

נ.נ בתכנון

משולב

חומרה/תוכנה

פרויקט

00460882

שם מגיש 2 : בר ארמה

ת.ז: 207708397

שם מגיש 1 : תומר בן ארוש

ת.ז: 209157692

תוכן

3	1. מבוא
4	2. Json Dumps benchmark
4	2.1 סקירה - Overview
4	2.10 תיאור ה-benchmark
4	2.11 השימוש בספריות
4	2.12 מבני נתונים שנעשה בהם שימוש
6	2.2 ניתוח ראשוני - Initial analysis
6	2.20 ניתוח ביצועים
8	2.21 Flame graphs
16	2.3 שיפורים ופעולות אופטימיזציה
17	2.4 השוואת ביצועים
18	2.5 שיפור חומרת
22	2.6 מסקנות
24	3. Logging
24	3.1 סקירה - Overview
24	3.10 תיאור ה-benchmark
24	3.11 השימוש בספריות
25	3.12 מבני נתונים שנעשה בהם שימוש
26	3.2 ניתוח ראשוני - Initial analysis
26	3.20 ניתוח ביצועים
27	3.21 Flame graphs
33	3.3 שיפורים ופעולות אופטימיזציה
36	3.4 השוואת ביצועים
37	3.5 שיפור חומרת
40	3.6 מסקנות
42	4. נספחים
42	נספח 1: בחירת מקרה הבדיקה - למה דווקא NESTED?
42	נספח 2: הסבר על שורות ההרצה בjson_dumps
43	נספח 3: השוואת לוגים - בדיקה שאכן מתקבל פלט זהה
44	נספח 4: הסבר על שורות ההרצה ב logging
45	נספח 5: Meme לכל benchmark שנבחר

1. מבוא

מטרת פרויקט זה היא לחקור, לנתח ולבצע אופטימיזציה לביצועי python interpreter והספריות שהוא משתמש בהן, על ידי עבודה עם benchmarks שונים מתוך pyperformance. שיפור ביצועי קוד פייתון, אשר נמצא כיום בשימוש רחב מאוד במערכות תוכנה מודרניות, עשוי להוביל לחיסכון ניכר בצריכת זמן CPU, משאבים ומימון חישובי.

במסגרת הפרויקט, התבקשנו לבחור שני benchmarks מתוך הרשימה הנתונה ולהעמיק בהם מבחינת אופן הביצוע, הארגון הפנימי של המימוש, וכן הקשר שלהם למבנה python interpreter כולל, bytecode, מנגנון הריצה, והספריות שהוא מפעיל. כאשר אחת מהמטרות המרכזיות הייתה לאתר bottlenecks בביצועים, להסביר את מקורם, ולהציע שיפורים תוכנתיים או חומרתיים אשר יאיצו את הפעולה.

לצורך זיהוי צווארי בקבוק נעשה שימוש בכלים כמו perf ו-FlameGraph אשר נלמדו בכיתה ומאפשרים ניתוח עומק של זמן ריצה וצריכת משאבים של תוכנית. לאחר זיהוי ה-hotspots ביצענו אופטימיזציה ממוקדת לאזורי הקוד הקריטיים, תוך שימוש בספריות יעילות יותר או בשינוי מבנה הקוד. בנוסף, נדרשנו להציע רעיון להאצת חומרה (Hardware Acceleration) עבור חלק מהפעולה שבוצעה, כולל תיאור ארכיטקטוני ברמת בלוק דיאגרם.

כאמור העבודה מתועדת במלואה ב-Git repository וכוללת קוד, סקריפטים, גרפים, ודוחות ביצועים.

בקישור הבא :

<https://github.com/barar953/HwSw>

במסגרת הפרויקט, בחרנו להתמקד בשני benchmarks מתוך **pyperformance**:

- **json_dumps** : המודד את ביצועי המרה של אובייקטי פייתון לפורמט JSON.
 - **logging** : הבוחן את מהירות ויעילות מנגנון יצירת הודעות לוג, כולל פורמט, הוספת מטא-דאטה (כגון PID ו-timestamp) והעברה ל-handlers השונים.
- שני benchmarks אלו מדמים תרחישים נפוצים במערכות מבוזרות, יישומי API, ומערכות הפעלה. הם נבחרו מכיוון שהם עוסקים בשירותי תשתית מרכזיים בפיתוח תוכנה: סידור נתונים וכתובת לוגים. ניתוחם מאפשר להבין לעומק את מנגנוני הסידור והניטור של פייתון, לזהות צווארי בקבוק מהותיים, ולבחון דרכים מעשיות ליעול תוכנתי ואף לשקול מימוש מאיץ חומרתי שיכול לשפר ביצועים במערכות עתירות תקשורת וניטור.

2. Json Dumps benchmark

2.1. סקירה - Overview

2.10. תיאור ה-benchmark

הבנצ'מרק עוסק בבחינת ביצועי הפונקציה `json.dumps()` של פייתון, אשר משמשת להמרת אובייקטי פייתון למחרוזות JSON. פונקציה זו נמצאת בשימוש נרחב בפרויקטים שדורשים שמירה, שליחה או לוגינג של נתונים בפורמט JSON, ולכן שיפור הביצועים שלה עשוי לתרום משמעותית למהירות הכוללת של מערכות רבות.

2.11. השימוש בספריות

הבדיקה התבצעה על בסיס הבנצ'מרק `json_dumps` מתוך חבילת `pyperformance` שהיא חבילת הבנצ'מרקים הרשמית של שפת פייתון למדידת ביצועים של פונקציות ומודולים סטנדרטיים.

לצורך העבודה, שודרג סקריפט הבנצ'מרק כך שיתמוך גם בטעינת קבצי JSON חיצוניים מותאמים אישית, ויאפשר בחירה בין שלוש ספריות/גישות שונות להמרת האובייקטים:

- **`json.dumps`** : המימוש המקורי של פייתון.
 - **`dumps_optimized`** : גרסה מותאמת שפותחה כחלק מהפרויקט. גרסה זו מוסיפה:
 - שימוש במפרידים קומפקטיים (',', ':', ')') לצמצום נפח המחרוזת.
 - נתיב מהיר (Fast-path) לטיפוסים פרימיטיביים. (None, bool, int, float, str)
 - שימוש במנגנון `cache` פנימי עבור אובייקטים שחוזרים על עצמם (כגון dict או list), כדי למנוע קידוד חוזר מיותר.
 - **`dumps_fast`** : עטיפה שמנסה להשתמש בספריית **`orjson`** (אם מותקנת), המממשת את ההמרה ל-JSON בקוד מהיר בשפת Rust/C. כאמור אם `orjson` אינה זמינה, מתבצעת נפילה (fallback) ל-`ujson` ואם גם היא אינה קיימת – חזרה ל-`dumps_optimized`.
- כל המימושים רוכזו בקובץ **`my_json_dumps.py`**, והבחירה ביניהם מתבצעת באמצעות פרמטר חיצוני (`--impl baseline|optimized|fast`) בעת הרצת הבנצ'מרק.

2.12. מבני נתונים שנעשה בהם שימוש

מבני הנתונים שנבדקו כוללים מילונים מקוננים (dict) ורשימות (list) אשר מהוות סימולציה למבנה JSON נפוץ באפליקציות אמיתיות:

- **`EMPTY`** : מבנה ריק
 - **`SIMPLE`** : מילון שטוח עם ערכים פשוטים.
 - **`NESTED`** : מבנה מקונן של מילונים ורשימות.
 - **`HUGE`** : מבנה שחוזר על `NESTED` 1000 פעמים.
- לצורך ניתוח מעמיק, בחרנו להתמקד במבנה `NESTED`, אשר מדמה באופן נאמן קלט JSON אמיתי, אך אינו כבד מדי למדידה ופרופילינג.
- מבנה זה כולל חזרות של אובייקטים, תווים מיוחדים ורמות שונות של היררכיה פנימית - ומכאן נובע הפוטנציאל לחשוף נקודות כשל בביצועים של פעולת ההמרה.

ההרצה בוצעה באמצעות כלי המדידה pyperf ובנוסף בוצע פרופילינג עמוק באמצעות perf ו-FlameGraph.

