World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:7, No:4, 2013

Empirical Evaluation of Performance Optimization Techniques Used in Mobile Applications

Nathar Shah, Bu Kiat Seng

Open Science Index, Computer and Information Engineering Vol:7, No:4, 2013 publications.waset.org/9997045/pdf

***Abstract*—**Mobile application development is different from regular application development due to the hardware resource limitations existed in the mobile platforms. In the mobile environment, the application needs to be optimized by the developer to produce optimal software with least overhead. This study discussed about performance optimization techniques that are employed in general application development, and how such techniques are performing on mobile platforms through some empirical evaluations on a mobile emulator, Nokia X3-02 and Nokia C5-03devices. The scope of the work is only confined to mobile platform based on Java Mobile edition architecture. The empirical results showed that techniques such as loop unrolling, dependency chain, and linearized getter and setter performed better by a factor of 3 to 7. Whereas declaration and initialization on the same line or separate line did not improve the performance.

***Keywords*—**Optimization Techniques, Mobile Applications, Performance Evaluation, J2ME, Empirical Experiments.

I. Introduction

NOW-A-DAYS, the mobile content industry is developing at a fast pace that almost all people using some kind of mobile based applications in their daily life. Many mobile application developers see this as an opportunity and a challenge. High performing mobile applications are important to engage user experience and to maintain market share and revenue streams of a company [15]. There exist several performance enhancement techniques normally employed in application development. This research address the curiosity if such optimization techniques would result in any performance loss or gain in the mobile platform where the machine architecture is with less resources compared to personal computers or workstations. Besides that the study also would like to measure the performance accuracy shown by the standard emulator to represent actual devices. The study is important to guide mobile application developers to select appropriate performance optimization techniques when developing java based mobile applications. The effectiveness of the standard emulators in measuring performance close to the devices would help in deciding reliance on the emulators or actual devices when measuring implementation optimizations.

This particular study is important in image processing applications on mobile devices. Several commonly used operations in mobile applications like array extraction, increment, sum, shift, multiply, divide, less or equal, less, and equal are measured for their performance on mobile devices, and it was found that the division operation was the slowest among all in J2ME based operations, followed by the relational operations [6]. The same observation was also noticed on native Symbian application [7]. With just-in-time compilation, it is 4 – 6 times faster compared to interpretation on Intel XScale Technology [8]. Further, applications running on optimized J2ME virtual machine when compared to native C based applications are also inferior by 1.7 times.

Several object-oriented features used in J2ME were also investigated for performance optimization. Limiting the level of inheritance, minimizing the number of object creation, using alternative synchronization approaches (than lock based), eliminating inner classes, avoiding string concatenation, using shorter method/class names, and using lazy class loading are among the approaches that can lead to better performance [9], [10].

Focus has also been made to improve performance by tailoring virtual machine to specific target environment/profile, and deployment level optimization that targets application to specific environment besides the runtime optimization described earlier [11]. Virtual machines, when it is customized for a certain profile (e.g. Foundation profile of J2ME or JCL Xtr, a custom profile) produced reduced application startup-time and also less memory footprint when compared to generic virtual machines which can handle any profiles. The deployment level optimization also targets to reduce the delay when launching an application besides lowering the resource usage of the virtual machine when loading Java classes. It was found that JXE format was efficient to cut the start-up time of applications than the default JAR format by resolving classes, addresses, and methods before runtime that the level of processing required by the virtual machine during the runtime is substantially reduced. Other techniques available for code optimization are in-lining methods [12], call site devirtualization [14], and pre-compiling classes/methods to native code using ahead of time compilation [13].

The objective of this study was to investigate variety of other optimization techniques like loop unrolling, dependency chain, reference type declaration vs. reference type declaration and initialization, if statement vs. switch statement, and removing getters and setters for performance enhancements on mobile applications. The study then empirically examines them on actual phones (Nokia X3-02 and Nokia C5-03) and emulator to compare their performance on the same set of inputs. A discussion ends the study highlighting important results.

Nathar Shah and Bu Kiat Seng are with Faculty of Computing and Informatics, Multimedia University, Cyberjaya, 63100, Malaysia (e-mail: nathar.packier@mmu.edu.my).

Open Science Index, Computer and Information Engineering Vol:7, No:4, 2013 publications.waset.org/9997045/pdf

II. Methods for Performance Optimizations

Performance optimization methods such as code rewriting, loop unrolling, "switch" statement instead of sequential "if" statements, and getter and setter elimination are used in our empirical experiments [1], [3].

