Performance vs. Power and Energy Consumption: Impact of Coding Style and Compiler

Hesham H M Hassan
Computer Science Department
Faculty of Computer and Information, Cairo University
Cairo, Egypt

Ahmed Shawky Moussa
Computer Science Department
Faculty of Computer and Information, Cairo University
Cairo, Egypt

Ibrahim Farag
Computer Science Department
Faculty of Computer and Information, Cairo University
Cairo, Egypt

***Abstract*—Reaching a balance between performance and energy consumption has always been a difficult objective to achieve for energy and power-aware applications. The work presented in this paper investigates the impact of using different coding styles to achieve a balance between performance and energy efficiency. The research also studies how different compilers may affect not only the performance of the code but also the energy consumption. The research demonstrates and concludes the process of choosing the right combination of the coding style and compiler, the combination which works best with the nature of the application and the target hardware, is necessary if the balance between performance and energy is a software design goal. The study addresses some experimental aspects of the impact of coding style and choice of the compiler on energy and performance efficiency. It also shows how different coding practices for the same problem could produce different performance and energy consumption rates.**

Keywords—Energy consumption; energy efficiency; power-aware; performance; coding styles; coding practice; compilers

I. Introduction

In software applications, code efficiency can mean different things depending on the system constraints. A time-constrained system is efficient when it runs fast, a power-constrained system is efficient when it runs on low power, and an energy constrained system is efficient when it consumes low total energy [1], [2]. Reaching the balance between performance, power and energy consumption has always been a difficult problem, as coding and compiling for performance do not always mean coding and compiling for power and energy [3]. When a program is executed on a computing device, it consumes energy based on how it uses the computing device's resources [4]-[7]. Each instruction inside the program contributes to the resources usage and to the total energy being consumed. Those instructions get generated by the compiler used and based on how the program's code is written.

In this study, the authors show that the coding style along with the compiler choice have a great impact on the application's performance, power, and energy consumption. The C++ will be used as the programming language in the case study, with code compiled by four different C++ compilers (MinGW GCC, Cygwin GCC, Borland C++, and Visual C++).

Following is an outline of this paper. Section II discusses some of the related work done in the software energy optimization techniques. Section III explains the setup of the experiments, the three different coding styles to be studied, the energy model to be used, and the target machine details. Sections IV, V, VI and VII analyze and explain the results of executing the different coding styles using each of the selected compilers. At the end of each section, the specific compiler results are summarized in terms of which coding style best suits the system constraints. In Section VIII, the four compilers are compared and contrasted. Section IX validates the introduced software improvement on one of the well-known open source C/C++ applications, showing how much energy can be saved. Section X highlights the future work. Finally, Section XI draws the conclusions.

II. Related Work

There are various hardware and software techniques and approaches to reducing energy consumption [1]. Although there is a considerable amount of work done in hardware power optimization, these techniques are best applied in early design stages [8]. Source-code level energy optimization is another way to reducing energy consumption, which is of particular importance when adhering to a strict power budget [9]. It is also believed to fill an important gap in providing a machine-independent computing cost reduction [10].

Ajit Pal in his study "Low-Power Software Approaches" [11], Vishal Dalal et al. in their study "Software power optimizations in an embedded system" [12], and Tajana Simunic et al. in their study "Energy-efficient design of battery-powered embedded systems" [13] have demonstrated various software optimization techniques for reducing energy consumption without modifying the underlying hardware. The studies discussed performing machine independent optimization techniques that do not require any knowledge of the underlying hardware architecture. Below are some of the code enhancement techniques examined in their studies, which can be applied manually or automatically as a compiler optimization:

- Reducing the code size by removing the unnecessary computations e.g. removing non-reachable code and non-used variables. This results in less Central Processing Unit (CPU) work and memory usage, resulting in less power consumption.
- Using local variables instead of global variables, so that variables can be easily assigned to a register.

- Avoiding multiple memory lookups by replacing pointers chain with a reference variable.
- Reusing the already computed results, instead of computing it again.
- Reducing conditional branches and jump statements, as it interferes with the prefetching of the instruction, causing a code slow down.
- Optimizing the common case, focusing on the fast path.
- Increasing the spatial locality of reference by placing the code and data together in memory, if they are accessed together in time. For example; exchanging inner loops with outer loops, when the loop variables index into an array.
- Replacing a function call with the body of that function. This may lead to better memory space utilization at runtime. However, it has a reverse impact on performance in some cases, if the code size did not fit in the cache memory.
- Unrolling the loops by duplicating the loop body multiple times to decrease the overhead of the loop conditions.
- Moving the code outside the loop, when possible.
- Replacing the slow mathematical operations with faster ones e.g. replacing a multiplication with a summation operation.
- Merging multiple loops into a single one, aiming at reducing the loop conditions overhead.
- Splitting the loop by removing the conditions that are only introduced to handle the first or last iterations.

III. Experimental Setup

Compilers apply several optimizations to improve the quality of the final code. Some optimizations may result in better performance and energy efficiency. In some cases, they may cause performance loss and increased energy consumption [14]. In this particular study, the selected compilers are used with their default settings, without applying any compilation time enhancements, flags, or directives. Furthermore, the study did not assume or investigate whether or not the compilers performed any of the compiler optimization techniques covered in [8], [15]-[19]. Also, investigations were not carried out on how the executable files (exe) are generated, or how the instructions are created inside the exe files.

A. Energy Model

In this study, Windows Performance Analyzer is used to measure the energy consumption of the software applications. Windows Performance Analyzer (WPA) is an analysis tool developed by Microsoft. It creates graphs and data tables of Event Tracing for Windows (ETW) [20]. WPA analyzes all execution parts of the Windows operating System. It opens the ETL files which the Windows Performance Recorder (WPR) creates and it also utilizes graphs and tables to show the collected data for analysis. The WPA tool enables us to see the system activities, computation and power usage [21]-[23].

B. Execution Setup

The case study is done on three coding styles of the Selection Sort algorithm. Sorting is done on an array of size 500,000 elements, filled with randomly generated numbers, ranging from 0 to 32768. The sort algorithms have been given the same initial set of random numbers, to ensure that the algorithms have the same amount of work for each run. The initial set of random numbers is pre-generated in an external file, which is then populated to an array at the beginning of the execution. Each coding style is executed 10 times, to avoid any changes and discrepancies in the processing time results, and also to avoid the other processes overhead, which are considered as noise. The execution time is calculated for each run. The 10 executions are captured in 10 different power measurement sessions. The power measurements are then extracted from the Windows Performance Analyzer via filtering out the data by the name of process, where multiple power measurements can be extracted. The standard deviation is then calculated, to find out how close the data are to the average power. Whenever an average is calculated, the standard deviation is shown as an error bar at the top of the graph, to show how close the data are to the average.

During the execution of the experiments, the Operating System may start a process that could impact the accuracy of our measurements. Whenever any of these anomalies are detected in the Windows Performance Analyzer, the measurement sample is discarded. Fig. 1 shows an example of the energy measurement anomaly, caused by the operating system.

