An Unobtrusive Entropy Based Compiler Optimization Comparator

With Introduction to Symbiotic Differential Comparison Algorithm

Mohammed F. Mokbel[α], Christopher D. Cambly[β]

XL C++ Front End and Run-time Development Department
IBM Toronto Lab, 8200 Warden Ave
Markham, ON, L6G 1C7, Canada

[α]{mokbel@uwindsor.ca} [β]{ccambly@ca.ibm.com}

✠ *Background: What is the problem solved?*

Often, compiler developers need a mechanism to detect the optimizer generated code at varying levels of optimization to tell whether any optimization has been applied at any level (OX) with any additional options implied by a specific level. The known method is by inspecting the compiler listing or the actual compiler optimizing transformations report to understand the performance characteristics of the generated code. This entails a careful analysis of the assembly, pseudo-assembly instructions or any high-level intermediate representation, which is a complex and overwhelming process, hence, the need for a better method and apparatus to handle this situation more efficiently and in less time. In some cases, there is no need to examine the actual code transformation or conduct low level code breakdown. If the main purpose is to identify that the executable binary has been modified under the course of a specific optimization level with any associated options, then a different approach is needed to achieve this lookup in a very short time. However, the complexity varies between a fully object oriented code (e.g. C++) and procedural code (e.g. C).

The proposed method is time efficient and doesn't require a regular intervention from the developer, only whenever needed and in the appropriate time. One of the consequences of the proposed method is that it helps in finding a relational correspondence between the optimization levels and the performance results as proved particularly using SPEC CPU2006 benchmarks [1].

✠ *Summary: Brief description of the proposed method.*

The idea we propose works at the binary level without taking into consideration the semantics of the language used to generate the executable binary whether it is dynamically or statically compiled, hence, an unobtrusive approach. The executable binary is considered as a black box. The core idea of the proposed method is based on Shannon entropy theory [2, 3], part of information theory, which measures the uncertainty associated with a random variable. The entropy analysis examines the statistical variations and quantifies the information contained in the binary (bytes in this case). The entropy equation has already been used in malware analysis and code encryption [4]. In biology, Shannon-Weiner-Index is used to measure the diversity in a categorical data.

The entropy equation works by taking the probability distribution of each random variable rather than the actual value. Therefore, an all-inclusive image is premeditated which accounts for any major

changes or repetitions in the executable file. Redundancy is another factor which measures the amount of repetition as part of the entropy maximal threshold. So, as the entropy increases the redundancy decreases (and vice versa), this means that more data have been added or transformed within the actual implementation.

For a code compiled at optimization level "0" up to "5", the executable binary image will change based on the optimizing transformations with any additional options at each level. Analyzing the entropy of each executable reveals very interesting facts as to whether any optimizing transformations have taken action. So, if you are testing an optimization feature, and you want to conclude if the transformation has happened, then taking the entropy of the executable file gives us a proof that an actual transformation has been taken. Hence, no need to look at the listing, only if needed to track the generated code itself. The entropy value is bounded to the actual optimization properties, for example, a loop unrolling will yield bigger entropy while for dead code elimination the entropy would be less.

We can look at it from a high level perspective by taking the entropy of each executable file compiled at optimization level 0 up to 5. The entropy will change as per the optimization level, which means that actual transformations have been applied. It turns out that this exhibited behavior exposes a close correlation with the performance results which proved using SPEC CPU2006 benchmarks binaries. It gives a tantalizing and heuristic indication about the mutual relationship between the actual optimizing transformation and the performance ratio. This mapped relationship cannot be taken as a definitive signal, because the implementation does not quantify or value the semantics and the optimizing transformations of the language whether it is the low or high level representation. This is most likely to be applicable to a compilation where the speed is favored over the code size.

We've already prototyped the idea to prove the validity of the proposal. The results we got are very promising, and it is well worth it to be part of an actual development environment. The next step would be to integrate it in the compiler optimizer so that it can act precisely to determine if an actual optimization must have happened as part of a specific transformation.

✠ *Detailed description of the proposed idea.*

The idea is very generic; it is not limited to a specific compiler or language. Each compiled executable file has a section header table which lists all the file's sections. What's important here is the code section which holds the executable instructions of a program. So, any transformation should take place in this section. And for other cases it might be in different sections as well, interlinked or separate, depending on the optimization characteristics. We've chosen the ELF32 (Executable and Linkage Format) file format to test the applicability of the proposed idea. The binaries were compiled under Linux OS, Power5+ machine with IBM XL C++ compiler.

The method works by first parsing the section header table of the binary to locate the offset of the code (.text) section. Second, we take the hexadecimal representation of the code (.text) section under inspection. And then take the entropy of that section according to Shannon Entropy equation:

$$H(X) = -\sum_{i=1}^{n} p(x_i)\, log_b p(x_i)$$

First, we present the following definitions:
$n = \{Hex|\ Hex\ is\ an\ element\ that\ takes\ a\ unique\ label\ from\ 0\ to\ 255\}\ and\ |n| = 256$
$\Phi = \{the\ set\ of\ all\ the\ optimization\ levels: O_i, \dots, O_{i+n}\}$

In this case, the set of the possible values for a 32-bit file format is 256. $p(x_i \in n)$ is the probability for each discrete random variable X out of the 256 possibilities which is defined as $p(x_i \in n) = \frac{Frequency(x_i \in n)}{length(.text)}$ and $length(.text) = \sum_{x_i \in n}^{|n|} Frequency(x_i \in n)$ ($Frequency(x_i \in n)$, the number of occurrences of each byte). The base of the logarithm is b = 2, hence the entropy unit is bit. Because $|n| = 256$, the entropy maximal value is $H_{max} = \log_2(n) = 8.0$. And the result is a floating-point number between 0 and 8. $\forall(x_i), 0 \leq p(x_i) \leq 1\ \therefore\ H(X) > 0$. The entropy is always positive. Redundancy is calculated in percentage: $R(X) = 100\% - H(X)\%$.