A. Reference Type Declaration vs. Reference Type Declaration and Initialization

An object which is initialized at declaration saves computation time compared to separate assignment (Fig. 1).

```
MyClass obj = data;
//instead of
MyClass obj;
obj = data;
```

Fig. 1 Object initialized during declaration

B. Loop Unrolling and Dependency Chain

For small sized iterations, unrolling the loop statements will provide better low machine level efficiency. Therefore, it is better to remove loops (i.e. loop unrolling) and repeat the body of the loop several times. The other approach is using dependency chain [1]. Dependency chain is a method to perform the intended operation several times in one loop and increase the loop variable by the number of times the operation was performed (see Fig. 2 for an illustration).

```
for( int i = 19; i >= 0; i -= 4 ){
sum += a[i];
sum += a[i - 1];
sum += a[i - 2];
sum += a[i - 3];}
```

Fig. 2 Example of dependency chain

C. If Statement vs. Switch Statement

A sequentially ordered switch statement is faster than if statement in most case. However, there wasn't a big difference in the computation time if the cases are arranged randomly (not in sequence).

D. Removing Getters and Setters

Additional overhead in calling getter and setter methods can be avoided if the data members are accessed directly especially in interpreted platforms like Android, J2ME, and Blackberry. However, this method is risky as it will break encapsulation.

III. Experiment Methods and Analysis

A. Experiment Design

The experiment was designed in such a way that a given technique was compared on different complexities. This was to study the performance pattern under different complexities. The complexities are specified differently based on the technique under study. For example in a technique that compares reference type declaration with another that does reference type declaration and initialization, the complexity is defined in terms of number of declaration and initialization lines. In the study, there are 4 ranges of complexities under study: simple, moderate, complex, and very complex. In other words, each technique will be measured for performance gain/loss in an increasing order of complexity. The rationale in doing so is to investigate if at different complexities, the performance will be affected by alpha components like caching efficiency, or overheads.

The experiment setup was to investigate the neutrality of the performance enhancement techniques when executed on different hardware platforms: Nokia X3-02 and Nokia C5-03. The experiment was also executed on a standard Java ME WTK emulator to compare the differences in the emulator profiling with actual hardware's profiling. Nokia X3-02 has 680 MHz speed processor, with 64 MB of RAM; Nokia C5-03 is with 600MHz ARM 11 processor and 128MB of RAM. The experiments were repeated 3 times to obtain the mean time and standard deviation. The standard deviation measurement indicates the degree of error in the experiments.

The experiments are driven to answer the following questions: Is an emulator a true representation of a device's performance? Which kind of optimization technique results in better performance? And is the optimization performance similar across the same platform mobile phones?

B. Statistical Analysis

1. Optimization Technique: Reference Type Declaration vs. Reference Type Declaration and Initialization

This technique was tested by increasing the complexity of the lines of code gradually from just five (scale: simple) to forty (scale: complex) variable declarations and initializations. Fig. 3 shows the mean and standard deviation readings for both approaches when experimented on different mobile devices. The graphs (Figs. 4-6) are the graphical visualizations of the mean time. Nokia X3-02 takes lesser time compared to Nokia C5-03 when the complexities of the declarations are increased. The 80MHz extra processing power of Nokia X3-02 resulted in the performance increase. This can be seen at the uniform rate change in the processing time across different complexities for Nokia X3 and Nokia C5. Undeniably the emulator which was running on a standard personal computer performed faster due to its faster processors compared to the mobile devices. It can be deduced that under a faster CPU clock tick, same line initialization and declaration marginally performs less than initialization after declaration.

These results signify that declaration and initialization on the same line or not does not influence much on the performance. Programmers do not need to worry on this order as its contribution to performance increase is negligible even at higher complexity. This observation is neutral to the device the study tested.

Open Science Index, Computer and Information Engineering Vol:7, No:4, 2013 publications.waset.org/9997045/pdf

Nokia X3-02		
Initialization during declaration		
Standard deviation	mean	Complexity
15.88	75.02	Simple
15.08	149.46	Moderate
15	308.87	Complex
18.89	620.65	Very complex
Initialization after declaration		
Standard deviation	mean	Complexity
15.56	74.75	Simple
14.48	148.73	Moderate
15.13	308.06	Complex
20.04	619.67	Very complex

(a)

Emulator		
Initialization during declaration		
Standard deviation	mean	Complexity
4.48	13.19	Simple
4.69	24.01	Moderate
5.5	45.49	Complex
7.23	88.03	Very complex
Initialization after declaration		
Standard deviation	mean	Complexity
2.4	12.25	Simple
2.33	21.33	Moderate
1.8	39.62	Complex
1.97	76.97	Very complex