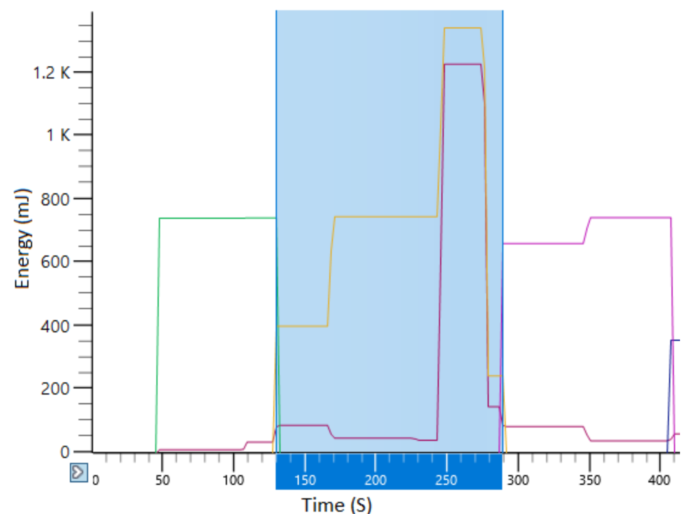

Fig. 1. Measurement anomaly – caused by the operating system.

C. Processor Affinity

Processor affinity exploits the way that remainders of a process running on a given processor may stay in that processor's state, even after another process is executed on that processor. Scheduling the process to run on the same processor may allow it to utilize the processor's cache and avoid cache misses [24]. In this experiment, the processor affinity will be set to a single processor all the time. Given that our

experimental applications are coded in a single threaded manner, using a processor affinity should not impact the experimented application's execution. However, it will attempt to standardize the way the processor handles the tasks execution; especially around the processor's local cache.

It is also important to note that the Processor Affinity is just a direction sent to the CPU, where there is no guarantee that the CPU will adhere to it. In specific situations, such as two processor-intensive tasks having the same affinity to a single processor while another processor is not utilized, a Scheduling-Algorithm implementation will switch a task execution to another processor to gain higher efficiency. In such cases, the application will bounce between different processors. However, this bouncing does not impact the sequential status of the application, and it also has never occurred in our experimentation results.

D. Selected Coding Styles

In any process, different instructions are performed by different components of the processor, resulting in different energy consumption for a variety of instructions. Since not every component of the processor is used for every instruction, components could be automatically switched off when they are not used; however, energy is also consumed when the components are switched on again [25]. In CMOS circuits, for example, power is dissipated in a gate when the gate output changes from 0 to 1 or from 1 to 0. A study on compiler optimization [26] claims that energy consumption enhancements can be made by minimizing the power dissipation at instruction-level, by scheduling instructions to reduce the power consumption on the instruction bus.

In this paper we chose three different coding styles for the same algorithm, which investigates the performance and energy consumption impact of switching between processing-intensive operations and other I/O operations, and to validate the above claims from a high-level coding perspective. Also, because the Operating System could interrupt the processing-intensive operations by giving a lower priority to the process, we have investigated this situation by deliberately setting the process to a sleep mode.

Table I shows code snippets of the three coding style, while the following subsections describes the three coding styles in more details.

1) First Coding Style

The first approach is to perform the sorting completely within its own loop, then print the output in a different loop. In this approach, there is a complete separation between the CPU intensive operations and the input/output (I/O) operations, so the impact on the energy consumption can be measured.

2) Second Coding Style

In the second coding style, the output is printed while the sorting is in progress. The second printing loop is left empty, so that the number of instructions remains the same as the first coding style, leaving the energy consumption difference focused on interrupting the CPU intensive instructions with the I/O instructions.

3) Third Coding Style

In the third coding style, everything is repeated from the first coding style, but with an extra sleep statement. The sleep statement is executed every 500 iterations, for 1 millisecond, giving a total of 1000 millisecond per algorithm run. The sleep statement shows whether interrupting the processor's activity with a sleep statement is different from interrupting it with an I/O operation switch.

TABLE I. CODING STYLES – CODE SNIPPET

First Coding Style Approach

```
void selectionSortTest1(int *numArray) {
  long i, j, first, temp;
  //performing the selection sort
  for (i = ARRAY_SIZE - 1; i > 0; i--) {
    first = 0;
    for (j = 1; j <= i; j++) {
      if (numArray[j] < numArray[first]) {
        first = j;
      }
    }
    temp = numArray[first];
    numArray[first] = numArray[i];
    numArray[i] = temp;
  }
  //printing loop
  for(i=ARRAY_SIZE-1;i>=0;i--) {
    cout<<numArray[i]<<endl;
  }
}
```

Second Coding Style Approach

```
void selectionSortTest2(int *numArray) {
  long i, j, first, temp;
  //performing the selection sort
  for (i = ARRAY_SIZE - 1; i > 0; i--) {
    first = 0;
    for (j = 1; j <= i; j++) {
      if (numArray[j] < numArray[first]) {
        first = j;
      }
    }
    temp = numArray[first];
    numArray[first] = numArray[i];
    numArray[i] = temp;
    //This line is moved up
    cout<<numArray[i]<<endl;
  }
  cout<<numArray[0];
  //printing loop
  for(i=0;i<ARRAY_SIZE;i++) {
  }
}
```

Third Coding Style Approach

```
void selectionSortTest3(int *numArray) {
  long i, j, first, temp;
  //performing the selection sort
  for (i = ARRAY_SIZE - 1; i > 0; i--) {
    first = 0;
    for (j = 1; j <= i; j++) {
      if (numArray[j] < numArray[first]) {
        first = j;
      }
    }
    temp = numArray[first];
    numArray[first] = numArray[i];
    numArray[i] = temp;
    if(i%500 == 0) {
      Sleep(1);
    }
  }
  //printing loop
  for(i=ARRAY_SIZE-1;i>=0;i--) {
    cout<<numArray[i]<<endl;
  }
}
```

E. Machine Preparation

Before the experiments are executed, some preparations were done to ensure the standardization of the experimental setup and neutralization of any external effect. The machine is prepared by switching off all network cards, in order to reduce the Operating System's (OS) background update activities. The screen brightness is also set to the lowest brightness level, and the machine is switched to battery-powered mode. The machine is then shut down for half hour to allow the machine to cool down. When the machine is started back up, the machine is left idle for 15 minutes to make sure all the OS's start-up activities are complete before the experiments are commenced. The case study was executed on a machine with the specifications mentioned in Table II.

TABLE II. MACHINE SPECIFICATIONS

Model	Dell Inspiron 7520
Processor	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz (8 CPUs), ~2.8GHz
Hard Drive	M.2 SSD / PCIe NVMe, OPAL2: 256GB
Power mode	Battery powered
Battery	6-cell (47 Wh), internal
Operating System	Windows 10 Pro 64-bit (10.0, Build 15063) (15063.rs2_release.170317-1834)
Display	NVIDIA GTX 1050 Ti 2GB DDR5 graphics

IV. EXPERIMENTATION ON MINGW 5.2.0 X86_64 GCC 5.2.0 COMPILER

This section compares the three coding styles mentioned in Section III. The code is compiled using MinGW32 GCC 6.3.0 compiler, using the default compiler settings. Table III, Fig. 2, 3, and 4 present the results of the experiment.

