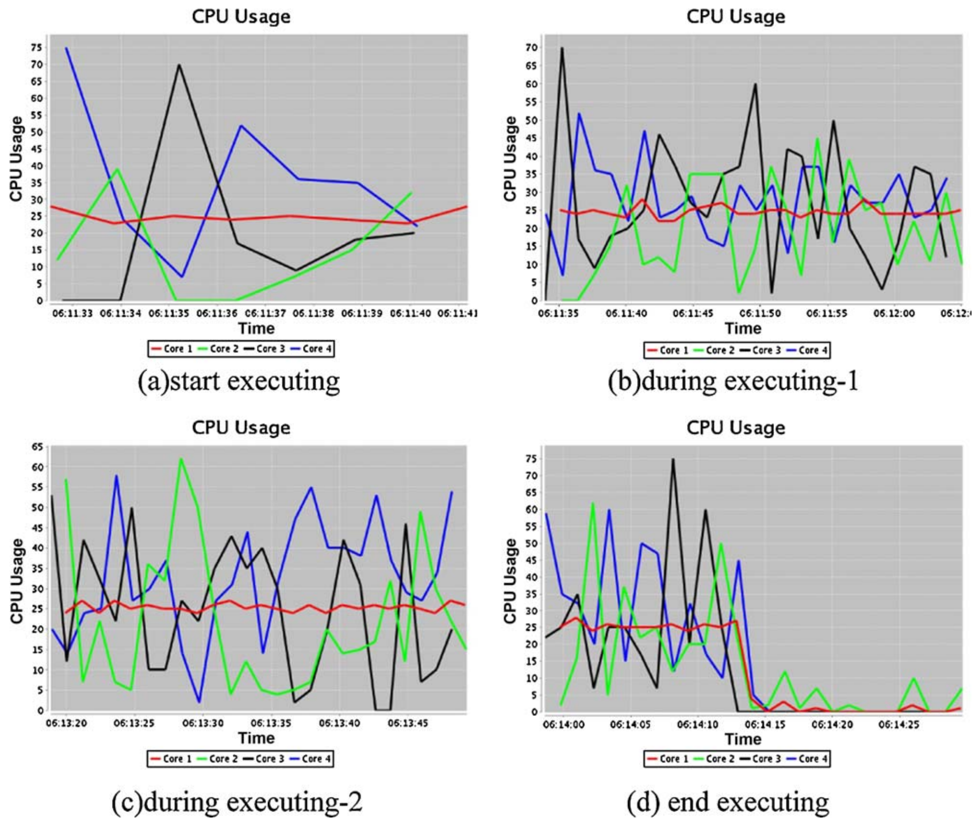


Fig. 32. Visualization of multi-core CPU utilization at different timing.

1 stays stable all the time. Core 2 has lower performance. Table 4 illustrates statistics information. In terms of average CPU utilization, Core 3 and core 4 have higher performance (29% and 31%). Core 1 performs the second (18%), and core 2 has the lowest 18% performance. Core 3 and Core 4 have the maximum performance value.

5.4. Parallelism benchmark experiment

Adopting ATEMES, this experiment aims to measure how execution performance of pipeline program using TBB library can be influenced under different CPU core numbers with of varied token numbers.

The experiment used parallel image program as target program adopting Intel TBB library. Stages of image effect processing were processed by TBB pipeline parallel technology. Image size is 1280 * 1024. The main tasks are gradient and smooth images. Pipeline is divided into four stages: read image, gradient, smooth, and output image. Read image and output image is processed by sequential execution, while gradient and smooth is done by parallel execution. ATEMES could generate test case for pipeline token number automatically. Each test case has 100 data sets, which are token numbers from 1 to 100. Multi-core utilization performance testing can be automatically executed multi-rounds. The system then concludes a suggested range of pipeline token numbers and presents utilization of each CPU core numbers.

Table 4
Multi-core CPU utilization.

