# An Interactive Code Generator and Profiler System

**Muhammad Azeem[1], Munir Ahmed[2], Ahthasham Sajid[3], Tahir Iqbal[4],**
**Asif Farooq[5], Prajoona Valsalan[6]**

[1, 2]Barani Institute of Information Technology, PMAS- AAUR, Rawalpindi. Pakistan
[3]Department of Computer Science , FICT, BUITEMS, Quetta, Pakistan
[4]Department of Computer Sciences, Bahria University Lahore Campus, Lahore, Pakistan
[5]Computer Science and IT University of Lahore, Lahore Pakistan
[6]Department of Electrical and Computer Engineering, College of Engineering, Dhofar University, Salalah, Oman

**Summary**
In the last few years with the advent of various compiler languages, it is perhaps difficult to choose one development language over another for a specific set of needs such that compatibility with a compiler for a particular problem is optimal as regards to caching, garbage handling and priority tasking. This hints at a need for automated optimization and compatibility. One possible way is to have general pseduocode which are optimized by profilers for different compilers to identify optimization opportunities and best fits using either sampling or event-based profilers for a software-mediated algorithm analysis. For devices with limited batteries, profilers may be used to optimize energy usage, as well. Additionally, if such an automated profiler is able to generate formal code from a pseduocode, much of Software Management may be automated. In this paper, an Android-based profiler integrated with a code generator is proposed in line with this rationale.

***Key Words:***
*Pseduocode, Symbol Table, Analyzer, Parser, Profiling.*

## 1. Introduction

The parameters to judge quality of software include performance, efficiency, scalability and accuracy, among others. Statistics show that from the internet age i.e. between 1990 and 1999, approximately 14 major programming languages were developed whereas from 2000 to 2015, 12 new languages were developed which means yearly, at least one new language evolves and gets prominent. Each language has its own convention, syntax and even memory requirements. With the increase in the number of languages, it is difficult for programmers to develop applications in various languages without in-depth knowledge about its compiler language. This requirement is tied with scalability and accuracy of any given program. In this regard, there exist software which aid developers in writing algorithms based on certain rules and convert this pseduocode into different languages, partially overcoming the need for an in-depth understanding of syntax based on a particular compiler language. To evaluate performance and efficiency, algorithmic analysis is employed, which acts as a diagnostic tool to identify areas for improvement. For this, there, too, exist automated software profilers which amalgamate information pertaining to software space complexity and running time complexity; this includes frequency of function calls. Such software are classified as either statistical profilers or event based profilers. The former samples a target program whereas the latter collects target program's information by analyzing loops, function calls, path frequencies (Ball, 1996), load unload capacities and even memory requirements. As a motivating example, an event-based profiler may be employed to manage resources and memory by, for exampling, overcoming memory leakage (Hill, 2013). The most common optimizers are Common Language Runtime (CLR) and its garbage collector. The application of profilers is not limited to software and may even be applied to an OS (for e.g., see (González, 2014) for a Raspberry PI profiler) along with its hardware considerations. Profilers may also be employed for value prediction (Gabbay, 1997) to eliminate true-data dependencies whereby the outcome values of a routine are predicted run-time and true-data dependent routines are executed based on that prediction. An alternate is to predict running time of different applications using heuristics (Smith, 1998) but this requires a history of execution and thus is inapplicable for novel software. Profilers may even focus on hardware codes to improve hardware performance (Matev, 2009) There also exist profilers which are online and thus especially suitable for Cloud deployment when actual hardware is unknown (Riou, January 2014) and source code may not be available. In this context, profiling is still possible across different platforms in the Cloud for High Performance Computing (Marinković, July 2016). Profilers may also be GUI-based (e.g., (Rubio, December 2015) (Chevalier, September, 2007). In this paper, an event-based profiler for program performance optimization is proposed explicitly to determine (run-time or otherwise) function calls, function iterations and function paths to address memory usages for Android devices. Profiling for Android systems is important to understand, say, the retardation experienced during browsing, streaming and booting (. Lin, (March 4, 2013). From the aegis of a function call stack and path, the most expensive function can be singled out. This will

Manuscript received January 5, 2019
Manuscript revised January 20, 2019

further allow the tracing of memory leaks for code optimization after measuring memory consumption. For added robustness, the proposed software is further integrated with the capacity to convert pseduocode, written out based on certain rules, into formal code via a lexical analyzer and optimized using Business Intelligence. This optimized code is compiled remotely on a Cloud infrastructure. This overcomes any hardware limitations and the compiler may even be engineered to follow different kernels. The optimized code is then supplied to the profiler, analyzed and may then be improved upon by the developer. For the time being, the export only supports C++ code generation but may be extended to incorporate different languages, thus allowing for further compatibility across different platforms. The need for a remote compiler is justified not only because of the apparent increase in performance but because it allows for more log space than the limited counterpart on the Android device and further liberates hardware constraints. Furthermore, the execution of an application requires multiple software layers in the Android platform, which may even be programmed in different languages. This paper is structured as follows: in Section 2, different methods of profiling and code generation in literature are reviewed to identify latest methods. In Section 3, the mechanism and algorithm of CGen-Profiler are detailed whereas running examples are presented in Section 4. In Section 5, an experiment is performed to compare how the proposed software fares against other existing software for similar ends. The paper is concluded in Section 6 with potential benefits and future additions.