The first approach, which separates the CPU intensive instructions from the input/output instructions, showed the best execution time, with an average of 499.549 seconds. However, it showed the worst application power with an average of 484.2 Milliwatt. It also showed a moderate application energy consumption, with an average of 219111 Millijoule.

The second approach, which is printing the output while the sorting is in progress, has slightly increased the execution time by 2.8%. However, it showed an Improvement in power & energy consumption; the power has decreased by 0.53%, and the energy consumption has decreased by 1.1%. This makes the second approach a good fit in power and energy-aware applications.

The third approach, which is interrupting the loop operation with a sleep statement, has increased the execution time by 1.71% due to the extra sleep statements added to the code. It also increased the total energy consumption by 1.28% from the first approach. However, it showed the best power measurement with a 1.11% decrease from the first approach. This means that the third approach is a good fit in power-aware applications, but not good in the energy-aware ones.

In conclusion, for the MinGW GCC compiler, switching between processing-intensive operations and I/O operations is a good approach for energy and power-aware systems. This approach reduces the power and energy consumption while having a small impact on the application performance.

TABLE III. MINGW GCC AVERAGE RESULTS

Measurements	Coding Styles		
	First Approach	***Second Approach***	***Third Approach***
Execution time (S)	499.549	513.525	508.123
Application Power (mW)	484	482	479
Application Energy consumption (mJ)	219112	216571	221920

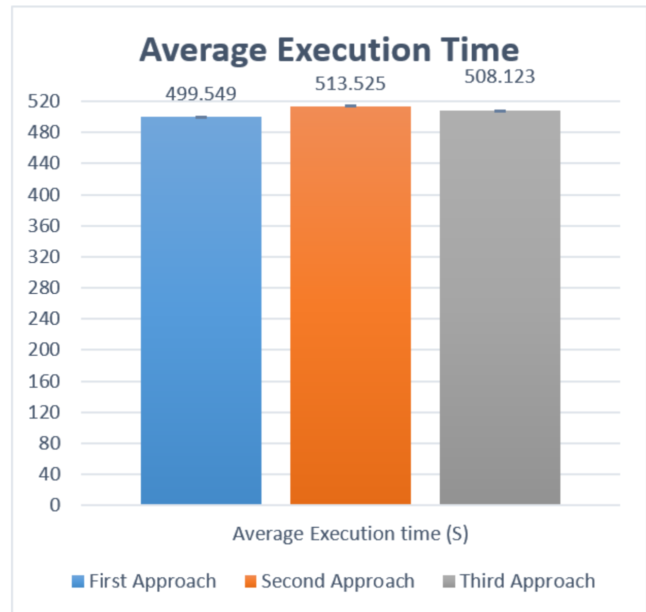

Fig. 2. MinGW average execution time graph. Error bars show standard deviation.

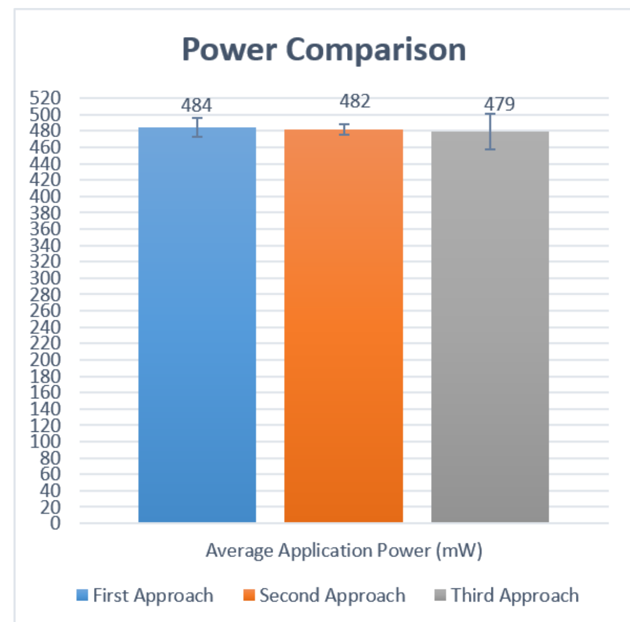

Fig. 3. MinGW average application power graph. Error bars show standard deviation.

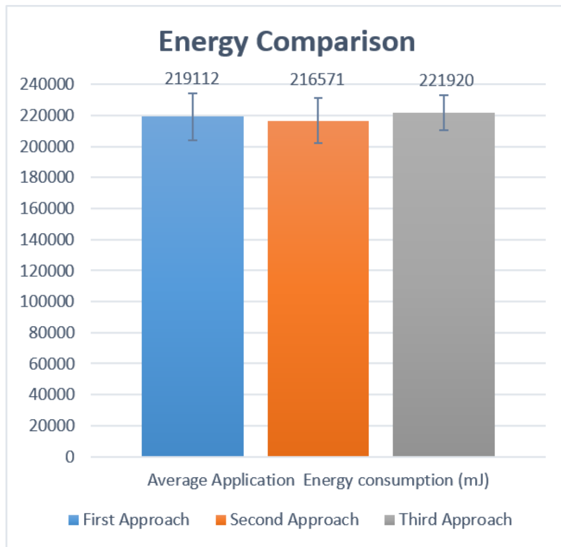

Fig. 4. MinGW average application energy consumption graph. Error bars show standard deviation.

V. Experimentation on Cygwin 2.0.4 X86_64 GCC 4.9.3 Compiler

This section compares the three coding styles mentioned in Section III. The code is compiled using Cygwin 2.0.4 x86_64 GCC 4.9.3 compiler, using the default compiler settings. Table IV, Fig. 5, 6 and 7 present the results of the experiment.

The first approach showed the worst application power, at an average of 516 Milliwatt, and the worst application energy consumption, at an average of 220651 Millijoule.

The second approach has the worst execution time with an increase of 0.9% from the first approach. However, it decreased the application energy consumption by 0.98%, and it showed the best application power with 3.1% decrease from the first approach.

The third approach unexpectedly showed 3.60% less execution time than the first approach. It showed a moderate application power, a decrease of 1.93% from the first approach. It also showed the best application energy consumption, a decrease of 1.35% from the first approach.

In conclusion, interrupting the processor's activity with a sleep statement is a good approach to use with Cygwin GCC compiler. It showed the best energy consumption and a balanced power measurement, without compromising the application performance.

TABLE IV. Cygwin Average Results

Measurements	**Coding Styles**		
	First Approach	***Second Approach***	***Third Approach***
Execution time (S)	318.848	321.999	307.361
Application Power (mW)	516	500	506
Application Energy consumption (mJ)	220651	218467	217659

Fig. 5. Cygwin average execution time graph. Error bars show standard deviation.

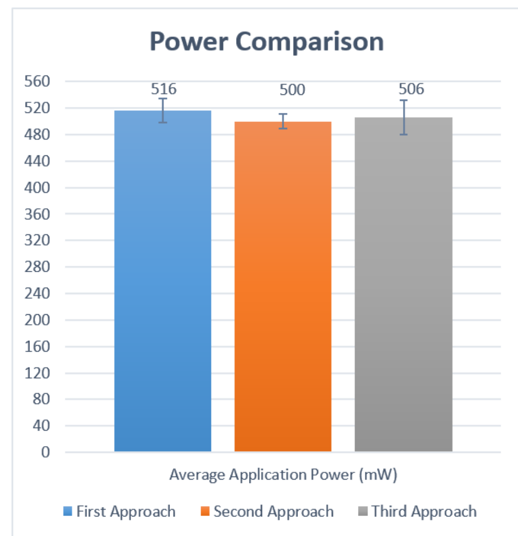

Fig. 6. Cygwin average application power graph. Error bars show standard deviation.