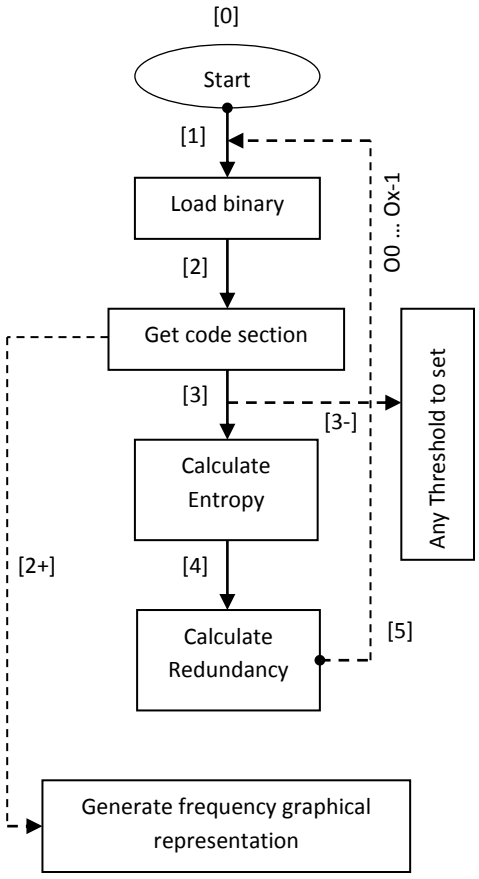

Figure 1. Entropy Optimization comparator Flowchart.

Figure 1 shows the steps needed to calculate the entropy of the code section (.text) in the specified order. First [0], load the executable binary [1], check if it is ELF32 file format, get .text section offset and size [2], generate frequency graphical representation (optional) [2+], check if there is any threshold has been set against a specific optimizing transformation (higher or lower than the specified threshold) [3-], proceed to calculate entropy [3] followed by redundancy [4] and repeat the same process [5] over binaries compiled with O0 up to O5 for comparison.

Frequency graphical representation is a very convenient method to look for any major changes at the byte level from optimization level to another. It is represented as a matrix where the number of columns and rows is equal to 16 (size = 16 x 16 = 256) which accounts for every byte in the binary file. Each cell contains the frequency of each byte. Next, we sort them in ascending order and assign a specific color for each range. The size (256) is divided over three ranges (0 - 85, 86 – 170, 171 - 256). The color intensity varies based on the frequency of each range.

Algorithm 1. OPTIMIZATION TRANSFORMATION COMPARATOR (Binary_File, Opt_Lvl, Opt_Feat)

_begin

1. FF ← File_Format(Binary_File)
   ▷ *This is to determine the executable binary File Format(e.g., ELF32, PE,…)*
2. Offset ← Get_Binary_XCode_Section_Offset(FF, Binary_File)
   ▷ *For ELF32 File Format the code section is ".text", or it could be other section*
3. Size ← Get_Binary_XCode_Section_Size(FF, Binary_File, Offset)
   ▷ *Size variable holds the size of XCode section*
4. Generate_Frequency_Graphical_Representation(Binary_File, Offset, Size)
   ▷ *Optional but easier to spot major changes at the byte level*
5. EntV ← Entropy_Analyzer(Binary_File, Offset, Size)

AsLongAs OPT_LVL.O$X \neq$ OPT_LVL.OY

▷ *The OL does not have to be different if you want to test a specific OT on the same binary file*

switch(OPT_LVL)

case -O$X \mid X \geq 0$ ▷ *Depends on which OL is taken as a reference point*

$\text{EntV}_{\text{reference}} \leftarrow$ EntV

case –O$Y \mid Y > 0$

if EntV $\neq \text{EntV}_{\text{reference}}$ **then**

OT ← OptimizationTransformationThreshold: "OK"

else ▷ *The threshold is leveled according to a specific transformat°*

OT ← OptimizationTransformationThreshold: "NO"

end

if OT **then**

EntV.OPT_FEAT = Increase ‖ Decrease

▷ *Depends on the actual transformation characteristics*

else

EntV.OPT_FEAT = 0

▷ *No Optimizing Transformation happened*

end

default:

break

6. Calculate_Redundancy(EntV)

_end

Figure 2. Optimization Transformation Comparator Algorithm.

Figure 2 shows the pseudocode for the transformation optimization comparator. The presented procedure called TRANSFORAMTION-OPTIMIZATION-COMPARATOR. The pseudocode version is a high level representation of the actual algorithmic implementation. Step 5 is where the decision to be made if an actual Optimizing Transformation (OT) happened as compared to the reference Optimization Level (OPT_LVL) Entropy Value ($EntV_{reference}$). Optimizing Transformation Threshold (OTT) is leveled based on the amount of changes that need to be reflected on the entropy value. And based on the OT characteristics (OPT_FEAT), the entropy value will register either an increase or decrease as compared to $EntV_{reference}$. It doesn't have to be against two different levels, it works as well with the same binary if you are experimenting with a specific OT feature (Some exceptions and low level details are omitted for clarity). Figure 3 depicts how the proposed method could be implemented in a real compiler.