בשימוש בפקודות הבאות :

גרסת הבסיס המקורית:

```
root@ubuntu:~/HwSw/json_dumps_bench# perf record -F 99 -g -- python3 custom_json_benchmark.py --cases NESTED --impl baseline
```

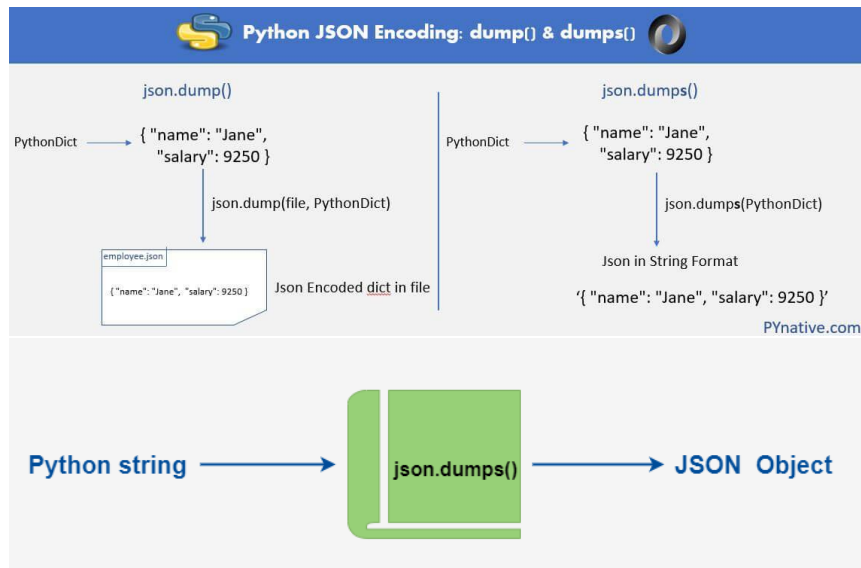
גרסת האופטימיזציה שבנינו:

```
root@ubuntu:~/HwSw/json_dumps_bench# perf record -F 99 -g -- python3 custom_json_benchmark.py --cases NESTED --impl optimized
```

גרסת האופטימיזציה תוך החלפת ספרייה:

```
root@ubuntu:~/HwSw/json_dumps_bench# perf record -F 99 -g -- python3 custom_json_benchmark.py --cases NESTED --impl fast
```

לצורך הסקירה וההבנה הכללית נתבונן ברעיון הכללי:



<https://pynative.com/python-json-dumps-and-dump-for-json-encoding>

2.2. ניתוח ראשוני - Initial analysis

2.20. ניתוח ביצועים

בשלב הראשון של הניתוח, ביצענו מדידות ביצועים לבנצ'מרק json_dumps באמצעות כלי הביצועים pyperf. המדידה בוצעה על מבנה נתונים מקונן (בחרנו לנתח את המקרה NESTED המדמה סידור של אובייקטים מורכבים כפי שקורה בשימוש אמיתי- לדוגמה, במערכות API, לוגים או העברת נתונים ברשת. (ראה נספח 1: [בחירת מקרה הבדיקה - למה דווקא NESTED?](#))

ההרצה הראשונה בוצעה על המימוש המקורי של פייתון - json.dumps. תוצאות ההרצה שמורות בקובץ baseline.json, וניתן מהן לראות את הנתונים הבאים:

```
root@ubuntu:~/HwSw/json_dumps_bench# python3 custom_json_benchmark.py --cases NESTED -o baseline.json
.....
WARNING: the benchmark result may be unstable
* Not enough samples to get a stable result (95% certainly of less than 1% variation)

Try to rerun the benchmark with more runs, values and/or loops.
Run 'python3 -m pyperf system tune' command to reduce the system jitter.
Use pyperf stats, pyperf dump and pyperf hist to analyze results.
Use --quiet option to hide these warnings.

json_dumps: Mean +- std dev: 7.35 ms +- 0.31 ms
```

- זמן ממוצע להרצה אחת : 7.35 ms
- סטיית תקן: $\pm 0.31\text{ ms}$

לאחר מכן, ביצענו הרצה על גרסת הקוד שהוחלפה ב־dumps_fast, אשר משתמשת בספריית orjson. התוצאות של הרצה זו נשמרו בקובץ fast.json.

```
● root@ubuntu:~/HwSw/json_dumps_bench# python3 custom_json_benchmark.py --cases NESTED -o fast.json
.....
WARNING: the benchmark result may be unstable
* Not enough samples to get a stable result (95% certainly of less than 1% variation)

Try to rerun the benchmark with more runs, values and/or loops.
Run 'python3 -m pyperf system tune' command to reduce the system jitter.
Use pyperf stats, pyperf dump and pyperf hist to analyze results.
Use --quiet option to hide these warnings.

json_dumps: Mean +- std dev: 1.30 ms +- 0.05 ms
```

- זמן ממוצע להרצה אחת : 1.30 ms
- סטיית תקן: $\pm 0.05\text{ ms}$

ומתוך ההשוואה קיבלנו:

```
● root@ubuntu:~/HwSw/json_dumps_bench# python3 -m pyperf compare_to baseline.json fast.json
Mean +- std dev: [baseline] 7.35 ms +- 0.31 ms -> [fast] 1.30 ms +- 0.05 ms: 5.66x faster
○ root@ubuntu:~/HwSw/json_dumps_bench#
```

במילים אחרות, השימוש ב־dumps_fast הוביל לשיפור ביצועים של יותר מפי 5.6, כלומר **ירידה של כ-82% בזמן הריצה** של הפעולה. ולמעשה קיבלנו את השיפור הנדרש.

נבדוק בנוסף את השיפור של גרסת ה optimized שהצענו:

```
root@ubuntu:~/HwSw/json_dumps_bench# perf record -F 99 -g -- python3 custom_json_benchmark.py --cases NESTED --impl optimized
Couldn't record kernel reference relocation symbol
Symbol resolution may be skewed if relocation was used (e.g. kexec).
Check /proc/kallsyms permission or run as root.
perf_event__synthesize_bpf_events: failed to synthesize bpf images: No such file or directory
Couldn't synthesize bpf events.
.....
WARNING: the benchmark result may be unstable
* Not enough samples to get a stable result (95% certainly of less than 1% variation)

Try to rerun the benchmark with more runs, values and/or loops.
Run 'python3 -m pyperf system tune' command to reduce the system jitter.
Use pyperf stats, pyperf dump and pyperf hist to analyze results.
Use --quiet option to hide these warnings.

json_dumps: Mean +- std dev: 831 us +- 45 us
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.179 MB perf.data (1129 samples) ]
```

- זמן ממוצע להרצה אחת : 831 us

- סטיית תקן: $\pm 35\text{ us}$

כלומר השימוש ב־ **optimized_dumps** במקרה **NESTED** הוריד את זמן הריצה ממוצע של כ־ 7.35 ms ל־ 0.831 ms בלבד – חיסכון של כ־ **88.7%** בזמן הריצה, כלומר ביצועים מהירים יותר בכ־ **פי 8.85** לעומת הבייסליין.

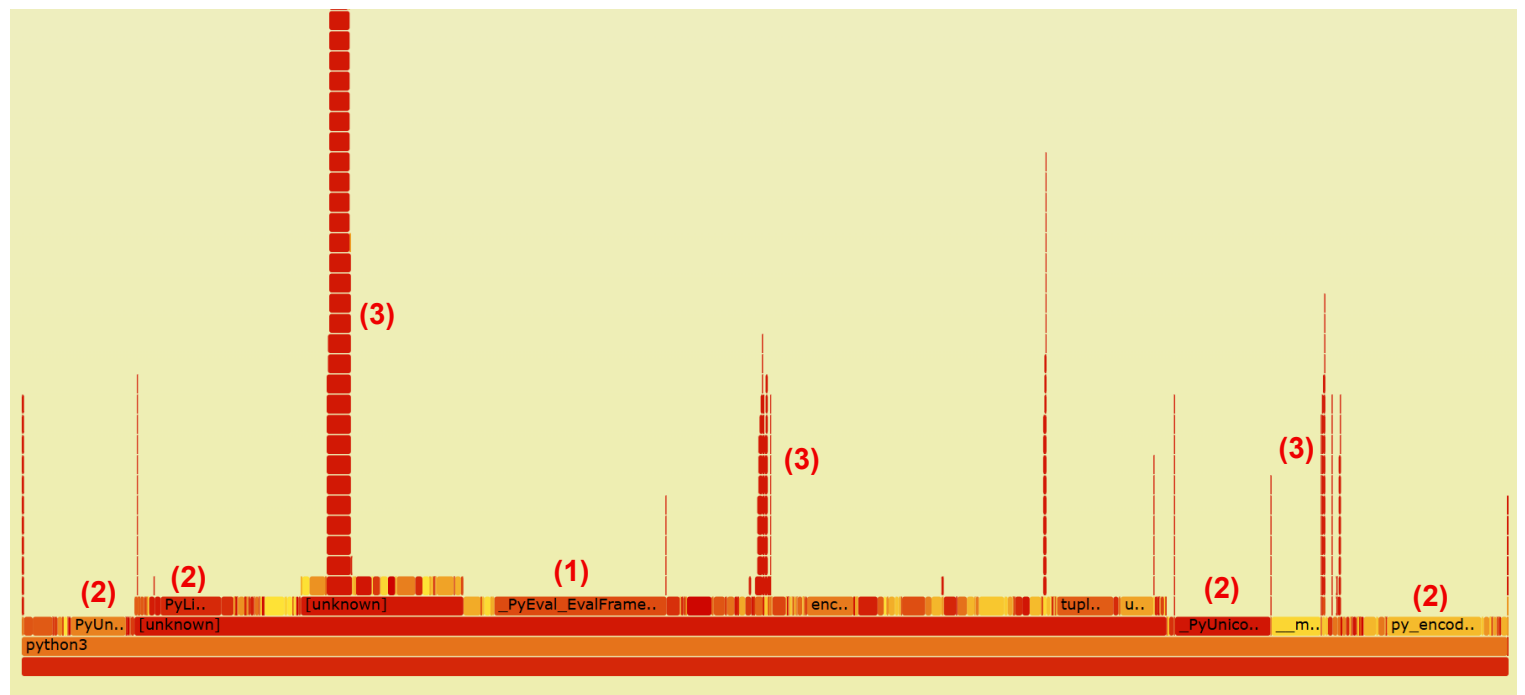
Flame graphs 2.21

כאמור ב-FlameGraph, ככל שהמלבן רחב יותר- כך הפונקציה צריכה יותר זמן CPU באופן ישיר או עקיף.
המשמעות היא:

- **רוחב המלבן** \approx כמה זמן בוצעה הפונקציה (או פונקציות שהיא קראה).
- **גובה הסטאק** = עומק הקריאה (מי קרא את מי).