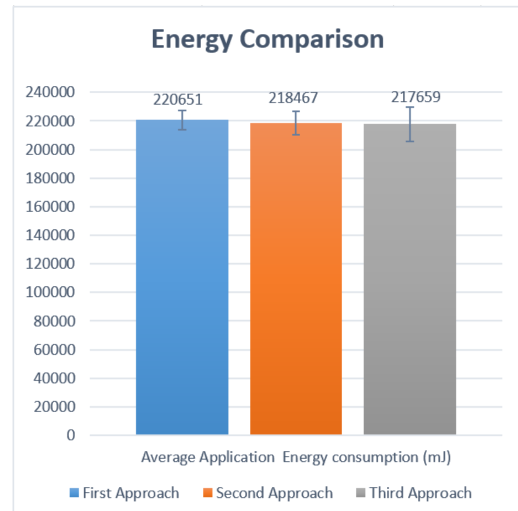

Fig. 7. Cygwin average energy consumption graph. Error bars show standard deviation.

VI. Experimentation on Borland C++ 5.5.1 for Win32 Compiler

This section compares the three coding styles mentioned in Section III. The code is compiled using Borland C++ 5.5.1 for Win32 compiler, using the default compiler settings. Table V, Fig. 8, 9 and 10 present the results of the experiment.

The first approach showed the best execution time of 202.723 seconds, and the best energy consumption of 79497 Millijoule. However, it showed the worst application power of all the three approaches, at an average of 555 Milliwatt. This makes the first approach a good fit for energy-aware systems, but not for power-aware ones.

The second approach showed balanced power, energy, and execution time measurements. It decreased the application power by 5.4% from the first approach, but increased the execution time by 1.98% and increased the energy consumption by 2.79%.

The third approach showed the best application power of 517 Milliwatt, with a 6.85% decrease from the first approach. However, it showed the worst execution time with an increase of 5.26% and the worst application energy consumption with an increase of 13.46% from the first approach.

In conclusion, if the target of our code enhancements is building an energy-aware application without compromising the performance, then separating the processing intensive operations from the I/O operations is a good approach to use with Borland C++ compiler. If our main target is to reduce the application power regardless of the total energy or performance, then interrupting the processor's activity with a sleep statement would be a good approach.

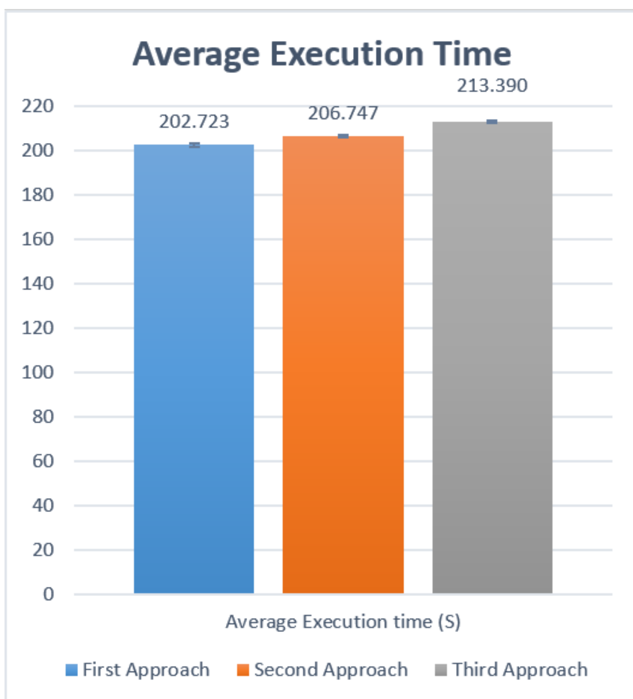

Fig. 8. Borland C++ average execution time graph. Error bars show standard deviation.

TABLE V. Borland C++ Average Results

Measurements	Coding Styles		
	First Approach	*Second Approach*	*Third Approach*
Execution time (S)	202.723	206.747	213.39
Application Power (mW)	555	525	517
Application Energy consumption (mJ)	79497	81719	90199

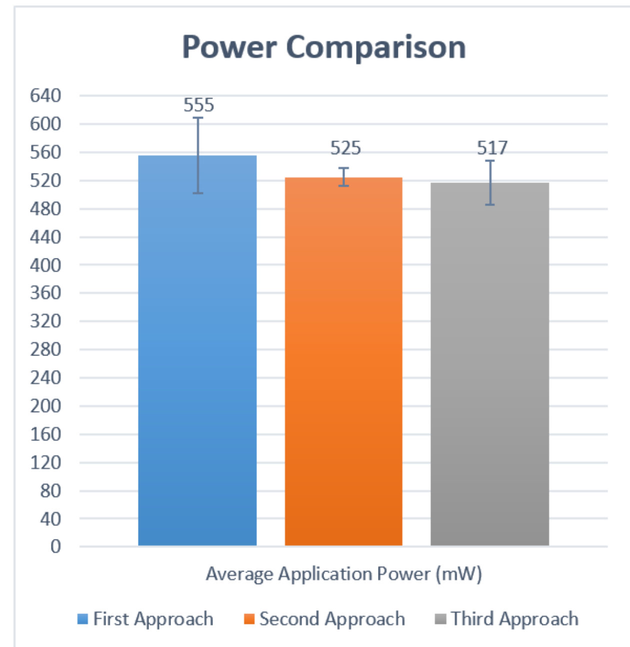

Fig. 9. Borland C++ average application power graph. Error bars show standard deviation.

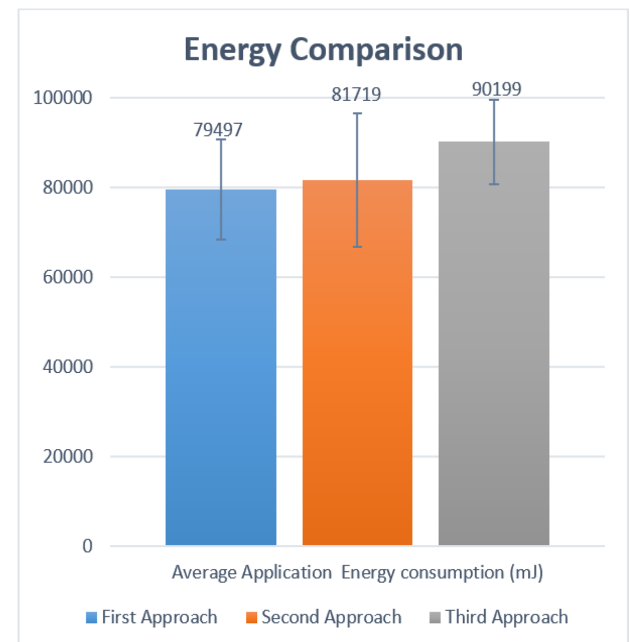

Fig. 10. Borland C++ average energy consumption graph. Error bars show standard deviation.