## 2. State of Art

Since the proposal concerned with this paper involves a profiler integrated with a code generator, a survey of state of art profilers and code generators is performed to highlight popular methodology and identify potential contributions. In general, it is noted that each profiler and code generator serves purpose for a particular class of application for professional and amateur developers, alike.

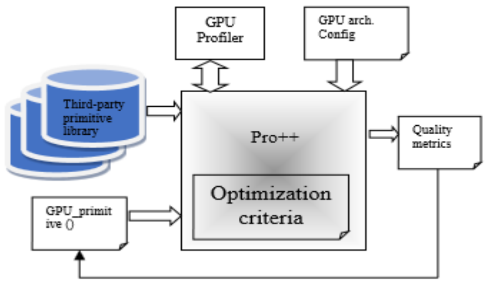


Fig. 1 Proposed framework of (Bombieri, 2015)

### 2.1 Profilers

Profiling techniques differ in their nature, method and reporting methods. For instance, a class of profilers are based on lazy allocators (Shi, China 2016) which profile objects and manipulate them by run-time events in Java-based applications running on a virtual machine. Recently, packed objects in the IBM J9 Virtual Machine have been profiled, too, to measure memory consumption (Pandya, 2016). Such packed objects offer a better control over the layout of objects in memory. Furthermore, they employ a flexible memory structure which can be leveraged to increase the efficiency of caching. Run-time profiling has the advantage of analyzing execution under heavy load conditions, which may escape analysis prior to execution. Such run-time profilers are expected to be flexible apart from being accurate. In this regard, a novel performance monitoring unit not based on instrumentation is introduced in (Gibert, (June 19, 2015).Casual Profiling is introduced in (Curtsinger, 2015, October) via a tool named COZ. Causal profiling method narrows down the focal points of optimization by running performance experiments in parallel to execution of program, the underlying assumption being that profiling only tells programmers where their program spent most time and this may not accurately reflect performance. Event-based profilers typically base their methods on byte code instrumentation. This assumes that the compiler does not have an in-built capacity for optimization via stack allocation and in lining. One aspect of CGenProfiler is based on this idea, whereby it uses a remote compiler. For compilers without such optimization capacities, a novel technique was introduced Zheng et al. (Zheng, 2015) on a Java Virtual Machine (JVM).

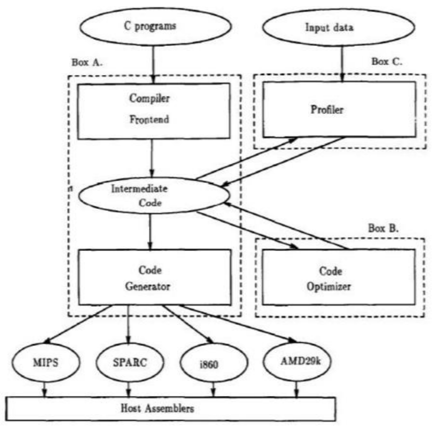


Fig. 2 Block diagram of proposed prototype of (Chang, (December 1991)

Bombieri et al. (Bombieri, 2015)have presented a profiling framework for GPU primitives called Pro++ (see Figure 1), which allows measuring of the implementation quality of a given primitive compiler by considering the target architecture characteristics. The framework collects the information provided by a standard GPU profiler and combines them based on optimization criteria. The criteria evaluations are weighed to distinguish the impact of each optimization on the overall quality of the primitive implementation. In (Chang, (December 1991), authors propose and implement a compiler equipped with optimization and profiling capacities for classical methods of code optimizations (see Figure 2). The novelty of the compiler is in its two components viz. an execution profiler and a profile-based code optimizer. The former inserts probes into the target program, performs several runs, gathers profile information and then feeds this information to the latter – the optimizer.