Flame Graph עבור json.dumps המקורי ללא אופטימיזציות :

בנוסף למדידה הכמותית, ביצענו ניתוח עומק באמצעות perf והפקת Flame Graph. גרף זה מאפשר להציג באופן חזותי את צריכת זמן המעבד של כל פונקציה במהלך הביצוע.



מן הגרף ומן הקובץ out_baseline.folded אשר מוזן לflame graph עלו הממצאים הבאים:

1. דומיננטיות של _PyEval_EvalFrameDefault עומס פרשני מתון אך משמעותי

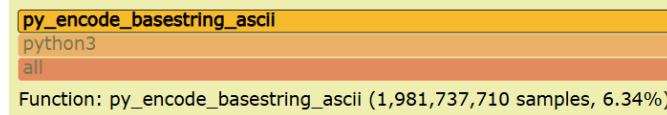
הפונקציה PyEval_EvalFrameDefault היא ליבת לולאת הbytecode interpreter loop. הופעתה ב-11.58% מכלל הדגימות מצביעה על כך שחלק לא קטן מהזמן מוקדש לפרשנות עצמה – כלומר, לפרש כל שורת קוד פייתון בזמן ריצה ולא להריץ קוד מהודר (compiled). במקרה שלנו, מדובר במאמץ פרשני שאינו מזערי, אך גם לא הקיצוני ביותר. עדיין, בהינתן שהבנצ'מרק מבצע סדרה ארוכה של פעולות סריאליזציה, הופעה משמעותית של הפונקציה הזו מעידה על overhead פרשני שראוי להקטין.
המשמעות: המימוש הבסיסי סובל מעומס בפרשנות של bytecode דבר שמאט ביצועים בפעולות רקורסיביות כמו json.dumps.

_PyEval_EvalFrameDefault
[unknown]
python3
all
Function: _PyEval_EvalFrameDefault (3,622,372,736 samples, 11.58%)

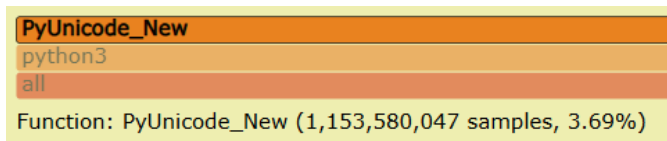
2. פונקציות מרכזיות בפרופיל הביצועים של json.dumps

במהלך ניתוח FlameGraph של json.dumps בגרסתו הבסיסית (ללא אופטימיזציה), זיהינו מספר פונקציות מרכזיות שתופסות חלק משמעותי מזמן הריצה. להלן פירוט של הפונקציות הבולטות ביותר:

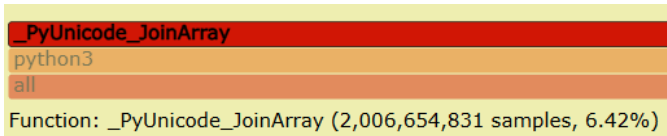
- **py_encode_basestring_ascii**: זוהי הפונקציה שאחראית על המרת מחרוזות למחרוזות JSON תקינות תוך ביצוע escape לתווים מיוחדים כמו \, \n ועוד. הפונקציה מופיעה באופן תדיר מאוד בפרופיל, בשל הריבוי של מחרוזות במבני JSON, ובכך מהווה צוואר בקבוק מובהק.



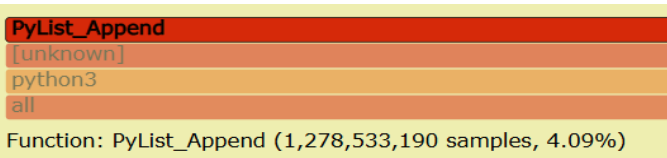
- **PyUnicode_New**: פונקציה זו מקצה אובייקט Unicode חדש בזיכרון, ונקראת בכל פעם שנוצרת מחרוזת חדשה במהלך הסריאליזציה. עצם השימוש החוזר בה מצביע על פעילות הקצאה אינטנסיבית ו overhead-ניכר בניהול מחרוזות.



- **PyUnicode_JoinArray**: אחראית על חיבור (join) של מחרוזות מתוך מערך מחרוזות. פעולה זו חיונית כאשר ממירים רשימות או אובייקטים למחרוזות JSON, שכן היא זו שמכניסה את הפסיקים והמבנה התחבירי של JSON. גם פונקציה זו מופיעה בעומק רב ב-FlameGraph.



- **PyList_Append**: אחראית על הוספת איברים לרשימות. זו פעולה פנימית שנעשית רבות בזמן יצירת רשימות או מילונים כתחליף זמני בעת המעבר ל-JSON, ולכן גם היא תורמת לזמן ריצה כולל.



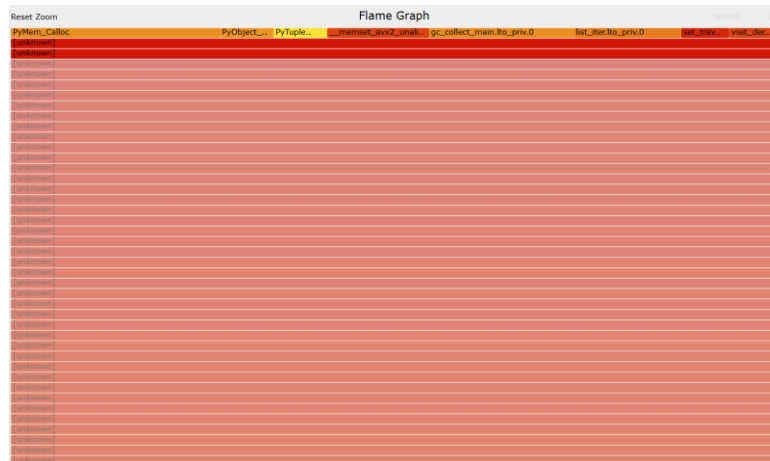
- **list_dealloc, tupledealloc, unicode_dealloc**: אלו הן פונקציות אחראיות לשחרור זיכרון של מבני נתונים זמניים (רשימות tuples, ומחרוזות). (הופעתן התכופה מעידה על כך שהרבה מאוד מבנים נוצרים ונמחקים שוב ושוב במהלך הריצה, מה שגורם ל- overhead משמעותי בניהול זיכרון.

הפונקציות הללו מייצגות את האזורים הקריטיים ביותר מבחינת צריכת זמן CPU. הממצאים מדגישים את הבעיות הקלאסיות של פייתון בביצועים: שימוש יתר בפרשנות במקום הידור, ניהול זיכרון בזבזני, והקצאות תכופות של מחרוזות. לכן, בבחירת אסטרטגיית אופטימיזציה, היה ברור שהחלפת json.dumps בספרייה מהודרת כמו orjson עשויה לצמצם משמעותית את השימוש בפונקציות הללו ולשפר את הביצועים באופן דרמטי.