VII. Experimentation on Visual Studio 2013 Visual C++ Win32 Console Application

This section compares the three coding styles mentioned in Section III. The code is compiled using Visual Studio 2017 Visual C++ compiler, as a Win32 Console Application, using the default compiler settings. Table VI, Fig. 11, 12 and 13 present the results of the experiment.

The first approach showed a moderate execution time, with an average of 90.1515 seconds, a moderate application power, with an average of 0.5837 watts, and a moderate application energy consumption, with an average of 52.6019 joules.

The second approach showed the worst execution time: it increased by 10.95% from the first approach. Although the second approach showed the best application power with a 4.23% decrease, it showed the worst total application energy consumption with a 6.20% increase from the first approach. This is another indication that lower power does not always translate to lower total energy consumption.

The third approach showed similar results to the first approach with a very low and insignificant difference. It decreased the execution time by 0.71%, increased the power by 0.18%, and decreased the energy consumption by 0.51%. This is considered to be the same result as the first approach.

As a conclusion, separating the I/O instructions from the CPU intensive instructions is a good approach to follow when compiling with Visual C++. Interrupting the CPU intensive instructions with I/O instructions is only recommended for power-aware applications, but not highly recommended for performance and energy-aware applications when compiling with Visual C++.

Fig. 11. Visual C++ 2013 average execution time graph. Error bars show standard deviation.

TABLE VI. Visual C++ Average Results

Measurements	Coding Styles		
	First Approach	*Second Approach*	*Third Approach*
Execution time (S)	484.719	495.374	495.36
Application Power (mW)	487	478	472
Application Energy consumption (mJ)	221655	216255	218420

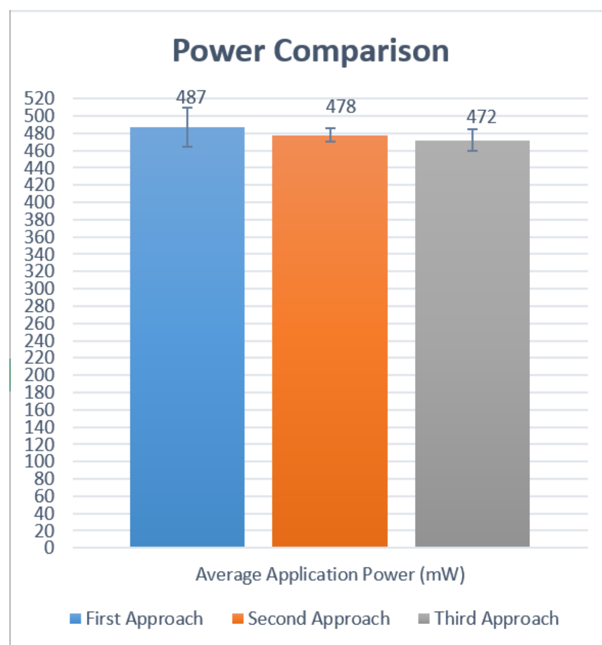

Fig. 12. Visual C++ 2013 average application power graph. Error bars show standard deviation.

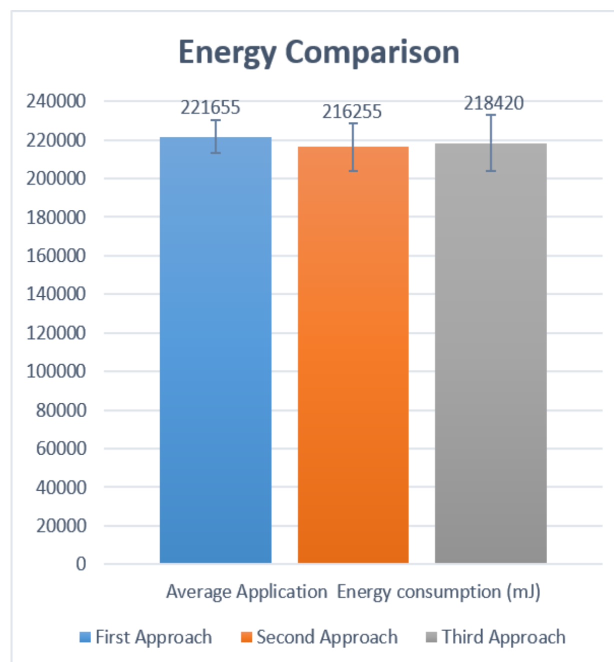

Fig. 13. Visual C++ 2013 average energy consumption graph. Error bars show standard deviation.

VIII. Compiler Comparison

It is evident that the choice of the compiler can heavily impact the execution time of an application. As shown in Fig. 15, Borland C++ compiler has an execution time of 58% ~ 59.73% less than the execution time of MinGW, 30.57% ~ 36.42% less than Cygwin, and 56.92% ~ 58.26% less than Visual C++.

Visual C++ and MinGW compilers are showing the lowest application power; their average power measurements are very close. They have an application power of 4.4% ~ 6.71% less than Cygwin, and 8.7% ~ 12.79% less than Borland C++, Fig. 16. These results show that Visual C++ & MinGW are good choices for power-aware applications, but not for energy-aware ones.

Because the Borland C++ compiled application has a low execution time, it used less total energy than all the other compilers. Fig. 14 shows that Borland C++ has an application energy consumption of 59.35% ~ 63.71% less than MinGW, 58.55% ~ 63.97% less than Cygwin, and 58.70% ~ 64.13 less than Visual C++.

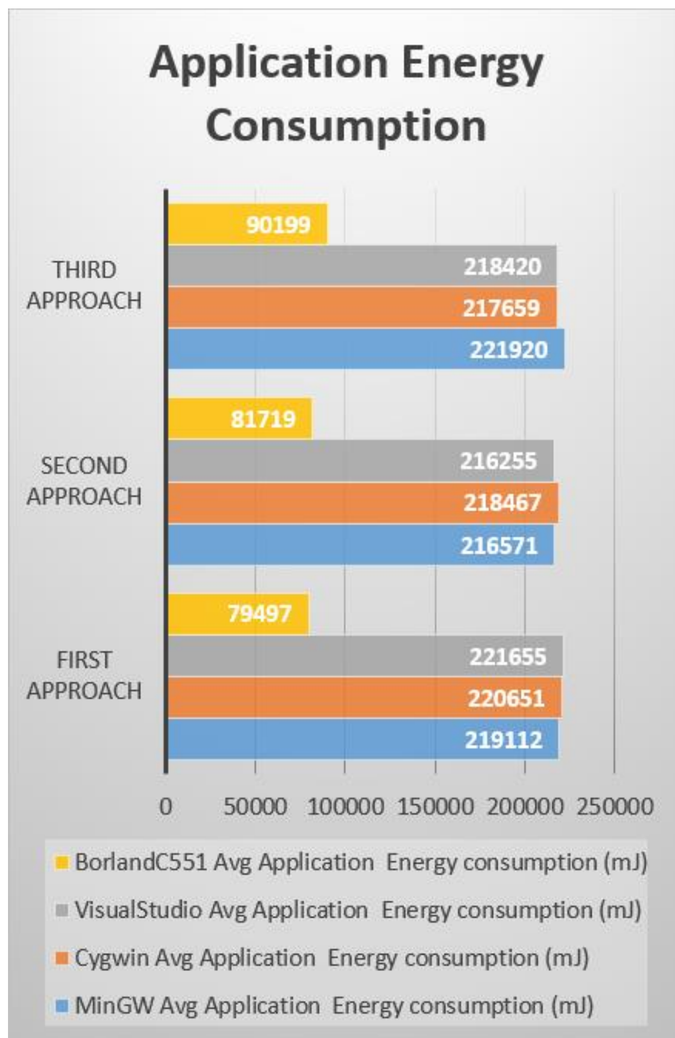

Fig. 14. Energy consumption comparison between different compilers.