Froyd et al. in (Froyd, 2003)z proposed a novel low-overhead call path profiler along with its implementation. The profiler samples frequency counts for call graph edges and, furthermore, managed to bypass code instrumentation. An efficient instrumentation technique is implemented by Mudduluru et al. (Mudduluru, 2016) which generates object flow profiles for programs written in Java, without modifying the underlying JVM. This is done by a method called Ball-Larus numbering on a specialized hybrid flow graph. In (Matev, 2009), Matev et al. designed a run-time profiler on instruction level with a focus on analysis of loops. Costly and fine grain blocks are searched for in lieu of coarse grain blocks to determine wall clock time. Data pertaining to position of blocks, its body, number of executions and its size are stored and analyzed by the profiler in separate hardware whereas a majority of profiling of code is done in software by introducing code into the target program which has time overhead. A profiling platform specifically for Android platform was made by Yoon (Yoon, (April 2012) to analyze the platform's performance via a modified debugging method and a trace tool. CGen-Profiler differs in that it allows for run-time profiling and includes a path-profiling with frequency of function calls to identify expensive iterations. This is complemented by the framework in (Steigerwald, 2012) as an attempt to optimize battery performance of Android devices. Such considerations stem from efficient use of energy. This is also another important consideration not only from an engineering and social perspective but is now an important practice for IT development, as well (Murugesan, October 2012) and is not limited to hardware; the term "green software" is becoming increasingly popular. It is important to realize that source code structure has an effect on energy consumption by virtue of compiler mechanisms. Several mechanisms exist to calculate energy consumption of different software. For example, ALEA was proposed in (Murugesan S. a., (March 2017). ) to profile software at the fine-grained level of loops and functions, yielding energy savings up to 2.97 times. A similar software which the authors call eCalc (Hao, 2012) was introduced for Android systems. To address this concern, the implementation of a remote compiler in the proposed mechanism is expected to contribute to a reduction of hardware waste

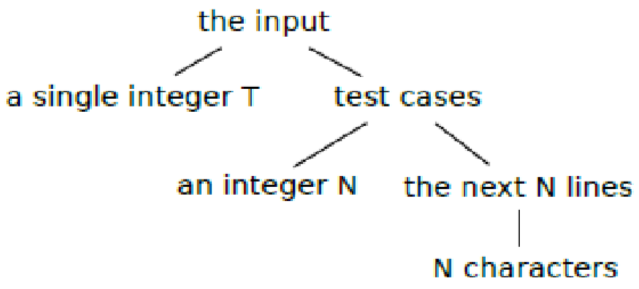


Fig. 3 A specification tree, as found in (Lei, 2013)

## 2.2 Code Generators

The discussion of compilers as translators of source language code to optimized target language code notwithstanding, a great advantage may be leveraged if such optimization compilers are complemented with translation of pseudo code to formal code, offering a user-friendly integrated tool. To this end, there are various standalone tools available called code generators which are either designed for narrowed down purposes or specific languages. For instance, in (Eid, 2010), a geometric algebra-based code generator is designed specifically to represent and process geometric data. In (Cataño, 2015), an Enterprise Resource Planning system is code generated and evaluated by code generators called EventB2SQL and Open Bravo POS. A code generator for Oracle embedded with business logic, a GUI and drag-and-drop tool box was developed in (Rathod, 2016). In (Lee, 2012), the authors develop a code generator which allows for semantic translation of code for cross-platform compatibility. Similarly, for hardware, a compiler framework converts an Open MP programming model to that for NVIDIA's CUDA (Lee S. M.-J., 2009). Similarly, in (Verdoolaege, 2013), a source-to-source compiler is introduced for CUDA's parallel execution paradigm. In (Li, 2015), an automatic code generator called STEPOCL for multicores is proposed which uses one piece of code and replicates it for other cores along, ensuring compatibility. A generic code generator for statistical translation of natural language to formal code may be found in a paper by Lei et al. (Lei, 2013), in which the authors implemented a statistical (Bayesian) model to translate natural language specification into a specification tree (see Figure 3). This specification tree is then used to build a C++ input parser.

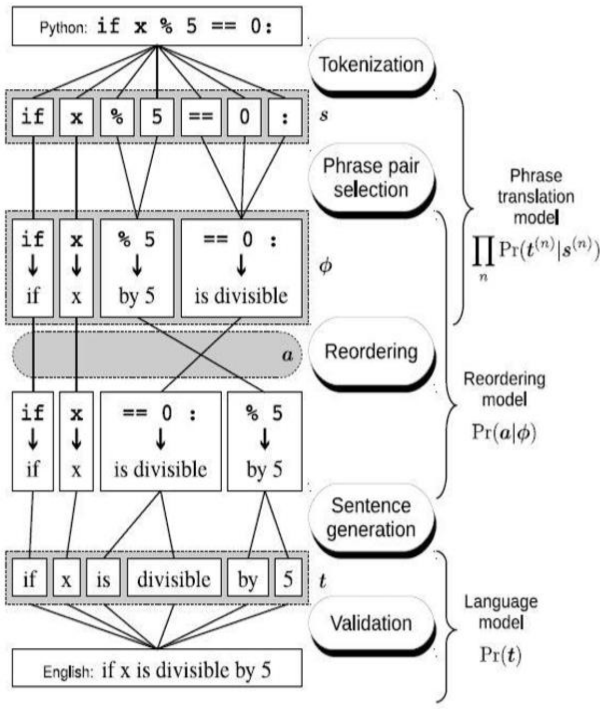


Fig. 4 Pseudocode generated from Python – example in (Oda, 2015)

The other way (from code to natural language) has been accomplished, as well (Oda, 2015) wherein the authors to use the Statistical Machine Translation (SMT) method for a tokenized Python code via a proposed tool called Pseudogen. Their main idea is show in Figure 4. In summary, profilers and code generators are used for a particular set of existing problems on a unique platform. CGen-Profiler is expected to contribute to the growing gap between diversity of software available and narrow expertise of developers to address broader platforms simultaneously. Since the architecture for kernels across platforms are usually different in their execution, the translation of data handling by across compilers needs careful attention. CGen-Profiler is expected to contribute to greener software, as well, for mobile devices powered by portable batteries.