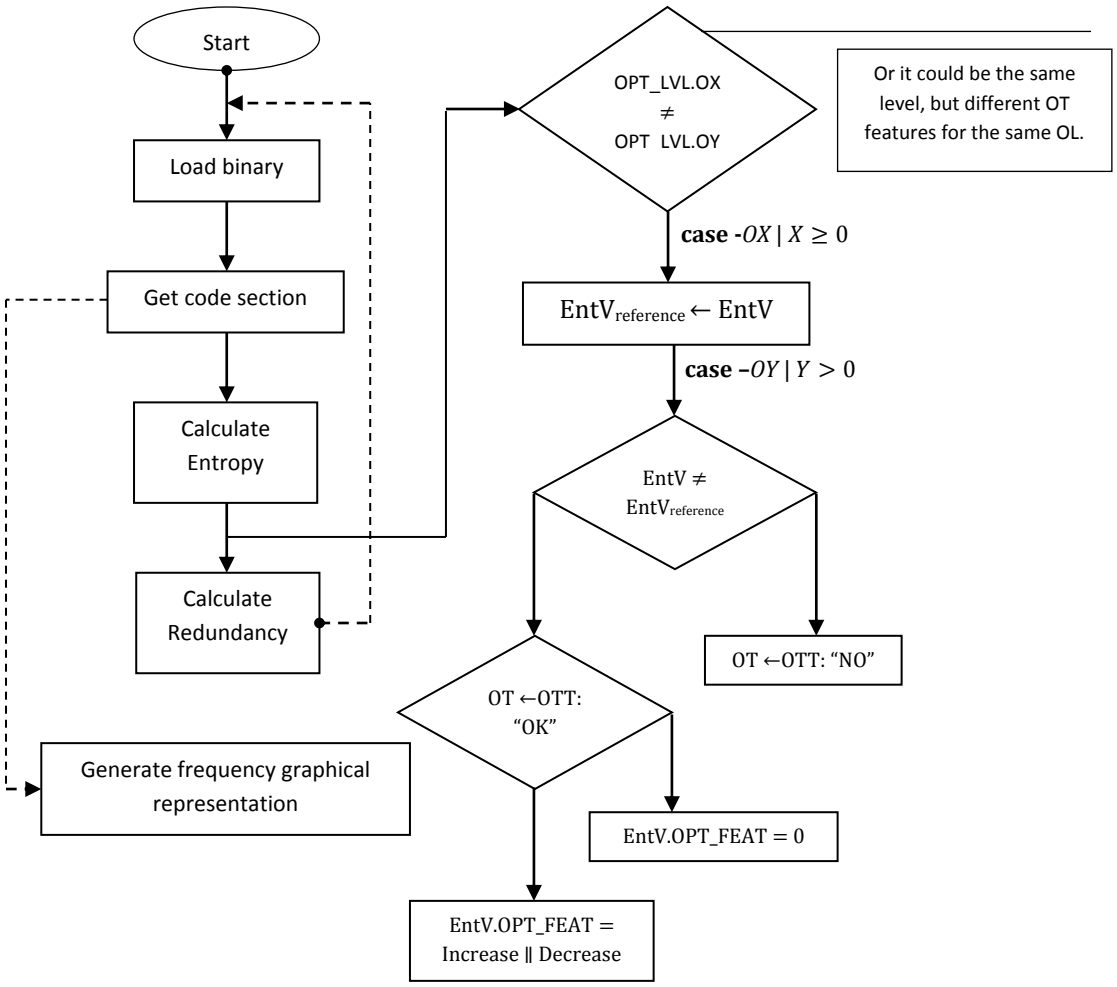

Figure 3. Entropy Optimization comparator Flowchart (detailed version).

Almost every optimization exhibits either of the two effects: code expansion or contraction. Thus, the entropy is bounded by these two effects to register the type of difference a specific optimization undertakes. On the other hand, the size of the .text section is taken as on disk and not as in memory.

Below is an actual demonstration of the proposed idea on SPEC CPU C++ Benchmarks binaries. Seven benchmarks have been chosen grouped as Integer and Floating-Point types, all compiled starting with O0 OL up to O5.

Table 1 shows the .text code section size in bytes for each benchmark and for each optimization level. O0 is the reference Optimization Level for each benchmark binary file. File size increases (O2 - O5) as the level of optimization increases.

	Benchmark	O0	O2	O3	O4	O5
INT	471.omnetpp	983292	691548	745580	808772	864164
	473.astar	55984	41664	62656	74964	83092
	483.xalan	7187960	4884810	6000460	5973560	6623060
FP	444.namd	376928	234048	596064	674964	669812
	447.dealII	6641840	5357840	5934190	2716720	2472370
	450.soplex	777180	466480	624976	716704	656560
	453.povray	1072430	854364	1220430	2474390	2911320

Table 1. .text code section size in bytes.

Table 2 shows the entropy value for each benchmark .text code section. We notice how the entropy value is increasing as the level of optimization increases. That's not only due to the increase in the file size but it is part of the actual transformation on every level, and it is evident in many cases. The optimizer is capable of exploiting any further optimization that is applicable on every level. An interesting fact arises as in 444.namd benchmark case, where the entropy values are very close from O2 to O5; it is in perfect correlation with the performance ratios (Note that the performance data is not shown) which confirm these observations. Thus, it is a benchmark characteristic as well.

	Benchmark	O0	O2	O3	O4	O5
INT	471.omnetpp	5.47576	5.8094	5.86235	6.13205	6.16308
	473.astar	5.2534	5.95447	6.20346	6.43754	6.45211
	483.xalan	5.16109	5.59981	5.67255	5.9661	5.98904
FP	444.namd	5.61111	6.67464	6.65392	6.66238	6.6609
	447.dealII	5.3011	6.05335	6.11919	6.42986	6.37772
	450.soplex	5.26943	5.90492	6.09232	6.31169	6.31902
	453.povray	5.53779	6.12024	6.16816	6.39128	6.43457

Table 2. .text code section entropy in bits.