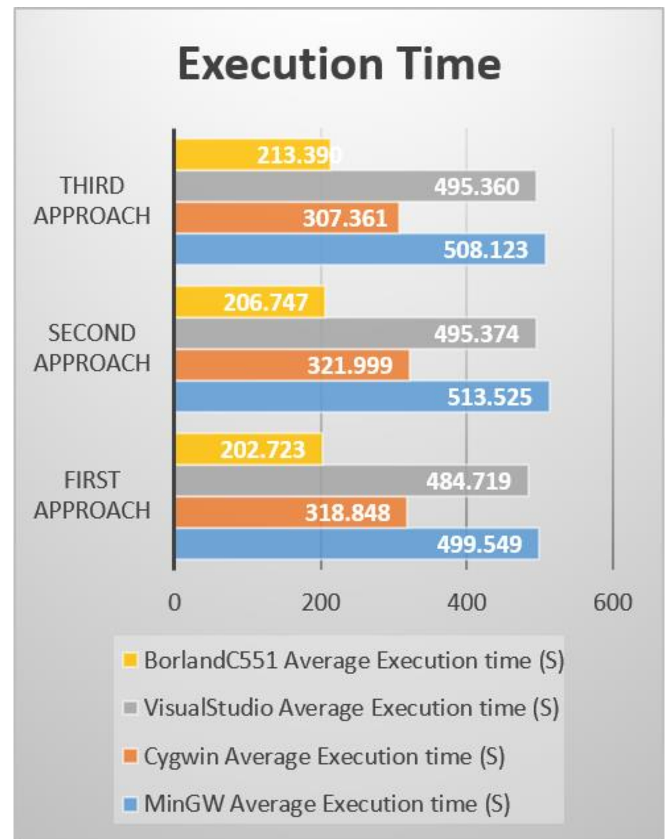

Fig. 15. Execution time comparison between different compilers.

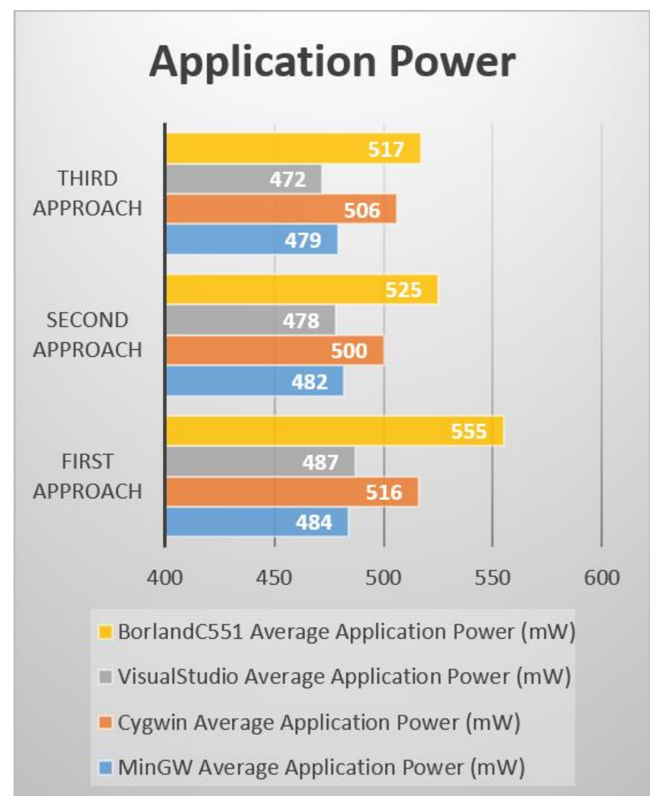

Fig. 16. Power comparison between different compilers.

IX. RESULTS VALIDATION

For validation, the authors applied the findings of the research to one of the well-known open source C/C++ applications, showing how much energy can be saved. According to the findings in Sections IV, V, VI, VII and VIII; the best performance and least energy consumption can be achieved by compiling the code with Borland C++ 5.5. We can also achieve a lower power by applying a sleep statement in the middle of the CPU intensive operations. The above sections claim that will be validated in this section is that the applications compiled by Borland C++ 5.5 should consume 58%~64 less energy than applications compiled with other compilers.

SQLite3 is a self-contained, serverless, zero-configuration, transactional SQL database engine. It is a lightweight database that supports SQL syntax of standard relational databases for complex queries [27].

SQLite3 open source application has been chosen for this experiment because of its wide usage in a large variety of software and products [28]-[30]. For example, SQLite is used in Apple's many native Mac OS-X and IOS applications, used as a meta-data storage for Firefox web browser and the Thunderbird e-mail reader from Mozilla, used as an application file format for Adobe Photoshop Lightroom, used as a primary data store on the client-side of the Dropbox file archiving and synchronization service, and it is also used in Google's Android OS and the Chrome Web Browser.

The source code has been downloaded from the official SQLite3 download page [31]; the downloaded package can be reached from [32]. The C code has been modified to fix the compilation issues raised from compiling with BorlandC55. It has also been modified to execute a sleep statement for 1 millisecond, every 500 loop rounds, in the DROP TABLE method. This experiment will focus only on the DROP TABLE statement, by executing 200 DROP TABLE statement. Each table contains 6,856,019 records, and each record consists of two integer columns. The DB is saved in a single file with a size over 4 GB.

The 200 statements are split over 5 execution sessions for power measurements. The reason for that split is to have multiple power measurements, where the standard deviation can be calculated, to find out how close the data are to average power. The average application power is then used along with the execution time to generate the average energy consumption. Whenever an average is calculated, the standard deviation is shown as an error bar at the top of the graph, to show how close the data are to the average.

Finally, the BorlandC55 result is compared to the result of the EXE file downloaded from the official SQLite3 website [33], to compare the energy savings, and the execution time performance.

A. *Experimentation Result*

As shown in Table VII, Fig. 17, 18 and 19, the application that has been compiled with BorlandC55 with the sleep statement has shown a 52.16% decrease in application power, a 58.13% decrease in application energy consumption, and a 13.14% decrease in execution time. This result is a strong indication of the validity of the findings in Sections IV, V, VI, VII and VIII.