## 3. Mechanism of CGen-Profiler

The CGen-Profiler analyzes the pseudocode with the help of a formally defined algorithm lexicon for conversion to formal code and for profiling. The profiling indicates key areas which need to be optimized for efficiency. The components of CGen-Profiler are listed out in detail in the following subsections, showed pictorially in Figure 5.

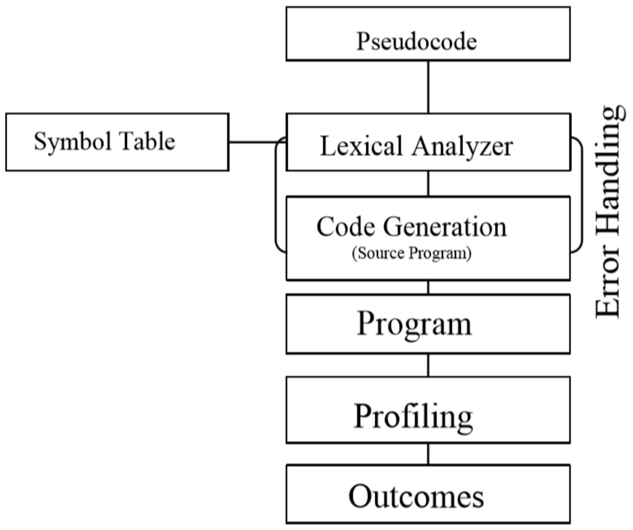


Fig. 5 Logic flow of CGen-Profiler

### 3.1 Pseudo code input

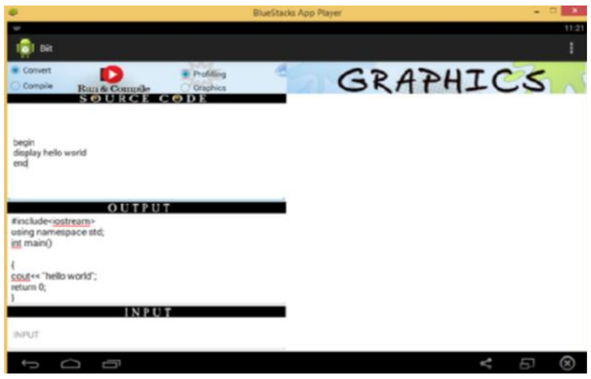
CGen-Profiler accepts an algorithm in the form of standard pseudo code in the particular standard format found in (Cormen, 2009) These include reserving formal names for standard C++ routines including while, for, switch if-else along with their indentation. Incorrect syntax is highlighted in red. It is important that the pseudo code must be conforming to standards as the inputs are tokenized. If the pseudo code is written without standard practices, then a token is considered unavailable in the rules.

Table 1: Symbolic Description

| Lexeme | Category | Code |
|---|---|---|
| = | Comparison | = = |
| := | Assignment | = |
| ↲ | Next statement | ; |
| input | Get input | cin>> |
| display | Give output | cout<< |

### 3.2 Lexical Analyzer

A lexical analyzer then performs a straightforward conversion of the given pseudo code into formal C++ code based on a symbol table. A part of the symbol table is shown in Table 1. This analyzer parses the algorithm by converting the given pseudo code in tokens of lexemes. The symbol table is used as a reference to replace tokens with formal code. These lexemes are matched with set of rules and converted into code parts using the symbol table. In case the pseudo code is not standard, the lexical analyzer generates an error message and stops the process at this stage.



```
begin
display hello world
end
```

```
#include<iostream>
using namespace std;
int main()

{
cout<< "hello world";
return 0;
}
```

Fig. 6 Screenshot of CGen-Profiler with code generated from pseduocode

## 3.3 Code Generation

The pseduocode is segmented into code directly from the input and data for identification of variables, necessary variables and inclusion of called-for libraries. The pseduocode itself identifies variables needed. The pseduocode is sent to a remote server for compilation to an online version of Cygwin supporting Android with POSIX. A quick reference on Android architecture may be found in (Yoon, (April 2012)). If the code has no compile time error, it saves a log file and generates a formal code. The metadata of each code is kept separate. This part includes function names, as well, determined directly by the lexical analyzer.