Table 3 shows the redundancy in percentage for each benchmark .text code section. O0 registers the highest redundancy, which is in perfect correlation with entropy values. The lower the entropy, the higher the redundancy and vice versa.

	Benchmark	O0	O2	O3	O4	O5
INT	471.omnetpp	31.55	27.38	26.72	23.35	22.96
	473.astar	34.33	25.57	22.46	19.53	19.35
	483.xalan	35.49	30	29.09	25.42	25.14
FP	444.namd	29.86	16.57	16.83	16.72	16.74
	447.dealII	33.74	24.33	23.51	19.63	20.29
	450.soplex	34.13	26.19	23.85	21.1	21.01
	453.povray	30.78	23.5	22.9	20.11	19.57

Table 3. .text code section redundancy %.

Another example is a small program with for-loop having a conditional expression set to 1000000; the stride and code statement are the same (increment by 1). The results are shown in table 4. It is evident from the table that code alterations and transformations are happening at O2, O3 and O4 levels. This X-program is compiled with XL C++ compiler. The listing reveals a code alteration at O2 (less instructions and the entropy value decreased to reflect this alteration). The further decrease in the entropy from O2 to O3 is due to some of the optimizing transformations (Loop Rolling, Constant Propagation, Loop Normalization, Copy Propagation and Subexpression Elimination). Again, more optimizing transformations at O4 (Loop Rolling, Value Range Propagation, Forward Store Motion, Loop Normalization, Constant Propagation, Copy Propagation, Dead Code Elimination, Compute Register Pressure, Loop Unrolling, Remove Loop, Wand Waving), hence the decrease in the entropy value, and the listing confirms this by having less instructions. At O5 (O4 = O5), no transformations at all, this is a strong indication that there is nothing left to optimize (or for whatever reason), thus, the same entropy value as O4.

X-Program	O0	O2	O3	O4	O5
Size (.text)	1408	1376	1424	1344	1344
Entropy	5.70323	5.67074	5.66801	5.64867	5.64867
Redundancy	28.71	29.12	29.15	29.39	29.39

Table 4. X-program with for-loop.

Figure 4 shows the frequency graphical representation for all levels of optimizations of X-program (Table 4). The differences between O0 and O2 are shown on O0 graph, between O2 and O3 on O3 and between O3 and O4 on O4. O4 and O5 are perfectly matching without any differences. The aforementioned explanation for table 4 is also valid for figure 4.