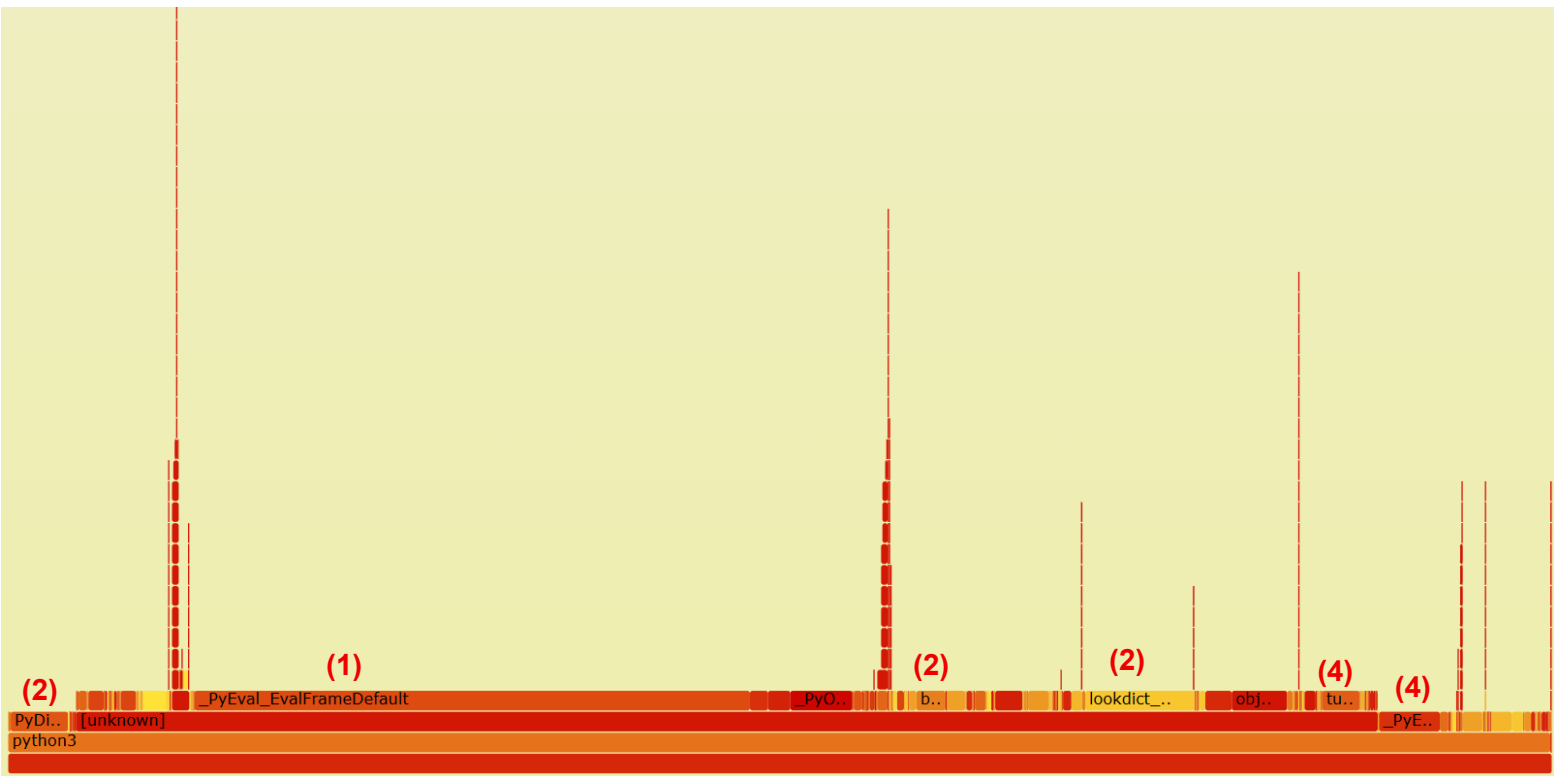
3. ניתוח מבני: עומק ה־FlameGraph

ב־FlameGraph של json.dumps בגרסה זו, ניתן להבחין במגדלים גבוהים מאוד- כלומר, ערימות של קריאות פונקציה עמוקות שנבנות זו על גבי זו. גובהם של מגדלים אלו מלמד על מספר רב של שכבות קריאה (deep call stacks), שמעידות על חוסר יעילות מבנית בקוד. המשמעות היא כפולה:

1. ה־**overhead של ה־interpreter**: הולך ותופס מקום מרכזי, שכן הקריאות נעשות בפייתון ולא בקוד מהודר. מסומן בגרף ב (3) סביב ה־unknown.
 2. **תחזוקת המחסנית** - כלומר, ניהול ההקשר של כל פונקציה בריצה - גוזלת משאבים באופן מצטבר.
- בנוסף, מגדלים אלו יוצרים הרבה **הקצאות זמניות בזיכרון**, שרובן משוחררות מיד לאחר מכן - מה שמסביר את הנוכחות הגבוהה של `dealloc functions` כמו `list_dealloc`, `unicode_dealloc` ועוד.

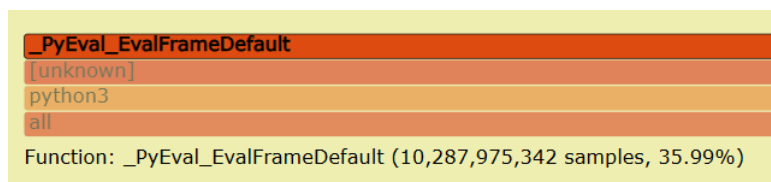


Flame Graph עבור json.dumps עם האופטימיזציות בגרסת optimized :



ביצענו הרצת פרופילינג מחודשת על הגרסה המאופטמת. מתוך גרף ה־ Flame המחודש והקובץ out_self_opt.folded ביצענו ניתוח מפורט של השינויים בפרופיל הביצועים. להלן הממצאים המרכזיים:

1. **דומיננטיות מוגברת של PyEval_EvalFrameDefault (~36%) :**
בזמן שהגרסה הבסיסית הראתה את הפונקציה הזו סביב 11.6% מכלל הדגימות, בגרסת ה־ optimized היא עלתה ל־~36%. ההסבר: זמן הריצה הכולל ירד משמעותית כך שפעולות encode כבדות כמעט נעלמו מהפרופיל. מה שנותר הוא בעיקר overhead פרשני – לולאת הבייטקוד של פייתון שמבצעת את בדיקות ה־ cache וה־ fast-path.



2. **הסת צווארי הבקבוק מפעולות Unicode לפעולות dict ו־id :**
הפונקציות ששולטות כעת הן:
 - lookdict_unicode_nodummy (~7.7%) - חיפוש מפתחות במילון (במיוחד בגישת (cache
 - PyDict_GetItemWithError (~3.7%) - שליפת ערכים ממילון.
 - builtin_id (~1.7%) - עקב שימוש ב־ id(obj) לתיג האובייקטים החוזרים.
 - _PyObject_GenericGetAttrWithDict (~4%) : קריאות getattr פנימיות.אלו מייצגות במדויק את עיצוב האופטימיזציה – פחות קידוד מחרוזות, יותר גישות ל־ dict של cache.

lookdict_unicode_nodummy.lto_priv.0
[unknown]
python3
all

Function: lookdict_unicode_nodummy.lto_priv.0 (2,023,995,639 samples, 7.08%)

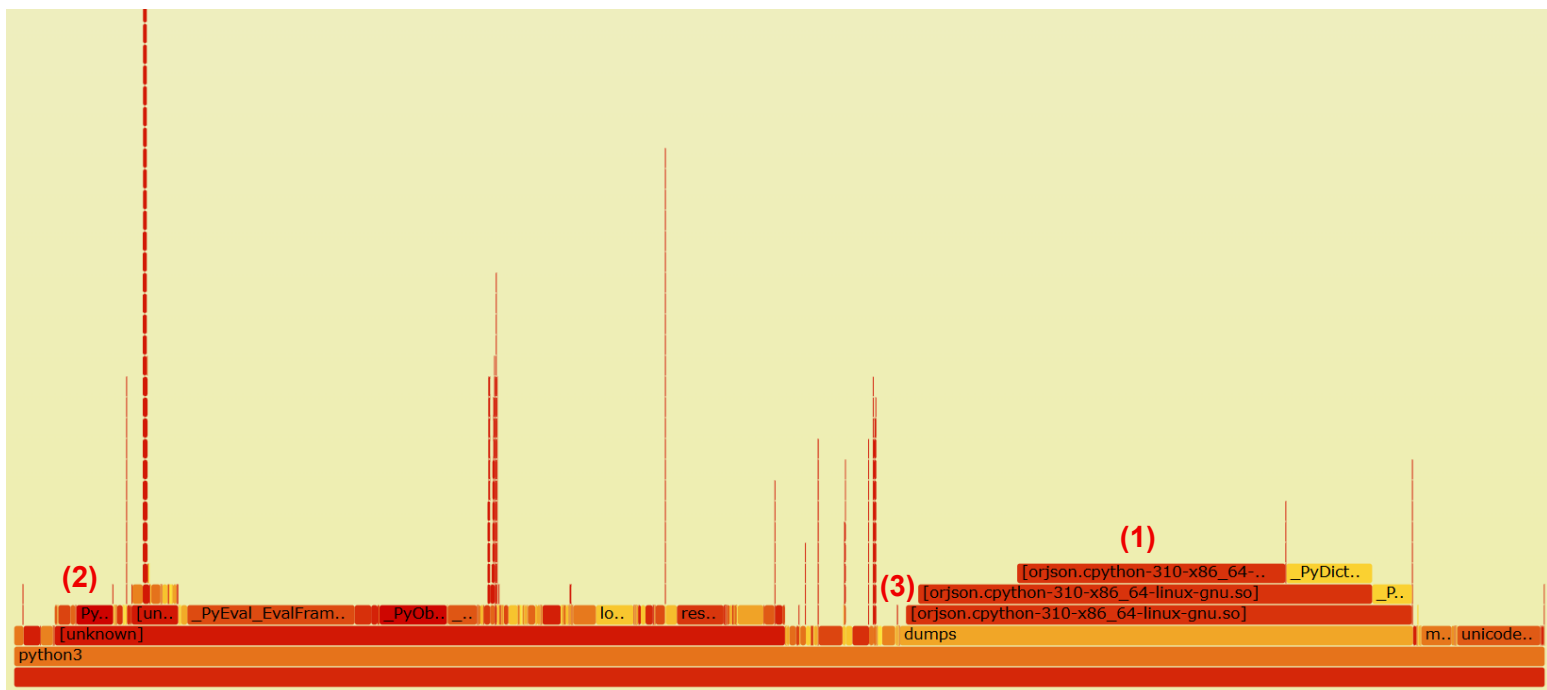
3. היעלמות צווארי בקבוק הקשורים למחרוזות

פונקציות כבדות שהיו דומיננטיות בבסיסליין כמו `py_encode_basestring_ascii` ו-`PyUnicode_New` ו-`PyUnicode_JoinArray` כמעט ואינן מופיעות ב-`optimized`, זה נובע מה-`cache` שמחזיר מחרוזות JSON מוכנות במקום לבצע `escape` והקצאות חוזרות.

4. ניהול פריימים ו- GC עדיין נוכח

יש נתח ניכר מ-`_PyEval_MakeFrameVector` (~3.95%) ו-`frame_dealloc` (~1.9%), מה שמעיד שעדיין יש יצירה והשמדה של פריימים פרשניים. נוסף לכך מופיעים `tupledealloc` (~2.4%) ו-`object_dealloc` (~1.7%), אך פחות משמעותיים מאשר בגרסה הבסיסית.