## 3.4 Profiler

The flow for this profiling is shown in Figure 7. In its architecture, the profiler is no different than the commonplace profiler for Android Dalvik Debug Monitor Service, which is a part of the Android Software Development Kit. The profiling tool has the capacity to determine function calls, function iterations, interdependence of functions, flow of operations and duration (in wallclock time) of function execution. Since Cygwin is employed as the virtual machine for remote compilation instead of a local Linux kernel, the commonplace profiler was modified to support dynamic profiling. Thus, parts of memory reserved for the kernel may be freed up and any garbage dump, poor stack allocation and method inlining may be managed accordingly.
In summary, all valid options are profiled and best options are compared with the prehistorical data and thus optimized. Afterwards, the profiler checks the outputs for all cases, ensuring that the output is the same for the given inputs and more efficient code is displayed as optimized code. The efficient code is sent back again to Cygwin for a final compilation.

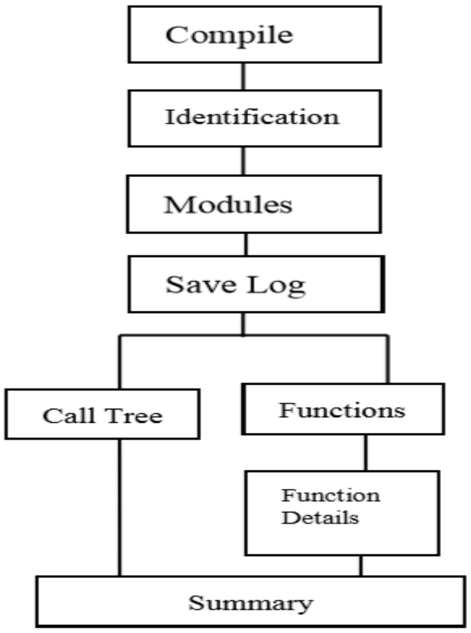
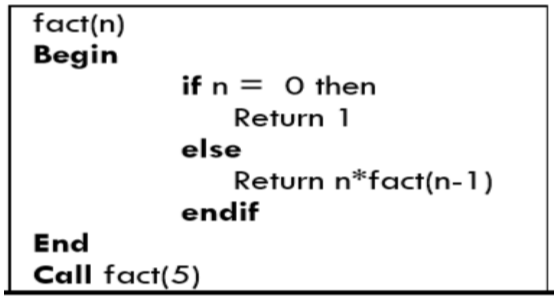


Fig. 7 Profiler logic flow of CGen-Profiler

## 4. Running examples

A pseudo code for finding the factorial of a given integer ($n$ = 5 in this case) is written using recursive functions. The algorithm spelled out to CGen-Profiler is as follows:



```
fact(n)
Begin
        if n =  0 then
            Return 1
        else
            Return n*fact(n-1)
        endif
End
Call fact(5)
```

Listing 1: Pseudocode for Factorial of 5 as input to CGen-Profiler

The software translates the pseudo code into a complete C++ code, adding any necessary libraries, function names and variables. The outcome obtained is present in Listing 2. The pseudo code is fed to the software again but this time, with a replacement of the recursive function with a loop. The output given is in Listing 3. To show that this enhances the algorithm, we profile both codes. These are listed in the Results and Discussion section.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int fact(int f)
{
if (f == 1)
                return 1;
else
                return f*fact(f -
1);
}
int main()
{
cout << fact(5);
return 0;
}
```

Listing 2: C++ code from pseudocode

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int fact(int f)
{
for (int i = 4; i > 0; i--)
                {
                                f = f*i;
                }
                return f;
}
int main()
{
                cout << fact(5);
                return 0;
}
```

Listing 3: C++ code from different pseudocode

These steps are repeated for the well-known binary search. The pseudo code is shown in Listing 4.

```
binary_search(a[], x, l, h)
Begin
        if (l > h)
          Print "Not found"
          Return 0
      else
          m = (l +h) / 2
      if (a[m] = x)
          Print "Found"
          Return 0
      elseif (x > a[m])
          binary_search(a, x, m+1, h)
      elseif (x < a[m])
          binary_search(a, x, l, m-1)
      endif
End
a[]={ 1,2,3,5,6,8,9,10,12,14 };
Input x
l:=0
h:=9
Call binary_search(a, x, l, h)
```

Listing 4: Pseudocode for Binary Search as input to CGen-Profiler

The result, without optimization, is shown in Listing 5. In Listing 6, the heuristic optimization process is applied and the function itself is integrated in the main function but instead of a recursive function, a while loop is added.

```
#include "stdafx.h"
#include  <iostream>
using namespace  std;
int binary_search(int a[], int x, int l,
int h)
{
                int m;
                if (l > h)
                {
                                cout << "Not
found" << endl;
                                return 0;
                }
                else
                                m = (l + h) / 2;
                if (a[m] == x)
                {
                                cout <<
"Found" << endl;
                                return 0;
                }
                else if (x > a[m])
                  binary_search(a, x, m +
1, h);
    else if (x < a[m])
        binary_search(a, x, l, m - 1);
}
int main()
{
        int a[] = {
1,2,3,5,6,8,9,10,12,14 };
        int x, h, l;
                cin >> x;
                l = 0;
        h = 9;
       binary_search(a, x, l, h);
         return 0;
}
```