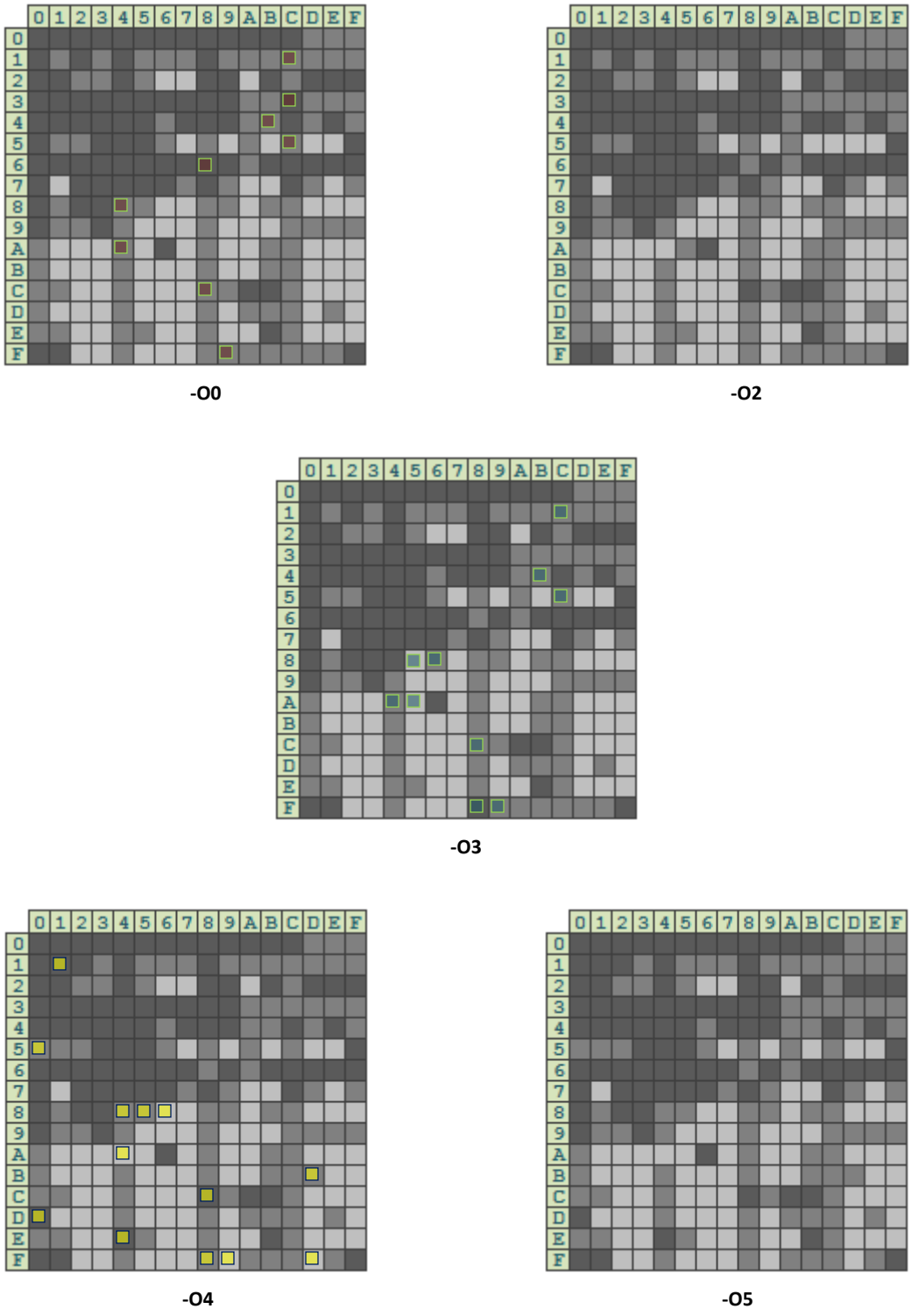

Figure 4. X-binary program: frequency graphical visualization.

The computation performed by the entropy does not account for the displacement of the same hexadecimal value across different positions. On a large scale, it is almost impossible to happen in a way such that a huge portion of the hexadecimal values (worth a functional equivalent) will exhibit such differences. However, the two binaries are different in this case (different positions) and the entropy does not reflect that.

✠ *A Symbiotic Differential Comparison (SDC) Algorithm*

In this section we present a coarse-grained approach that helps in quantifying and classifying the binary differences at the byte level in integrated and phased steps. The algorithm accounts for all the variations across all the optimization levels (opt. lvl) in a descriptive and relational manner such that, a change at one optimization level is also reflected and examined to track the effect that change has on the overall variations. Only the ".text" code section is examined. The algorithm is presented in algorithm 2.

The algorithm encapsulates a staged analysis approach between every two optimization levels as well as a holistic (integrated) one which enumerates over all the optimization levels at once.

The variations are accounted for by comparing the difference of the frequency of each hexadecimal byte (total of 256) between every two optimization levels such that the first opt lvl (binary file) of the second comparison is the second of the first comparison. Thus, a successive interrelated analysis is provided to track all the changes reliably. And this takes place in the following order: $(O_i \; vs \; O_{i+1}), (O_{i+1} \; vs \; O_{i+2}), \dots, (O_{i+n} \; vs \; O_{i+n+1})$.

For the integrated approach, the difference of the frequency of each hexadecimal byte is calculated across all the optimization levels at once in the following order: $O_i, \dots, O_{i+n}$.

The logic of the algorithm is as follows:

If the relational comparison between the frequencies of the same byte across two optimization levels is decreasing, a **N**egative counter is incremented to keep track of the number of bytes that follows this pattern. If it is increasing, a **P**ositive counter is incremented, otherwise, a **Z**ero counter is incremented (they match) (L. $04 - 12$). In the integrated approach, across all the optimization levels, the same logic applies. However, the counters are incremented appropriately only if the end result is either, negative, positive or zero (L. $13 - 21$).

Among the optimization levels, the differences distribution may vary randomly, but the end result is either **P**ositive, **N**egative or **Z**ero (neutral) according to the SDC algorithm rules in the integrated approach.

The reason behind this categorization is the abstraction needed to gage the high level changes among various levels of optimizations in an orderly manner using a semantically constructed set of simple yet complex key parameters.

The hypotheses are as follows:

A. **P** $[O_i \; vs \; O_{i+1}]$:

$if\ |\mathbf{P}|_{\gg}\ compared\ to\ |\mathbf{N}| \Longrightarrow More\ optimization\ transformations\ are\ taking\ place.$

B. $\mathbf{N}\ [\mathrm{O_i}\ vs\ \mathrm{O}_{i+1}]$:

$if\ |\mathbf{N}|_{\gg}\ compared\ to\ |\mathbf{P}| \Longrightarrow Less\ optimization\ transformations\ are\ taking\ place.$

C. $\mathbf{Z}\ [\mathrm{O}_i\ vs\ \mathrm{O}_{i+1}]$:

1. $if\ |\mathbf{Z}|_{\gg}\ compared\ to\ (|\mathbf{P}| \wedge |\mathbf{N}|) \Longrightarrow Few\ opt\ transformations\ are\ taking\ place.$
   $Or\ not\ at\ all\ ,it\ depends\ on\ the\ code\ size.$

   $(no\ changes\ at\ all\ for\ a\ specific\ 'h'\ or\ the\ differences\ are\ cancelling\ each\ other)$

2. $if\ |\mathbf{Z}|_{\ll}\ compared\ to\ (|\mathbf{P}| \wedge |\mathbf{N}|) \Longrightarrow More\ opt\ transformations\ are\ taking\ place.$

D. $\lambda := \sigma\ (|\mathbf{P}|)[(O_i\ vs\ O_{i+1}), \dots, (O_{i+n}\ vs\ O_{i+n+1})] \approx_{k\ (for\ a\ very\ small\ k)}$
$\sigma\ (|\mathbf{N}|)[(O_i\ vs\ O_{i+1}), \dots, (O_{i+n}\ vs\ O_{i+n+1})] \wedge (\mathrm{C.2})$

$if\ (\lambda) \Longrightarrow The\ changes\ among\ the\ optimization\ levels\ are\ in\ congruent\ state.$

$\underline{Note: \sigma\ symbol\ denotes\ the\ standard\ deviation.}$

The SDC algorithm is demonstrated by applying it to various SPEC CPU benchmarks compiled at all the major possible optimization levels provided by the XLC compiler. An analysis is provided for some of them that outline how the algorithm works. The computation performed by the algorithm still requires an insightful look to make sense out of the generated numbers. On the other hand, the algorithm exposes other analytical dimensions in correspondence with other factors (e.g. file size, number of instruction, program size, the entropy ratio) based on the Optimizing Transformation Comparator Algorithm.