(b)

Nokia C5-03		
Initialization during declaration		
Standard deviation	mean	Complexity
0.44	96.08	Simple
2.43	190.58	Moderate
2.25	385.61	Complex
1.88	805.42	Very complex
Initialization after declaration		
Standard deviation	mean	Complexity
1.3	96.62	Simple
0.71	191.53	Moderate
3.47	363.92	Complex
2.21	810.95	Very complex

(c)

Fig. 3 (a)–(c) Mean and standard deviation time (in milliseconds) taken by declaration and initialization for different devices

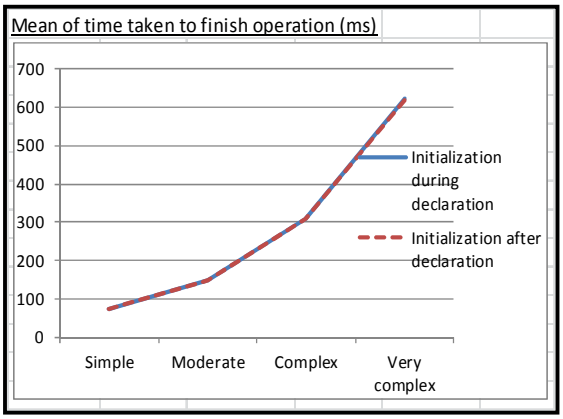


Fig. 4 Mean time on Nokia X3-02

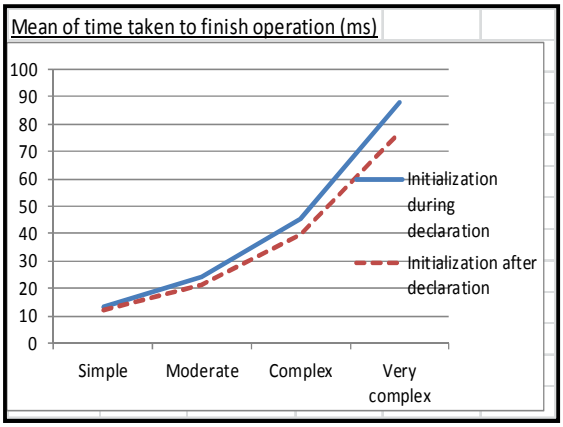


Fig. 5 Mean time on Emulator

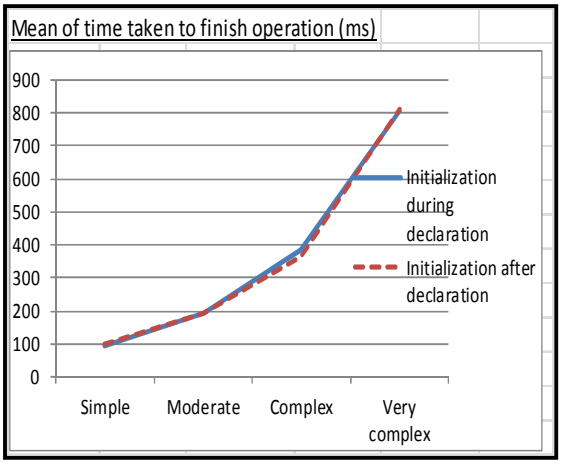


Fig. 6 Mean time on Nokia C5-03

2. Optimization Technique: Loop Unrolling

The complexity of this test was based on the number of loops unrolled. The number of loops was ranged from 5 (simple) to 40 (very complex). Comparing the results presented in Figs. 7-9, Nokia C5-03 takes between 8 to 10 times more time to process both rolled and unrolled loops as opposed to the time taken by Nokia X3-02 and the Emulator. Unrolling is an effective performance optimization technique across all the mobile devices. As can be seen in Figs. 10-12,

Open Science Index, Computer and Information Engineering Vol:7, No:4, 2013 publications.waset.org/9997045/pdf

the margin of efficiency gets better on higher loop complexities. This can be seen through the spread between the blue (loop unrolling) line and the red (standard for loop) line in those graphs. Among all the graphs, the efficiency growth of loop unrolling (vs. standard for loop) between the complex and the very complex is by a factor 2 to 3.The margin of error shown by the standard deviation is small for this experiment. The high effectiveness of loop unrolling is due to no overhead of loop variable creation and destruction in the loop body.