Core No.	Average (%)	Max (%)	Min (%)
Core 1	25.05969	28	22
Core 2	18.97810	62	0
Core 3	29.93431	76	0
Core 4	31.05839	75	0

5.4.1. Experiment result

Fig. 33 shows that the more CPU core number there are, the more efficient target program is. Take pipeline token number for instance, when token number is smaller than 5, target program is less efficient. However, target program is more efficient when token number is larger than 5. As for TBB pipeline, when token numbers reach to a certain point, not only little efficiency is raised but also system burden is increased. Following is the execution illustration under different CPU core numbers with varied token numbers, and our suggested range of token numbers (Table 5).

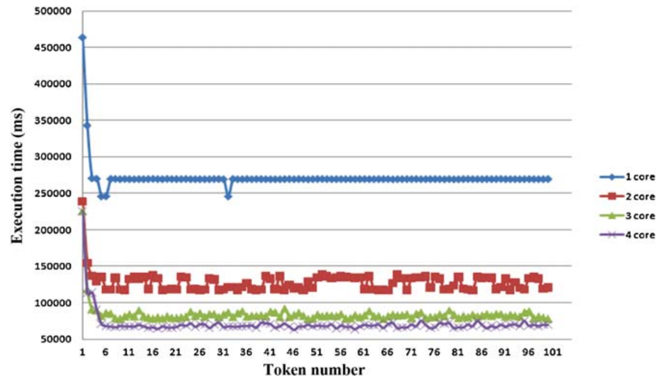


Fig. 33. TBB pipeline execution performance measurement under 1–4 CPU cores.

Table 5
Suggested token number under 1–4 CPU core.

Core numbers	1	2	3	4
Token numbers	5–9	6–10	7–11	8–12

Table 6
Item reliability analysis.

	Mean	Variance	Cronbach's alpha	Std. Cronbach's alpha	Number of items
Value	4.533	.012	.928	.927	44

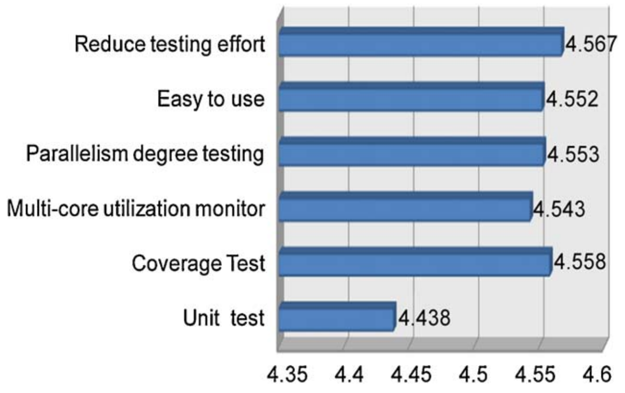


Fig. 34. Mean value of each dimension.

5.5. Usability assessment experiment

To test our system usability, in terms of efficiency and acceptability, we asked 30 graduate students from computer and information science department as subjects to engage user trial testing and fill in post-testing questionnaire.

Experiment arrangement: Subjects use ATEMES to perform a series of testing task on unit testing, coverage testing, multi-core utilization performance monitor, and TBB pipeline parallelism benchmark experiment.

Target program: Subjects are provided with sample programs to be edited as wished. Sample programs include data structure, array multiplication, and image processing programs.

Duration: Two hours.

Pearson's correlation coefficient two-tailed test is adopted in this research. Total questionnaire items are forty four, divided by functional and non-functional categories of 6 dimensions. Four functional dimensions, including *unit test*, *coverage test*, *multi-core utilization monitor*, and *TBB pipeline parallelism degree*, are used for testing operation precision of the system. Two non-functional dimensions are employed for checking user's acceptability toward *easy to use* and *reduce testing effort* (time-saving for writing testing program).Questionnaire reliability adopts Cronbach's alpha (α) to test scale's internal consistency. Table 6 indicates Cronbach's alpha (α) of total item is 0.928, which is larger than 0.7. Reliability is thus credited.