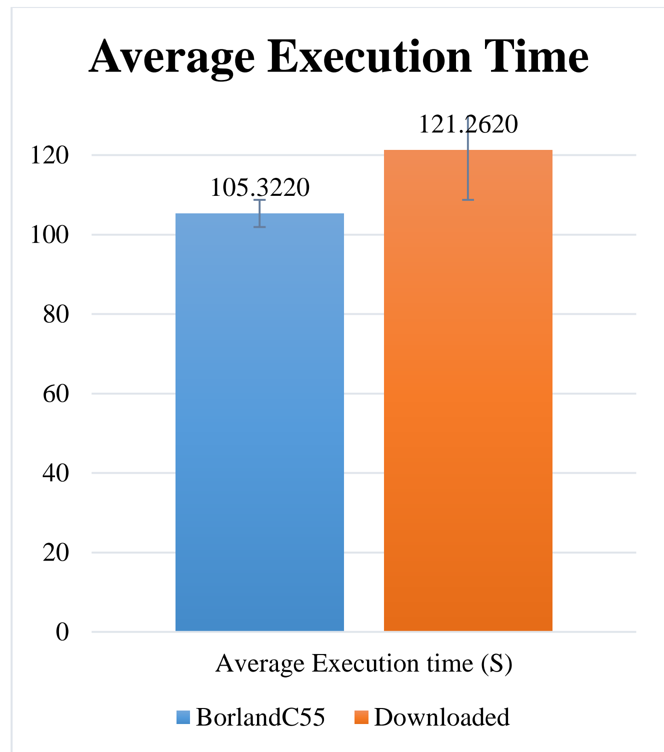

Fig. 17. SQLite3- Average Execution Time – Error Bar shows standard deviation.

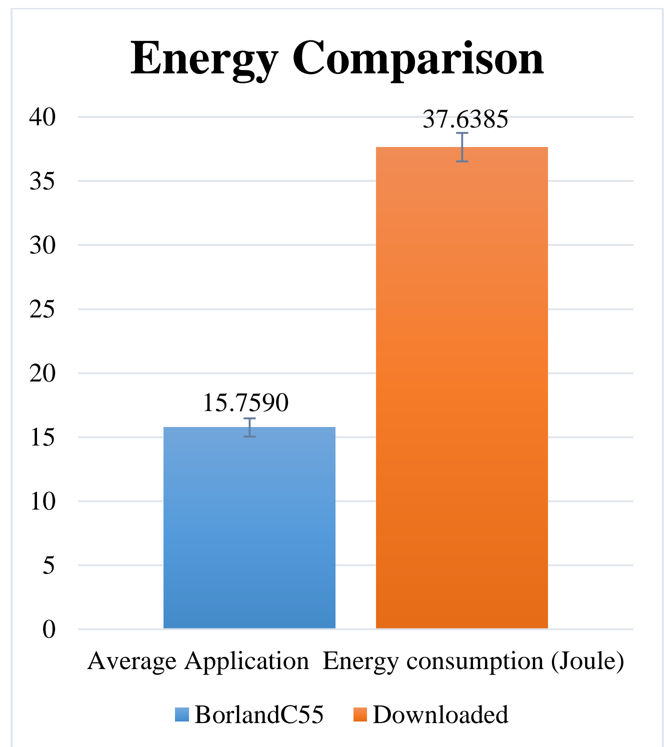

Fig. 18. SQLite3 - Energy Comparison - Error Bar shows standard deviation.

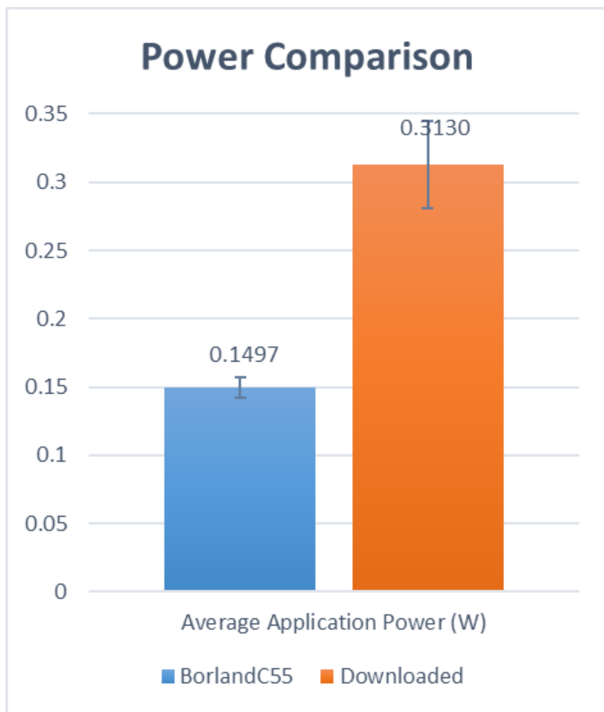

Fig. 19. SQLite3 - Power Comparison - Error Bar shows standard deviation.

TABLE VII. SQLITE3 - EXPERIMENTATION RESULT

Measurements	BorlandC55 Compiled EXE	Downloaded EXE
Execution time (S)	105.3220	121.2620
Application Power (W)	0.1497	0.3130
Application Energy consumption (Joule)	15.7590	37.6385

X. FUTURE WORK

The main contribution of this research lies in highlighting the impact of coding style and choice of the compiler on energy and performance efficiency. However, there are several lines of research arising from this study which should be pursued. The experimentation results show that the magnitude of the differences between the four compilers is significant. However, further investigation of why the difference is large is yet to be done, e.g. how different is the assembly code? What optimizations does each compiler use by default? What libraries do the compilers use? How is initialization done? Where in memory does the program get loaded? Does this matter?

The following points are some other ideas that could be an extension of this research:

- Applying compilation time enhancements, flags, and directives, instead of using the compiler's default settings, and detect the difference in performance and energy consumption.
- Implementing a dynamic power-aware framework, that automatically reduces the application's power when it reaches a certain level.
- Investigating the power-aware techniques for the Virtual Machine based programming languages, e.g. JAVA.

XI. CONCLUSION

This research has demonstrated that an important solution for finding the balance between performance, power, and energy consumption could be choosing the right coding style along with the right compiler that works best with the nature of the application and the target machine.

It also has shown that although in most of the cases high CPU performance means high application power, this is not universally valid, and low power does not always translate to lower total energy consumption.

The research also revealed that one coding style could work best for one compiler, but not for another compiler and that the most efficient coding style varies based on the system goals and constraints. In addition, enhancing the program's energy efficiency is not only dependent on the target machine and the type of program [34], but it is also dependent on how the program is written and compiled.

Furthermore, the research showed that for some compilers, interrupting the CPU intensive instructions with a sleep statement could be a simple and easy way of controlling the application's power. However, it may slightly impact the performance and total consumed energy. We have also shown that interrupting the CPU intensive instructions with I/O instructions is another way of reducing application power. However, in most of the cases, it negatively impacts the performance and the total energy consumption.

All the experimental results were then put into one comparison between compilers, showing how compiler choice can impact the performance, the power, and the total energy consumption of the application.

From the experimentations done on the selected four compilers and three coding styles, it has been found that the best performance and energy saving result can be achieved by compiling the application with Borland C++ 5.5 while separating the CPU intensive instructions from the input/output instructions. The lowest power, however, could be achieved by compiling the application with Visual C++ compiler while interrupting the CPU intensive instructions with a sleep statemen.