The results are significant to show that loop unrolling is an effective optimization technique on the mobile phones as the performance gain can be observed from multiple phones in the market. It is recommended to unroll loop constructs in mobile applications having dense loops/nested loops as the performance gain is worth the hassle of unrolling it. The emulator can be used as a relative measure of performance gain as the reading is consistent with the performance gain on the physical devices.

loop unrolling		
Standard deviation	mean	Complexity
0.63	1.14	Simple
0.49	1.42	Moderate
1.41	2.99	Complex
0.74	4.22	Very complex
standard for loop		
Standard deviation	mean	Complexity
0.55	2.4	Simple
1.45	4.3	Moderate
1.33	6.88	Complex
4.12	11.71	Very complex

Fig. 7 Loop unrolling on Nokia X3-02

loop unrolling		
Standard deviation	mean	Complexity
0.43	0.76	Simple
0.3	1.1	Moderate
0.5	1.45	Complex
0.52	2.32	Very complex
standard for loop		
Standard deviation	mean	Complexity
0.52	2.23	Simple
0.42	4.04	Moderate
1.43	7.16	Complex
0.99	14.85	Very complex

Fig. 8 Loop unrolling on an Emulator

loop unrolling		
Standard deviation	mean	Complexity
0.6	3.75	Simple
0.59	5.31	Moderate
0.59	8.47	Complex
0.45	14.77	Very complex
standard for loop		
Standard deviation	mean	Complexity
0.45	16.49	Simple
0.84	29.45	Moderate
0.49	55.12	Complex
0.72	106.8	Very complex

Fig. 9 Loop unrolling on Nokia C5-03

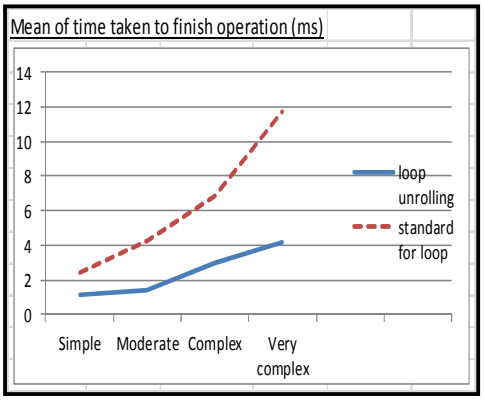


Fig. 10 Loop unrolling on Nokia X3-02

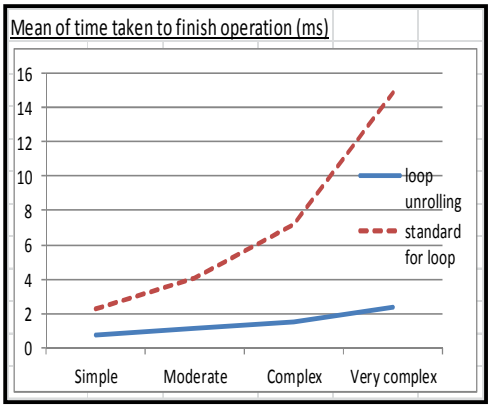


Fig. 11 Loop unrolling on an Emulator

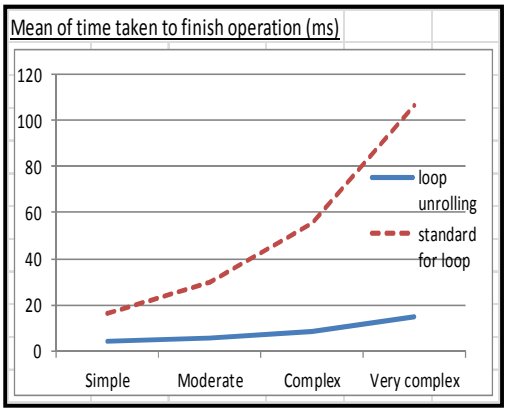


Fig. 12 Loop unrolling on Nokia C5-03

Open Science Index, Computer and Information Engineering Vol:7, No:4, 2013 publications.waset.org/9997045/pdf

3. Optimization Technique: Dependency Chain

The complexity of this test was based on the number of dependency unrolled. The number of loops was ranged from 10 (simple) to 1000 (very complex). As shown in Figs. 16-18, looping using dependency chain performs better than the ordinary "for" loop. Though, the performance pattern is the same for both Nokia X3-02 and Nokia C5-03, however, there is a ten fold efficiency in Nokia X3-02. The performance efficiency is drastic when the complexity increases. The study observed low margin of error (i.e. standard deviation) in all the experiments as shown in Figs. 13-15. When run on an emulator, the results of conventional "for" loop and dependency chain loop was close and both was running at a higher efficiency than those on the mobile device.