Fig. 34 displays results of questionnaire analysis. Total mean is above 4.5(highest point is 5, and lowest point is 1). The result suggests that subjects are satisfied with the questionnaire items and functions supported by the system.

For *unit test dimension*, results of questionnaire analysis confirm functions provided fulfill subjects' demands. When unit test is in process, test data, test case, and test driver required by unit testing can be generated automatically. In addition, the system can help checking testing result for subjects so as to save time and enhance task efficiency.

For *coverage test dimension*, subjects recognize system's advantages in supporting automatically generated test data, test case, test driver, and the dynamically visualized presentation of result during runtime. Coverage value collected from multi-round coverage tests were either reached to expected level, which can save subjects' time and improve working efficiency.

For *multi-core CPU utilization monitor dimension*, subjects acknowledge multi-round cross-testing functions and presenting visualized result dynamically. Subjects consider it improve embedded software performance by allowing them to observe how their developed embedded software is actually being executed on the embedded platform.

For *TBB pipeline parallelism dimension*, subjects approve system's merits in supporting generating test driver automatically, executing precise cross-testing, presenting CPU utilization dynamically, and suggesting better token number when TBB pipeline performance monitor is executing. Subjects grant the system is effective in saving time on testing software parallelism performance and can increase efficiency on editing and adjusting embedded software for better parallelism.

For *easy to use dimension*, results indicated that users find the operation environment friendly. The low user threshold makes application easy for even the first time user. The accessibility of the system increased subjects' interests in software testing domain. Subjects are willing to use the software and would put it on their recommendation list.

For *reduce testing effort*, outcome showed that whether it is on the generation of target data, testing program code, executing various testing and monitor, or the giving of recommended value for parallel program and parallel parameter, subjects' workloads are reduced, time is saved, and efficiency is increased.

To conclude, ATEMES facilitates test engineer in multifold.

In terms of reducing workloads, automatic testing functionalities including the automatically generated test data, test case, and test driver save test engineer's energy from inputting and checking numerous detailed but crucial data. The dynamically visualized presentation of result during runtime not only keeps test engineer informed of how his embedded software is actually being executed on the embedded platform, but also makes modification easy.

In terms of increasing efficiency, execution of cross-testing, presenting CPU utilization dynamically, and suggesting better token number when TBB pipeline performance monitor is executing support test engineer with effective tool for testing and adjusting for better embedded software parallelism performance. The low user threshold and the accessibility the system offered on generating test data, testing program code, performing various testing functions and monitor, and giving recommended value for parallel program and parallel parameter all help increasing test engineer's efficiency.

According to the collected result, a prevalent high acceptability is reflected on the experiment. Two inferences can be concluded. For one, functionality supported by ATEMES can satisfy subjects' demands. For the other, under-graduate students are less exposed to testing task. Since subjects are less familiar with using automatic testing tool to develop software system, especially on enhancing and adjusting multi-core embedded software performance and parallelism, they are likely to be intrigued by the automatic functionality supported by ATEMES without their efforts to write any testing program.

There are, however, suggestions for ATEMES to improve: (1) software user interface is not esthetically appealing; (2) stability needs more improvement; (3) developed result is expected to be saved on the website; (4) supply of operation instruction file is required.

5.6. Discussion

As results of experiments 1–5 indicated, the fact that our system ATEMES can enhance testing efficiency and earn prevailing acceptability shows to be an effective testing tool.

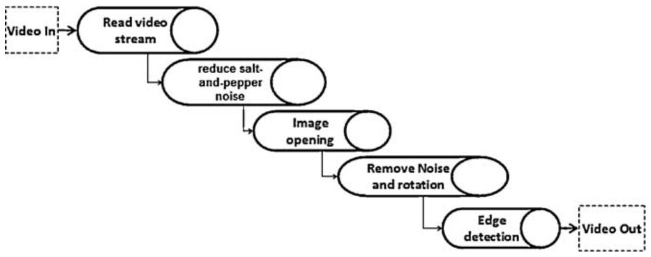


Fig. 35. Pipeline design description of target program.