Finally, and most importantly, the research team validated the newly introduced software improvement by applying it to one of the well-known open source C/C++ applications (SQLite3). The results have shown 52.16% decrease in application power, 58.13% decrease in the application's energy consumption, and 13.14% decrease in execution time.

REFERENCES

[1] Hassan, Hesham, and Ahmed Shawky Moussa. "Power Aware Computing Survey." International Journal of Computer Applications 90, no. 3, 2014.

[2] Osman S. Unsal. "System-Level Power-Aware Computing in Complex Real-Time and Multimedia Systems" Doctor of Philosophy Doctoral Dissertation, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, 2008.

[3] Valluri, Madhavi, and Lizy K. John. "Is Compiling for Performance==Compiling for Power?" In Interaction between Compilers and Computer Architectures, pp. 101-115. Springer US, 2001.

[4] Orgerie, Anne-Cecile, Laurent Lefevre, and Jean-Patrick Gelas. "Demystifying energy consumption in grids and clouds." In Green Computing Conference, 2010 International, pp. 335-342. IEEE, 2010.

[5] Mittal, Radhika, Aman Kansal, and Ranveer Chandra. "Empowering developers to estimate app energy consumption." In Proceedings of the 18th annual international conference on Mobile computing and networking, pp. 317-328. ACM, 2012.

[6] Carroll, Aaron, and Gernot Heiser. "An Analysis of Power Consumption in a Smartphone." In USENIX annual technical conference, vol. 14. 2010.

[7] Perrucci, Gian Paolo, Frank HP Fitzek, and Jörg Widmer. "Survey on energy consumption entities on the smartphone platform." In Vehicular Technology Conference (VTC Spring), 2011 IEEE 73rd, pp. 1-6. IEEE, 2011.

[8] Ortiz, David A., and Nayda G. Santiago. "Impact of source code optimizations on power consumption of embedded systems." In Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on, pp. 133-136. IEEE, 2008.

[9] Pallister, James, Simon J. Hollis, and Jeremy Bennett. "Identifying compiler options to minimize energy consumption for embedded platforms." The Computer Journal 58, no. 1 (2015): 95-109.

[10] Ayse, Md Ashfaquzzaman Khan Can Hankendi, and Kivilcim Coskun Martin C. Herbordt. "Application Level Optimizations for Energy Efficiency and Thermal Stability." Power (W) 20, no. 10: 0.

[11] Pal, Ajit. "Low-Power Software Approaches." In Low-Power VLSI Circuits and Systems, pp. 355-386. Springer India, 2015.

[12] Dalal, Vishal, and C. P. Ravikumar. "Software power optimizations in an embedded system." In VLSI Design, 2001. Fourteenth International Conference on, pp. 254-259. IEEE, 2001.

[13] Simunic, Tajana, Luca Benini, and Giovanni De Micheli. "Energy-efficient design of battery-powered embedded systems." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9, no. 1 (2001): 15-28.

[14] De Lima, Ewerton Daniel, Tiago Cariolano de Souza Xavier, Anderson Faustino da Silva, and Linnyer Beatryz Ruiz. "Compiling for performance and power efficiency." In Power and Timing Modeling, Optimization and Simulation (PATMOS), 2013 23rd International Workshop on, pp. 142-149. IEEE, 2013.

[15] Pallister, James, Simon J. Hollis, and Jeremy Bennett. "Identifying compiler options to minimize energy consumption for embedded platforms." The Computer Journal 58, no. 1 (2015): 95-109, 2015.

[16] Purini, Suresh, and Lakshya Jain. "Finding good optimization sequences covering program space." ACM Transactions on Architecture and Code Optimization (TACO) 9, no. 4 (2013): 56

[17] Huang, Qijing, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Shannon Brown, and Jon Anderson. "The effect of compiler optimizations on high-level synthesis for FPGAs." In Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on, pp. 89-96. IEEE, 2013.

[18] Pan, Zhelong, and Rudolf Eigenmann. "Fast and effective orchestration of compiler optimizations for automatic performance tuning." In Code Generation and Optimization, 2006. CGO 2006. International Symposium on, pp. 12-pp. IEEE, 2006.

[19] Haneda, Masayo, Peter MW Knijnenburg, and Harry AG Wijshoff. "Automatic selection of compiler options using non-parametric inferential statistics." In Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on, pp. 123-132. IEEE, 2005.

[20] Windows Performance Analyzer, https://msdn.microsoft.com/en-us/library/windows/hardware/hh448170.aspx (accessed at: 12/10/2017 3:00pm).

[21] Introduction to WPA, https://msdn.microsoft.com/en-us/library/windows/hardware/hh448171.aspx (accessed at: 12/10/2017 3:30pm).

[22] WPA Features, https://msdn.microsoft.com/en-us/library/hh448220.aspx (accessed at: 12/10/2017 4:00pm)

[23] Introduction to the WPA User Interface, https://docs.microsoft.com/en-us/windows-hardware/test/wpt/introduction-to-the-wpa-user-interface. (accessed at: 12/10/2017 4:00pm)

[24] Markatos, Evangelos P., and Thomas J. LeBlanc. "Using processor affinity in loop scheduling on shared-memory multiprocessors." IEEE Transactions on Parallel and Distributed systems 5, no. 4 (1994): 379-400.

[25] Hannah Bayer and Markus E. Nebel, 2009, "Evaluating Algorithms according to their Energy Consumption", Mathematical Theory and Computational Practice.

[26] Chingren Lee, Jenq Kuen Lee, and TingTing Hwang, 2000, "Compiler Optimization on Instruction Scheduling for Low Power", 13th International Symposium on System Synthesis (ISSS'00), Madrid, Spain, 20-22.

[27] Zhao, Aite, Zhiqiang Wei, and Yongquan Yang. "Research on SQLite Database Query Optimization Based on Improved PSO Algorithm." International Journal of Database Theory and Application 9, no. 4 (2016): 239-246, 2016

[28] Jeon, Sangjun, Jewan Bang, Keunduck Byun, and Sangjin Lee. "A recovery method of deleted record for SQLite database." Personal and Ubiquitous Computing 16, no. 6 (2012): 707-715, 2012.

[29] Owens, Mike, and Grant Allen. SQLite. Apress LP, 2010.

[30] Well-Known Users of SQLite, https://www.sqlite.org/famous.html (accessed at: 06/09/2016 08:26pm)

[31] SQLite official download page, https://www.sqlite.org/download.html (accessed at: 08/07/2016 4:01pm)

[32] C source code as an amalgamation, version 3.13.0. (sha1: b46e199f06aa6f644989076da40227da68db7b6a)https://www.sqlite.org/2016/sqlite-amalgamation-3130000.zip - (Downloaded at: 08/07/2016 4:01pm)

[33] bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff.exe program, and the sqlite3_analyzer.exe program. (sha1: d74226e1cd38853f792266b221ae70c6c7b26835)https://www.sqlite.org/2016/sqlite-tools-win32-x86-3130000.zip - A (Downloaded at: 08/07/2016 4:01pm).

[34] Cooper, Keith D., and Todd Waterman. "Understanding energy consumption on the c62x." In Workshop on Compilers and Operating Systems for Low Power (COLP 02, co-located with PACT 02). 2002.