The results signify that at higher loop complexities, the programmers using the dependency chaining approach would be able to take advantage of the pipelining done in the processor [2] to increase the performance of their application. The emulator does not capture the same pattern of improvement due to the differences in the cache lines of the mobile and the desktop processors. In this instance, it is seen that the emulator is not a direct representative of performance improvements in the mobile phones.

dependency chain		
Standard deviation	mean	Complexity
0.75	2.71	Simple
2.92	8.11	Moderate
3.35	14.18	Complex
4.6	122.67	Very complex
standard for loop		
Standard deviation	mean	Complexity
1.77	4.53	Simple
2.64	14.1	Moderate
4.15	26.99	Complex
4.67	252	Very complex

Fig. 13 Dependency unrolled measurements on Nokia X3-02

dependency chain		
Standard deviation	mean	Complexity
0.7	3.33	Simple
2.15	11.39	Moderate
6.32	17.59	Complex
7.05	31.51	Very complex
standard for loop		
Standard deviation	mean	Complexity
0.76	4.11	Simple
4.8	14.55	Moderate
6.79	29.49	Complex
14.51	37.04	Very complex

Fig. 14 Dependency unrolled measurement on Emulator

dependency chain		
Standard deviation	mean	Complexity
0.87	17.4	Simple
0.54	72.47	Moderate
0.51	141.41	Complex
2.45	1389.4	Very complex
standard for loop		
Standard deviation	mean	Complexity
0.59	29.36	Simple
0.57	132.49	Moderate
0.63	261.41	Complex
7.13	2615.47	Very complex

Fig. 15 Dependency unrolled measurements on Nokia C5-03

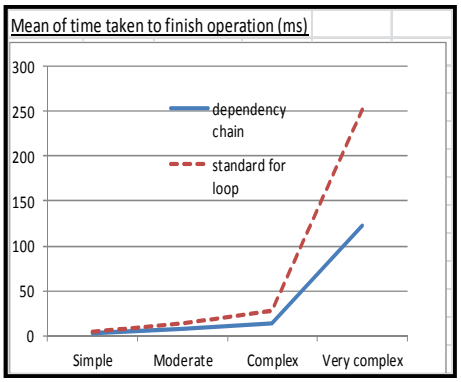


Fig. 16 Dependency unrolled result on Nokia X3-02

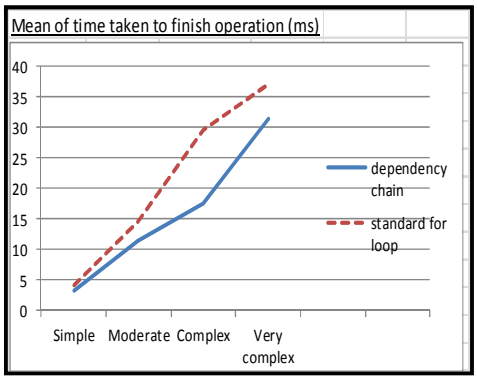


Fig. 17 Dependency unrolled result on Emulator

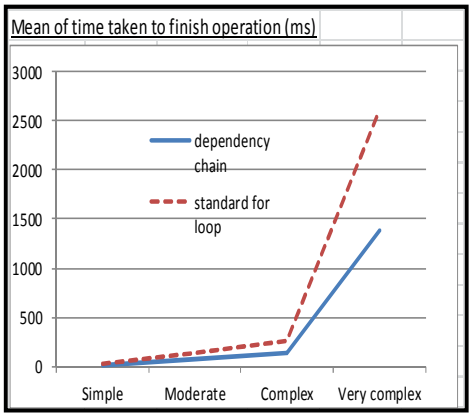


Fig. 18 Dependency unrolled result on Nokia C5-03

Open Science Index, Computer and Information Engineering Vol:7, No:4, 2013 publications.waset.org/9997045/pdf

4. Optimization Technique: "If" Statement vs. "Switch" Statement

The complexity of this test was based on the number of cases for the "If-else" and "Switch" statements. The determining condition was randomly generated to provide fair results. The number of conditions was ranged from 5 (simple) to 25 (very complex). As shown in Figs. 19-21, the time taken by "if-then-else" execution is comparatively larger than executing a "switch" statement. In Nokia C5-03, the "switch" performance was two fold compared to Nokia X3-02 at high complexity. At moderate complexity, as can be seen in Figs. 22-24, the performance is the best for both the techniques across device platform. In Nokia X3-02, the "if-then-else" technique even managed to get better than the "switch" statement at moderate complexity.