Flame Graph עבור json.dumps עם האופטימיזציות בגרסת fast :

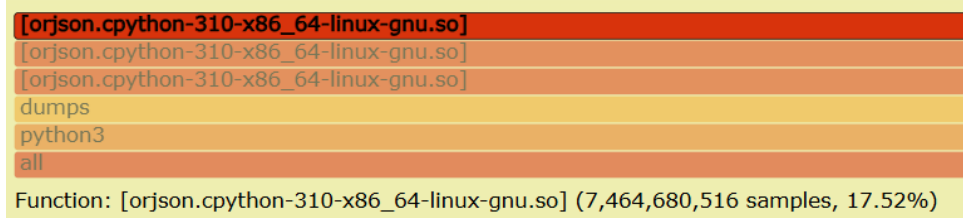


לאחר החלפת json.dumps ב-orjson.dumps , ביצענו הרצת פרופילינג מחדשת, שבסופה התקבלו כ-42.6 מיליארד דגימות. מתוך גרף ה־ Flame המחודש והקובץ out.folded ביצענו ניתוח מפורט של השינויים בפרופיל הביצועים. להלן הממצאים המרכזיים:

1. ביצוע ריכוזי ויעיל באמצעות ספרייה מהודרת (compiled C) :

בגרסה המאופטמת, הפונקציה json.dumps הוחלפה בקריאה ל-orjson.dumps , שהיא ספריית C מהודרת (compiled) . מהגרף ניתן לראות כי רוב העומס מרוכז בפונקציה אחת: python3;dumps;[orjson.cpython-310-x86_64-linux-gnu.so]

הופעתה כ"מגדל" בולט עם מעל **7 מיליארד דגימות** מצביעה על כך שכל פעולת הסריאליזציה מתבצעת בצורה מהודרת ואופטימלית בתוך קוד C , ללא הצורך בפענוח שורת פייתון אחר שורת פייתון כפי שהיה ב־json.dumps .
המשמעות : overhead פרשני נמנע כמעט לחלוטין, ופעולות כמו ניהול מחרוזות, הקצאות זיכרון ו־ join של מחרוזות אינן נדרשות יותר ברמת Python - אלא מבוצעות כולן בליבה יעילה ומהירה של orjson .



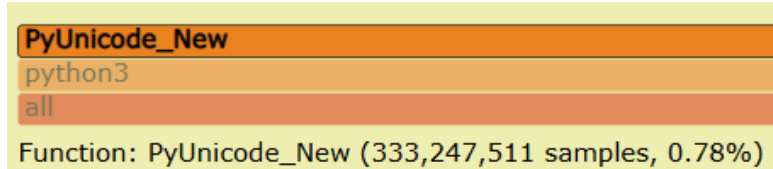
2. צמצום צווארי הבקבוק מממשק הניהול של מחרוזות

פונקציות כגון:

- `py_encode_basestring_ascii`
- `_PyUnicode_JoinArray`
- `PyUnicode_New`
- `unicode_dealloc`

אשר היו דומיננטיות בפרופיל הבסיסי - אינן מופיעות כלל בפרופיל האופטימיזציה, או שהופעתן שולית ביותר.

המשמעות היא ש- `orjson` מדלג על שכבות העיבוד הכבדות של `Unicode` שמבוצעות ב- `Python`, על ידי מימוש ישיר ומהודר של סריאליזציה ב- `Rust`, עם ניהול זיכרון יעיל בהרבה.

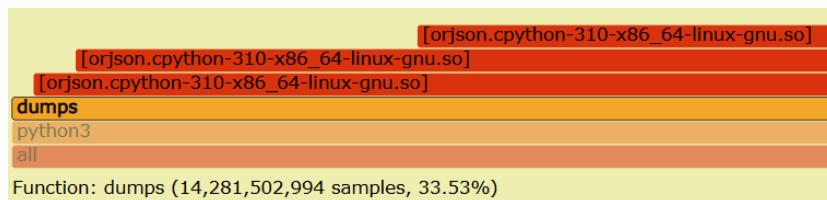


3. הופעת פונקציות חדשות ממודול `orjson`

בגרסה האופטימלית ניתן לזהות הופעה של פונקציות ממודול `orjson` כגון:

- `orjson.dumps`
- `orjson_encode`

פונקציות פנימיות של `Rust` לרוב מוצגות כאנונימיות או כחלק ממודול `C/Rust` אלה משקפות את המעבר לביצוע בפונקציות מהודרות ויעילות, ואת צמצום עומס החישוב מהפרשן של פייתון עצמו. זה הופך את הביצועים לטובים יותר, אך גם את הגרף לפשוט יותר, כי רוב העבודה נעשית בפונקציה אחת מהירה.



4. שיפור ניכר בניהול זיכרון

בגרסה הבסיסית הופיעו פונקציות רבות הקשורות להקצאה ושחרור זיכרון `list_dealloc`, `tupledealloc`, `unicode_dealloc`, `PyList_Append` האופטימלי.

זוהי עדות לכך ש- `orjson` מקצה מבני נתונים בצורה מרוכזת ויעילה יותר, תוך הימנעות מיצירה ופירוק תכופים של מבנים זמניים במהלך הסריאליזציה.

5. דומיננטיות של `PyEval_EvalFrameDefault`

למרות השימוש בספרייה המהודרת `orjson` בגרסה האופטימלית (`dumps_fast`), והירידה הדרמטית בזמני הריצה ובצריכת הזיכרון, עדיין ניתן לראות ב- `FlameGraph` כי הפונקציה `PyEval_EvalFrameDefault` נותרה אחת השכבות הבולטות בפרופיל – גם אם בעומק מופחת ובשכבות צרות יותר. הסיבה לכך היא ש- `PyEval_EvalFrameDefault` היא לולאת הפרשנות (`interpreter loop`) של פייתון – והיא לא קשורה רק לפונקציית הסריאליזציה עצמה, אלא גם לכל שאר הקוד שמריץ את הבנצ'מרק, כולל:

- קריאה לפונקציית `dumps_fast`
 - טיפול בנתונים לפני ואחרי הסיריזציה
 - בדיקת ביצועים, חזרות (loops) והרצת הבנצ'מרק (כחלק מהקוד של `pyperf`)
- כל עוד הבנצ'מרק כתוב בפיתון (כולל הלוגיקה שמקיפה את הקריאה ל-`orjson`) לא ניתן להימנע לגמרי מהשפעת לולאת הפרשנות. מה שכן השתנה הוא שפונקציות אחרות שתפסו נפח משמעותי בפרופיל כמו `PyUnicode_New`, `py_encode_basestring_ascii`, `unicode_dealloc` ועוד כמעט ונעלמו, משום שהן הוחלפו על ידי מימוש מהודר (compiled) ב-`Rust` בספריית `orjson`.
- לכן, הופעת `PyEval_EvalFrameDefault` גם בגרסה האופטימלית אינה מעידה על כשל באופטימיזציה, אלא על מגבלה מובנית בכך שהריצה עצמה מתבצעת בתוך סביבת פיתון.

2.3. שיפורים ופעולות אופטימיזציה

במהלך הפרויקט ביצענו אופטימיזציה לביצועי הבנצ'מרק `json.dumps`, אשר בגרסתו הבסיסית משתמש בפונקציה המובנית `json.dumps()` של פייתון. פונקציה זו מבצעת סריאליזציה של מבני נתונים לפורמט JSON באמצעות לולאת הפרשנות של פייתון, מה שגורם לעומס רקורסיבי וניהול מחרוזות אינטנסיבי.

לצורך העבודה יצרנו קובץ ייעודי `my_json_dumps.py`, ששמר על התאימות המלאה ל-API המקורי (`loads`, `load`, `dumps`, `dump`), אך איפשר לנו להכניס שיפורים מבוקרים. נוספו בו שתי גרסאות חדשות: `dumps_optimized` ו-`dumps_fast`.

`dumps_optimized`

גרסה זו כללה מספר שיפורים עיקריים:

- **Compact separators**: כאשר לא נמסרים ערכים אחרים, נעשה שימוש כברירת מחדל ב-`(',', ':')`, דבר שהוביל לקיצור מחרוזות הפלט, הפחתת תווים מיותרים והקטנת צריכת זיכרון.
- **Fast-path לפרמיטיבים**: טיפוסים בסיסיים (`None`, `bool`, `int`, `float`, `str`) מקודדים ישירות, ללא קריאה למנגנון הכבד של `JSONEncoder`.
- **Caching**: עבור אובייקטים שחוזרים על עצמם (למשל `dict` שחוזר ברשימה גדולה) נשמרת תוצאת הסריאליזציה במילון פנימי. בפעמים הבאות מוחזרת המחרוזת מה-`cache` במקום לבצע `encode` מחדש.

שילוב של שלושת השיפורים האלו הפחית בצורה דרמטית את הקריאות לפונקציות כבדות של Unicode והקצאות חוזרות, ובמיוחד במקרים כמו `HUGE` (אלפי מופעים של אותו `dict` או `NESTED`) הוביל לשיפור משמעותי מאוד.

`dumps_fast`

גרסה זו נועדה לספק את הביצועים המשופרים כאשר מותקנות ספריות חיצוניות:

- קודם נעשה ניסיון לייבא את `orjson` הממומשת ב-`Rust`. אם היא קיימת, היא משמשת כברירת מחדל, ולאחר ההמרה מ-`bytes` ל-`string` מספקת ביצועים יוצאי דופן.
- אם `orjson` אינה זמינה, נעשה ניסיון לייבא את `ujson`.
- אם אף אחת מהספריות אינה מותקנת, מתבצע fallback לגרסת `dumps_optimized` המקומית.