The SDC algorithm is also applicable to non-strict pure optimization levels comparisons. It is possible to specify other opt lvl sub-options (opt features) to conduct a reasonable comparison. However, the optimization levels order must be preserved, and any non-strict opt lvl comparisons should take into account the level being compared against. For instance,

$$\left\{O_i \quad O_{i+1} \quad O_{(i+1)*f_x} \quad O_{(i+1)*f_y} \quad O_{i+2}\right\}\ where\ f_x\ denotes\ some\ opt\ sub-options\ feature\ 'x'$$

These types of comparison call for new pattern detection rules that are not considered in the current formulations. Nonetheless, the phased analysis part does detect those variations.

Algorithm 2. SYMBIOTIC DIFFERENTIAL COMPARISON

01. **Input**: *The set of all binaries compiled at varying levels of opimization, the sets n and* Φ

02. **Output**: $\left\{ \begin{matrix} \left[{}^{O_{i+1}}_{O_i}\boldsymbol{SDC}.diff\langle \boldsymbol{X}\rangle, \dots, {}^{O_{i+n+1}}_{O_{i+n}}\boldsymbol{SDC}.diff\langle \boldsymbol{X}\rangle \right] \forall(\boldsymbol{X}) \\ \left[\mathbf{SDC}[h]^{\boldsymbol{Opt.All}}.diff\langle \boldsymbol{X}\rangle\right] \forall(\boldsymbol{X}) \end{matrix} \right\}$ *such that* $\boldsymbol{X} \in \{\boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Z}\}$

03. **_begin**

04. **for** $\forall(O_i \in \Phi)$ /* *The phased analysis steps, loop over all the optimization levels* */

05. **for** $\forall(h \in \mathrm{n})$ /* *loop over all the hexadecimal numbers* */

06. ${}^{O_{i+1}}_{O_i}\mathbf{Diff}[h] = Freq(h).O_{i+1} - Freq(h).O_i$ /* *Calculate the intermediate difference* */

07. **if** $[\boldsymbol{Freq}(h).\boldsymbol{O_i} > \boldsymbol{Freq}(h).\boldsymbol{O_{i+1}}]$ **then** $\left\{ {}^{O_{i+1}}_{O_i}SDC.diff\langle \boldsymbol{N}\rangle + +; \right\}$

08. **else if** $[\boldsymbol{Freq}(h).\boldsymbol{O_i} < \boldsymbol{Freq}(h).\boldsymbol{O_{i+1}}]$ **then** $\left\{ {}^{O_{i+1}}_{O_i}SDC.diff\langle \boldsymbol{P}\rangle + +; \right\}$

09. **else if** $[\boldsymbol{Freq}(h).\boldsymbol{O_i} = \boldsymbol{Freq}(h).\boldsymbol{O_{i+1}}]$ **then** $\left\{ {}^{O_{i+1}}_{O_i}SDC.diff\langle \boldsymbol{Z}\rangle + +; \right\}$

10. **end if**

11. **end for** /* **call** *hypotheses* $\left({}^{O_{i+1}}_{O_i}SDC.diff\langle \boldsymbol{X}\rangle\ [\forall(\boldsymbol{X})]$ && *Activate* $\{\boldsymbol{A}., \boldsymbol{B}., \boldsymbol{C}.\}\right)$ */

12. **end for** /* **call** *hypothesis* $\left({}^{O_{i+1}}_{O_i}SDC.diff\langle \boldsymbol{X}\rangle, \dots, {}^{O_{i+n+1}}_{O_{i+n}}SDC.diff\langle \boldsymbol{X}\rangle\ [\forall(\boldsymbol{X})]$ && *Activate* $\{\boldsymbol{D}.\}\right)$ */

13. **for** $\forall(O_i \in \Phi)$ /* *across all the optimization levels at once* */

14. **for** $\forall(h \in \mathrm{n})$

15. $\mathbf{SDC}[h]^{Opt.All} = {}^{O_{i+1}}_{O_i}\mathbf{Diff}[h] + \dots + {}^{O_{i+n+1}}_{O_{i+n}}\mathbf{Diff}[h]$ /* *Sum over all the* **Diff′s** (L. 06) */

16. **if** $\left[\mathbf{SDC}[h]^{\boldsymbol{Opt.All}} > \mathbf{0}\right]$ **then** $\left\{\mathbf{SDC}[h]^{\boldsymbol{Opt.All}}.diff\langle \boldsymbol{P}\rangle + +;\right\}$

17. **else if** $\left[\mathbf{SDC}[h]^{\boldsymbol{Opt.All}} < \mathbf{0}\right]$ **then** $\left\{\mathbf{SDC}[h]^{\boldsymbol{Opt.All}}.diff\langle \boldsymbol{N}\rangle + +;\right\}$

18. **else if** $\left[\mathbf{SDC}[h]^{\boldsymbol{Opt.All}} = \mathbf{0}\right]$ **then** $\left\{\mathbf{SDC}[h]^{\boldsymbol{Opt.All}}.diff\langle \boldsymbol{Z}\rangle + +;\right\}$

19. **end if**

20. **end for**

21. **end for**

22. **_end**

Below are the numbers generated by the algorithm for various SPEC CPU benchmarks.