According to results from experiments 1 and 2, efforts demanded from test engineer can be greatly reduced by the functionalities of automatically generated test input data, test case programs and test driver programs. Likewise, mechanism supported by ATEMES also provided efficient solutions including multi-round cross-testing between workstation and embedded system.

According to results from experiments 3 and 4, ATEMES can facilitate test engineer with multiple tasks. Test engineer can remotely monitor target program performance on multi-core embedded platform. When executing multi-round embedded parallel program, the incorporation of different CPU core numbers with various token numbers can help obtaining better parallel parameter. The method not only can alleviate resource limitations but also can improve embedded program performance and parallelism.

According to results from experiment 5, subjects approve ATEMES's functionalities in saving testing time, increasing testing efficiency, and credit the system with high acceptability.

5.7. Response time experiment for parallel program to handle different interrupt intervals with different core numbers

The experiment aims at measuring the response time of target program(parallel program) with different interrupt intervals under embedded platform using different core processor numbers. Our tool is employed to automatically generate test driver and different test data, namely, core processor number, interrupt time interval, and interrupt times. Automation in the context refers to tester must be capable of identifying the function name of interrupt handler under the testing tool. With the multi-round testing functionality, the testing tool executes testing automatically and repetitively. Further, the tool can also simulates embedded platform with different core processor numbers to handle different interrupt time intervals. Based on each time interval, the tool would generate 20 interrupt events sequentially for target program to measure the response time of each interruption.

Using TBB library, target program executes actions with TBB pipeline. Main program of target program would execute video processing based on video streaming data. The frame size of each streaming video is 512 * 384. Actions of image processing are divided into six steps: (1) Read video stream; (2) Reduce salt-and-pepper noise; (3) Image opening (Erosion and Dilation); (4) Remove Noise and Rotation; (5) Edge detection (Laplacian); and (6) Write video stream, among which, step 2 through step 6 are being under parallelized processing by TBB pipeline stage. Whenever interrupt event occurs, main program would call interrupt handler to execute image thinning. Fig. 35 displays the pipeline design description of target program. Fig. 36 shows the executing flow of tested program.

Figs. 37–40 present testing results generated automatically by the testing tool, core processor numbers are 1–4. As can be seen from the results, when target program uses embedded platform with 1 core processor, the system can handle the workload, if interrupt time interval is greater than or equal to 4 s. Similarly, when target program uses embedded platform with 2 core processors, the workload is also manageable, if interrupt time interval is greater than or equal to 2 s. When using embedded platform with 3 or 4 core processors, video processing execution is smooth and stays intact from interrupt. The existing test results only reflect how current target program responded to workload. Test results may vary accordingly to different workloads posed on main program and interrupt handler. In general, repetitive testing is required before assessing outcome of what lower budget hardware should embedded software use to properly handle external interrupt workload. In other words, the result is obtained with the cost of massive time and efforts. Instead, our testing tool provides an easier option to detect under what embedded platform should embedded software operate to handle external interrupt workload.

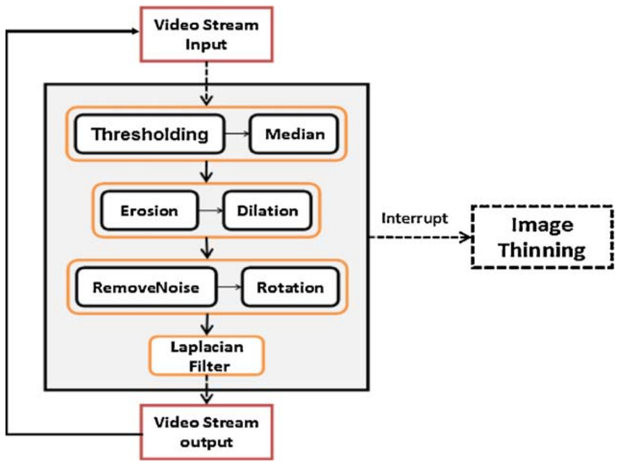


Fig. 36. Executing flow of tested program.