להלן מצורפים קטעי הקוד שהוספנו ושינינו בכדי להגיע לשיפור הדרוש:

```
def dumps_optimized(obj, *, skipkeys=False, ensure_ascii=True, check_circular=True,
                    allow_nan=True, cls=None, indent=None, separators=None,
                    default=None, sort_keys=False, **kw):
    """
    Optimized dumps with:
    - Primitive fast-paths (None/True/False/int/float)
    - Caching of repeated objects
    - Fallback to standard JSONEncoder for all other cases
    """

    if separators is None and indent is None:
        separators = (',', ':')

    # Primitive fast path
    if obj is None:
        return "null"
    if obj is True:
        return "true"
    if obj is False:
        return "false"
    if isinstance(obj, (int, float)):
        return str(obj)
    if isinstance(obj, str):
        return json.dumps(obj, ensure_ascii=ensure_ascii)
```

```
# Check cache
if isinstance(obj, (dict, list, tuple)):
    obj_id = id(obj)
    if obj_id in _cache:
        return _cache[obj_id]

# Fallback: use the standard dumps (default encoder)
result = dumps(obj, skipkeys=skipkeys, ensure_ascii=ensure_ascii,
               check_circular=check_circular, allow_nan=allow_nan,
               cls=cls, indent=indent, separators=separators,
               default=default, sort_keys=sort_keys, **kw)

if isinstance(obj, (dict, list, tuple)):
    _cache[id(obj)] = result
return result
```





```

317 # Example of how to integrate faster JSON libraries
318 def dumps_fast(obj, **kwargs):
319     """
320     Fastest possible JSON dumps using external libraries.
321     Falls back to standard library if not available.
322     """
323     try:
324         import orjson
325         # orjson returns bytes, so decode to string
326         return orjson.dumps(obj).decode('utf-8')
327     except ImportError:
328         try:
329             import ujson
330             return ujson.dumps(obj)
331         except ImportError:
332             # Fall back to optimized standard library version
333             return dumps_optimized(obj, **kwargs)
334

```

2.4 השוואת ביצועים

נשווה את הביצועים ונציג את השיפור:

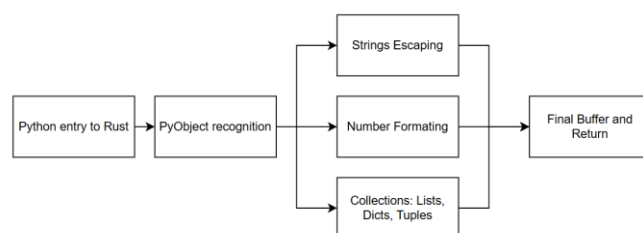
Metric	Original (json.dumps)	Optimized(orjson.dumps)	Optimized(dump_optimized)
Average Runtime	<ul style="list-style-type: none"> זמן ממוצע להרצה אחת : 7.35 ms סטיית תקן: $\pm 0.31\text{ ms}$ 	<ul style="list-style-type: none"> זמן ממוצע להרצה אחת : 1.30 ms סטיית תקן: $\pm 0.05\text{ ms}$ 	<ul style="list-style-type: none"> זמן ממוצע להרצה אחת : 831 us סטיית תקן: $\pm 35\text{ us}$
Speedup	○	 ↓ 82.3% Faster × 5.66	 ↓ 88.7% Faster × 8.85
FlameGraph Total Samples	31,277,168,328 samples	42,595,973,666 samples	28,585,577,903 samples
String Encoding Overhead	גבוה - py_encode_basestring_ascii + PyUnicode_New דומיננטיים	מופחת מאוד בזכות מימוש Rust ב־ מהודר	כמעט נעלם – שימוש ב־ cache והפחתת קריאות ל־Unicode
Improved memory management	ניהול הקצאות לא יעיל, הופעות list_dealloc / רבות של unicode_dealloc	הקצאות יעילות מאוד ברמת native	הקטנת הקצאות חוזרות באמצעות cache, שחרור זיכרון מופחת

2.5. שיפור חומרתי

נציין כי את האינטגרציה של המימוש החומרתי נבחן למול הגרסה **המשופרת** של הקוד שלנו, כאשר מירב ההתעסקות שלנו תהיה בהאצת האלגוריתם של orjson.

- **Software analysis - Orjson algorithm:**

- a. **block diagram:**



*After the entry to Rust, a new byte buffer is created and filled at the end.

- **צווארי בקבוק תוכנתיים:**

- לאחר הזיהוי של סוג PyObjectn , האלגוריתם מגיע לעיקר תכליתו – ביצוע פעולת הסריאליזציה. כאשר ניתוח הביצועים משתנה באופן משמעותי בהתאם למידע המגיע:

- עבור אובייקטי string גדולים – קיטעון string יכול לקחת עד כ-60% מזמן הריצה.
 - עבור אובייקטי מספר גדולים (ורבים) – התוצאה זהה, וההעברה לפורמט הנדרש גם היא יכולה לקחת עד כ-60% מזמן הריצה.
 - עבור אובייקטי מילון ואוספים שונים – התוצאה שוב זהה והזמן הדרוש יכול להגיע לעד כ-60% מזמן הריצה.
 - למול ניתוח זה, נרצה לשלב את המאיץ החומרתי שלנו בלב האלגוריתם שהוא גם עיקר צוואר הבקבוק התוכנתי.

- **נק' מפתח בשילוב החומרה:**

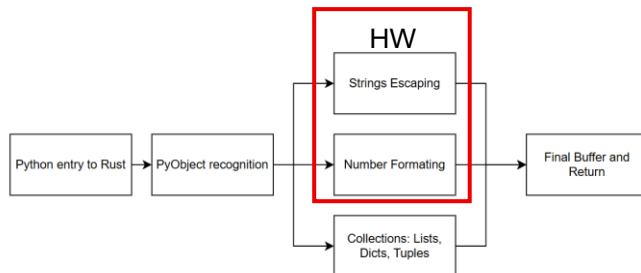
- **חסרונות החומרה:**

- **זיהוי PyObject** – זיהוי האובייקט נעשה כרגע ע"י תוכנה תוך שימוש בדגלים ופוינטרים לטובת הסיווג. פעולה זו זולה ומיידידת מבחינת משאבים, שילוביות שלה בחומרה מורכבת (בשל השימוש בפוינטרים) ולחומרה לא תהיה יעילות על התוכנה.
 - **סריאליזציה של אוספים** – באוספים (רשימות/מילונים וכו') לעיתים קרובות נראה שמירה של משתנים באופן רקורסיבי, כדוגמת רשימות מקוננות. השימוש באובייקטים בצורה רקורסיבית מקשה על ביצועו באמצעות חומרה ולא יהיה יעיל למול השימוש בתוכנה, זאת מכיוון שתהיה תקורה משמעותית של תקשורת חומרה תוכנה.
 - **תקשורת חומרה תוכנה** – תקורה שאין לנו כאשר אנו משתמשים בתוכנה בלבד. לטובת חישובי ההמשך נניח שימוש בפרוטוקול PCIe-3 במהירות של 8Gb/s , כאשר העברת קובץ (חד כיוונית) בגודל 1kb תיארך כ-1 מיקרו שנייה.

○ יתרונות החומרה –

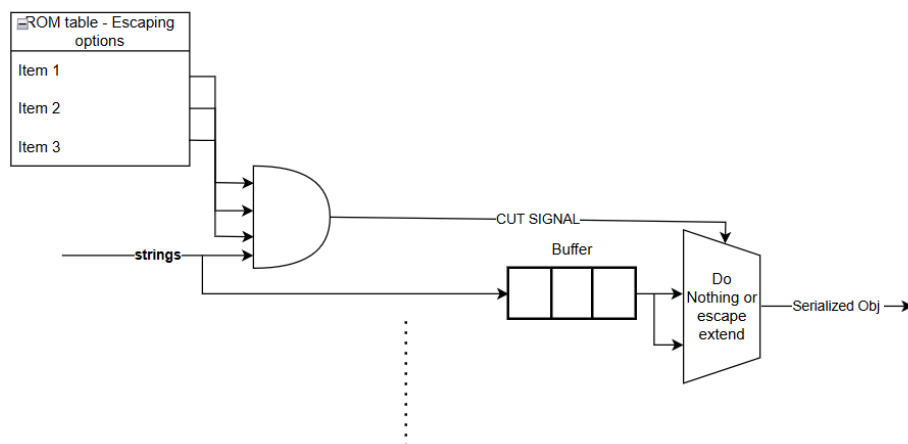
- **קיטעון של string** – כרגע נעשה בתוכנה באמצעות פקודות branch. חוסר הקונסיסטנטיות של המידע לא מאפשר חיזוי איכותי של פקודות branch ומוביל לכמות גבוהה של branch misses. כאשר מימוש חומרתי יהיה פשוט ומהיר.
- **העברת פורמט של float** – כרגע נעשה בתוכנה באמצעות אלגוריתם RYU, נחשב לאלגוריתם המהיר ביותר להמרה של float לstring. בחומרה הפעולה יכולה להתבצע במקביל במספר מעגלים מצומצם.

• מימוש המאיץ החומרתי:



כפי שהוסבר קודם, עבור המאיץ החומרתי ניישם string escaping & number formatting. כאשר עבור ממשק החומרה-תוכנה נשתמש ביכולת DMA ונעביר למאיץ החומרתי pointer לזיכרון ואורך.