This experiment signifies what sort of programming pattern that would result in higher performance. The "switch" outperform "if-else" if the switch cases are arranged in a consecutive manner. This is because such order gets accessed in a table manner where the accesses complexity is by order of n, the number of elements.

switch		
Standard deviation	mean	Complexity
2.21	11.83	Simple
2.18	11.79	Moderate
2.84	12.21	Complex
3.18	12.26	Very complex
if then else		
Standard deviation	mean	Complexity
3.07	12.13	Simple
2.25	11.6	Moderate
3.34	12.85	Complex
2.2	13.45	Very complex

Fig. 19 "switch" vs. "if-then-else" measurements on Nokia X3-02

switch		
Standard deviation	mean	Complexity
0.56	4.73	Simple
0.64	4.63	Moderate
0.38	4.85	Complex
0.48	4.64	Very complex
if then else		
Standard deviation	mean	Complexity
0.61	5.18	Simple
0.56	5.26	Moderate
0.51	5.76	Complex
0.59	6.56	Very complex

Fig. 20 "switch" vs. "if-then-else" measurements on an Emulator

switch		
Standard deviation	mean	Complexity
0.78	12.47	Simple
0.55	10.33	Moderate
0.89	12.51	Complex
0.91	12.61	Very complex
if then else		
Standard deviation	mean	Complexity
1.02	15.5	Simple
0.94	13.54	Moderate
0.71	18.98	Complex
1.1	24.11	Very complex

Fig. 21 "switch" vs. "if-then-else" measurement on Nokia C5-03

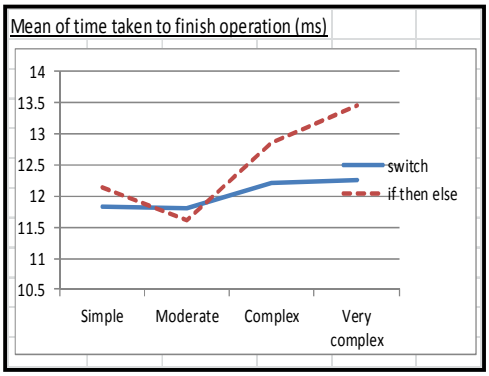


Fig. 22 "if" vs. "switch" performance on Nokia X3-02

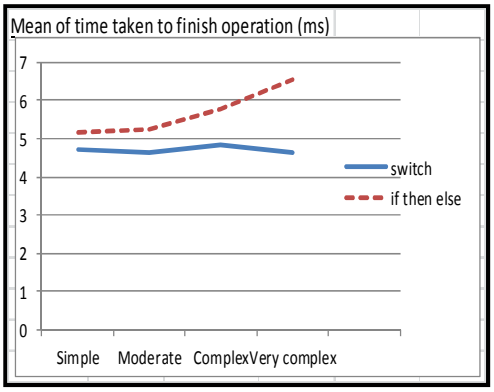


Fig. 23 "if" vs. "switch" performance on an emulator

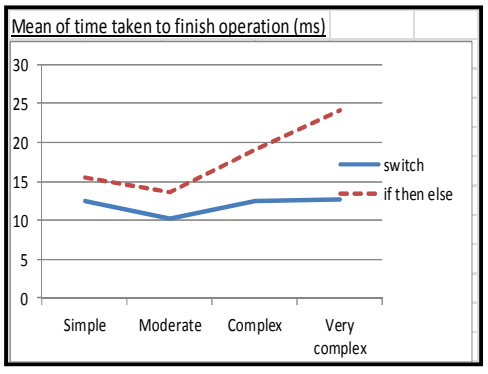


Fig. 24 "if" vs. "switch" performance on Nokia C5-03

Open Science Index, Computer and Information Engineering Vol:7, No:4, 2013 publications.waset.org/9997045/pdf

5. Optimization Technique: Removing Getter and Setter

When the getter and the setters are removed, the result is linear code without any encapsulation. This removes the overhead of method calls. Figs. 25-27 show the reading obtained for different complexities of getter/setter compared to without getter/setter. As can be seen in Figs. 28-30, regardless of the hardware platforms, the results unanimously show that without the getter and setter, the performance got better. For example, Nokia X3-02 and the emulator experienced many fold improvements. In Nokia C5-03, the performance got significantly better on increasing complexity. The margin of error (shown by the standard deviation) was significantly low in most cases.