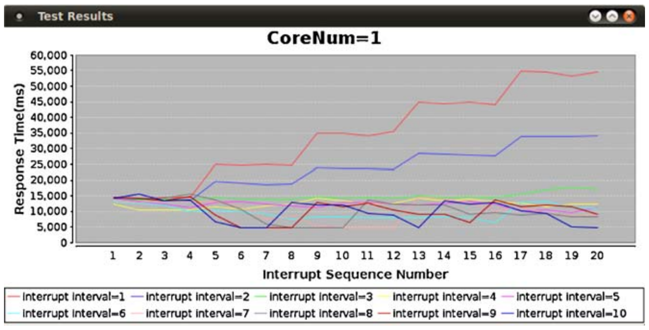


Fig. 37. Response time on different interrupt intervals using 1 core processor.

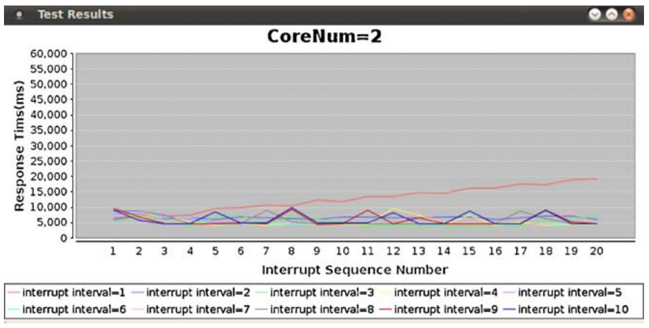


Fig. 38. Response time on different interrupt intervals using 2 core processors.

Table 7
Functionality comparison chart between our system ATEMES and relevant techniques.

Features	Tools			
	ATEMES	Erik Putrycz	Alexei Alexandrov	William Thies
Pure software solution	Yes	Yes	Yes	Yes
Automatic multi round testing	Yes	None	None	None
Automatic generate test driver	Yes	None	None	None
Automatic generate test data/cases	Yes	Yes	None	None
Supporting Complexity data structure	Yes	None	None	None
Supporting parallel /TBB performance testing	Yes	None	Yes	Yes
Test result visualization	Yes	Yes	Yes	None
Run time test result presentation	Yes	None	None	None
Cross Platform Testing	Yes	Yes	None	None
Supporting in Multi-Core Platform	Yes	None	Yes	Yes
Supporting interrupt performance test	Yes	None	None	None
Supporting Coverage test	Yes	None	None	None

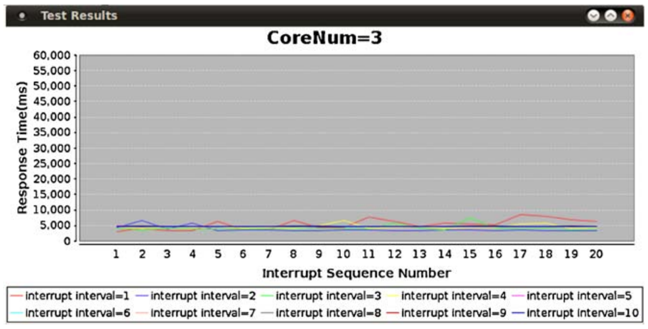


Fig. 39. Response time on different interrupt intervals using 3 core processors.

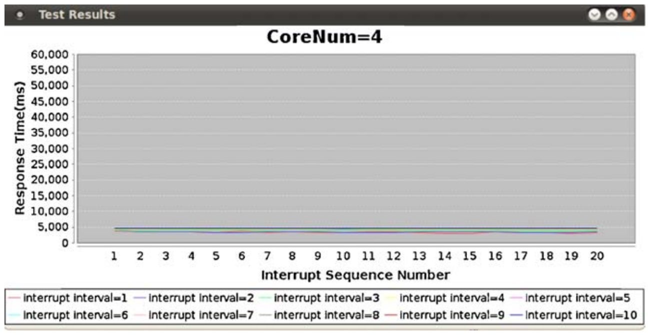


Fig. 40. Response time on different interrupt intervals using 4 core processors.