Listing 5: C++ code from pseudocode

```
#include "stdafx.h"
#include  <iostream>
using namespace  std;
int main()
{
   int h, l, m, loc, x, a[] = {
1,2,3,5,6,8,9,10,12,14};
    l = 0;
    h = 9;
    loc = NULL;
    cin >> x;
    while (h - l > 0)
    {
m = (h + l) /  2;
            if (x == a[m])
            {
  loc = m;
break;
            }
   else if (x < a[m])
            {
     h = m - 1;
     if (h < l)
     h = l;
            }
   else if (x > a[m])
            l = m + 1;
    }
    if (x == a[h])
    loc = h;
                cout << "The location is = "
<< loc << endl;
     return 0;
}
```

Listing 6: Changed C++ code for binary search

## 5. Results and Discussion

CGen-Profiler is now compared with Microsoft Visual Studio's inbuilt profiler. The same codes were fed to the profilers for both binary searches and factorial algorithm, once for each version of the algorithm. The runs were performed on a Core i-3-3227U CPU @ 3MB Cache, 2 cores, 4 threads, 1.90GHz base processor frequency, 5GT/s bus speed and a TDP of 17W with 3.89GB usable RAM. Microsoft Visual Studio was also employed as a benchmark to establish the accuracy of the profilers and, furthermore, to establish the veracity of the remote profiler.

### 5.1 Processing Speed Comparison

For a measurement of time corresponding to realistic epochs, ideally emphasis should be on the wall clock time. We thus choose to measure the wall clock speed for each code. Such runs may be plotted against frequency of function call to measure rates at which the function is summoned. The comparison in this section is much simpler in that the focus is on CPU usage plotted against wallclock time (in seconds). The gradient of the graphs may be used to determine processing speed and offer a comparison of the profilers. Wall clock measurements for CGen-Profiler were accomplished using std::chrono::time_point. The number of samples collected for each code were left unaltered. The outcomes for profiling are recorded in Figures 8 and 9 for the two factorial algorithms whereas Figures 10 and 11 record profiling outcomes for the binary search algorithm.

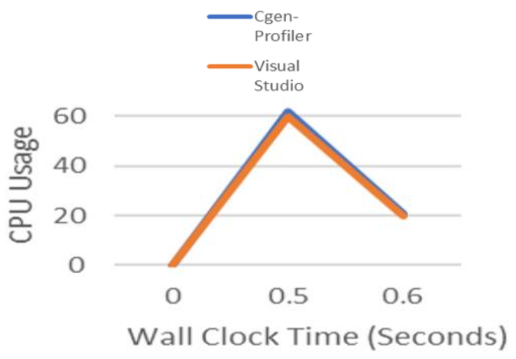


Fig. 8 Profiling for Listing 2

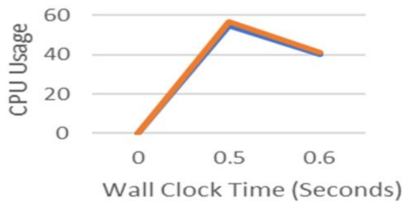


Fig. 9 Profiling for Listing 3

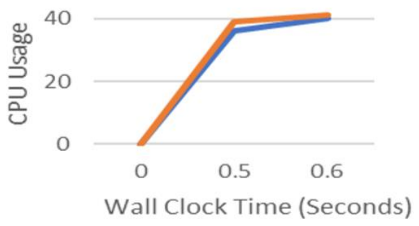


Fig. 10 CPU Usage for Listing 6

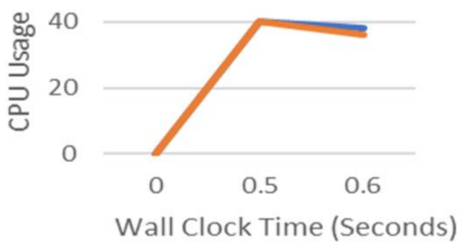


Fig. 11 CPU Usage for Listing 5

Listing 3 and Listing 6 are an optimized version of Listing 4 and Listing 5, respectively. To see this, observe that the CPU usage, as indicated by both profilers, peaks at 0.5 with different memory usage. The gradient of the graphs for the first 0.5s differs in that the first slope is higher, indicating that the rate at which memory is consumed is higher for Listing 2 (respectively, Listing 5) than that for Listing 3 (respectively, Listing 6). This could be explained by the frequency of function calls, memory leaks due to different algorithms or because one algorithm occupies more memory than the other by virtue of running memory requirements. Similar results are displayed.