The results below signify many fold increase in the performance if encapsulation not used. A programmer should write mobile application in a plain flat manner without encapsulation to take the performance advantage. It is worthwhile to repeat some code over and over again for this purpose although it badly affects the maintenance of the code.

without getter or setter		
Standard	mean	Complexity
0.33	1.92	Simple
0.49	3.35	Moderate
1.13	5.81	Complex
2.89	10.41	Very complex
with getter or setter		
Standard	mean	Complexity
0.39	1.84	Simple
1.82	3.86	Moderate
2.32	5.96	Complex
28.89	2376.47	Very complex

Fig. 25 Performance with and without the assessors on Nokia X3-02

without getter or setter		
Standard	mean	Complexity
0.65	2.8	Simple
0.61	4.14	Moderate
0.92	7.47	Complex
0.34	14.98	Very complex
with getter or setter		
Standard	mean	Complexity
3.77	14.7	Simple
2.7	29.73	Moderate
5.21	30.28	Complex
2.81	28.69	Very complex

Fig. 26 Performance with and without the assessors on Emulator

without getter or setter		
Standard	mean	Complexity
0.6	13.95	Simple
1.1	25.96	Moderate
0.54	49.69	Complex
0.47	97.24	Very complex
with getter or setter		
Standard	mean	Complexity
0.99	18.41	Simple
1.31	34.86	Moderate
1	67.63	Complex
1.19	133.1	Very complex

Fig. 27 Performance with and without the assessors on Nokia C5-03

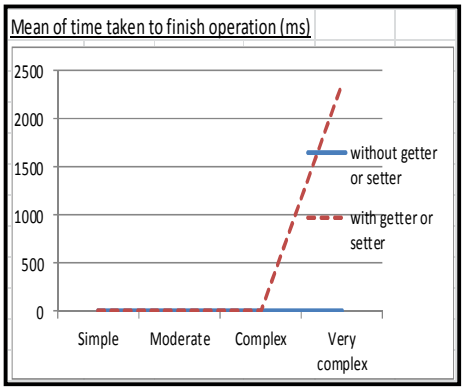


Fig. 28 Performance with and without the assessors on Nokia X3-02

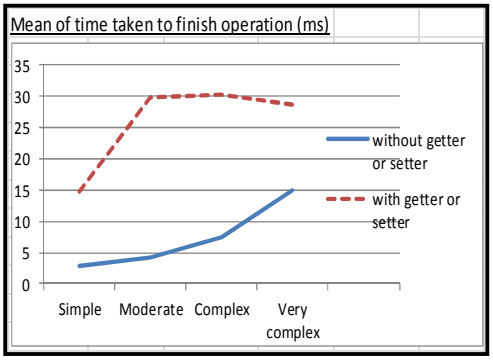


Fig. 29 Performance with and without the assessors on Emulator

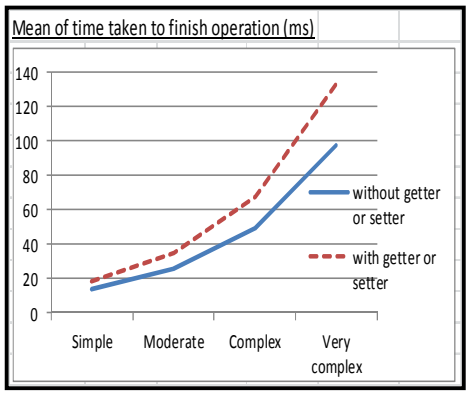


Fig. 30 Performance with and without the assessors on Nokia C5-03

Open Science Index, Computer and Information Engineering Vol:7, No:4, 2013 publications.waset.org/9997045/pdf

IV. Discussions

The study lists several optimization techniques and their degree of improvements from the empirical experiments the study has conducted. As shown in the earlier graphs, initialization during declaration or later did not show difference in performance although there was difference during the testing on the emulator when tested on different complexities. It is taking longer for the operations to be performed on the actual devices than on the emulator. Logically, the initialization should take longer when performed with the declaration (done on the same line or separately), but on mobile devices, they are negligible compared to the total time to perform the operations.

The loop unrolling technique shows a factor of 3 to 7 improvements compared to the standard loop approach. The removal of the overheads in the standard loop contributed to such efficiency and it is recommended to repeat statements when they are a few to leverage on the performance than using looping constructs. The experiments on different platforms revealed the same pattern of improvements and the standard deviations between multiple experiments are very small, therefore it is reliable. If the loop is large there will be insufficient registers available to hold the variables' values which will force the compiler to use the disk memory which in return will slow down execution [1], [2], [4].

Dependency chain optimization method performs best at higher complexity. Experiments show at higher complexity setting, the technique offers two times improvement compared to the standard looping approach. Similar to loop unrolling, chain dependency saves the compiler from loop overheads and keeps the execution pipeline full instead of being idle until the loop finishes [2].