6. Conclusion

This paper recognized the need for proper automatic testing tool and developed an automatic cross-testing environment (ATEMES) to support multi-core embedded software testing.

Table 7 is the functionality comparison chart between our system ATEMES and Erik Putrycz, Alexei Alexandrov, and William Thies. The features of the various algorithms implemented by this test tool set are listed and compared with those of the relevant testing suites. The significance of these algorithms over those of other systems is evident immediately.

The unique features of this technique not seen in other relevant techniques are highlighted in the table.

The ATEMES can not only automatically generate test data with primitive type, structure type, object type and array type but also generate CppUnit-based test case and test driver. The system can automatically initiate testing without having test engineer to edit any code, which can help reducing massive efforts demanded from test engineer. Functionalities supported by ATEMES are multifold.

With the automatic multi-round mechanism, unit testing and coverage testing can be implemented to save test engineer from massive repetitive tasks. With the cross-testing technology between host-side (workstation) and target-side (embedded system), factors resulted from embedded system resource restraints can be reduced for testing. With the cross-testing technology, test case, test driver and target program can be cross-compiled automatically and uploaded to target-side for automatic implementation. Target-side test log data including runtime data, output result, and data of each core utilization during runtime from target-side CPU can be passed to host-side for runtime analysis, results of which can also be visually presented.

Moreover, ATEMES can execute automatic multi-round performance testing over multi-core embedded software adopting Intel TBB library. The system can support locating recommended value for better parallel parameter token number, which not only allows embedded software parallelism but also facilitates computing capacity of multi-core CPU to operate more efficiently.

The results of measuring the response time of target program(parallel program) with different interrupt intervals using different core processor numbers on the ARM11 multi-core platform show that our constructed testing environment can not only reduce burdens of test engineer, but also enhance efficiency of multi-core embedded testing task.

Limitations remaining for our system to overcome are that unit testing still demands manual input to check result, and that coverage testing so far can only support line coverage and branch coverage.

The aim of this paper is to develop a more precise algorithm to automatically generate input test data, test case and test driver. In the future, more research should be undertaken adopting random testing method in the experiment. Based on our developed regression testing function, this study aims to expand the system to a fuller regression testing functionality with the integration of web technology. In addition, a broader scale of developing testing functions aiming at multi-core embedded software, such as multi-thread race condition testing and pipeline program efficiency testing, are strongly suggested.

Acknowledgement

This research was supported in part by National Science Council, Taiwan, ROC, under grants no. NSC 97-2218-E-142-001, NSC 98-2220-E-142-001, NSC 99-2220-E-142-001 and NSC 99-2220-E-029-001.

References

Bertolino, A.,2007. Software testing research: achievements, challenges, dreams. In: 2007 Future of Software Engineering (FOSE'07). IEEE Computer Society, Washington, DC, USA, pp. 85–103.

Bird, D.L., Munoz, C.U., 1983. Automatic generation of random self-checking test cases. IBM Systems Journal 22, 229–245.

Broekman, B., Notenboom, E., 2002. Testing Embedded Software. Addison-Wesley, London.

Chan, K.P., Chen, T.Y., Towey, D.P., 2006. Restricted random testing: adaptive random testing by exclusion. International Journal of Software Engineering and Knowledge Engineering 16 (4), 553–584.

Chen, T.Y., Kuo, F.C., Liu, H., 2009. Adaptive random testing based on distribution metrics. Journal of Systems and Software 82, 1419–1433.

Chen, T.Y., Kuo, F.C., Merkel, R.G., Tse, T.H., 2010. Adaptive random testing: the ART of test case diversity. Journal of Systems and Software 83, 60–66.