○ String Escaping:

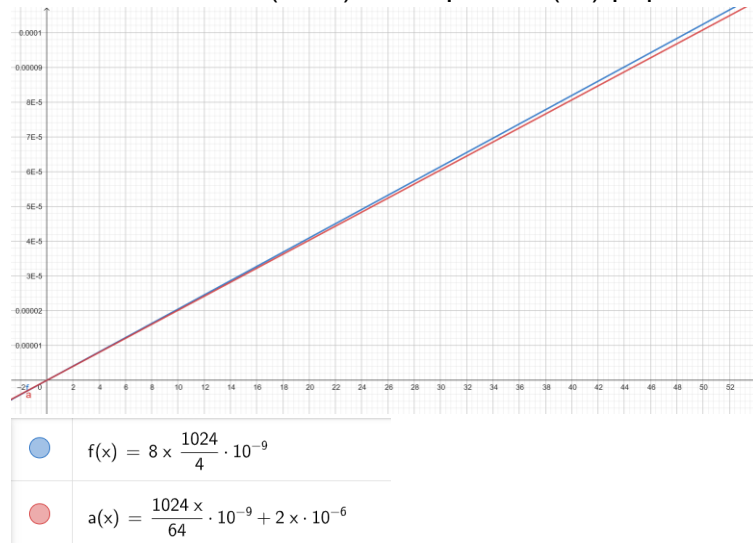


אובייקט string מגיע מהזיכרון ונבדקת התאמה שלו, ע"י שימוש בלוגיקת AND לאחד מדפוסי הקיטעון המוגדרים המאוחסנים בטבלת ROM. אם יש רצף המתאים לקיטעון ביציאה נקבל את הרצף משוכפל (כנדרש), אחרת – נשרשר קדימה את האובייקט כפי שהוא. המימוש כולו הוא עם לוגיקה א-סינכרונית (פרט לתורים בכניסה ויציאה לשמירה על משטר הזמנים וסנכרון עם הסביבה החיצונית) ואורך מחזור שעון יחיד. מיקרו-ארכיטקטורה זו משוכפלת לפי רוחב ה-BUS האפשרי מהזיכרון (בניח 64 בייט) ובעלת תפוקה של שרשר 64 בייטים של string בכל מחזור שעון.

• ניתוח ביצועים:

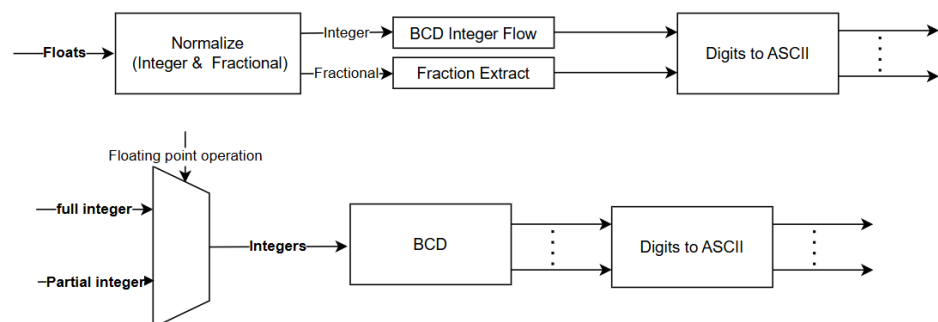
במימוש תוכנה בלבד פעולת ה-escaping תיקח בין 5-10 מחזורי שעון לכל בייט (branch + escape decision + copy). בהנחה של ריצת 4 חוטים במקביל בתהליך, עבור קובץ string בגודל 32kb, המימוש באמצעות תוכנה ייקח כ-65.5us. המימוש באמצעות חומרה (בהנחת DMA ופרוטוקול PCIe-3) ייקח: 512 מחזורי שעון עבור הלוגיקה עצמה וסך הכל 64.5us. עבור מקרה נקודתי זה נקבל שיפור של כ-2% בשימוש במאיץ חומרתי הייעודי. נציין שמבחינת שיקולי צריכת אנרגיה, האנרגיה הנצרכת ע"י המאיץ החומרתי זניחה לעומת האנרגיה הנצרכת ע"י פעולות המעבד.

נצרך גרף המציג את השיפורים:
גודל הקובץ (kb) למול זמן הביצוע (שניות)



בכחול – ללא המאיץ החומרתי, באדום – בשימוש המאיץ החומרתי. ניתן לראות שעל אף שהפעולה הלוגית במאיץ החומרתי מתבצעת בזמן הקצר ביותר האפשרי (מחזור שעון יחיד), התקורה המשמעותית הנוספת בשל תקשורת חומרה-תוכנה מראה שאין כדאיות בשימוש במאיץ עבור מקרה זה.

○ Number Formatting:

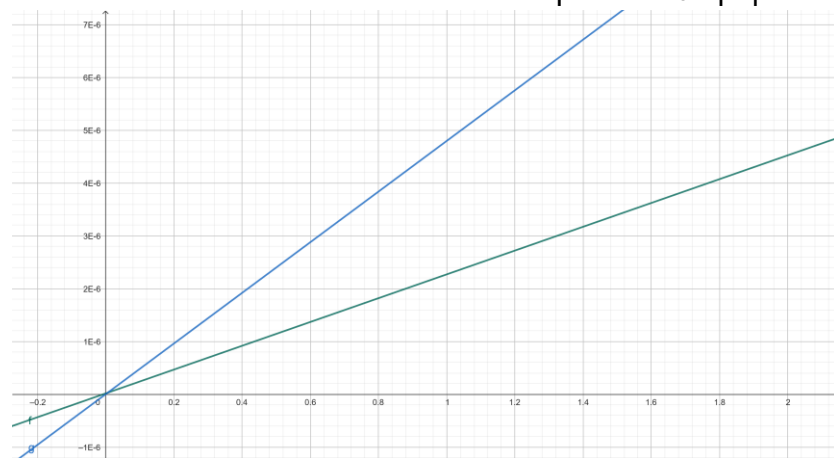


ניגע בקצרה בתתי המודולים המשולבים ולאחר מכן נבצע ניתוח ביצועים:

1. **בלוק הנורמליזציה** – אחראי על פיצול החוטים המייצגים את משתנה ה-FLOAT (mantissa/exponent/sign) ולאחר מכן, בעזרת שימוש במכפילים והזזות מפריד את החלק העשרוני והחלק השלם.
2. **בלוק ה-BCD** – בעל שימוש כפול (הן ל-integers והן ל-floats לחישוב החלק השלם). בלוק גנרי הכולל שימוש בהזזות Adders ע"פ אלגוריתם Double-dabble.

3. **בלוק חישוב השבר** – כולל שימוש במכפילים ב10 ומפריד את הספרות העשרוניות.
4. **בלוק ההמרה לASCII** – כולל המרה של כל ספרה לייצוג המתאים בASCII.

מבחינת ביצועים, המערכת הינה מערכת מצונרת, כאשר התפוקה הסופית שלה תהיה 1 floating point per cycle. עבור קובץ שמכיל 32kb, ע"פ הנחות היסוד לתקשורת שצינו קודם, נחשב את ההבדל בין התוכנה לשימוש במאיץ חומרתי (נניח גודל משתנה הוא 4 – בייט) חומרה: בהערכה שמסלול הצינור הינו באורך של 15 מחזורי שעון – $72.2 = 64us + 8.2us$ תוכנה: זמן עבודה ממוצע של אלגוריתם RYU הינו 75 מחזורי שעון, נניח עבודה במקביל של 4 חוטים ונקבל – 153.6us. קיבלנו שיפור של כפי 2 בין השימוש בתוכנה בלבד לבין השימוש במאיץ חומרתי. גודל הקובץ בkb למול זמן בשניות:



הגרף הירוק מציג שימוש במאיץ חומרתי והגרף הכחול מציג שימוש בתוכנה בלבד.

●	$f(x) = 2 \times 10^{-6} + \left(15 + x \cdot \frac{1024}{4}\right) \cdot 10^{-9}$
●	$g(x) = x \cdot \frac{1024}{4 \cdot 4} \cdot 75 \cdot 10^{-9}$

ניתן לראות ששימוש במאיץ חומרתי מציג שיפור הולך וגדל כתלות בגודל הקובץ.

נבצע ניתוח סופי ע"פ חוק אמדל לשימוש במאיץ החומרתי עבור קובץ המכיל בעיקר מספרים שנדרש להמיר לפורמט ASCII:

$$speed\ up = \frac{1}{0.3 \cdot \frac{0.5}{2} + 0.7} \sim 1.3$$

בשימוש במאיץ נקבל האצת ביצועים של כ-30%, האצה משמעותית.

2.6. מסקנות

בפרויקט זה בחנו את ביצועי ה- `benchmark json_dumps` הן במימוש הבסיסי של Python והן לאחר שילוב ספריות חיצוניות ושיפורים פנימיים שהוטמעו בקובץ `my_json_dumps.py`. באמצעות FlameGraph ו- `profiling` זיהינו שצווארי הבקבוק המרכזיים במימוש הבסיסי הם:

- טיפול במחרוזות (string escaping).
- הקצאות חוזרות ושחרור מחרוזות (`PyUnicode_New`, `unicode_dealloc`).
- פעולות `join` של `Unicode` בעת עיבוד רשימות ואובייקטים.
- עומס פרשני בלולאת ה- (`PyEval_EvalFrameDefault`) `bytecode`.

שיפורים ברמת התוכנה

במהלך העבודה נוספו שתי גרסאות משופרות:

1. `dumps_fast` (מבוסס `orjson/ujson`):

- אשר סיפקה את הביצועים הטובים ביותר עבור מבנים פשוטים, כגון **SIMPLE**, בזכות מימוש יעיל ב- `Rust/C`.
- במקרה **SIMPLE** נמדד זמן ריצה נמוך יותר מאשר ב- `dumps_optimized` מה שמעיד ש- `orjson` יעילה במיוחד במבנים שטוחים ומבוססי מחרוזות.