From the result, the study has proved that "switch" statement is faster than "if-else" statement. However, the "switch" statement need to be consecutively arranged for it to be faster than the "If-else" statement. A "switch" structure in general is cheaper execution-wise than "if-else" chains. However, there are cases where a "switch" performs as bad as "if-else". If the case expressions in a switch statement are continuous or nearly continuous then the compiler translates them into a jump table instead of a comparison chain. A jump table improves performance because it reduces the number of branches to a single procedure call and shrinks the size of the control-flow code no matter how many cases there are. It will enable the processor to perform successful branch prediction which will reduce the jumping time and improve performance. On the other hand, if the cases are not continuous the compiler will spend time converting them into a comparison chain like if-else statements and this uses dense sequential conditional branching [5].

When the getter and setter are linearized there is a noticeable improvement when the data is accessed directly without using the getter and setter. This was because the overhead of calling and returning of the getter and setter was removed. Although it removes the number of operations needed to get or set the data, it also removes the encapsulation from the object. Therefore, there is a tradeoff to make between performance and maintainability through encapsulation.

V. Conclusion

In conclusion, from the experiments that has been conducted in the study, variable declaration or declaration with initialization does not give any major performance efficiency. On the other hand, loop unrolling is an optimization technique that gives the highest magnitude of efficiency especially when the complexity increases. On the same note, the study also observed that the dependency chain approach gave two fold efficiency compared to the one which did not use the approach. However, the "switch" statement does not always give performance advantage compared to the "if–else" statement. The advantage of the "switch" statement is only when the "case" statements are consecutive in nature. Finally, removing the getter and the setter did give an improvement to the performance, but the trade-off is with the modularity and maintainability of such implementation.

References

[1] Chehimi, F., P. Coulton and R.F. Edwards, 2006. C++ optimizations for mobile applications. In: Proceedings of the IEEE 10th International Symposium on Consumer Electronics, June 28-July 1, 2006, Russia, pp: 1-6.

[2] Hanson, J., 2006. Accelerating compute intensive functions using C. http://www.drdobbs.com/windows/184406483?pgno=2.

[3] Jeffrey, S., 2009. Coding for life -- battery life, that is. San Francisco, CA, USA., http://www.google.com/events/io/2009/sessions/CodingLifeBatteryLife.html

[4] Kalev, D., 2001. The top 20 C++ tips of all time. http://www.devx.com/cplus/Article/16328

[5] Zeichick, A., 2004. Optimizing Your C/C++ Applications, Part 1. http://developer.amd.com/documentation/articles/pages/6212004126.aspx.

[6] Tierno, J. and C. Campo, 2005. Smart Camera Phones: Limits and Applications. Pervasive Computing, IEEE, Vol 4, Issue 2, pp. 84-87.

[7] Campo, C., C. Garcia-Rubio and A. Cortes, 2009. Performance Evaluation of J2ME and Symbian Applications in Smart Camera Phones, 3rd Symposium of Ubiquitous Computing and Ambient Intelligence 2008, Vol 51/2009, pp. 48-56.

[8] Domer, J., M. Nanja, S. Srinivas and B. Keshavachar, 2004. Comparative performance analysis of mobile runtimes on Intel XScale technology. Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators, June 7, 2004, Washington, DC., USA., pp: 51-57.

[9] Shirazi, J., 2003. Java Performance Tuning (2nd Edition), O'Reilly Media, USA., Pages: 570.

[10] Wilson, S. and J. Kesselman, 2000. Java Platform Performance: Strategies and Tactics, Addison-Wesley, New York, ISBN: 9780201709698, Pages: 230.

[11] Mihailescu, P., H. Lee and J. Shepherdson, 2005. Optimization techniques for J2ME based mobile application. Proceedings of the 5th WSEAS International Confernce on Applied Informatics and Communication, September 15-17, 2005, Malta, pp: 181-186.

[12] Tyma, P., 1996. Tuning Java Performance, Dr. Dobb's Journal, Vol 21, No. 4, pp 52 – 58.

[13] Muller, G., B. Moura, F. Bellard and C. Consel, 1996. Harissa: A flexible and efficient java environment mixing bytecode and compiled code. Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems, June 17-21, 1996, Toronto, Ontario, Canada, pp: 1-20.

[14] Ishizaki, K., M. Kawahito, T. Yasue, H. Komatsu and T. Nakatani, 2000. A study of devirtualization techniques for a Java Just-In-Time compiler. Proceedings of the 15th ACM SIGPLANconference on Object-oriented programming,systems, languages and applications, October 15-19, 2000, Minneapolis, MN, USA., pp: 294-310.

[15] Tomlinson, M., 2012. SOA world magazine. http://soa.sys-con.com/node/2116107.