Namd	Oref vs. O2	O2 vs. O3	O3 vs. O4	O4 vs. O5	SDCAlg
P	172	248	179	133	230
N	84	8	76	122	26
Z	0	0	1	1	0

For Namd, notice that for every two optimization levels, the number of P's is always higher than the rest of the counters, which means (theoretically speaking) that more changes are taking place. Between O2 & O3, it registers the highest number of P's, probably, it is at this level where most of the transformations took place with respect to the other opt levels. For the positive changes, we notice an asymmetric relation such that when P increases N decreases. Between O3 & O4, the changes are sort of rolling back (!), since fewer transformations are taking place at this level. Between O4 & O5, the difference between the number of P's and N's is very less as compared to the previous comparisons. SDCAlg shows that this benchmark went through a lot of changes by the virtue of the number of P's being the highest. The steady aspect of the number of Z's across all the opt levels (comparison) confirms the observation that this benchmark undergone through a lot of changes.

DealII	Oref vs. O2	O2 vs. O3	O3 vs. O4	O4 vs. O5	SDCAlg
P	134	213	13	46	76
N	122	43	243	209	180
Z	0	0	0	1	0

For dealII, things are very different. The number of N's registers the highest in O3 vs. O4 & O4. vs. O5. This is the only benchmark that registers the highest number of N's which is also confirmed by SDCAlg. In fact, a huge drop in the size of the executable (54.22%) happens between O3 and O4 (could be due to this factor!). On the other hand, the entropy registers the highest increase between O3 and O4.

Soplex	Oref vs. O2	O2 vs. O3	O3 vs. O4	O4 vs. O5	SDCAlg
P	125	242	199	63	192
N	131	13	57	192	64
Z	0	1	0	1	0

Povray	Oref vs. O2	O2 vs. O3	O3 vs. O4	O4 vs. O5	SDCAlg
P	167	248	252	234	255
N	80	8	4	22	1
Z	0	0	0	0	0

Omnetpp	Oref vs. O2	O2 vs. O3	O3 vs. O4	O4 vs. O5	SDCAlg
P	112	178	181	192	179
N	144	78	75	63	77
Z	0	0	0	1	0

Astar	Oref vs. O2	O2 vs. O3	O3 vs. O4	O4 vs. O5	SDCAlg
P	158	237	187	180	230
N	94	15	67	74	26
Z	4	4	2	2	0

Xalan	Oref vs. O2	O2 vs. O3	O3 vs. O4	O4 vs. O5	SDCAlg
P	117	225	165	197	177
N	139	31	91	59	79
Z	0	0	0	0	0

F - Loop	Oref vs. O2	O2 vs. O3	O3 vs. O4	O4 vs. O5	SDCAlg
P	15	28	15	0	16
N	30	16	39	0	39
Z	211	212	202	256	201

For F- Loop example (**X**-Program), note that the size (table 4) of the executable is small as compared to SPEC benchmarks. This example contains only a for-loop which is open to a limited number of transformations. An interesting fact to note here is that the number of instructions (based on the listing) is as follows: $F\langle O\mathbf{0}\rangle \dashrightarrow (20), F\langle O\mathbf{2}\rangle \dashrightarrow (12), F\langle O\mathbf{3}\rangle \dashrightarrow (21), F\langle O\mathbf{4}\rangle \dashrightarrow (2), F\langle O\mathbf{5}\rangle \dashrightarrow (2)$. Small variations count due to the size (small) of the executable. A drop in the number of instructions shows an increase in the number of P's (decrease in the number of N's). The number of Z's is very large here and that's due to the fact that the executable code is so minimal (only a for-loop), hence, no major changes. When P = 0 and N = 0 ($\Longrightarrow$ Z = 256), means there is absolutely no change at this level (e.g. O4 vs. O5).

An interesting observation to note is that the standard deviation between the **P**'s and **N**'s numbers across all the staged analysis comparisons is almost equal (except Povray, the std. dev. difference is around 10.7%). This happens when the number of **Z**'s across all the comparisons is negligible (in contrary to the F - Loop example). This is a reasonable conclusion, since the number of transformations is not steady across all the optimization levels (it could be more or less as the opt lvl increases, code design, etc.), hence, the number of **N**'s changes accordingly (hypothesis D).

✠ *A Complete Juxtaposation of All Optimization Levels Using Kullback-Leibler Divergence*

Analyzing the level of dispersion among the optimization levels provides a way to assess the effect of each optimization on the original distribution (e.g. a binary compiled at O0). More specifically, it shows how far or close each two unique OL are from each other.

In order to quantify the difference between two optimization levels, we use Kullback-Leibler Divergence [5] (KLD, also called the Relative Entropy) metric to measure the difference between two probability distributions (between every two unique optimization levels) of a discrete random variable x_i. We enumerate over all the OL in a complete way such that for every two unique OL, KLD is applied. KLD is defined as follows:

$$KLD(P \parallel Q) = \sum_{\forall(x_i \in n)} P(x_i) \log_2 \left(\frac{P(x_i)}{Q(x_i)} \right) \mid (P, Q) \in \Phi$$

P and Q are two probability mass functions. KLD is asymmetric $\big(KLD(P \parallel Q) \neq KLD(Q \parallel P)\big)$ and it does not satisfy the triangle inequality, hence KLD is not a true distance metric. KLD is only defined if both P and Q sum to 1 and $Q(x_i) > 0 \; \forall \; x_i \mid P(x_i) > 0$. If $P(x_i) = 0$ and $Q(x_i) \neq 0$ then $KLD(P \parallel Q) = \infty$.

KLD is always positive, $KLD(P \parallel Q) \geq 0 \; \forall (P, Q)$ and $KLD = 0$ iff $P = Q$. We define the function $\aleph: n \xrightarrow{KLD} [0,1]$ such that the input is the hexadecimal representation (n) of the input binary file and the output is a discrete value between 0 and 1.