Chen, T.Y., Leung, H., Mak, I.K., 2004b. Adaptive random testing. In: Proceedings of the Ninth Asian Computing Science Conference (ASIAN'04), Lecture Notes in Computer Science, vol. 3321, pp. 320–329.

Cho, Y., Choi, J., 2008. An Embedded Software Testing Tool Supporting Multi-paradigm Views, Computational Science and its Applications – ICCSA 2008 , pp. 780–789.

Ciupa, I., Leitner, A., Oriol, M., Meyer, B.,2008. ARTOO: adaptive random testing for object-oriented software. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008). ACM Press, New York, NY, pp. 71–80.

Claessen, K., Hughes, J., 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ACM.

CppUnit. http://cppunit.sourceforge.net/.

Delamaro, M.E., Vincenzi, A.M.R., Maldonado, J.C.,2006. A strategy to perform coverage testing of mobile applications. In: Proceedings of the 2006 International Workshop on Automation of Software Test. ACM, Shanghai, China.

Gcov. http://gcc.gnu.org/.

Hailpern, B., Santhanam, P., 2002. Software debugging, testing, and verification. IBM Systems Journal 41, 4–12.

Hung, S.H., Huang, S.J., Tu, C.H., 2008. In: Shu-Jheng, H., Chia-Heng, T. (Eds.), New Tracing and Performance Analysis Techniques for Embedded Applications. , pp. 143–152.

JFreeChart. http://www.jfree.org/jfreechart/.

Kaner, C., Falk, J.L., Nguyen, H.Q., 1999. Testing Computer Software, 2nd ed. John Wiley & Sons, Inc., New York, NY, USA.

Ki, Y., Seo, J., Choi, B., La, K.,2008. Tool support for new test criteria on embedded systems: Justitia. In: Proceedings of the 2nd International Conference on Ubiquitous Information Management and Communication. ACM, Suwon, Korea.

King, J.C., 1976. Symbolic execution and program testing. Communications of the ACM 19, 385–394.

Lyu, M.R., Horgan, J.R., London, S., 1994. A coverage analysis tool for the effectiveness of software testing. IEEE Transactions on Reliability 43, 527–535.

Michael, J.B., Bossuyt, B.J., Snyder, B.B., 2002. Metrics for measuring the effectiveness of software-testing tools. In: Proceedings of 13th International Symposium on Software Reliability Engineering, 2002 (ISSRE 2002), pp. 117–128.

Myers, G.J., 2004. The Art of Software Testing, 2nd ed. John Wiley & Sons Inc., New York.

Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.,2007. Feedback-directed random test generation. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007). IEEE Computer Society Press, Los Alamitos, CA, pp. 75–84.

Sen Fs K., Marinov, D., Agha, G.,2005. CUTE: a concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, Lisbon, Portugal.

Tasse, G., 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards and Technology.

TBB. Intel Threading Buildng Blocks. http://http://threadingbuildingblocks.org/.

Yin, Y., Liu, B., 2009. A method of test case automatic generation for embedded software. In: International Conference on Information Engineering and Computer Science, 2009 (ICIECS 2009), pp. 1–5.

Chorng-Shiuh Koong received M.S. and Ph.D. degree in Computer Science and Information Engineering from National Chiao-Tung University (Hsinchu, Taiwan) in 1995 and 2000 respectively. Currently, he is an associate professor in the Department of Computer and Information Science at National Taichung University of Education (Taichung, Taiwan). His research interests include Software Testing Technology, Object-Oriented Technology, Component Technology, Visual Programming and E-learning Technology.

Chihhsiong Shih received his BSc degree from ChungYuan Christian University, Taiwan, in 1984 and his Msc of Computer Science and Ph.D degree of Mechanical Engineering in 1997 all from Rensselaer Polytechnic Institute. Since then, he has been working in the CAD software industry. From 1997∼2000, he has worked for a CAD simulation company, simmetrix, while from 2000∼2002, he worked for EDA team of microelectronic division of IBM corp. He has broad interests in the software assisted CAD applications including engineering simulation and electrical properties analysis. He is currently teaching embedded software in Tunghai University, Taiwan, as an assistant professor.