### 5.2 Accuracy

Microsoft Visual Studio notes 90 allocations for Listing 2, with 41 samples collected. The number of modules assigned for each algorithm differ with 14, and 16 for Listing 1 and Listing 3, respectively. CGen-Profiler, on the other hands, notes 85 allocations for Listing 2 with 28 samples collected with modules 18 and 21 for Listing 1 and Listing 3, respectively. The allocation of more modules reflects the remote nature of the compiler. For each module, the function names and its calls were assigned and were in agreement with both profilers. The iterations for the fact function were reported to be 23 for both.

## 6. Conclusions and Future Work

A program called CGen-Profiler is proposed which simultaneously generates formal code from a given pseudocode and profiles it. This combination is optimized according to reference architecture for business intelligence. Currently, the optimized output formal code is generated for C++ only but may be extended to cover other languages, as well, by modifying the symbol table. With a broader “conversion capacity”, a smart system could be devised which automatically determines the best language for a given pseudocode, based on memory requirements, compilation speed etc. This is hoped to circumvent the need for in-depth understanding of different syntax. It is hoped that CGen-Profiler may encourage beginners and professionals in their endeavors to develop software. The execution of a final compatibility check of the optimized program could also be added in future versions.

## References

[1] Lin, Y.-D. H.-Y.-C.-H.-L. ((March 4, 2013). Booting, browsing and streaming time profiling, and bottleneck analysis on androidbased systems. Journal of Network and Computer Applications, 36, 4 . 1208–1218. .

[2] Ball, T. a. (1996). Efficient Path Profiling. In MICRO 29 Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture (pp. 46-57). Paris, France 1996: IEEE.

[3] Bombieri, N. B. ( 2015). An enhanced profiling framework for the analysis and development of parallel primitives for GPUs. In 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC) (pp. 1-8). (Turin, Italy : IEEE.

[4] Cataño, N. a. (2015). A Case Study on Code Generation of an ERP System from Event-B. In International Conference on Software Quality, Reliability and Security. IEEE.

[5] Chang, P. P.-m. ((December 1991). Using profile information to assist classic code optimizations. Software -- Practice & Experience, 21, 12. (pp. 1301 - 1321). Software: Practice and Experience.

[6] Chevalier, F. A. (September, 2007). Structural Analysis and Visualization of C++ CodeEvolution using Syntax Trees. In Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE (pp. 90-97). Dubrovnik, Croatia 3-4 : IEEE, 90-97. .

[7] Cormen, T. H. (2009). Introduction to Algorithm. MIT Press.

[8] Curtsinger, C. a. (2015, October). Coz: Finding code that counts with causal profiling. In Proceedings of the 25th Symposium on Operating Systems Principles. (pp. 184-197). (Monterey, California: Elseveir.

[9] Eid, A. H. (2010). Optimized Automatic Code Generation for Geometric Algebra Based Algorithms with Ray Tracing Application. Port-Said.

[10] Froyd, N. M.-C. ( 2003). Low-overhead call path profiling of unmodified, optimized code. In Proceedings of the 19th ACM Annual International Conference on Supercomputing (pp. 81-90). (Cambridge, Massachusetts, USA : IEEE.

[11] Gabbay, F. a. (1997). Can Program Profiling Support Value Prediction? In Proceedings of Micro-30 (pp. 270-280). (North Carolina: IEEE.

[12] Gibert, E. M. ((June 19, 2015). Support for Runtime Managed Code: Next Generation Performance Monitoring Units. Computer Architecture Letters, 14, 1 (pp. 62-65). IEEE.

[13] González, P. M. (2014). Profiling and optimizations for Embedded Systems. In 2014 Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE). (Lausanne, Switzerland: IEEE.

[14] Hao, S. L. ( 2012). Estimating Android Applications' CPU Energy Usage via Bytecode Profiling. In GREENS '12 Proceedings of the First International Workshop on Green and Sustainable Software (pp. 1-7. ). Zurich, Switzerland: IEEE.

[15] Hao, S. L. (2012). Estimating Android Applications' CPU Energy Usage via Bytecode Profiling. In GREENS '12 Proceedings of the First International Workshop on Green and Sustainable Software (pp. 1-7). Zurich, Switzerland: IEEE.

[16] Hill, E. T. (2013). GrowthTracker: Diagnosing Unbounded Heap Growth in C++ Software. In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (pp. 134-143). Luxembourg, Luxembourg: IEEE.

[17] Lee, S. M.-J. (2009). OpenMP to GPGPU: A Compiler Framework for AutomaticTranslation and Optimization. In Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming . Raleigh, NC, USA : IEEE.

[18] Lee, Y. a. (2012). A Study on the Smart Virtual Machine for Executing Virtual Machine Codes on Smart Platforms. International Journal of Smart Home, 93-106.

[19] Lei, T. L. (2013). From Natural Language Specifications to Program Input Parsers. In The 51st Annual Meeting of the Association for Computational Linguistics (pp. 1294-1303). Sofia, Bulgaria : Omnipress, Inc.