In [6], Johnson and Sinanović proposed a symmetrized $\big(KLD(P \parallel Q) = KLD(Q \parallel P)\big)$ version of KLD via the harmonic mean and they called it the Resistor-Average distance. However, our experimentation shows that it does not satisfy the triangle inequality but it is still useful as a semimetric. The symmetrized version of KLD is defined as follows:

$$\frac{1}{\mathcal{R}(P, Q)} \equiv \frac{1}{KLD(P \parallel Q)} + \frac{1}{KLD(Q \parallel P)}$$

In the symmetrized version, $(P \wedge Q) > 0$. In case this condition is not satisfied, the Resistor-Average $\mathcal{R}$ is undefined, otherwise a negative result will be reported since the probability distribution is not continuous between P and Q.

Also, we only take the .text section hexadecimal representation in $\mathcal{R}$ as in the entropy analysis. Everything else is the same as defined in the detailed description of the proposed idea section.

We applied $\mathcal{R}$ on SPEC benchmarks compiled at varying levels of optimizations as shown below. The reported numbers are in percentage. Every two unique optimization levels of every benchmark is completely covered by $\mathcal{R}$. The more different (OL) they are the greater the probability.

O0 - O2	O0 - O3	O0 - O4	O0 - O5
13.0691	14.1314	17.1819	18.6783
O2 - O3	**O2 - O4**	**O2 - O5**	
1.08332	3.85935	4.20106	
O3 - O4	**O3 - O5**		
2.13521	2.09068		
O4 - O5			
0.776058			

[DealII]

O0 - O2	O0 - O3	O0 - O4	O0 - O5
6.86391	8.06173	10.4102	11.0178
O2 - O3	**O2 - O4**	**O2 - O5**	
1.3571	3.56543	4.00785	
O3 - O4	**O3 - O5**		
2.69221	3.04737		
O4 - O5			
0.654044			

[Omnetpp]

We notice that the distance between O0 and every other consecutive optimization level is increasing and it registers the highest at O5 since for Ox > 0 a lot of transformations are taking place. And between O4 and O5, it registers the lowest since fewer transformations are taking place.

O0 - O2	O0 - O3	O0 - O4	O0 - O5
8.55949	9.90728	15.2353	15.6402
O2 - O3	**O2 - O4**	**O2 - O5**	
0.760437	3.80956	3.80035	
O3 - O4	**O3 - O5**		
2.12957	2.20293		
O4 - O5			
0.49264			

[Xalan]

O0 - O2	O0 - O3	O0 - O4	O0 - O5
12.6891	15.8796	24.7075	22.8374
O2 - O3	**O2 - O4**	**O2 - O5**	
1.40003	7.69281	5.90794	
O3 - O4	**O3 - O5**		
4.29176	2.9189		
O4 - O5			
0.96609			

[Soplex]

O0 - O2	O0 - O3	O0 - O4	O0 - O5
8.40958	9.45661	9.45661	14.9026
O2 - O3	**O2 - O4**	**O2 - O5**	
1.00091	3.69465	3.68256	
O3 - O4	**O3 - O5**		
2.32507	2.21677		
O4 - O5			
1.18532			

[Povray]

O0 - O2	O0 - O3	O0 - O4	O0 - O5
Undef.	Undef.	Undef.	Undef.
O2 - O3	**O2 - O4**	**O2 - O5**	
5.78476	6.29224	6.69692	
O3 - O4	**O3 - O5**		
1.54833	1.53252		
O4 - O5			
0.519442			

[Namd]

O0 - O2	O0 - O3	O0 - O4	O0 - O5
Undef.	Undef.	Undef.	Undef.
O2 - O3	**O2 - O4**	**O2 - O5**	
Undef.	Undef.	Undef.	
O3 - O4	**O3 - O5**		
Undef.	4.41646		
O4 - O5			
Undef.			

[Astar]

O0 - O2	O0 - O3	O0 - O4	O0 - O5
Undef.	Undef.	Undef.	Undef.
O2 - O3	**O2 - O4**	**O2 - O5**	
Undef.	Undef.	Undef.	
O3 - O4	**O3 - O5**		
Undef.	Undef.		
O4 - O5			
Undef.			

[F-Loop]

Some benchmarks exhibit unique differences and that requires further investigation. In the case of Namd and Astar benchmarks and F-Loop kernel an undefined situation is encountered in which the frequency of one or some of the hexadecimal values are/is zero.

$\mathcal{R}$ could also be used as an OL detector based on the level of dispersion between the OL.

Bibliography

[1] Henning, J. L. (2006) SPEC CPU2006 Benchmark Descriptions. ACM SIGARCH Computer Architecture News, Vol. 34, Issue 04, pp. 1-17.

[2] Schneider, T. D. (2010) Information Theory Primer. Http://alum.mit.edu/www/toms/paper/primer/

[3] Cover, T. M. and Thomas, J. A. (2006) Elements of Information Theory. 2nd ed. John Wiley & Sons, Inc., Hoboken, New Jersey.

[4] Lyda, R. and Hamrock, J. (2007) Using Entropy Analysis to Find Encrypted and Packed Malware. IEEE Security & Privacy, Vol. 05, Issue 02, pp. 40-45.

[5] Kullback, S. and Leibler, R. A. (1951) On Information and Sufficiency. Annals of Mathematical Statistics, Vol. 22, N. 1, pp. 79-86.

[6] Johnson, D. H. and Sinanović, S. (2001) Symmetrizing the Kullback-Leibler Distance. Technical Report.