Pao-Ann Hsiung received his B.S. degree in mathematics and his Ph.D. degree in electrical engineering from the “National Taiwan University”, in 1991 and 1996, respectively. Since 2007, he has been a full professor in the Department of Computer Science and Information Engineering, “National Chung Cheng University”. He has published more than 200 papers in international journals and conferences. He was a recipient of the 2010 Excellent Research Award and the 2004 Young Scholar Research Award, “National Chung Cheng University”, and the 2001 ACM Taipei Chapter Kuo-Ting Li Young Researcher award. He is a senior member of the IEEE and the ACM, and a life member of the IICM. He is on the editorial board of several international journals and on the program committee of more than 80 international conferences. His main research interests include reconfigurable computing and system design, multi-core programming, cognitive radio architecture, embedded design and verification, embedded software synthesis and verification, real-time system design and verification, hardware-software codesign and coverification, and component-based object oriented application frameworks.

Hung-Jui Lai received M.S. in the Department of Computer and Information Science from National Taichung University (Taichung, Taiwan) in 2010. His research interests include Software Testing Technology, Embedded System and Multimedia Technology.

Chih-Hung Chang received his BE, MS, and Ph.D degrees in Computer Science from Feng Chia University in 1995 and 1997, 2004 respectively. Currently, he is an associate professor at Hsiuping Institute of Technology. His research interests include Object-Oriented Technology, Software Re-engineering, Software Reuse & Maintenance, Software Engineering, Software Architecture, and XML. He has published more than 70 papers in international journals and conferences. He served as local arrangement co-chair for SAC 2011, and co-chair for SAC 2011 APGCC Track, WESQA'10.

Prof. William C. Chu, a professor of the Department of Computer Science, and the Director of Software Engineering and Technologies Center of Tunghai University. He had served as the Dean of Research and Development office at Tunghai University from 2004 to 2007, and the dean of Engineering College of Tunghai University from 2008 to 2011, Taiwan. From 1994 to 1998, he was an associate professor at the Department of Information Engineering and Computer Science at Feng Chia University. He was a research scientist at the Software Technology Center of the Lockheed Missiles and Space Company, Inc., where he received special contribution awards in both 1992 and 1993 and a PIP award in 1993. In 1992, he was also a visiting scholar at Stanford University. He is serving as the associate editor for Journal of Software Maintenance and Evolution (JSME), International Journal of Advancements in Computing Technology (IJACT) and Journal of Systems and Software (JSS). His current research interests include software engineering, embedded systems, and E-learning. Dr. Chu received his MS and PhD degrees from Northwestern University in Evanston Illinois, in 1987 and 1989, respectively, both in computer science. He has edited several books and published over 100 referred papers and book chapters, as well as participating in many international activities, including organizing international conferences, serving as steering committee for COMPSAC, APSEC and the program committee of more than 70 international conferences.

Nien-Lin Hsueh is an associate professor in the Department of Information Engineering and Computer Science and the Chief in the System Development Section of the Office of Information Technology at Feng Chia University in Taiwan. His research interests include software engineering, design pattern, software process improvement, and software testing. He received his PhD in Computer Science from National Central University in Taiwan in 1999.

Chao-Tung Yang is a Professor of Computer Science at Tunghai University in Taiwan. He received the PhD in Computer Science from National Chiao Tung University in July 1996. In August 2001, he joined the Faculty of the Department of Computer Science at Tunghai University. He is serving in a number of journal editorial boards, including International Journal of Communication Systems, “Grid Computing, Applications and Technology” Special Issue of Journal of Supercomputing, and “Grid and Cloud Computing” Special Issue of International Journal of Ad Hoc and Ubiquitous Computing. Dr. Yang has published more than 170 papers in journals, book chapters and conference proceedings. His present research interests are in cloud and grid computing, parallel computing, and multicore programming. He is a member of the IEEE Computer Society and ACM.