[20] Li, P. B. (2015). Automatic OpenCL code generation formulti-device heterogeneous architectures. In 44th International Conference on Parallel Processing (pp. 959-968). Beijing, China: IEEE.

[21] Marinković, M. K. ( July 2016). Platform independent profiling of a QCD code. In 34th Annual International Symposium on Lattice Field Theory . (University of Southampton, UK : PoS.

[22] Matev, V. T. ( 2009). Method for run time hardware code profiling for algorithm acceleration. In Proc. SPIE 7363, VLSI Circuits and Systems IV (p. 736304). SPIE Proceedings.

[23] Mudduluru, R. a. ( 2016). Efficient flow profiling for detecting performance bugs. In Proceedings of the 25th International Symposium on Software Testing and Analysis (pp. 413-424). (Saarbrücken, Germany : ACM.

[24] Murugesan, S. a. ( October 2012). R., eds. Harnessing Green IT: Principles and Practices. Wiley Publishing.

[25] Murugesan, S. a. ((March 2017). ). de, and Leather, Hugh. ALEA: A Fine-Grained Energy Profiling Tool. ACM Transactions of Architecture Code and Optimization, 14, 1.

[26] Oda, Y. F. (2015). Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation. In 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 574-584). Lincoln, NE, USA: IEEE.

[27] Pandya, U. K. (2016). In 2016 IEEE Canadian Conference on Electrical and Computer Engineering. In Electrical and Computer Engineering (CCECE) (pp. 1-6). (Vancouver, British Columbia, Canada: IEEE.

[28] Rathod, S. D. (2016). A new Approach to Capture Business Logic from UI with Automatic Code Generation and Database Creation. International Journal of Innovative Research & Development, 119-126.

[29] Riou, E. R. (January 2014). PADRONE: a Platform for Online Profiling, Analysis and Optimization. In

International workshop on Dynamic Compilation Everywhere . In DCE 2014-International workshop on Dynamic Compilation Everywhere. Vienna, Austria : HAL.
[30] Rubio, A. a. (December 2015). R: A Review of Existing Methods and an Introduction to Package GUIProfiler. The R Journal, 7, 2 , (pp. 275-287).
[31] Shi, J. J. ( China 2016). Profiling and analysis of object lazy allocation in Java programs. In Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD) (pp. 591-596). Shanghai: IEEE.
[32] Smith, W. F. (1998). Predicting Application Run Times Using Historical Information. In IPPS/SPDP '98 Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (pp. 122-142). Orlando: SpringerVerlag London, UK, 122-142.
[33] Steigerwald, B. a. (2012). Green Software. In Murugesan, San and Gangadharan, G. R., eds., Harnessing Green IT. John Wiley & Sons.
[34] Verdoolaege, S. J. (2013). Polyhedral Parallel Code Generation for CUDA. ACM Transactions of Architecture and Code Optimization - Special Issue on HighPerformance Embedded Architectures and Compilers.
[35] Yoon, H.-J. ((April 2012). A Study on the Performance of Android Platform. International Journal on Computer Science and Engineering, 4, 4 , (pp. 532-537).
[36] Zheng, Y. B. ( 2015). Accurate Profiling in the Presence of Dynamic Compilation. In roceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (pp. 433-450). (Pittsburgh, PA, USA : ACM, .

**Muhammad Azeem** Received MCS and MSCS degree from PMAS Arid Agriculture University Rawalpindi, Pakistan. He is currently associated with Barani Institute of Information Technology as Lecturer. His research interests are Data Mining, Algorithms.

**Munir Ahmad** Born in Punjab, Pakistan 1983.Get P h D degree from Capital University of Science and Technology Islamabad, Pakistan Current Research interests Data Science, Data Mining, Algorithms, Semantic Cache.

**Ahthasham Sajid** working as Assistant Professor in Department of Computer Science Since 2009, He did his PhD in Opportunistic Networks. His areas of interest are Wireless & Sensor Networks, Mobile Communication, and Network Security. He has more than 15 International and national publications.

**Tahir Iqbal** has a master degree in CS currently working as assistant professor in Computer Science Department at Bahria University Lahore. His current areas of research interests lie in wireless communication and Networks, Cloud Computing, and Data Science. He has number of national and international journal publications.

**Asif Farooq** is working as Lecturer in Department of Computer Science, University of Lahore. He is very energetic, hardworking and passionate about his profession. He is versed with teaching experience spread over 4 years including public and private sectors. He did MS Computer Science from The University of Management and Technology. His area of the interest and working domain is Cloud Computing.

**Prajoona Valsalan** is currently working as Assistant Professor in the Department of Electrical and Computer Engineering, Dhofar University. Her research areas include Low power VLSI, Sensors, Digital Design and Networks. She has published more than 15 papers in various reputed journals and conferences.