2. `dumps_optimized` (השיפורים שלנו):

- התגלתה כעדיפה במבנים מקוננים או חוזרים על עצמם, כגון **NESTED** ו- **HUGE** למשל.
- בזכות שימוש ב- `cache` ובנתיבי `fast-path` לפרימיטיבים, הצליחה להימנע מקידוד מחרוזות חוזר ולקצר משמעותית את זמן הריצה.
- מסקנה: אין "מנצח יחיד" – הספרייה החיצונית `orjson` עדיפה עבור מבנים פשוטים, בעוד שהשיפורים הפנימיים מספקים יתרון עצום במבנים חוזרים ומקוננים.

שיפורים ברמת החומרה

בחנו שילוב של מאיץ חומרתי ייעודי בשני תחומים:

- **String Escaping**: למרות שמימוש חומרתי מבצע את הפעולה במחזור שעון יחיד, תקורת התקשורת בין החומרה לתוכנה מבטלת את היתרון. בפועל התקבל שיפור זניח של כ-2% בלבד, ולכן אין כדאיות ממשית למימוש חומרתי עבור מחרוזות.
- **Number Formatting**: מאיץ חומרתי צינורי מאפשר עיבוד של מספר בכל מחזור, והציג שיפור של עד פי 2 בביצועים. בחישוב לפי חוק אמדל עבור קובץ עשיר במספרים התקבלה האצת ביצועים כוללת של כ-30% המהווה שיפור משמעותי ביחס לתוכנה בלבד.

סיכום

1. **האופטימיזציה התוכנית** באמצעות orjson הביאה לשיפור חד בביצועים - פי 5.66.
2. **האופטימיזציה המקומית** (dumps_optimized) סיפקה יתרון משמעותי במבנים מורכבים או חוזרים על עצמם – לדוגמה, ב־NESTED נמדד שיפור של פי 8.85 (88.7% ירידה בזמן הריצה), וב־HUGE התקבל שיפור משמעותי מאוד בזכות מנגנון cache שמונע קידוד חוזר של אובייקטים.
3. **הבחירה בפתרון תלויה במבנה הנתונים**: עבור מבנים שטוחים עם הרבה מחרוזות (SIMPLE) עדיף להשתמש בספרייה חיצונית מהודרת כמו orjson, בעוד שעבור מבנים מקוננים או חוזרים (NESTED, HUGE) הפתרון המותאם שלנו יעיל בהרבה.
4. **האצת החומרה אינה אחידה בערכה** –בעוד שב־string escaping לא הושג שיפור מהותי (כ־2% בלבד) בשל תקורת תקשורת חומרה–תוכנה, בהמרת מספרים התקבל שיפור משמעותי של פי 2, ובחישוב כולל לפי חוק אמדל שיפור ביצועים של כ־30%. הדבר מראה שחומרה מתאימה רק במקומות בהם החישוב צפוי, חוזר על עצמו, ובעל מבנה נתונים פשוט יחסית.
5. **שיקולי עלות–תועלת** –ההשקעה בפיתוח מאיץ חומרתי ייעודי מצדיקה את עצמה רק במקרים שבהם הנתונים נשלטים והעיבוד מתמקד בפעולות מספריות חוזרות. לעומת זאת, כאשר נפחי העבודה מגוונים (מחרוזות, מילונים, אוספים רקורסיביים), פתרון תוכנה יעיל יותר מספק מענה רחב יותר.
6. **השילוב של תוכנה משופרת עם מאיץ חומרתי ייעודי** מראה פוטנציאל לאיזון בין שיפור ביצועים לבין מורכבות הפיתוח, כאשר האצת מספרים היא התחום שבו ניתן להשיג את הערך הרב ביותר. המאיץ החומרתי אינו תחליף, אלא השלמה שמאפשרת מיצוי נוסף
7. **אסטרטגיה להמשך** –השילוב בין תוכנה אופטימלית לחומרה ייעודית מצביע על מודל עבודה נכון: קודם כל למצות את שיפור התוכנה (ספריות מתקדמות, אופטימיזציה אלגוריתמית), ורק לאחר מכן להכניס האצת חומרה ממוקדת במודולים בהם יש ערך מוסף אמיתי.

3. Logging

3.1. סקירה - Overview

כעת בחרנו לנתח ולבצע אופטימיזציה לבנצ'מרק Logging מתוך חבילת pyperformance . מטרת הבנצ'מרק היא למדוד את העלות הכרוכה בהפקה, עיבוד והפצה של הודעות לוג בסביבת פייתון. מכיוון שלוגינג הוא רכיב מרכזי כמעט בכל מערכת תוכנה - משרתי ווב ועד מערכות משובצות - שיפור ביצועי המנגנון יכול להצטבר לרווח ניכר בזמן ריצה ובעומסים גבוהים.

הסקירה שלנו מתמקדת בהבנת תהליך יצירת הודעת לוג: מהשלב שבו המתכנת קורא ל- `logger.debug()` או `logger.info()`, דרך יצירת אובייקט `LogRecord`, עיצוב ההודעה (formatting), ועד ההפצה ל- `handlers` השונים. בכל שלב יש נקודות שבהן מצטבר `overhead` שניתן לייעל.

3.10. תיאור ה-benchmark

כפי שצינו, המטרה היא להבין עד כמה פעולות הלוג עצמן מוסיפות `overhead` למערכת, ואיך ניתן לשפר זאת באמצעות התאמות קטנות בקוד.

הבנצ'מרק Logging בפייתון בודק את הביצועים של ספריית ה- `logging` המובנית על ידי יצירת והדפסת מספר רב של הודעות לוג בתצורות שונות. בגרסה שלנו (`custom_logging_benchmark.py`) הוספנו יכולות מורחבות:

- בחירה בין ספריית `logging` המקורית (STD) לבין גרסה מותאמת (`my_logging.py`) הכוללת אופטימיזציות.
- **שליטה בפרמטרים:** מספר ההודעות, סוג ה- `Handler` (`stream`, `null`, `file`), רמת הפירוט של הפורמט `message` בלבד, פורמט פשוט, פורמט מפורט כולל (`thread/process` שימוש ב- `QueueHandler`, בדיקה עם או בלי `isEnabledFor`).
- **מידה קפדנית:** חישוב זמן ממוצע, סטיית תקן, קצב הודעות לשנייה (`throughput`), ופלט JSON לתיעוד.
- שילוב עם כלי `perf` לניתוח פרופילינג והפקת `FlameGraphs` לפני ואחרי האופטימיזציה.

3.11. השימוש בספריות

בבנצ'מרק נעשה שימוש בעיקר ב-:

- **Logging** - ספריית הלוגינג הסטנדרטית של פייתון.
- **Pyperformance** - חבילת הבנצ'מרקים הרשמית, ממנה לקחנו את המסגרת ואת רעיון המדידה.
- **argparse, json, subprocess** - ספריות עזר לקריאת פרמטרים, שמירת תוצאות בפורמט JSON, והרצת פקודות מערכת.
- **perf (Linux perf tool)** - כלי פרופילינג לאיסוף נתוני CPU ודגימות, מאפשר לנתח `bottlenecks` ולהפיק `FlameGraphs`.

3.12. מבני נתונים שנעשה בהם שימוש

LogRecord - האובייקט המרכזי שמייצג הודעת לוג. כולל רמת לוג, טקסט, שם מודול, thread, process, זמן וכו'.

רשימות (lists) - משמשות לאגירת הודעות לדוגמה ולבניית רצף רמות הלוג (DEBUG/INFO/WARNING/ERROR).

מילונים (dict) - לניהול meta-data ולמיפוי מהיר מרמת לוג לשיטת הקריאה (DEBUG → logger.debug).

תור (queue.Queue) - בתרחישים עם QueueHandler, לניהול הודעות בין חוטים.

Handlers כמו (StreamHandler, FileHandler, QueueHandler) - האובייקטים המקבלים את ההודעה ומבצעים את ההדפסה או הכתיבה ליעד.

ההרצה בוצעה באמצעות כלי המדידה pyperf ובנוסף בוצע פרופילינג עמוק באמצעות perf ו-FlameGraph.

בשימוש בפקודות הבאות:

הרצה של הגרסה הרגילה:

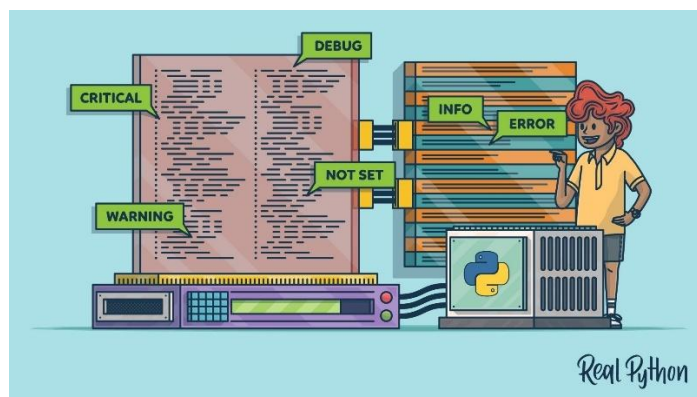
```
root@ubuntu:~/HwSw/logging_bench# perf record -F 99 -g -- python3 custom_logging_benchmark.py --mode std -n 300000 --enabled-checks --handler null -r 5
```

הרצה של הגרסה האופטימלית:

```
root@ubuntu:~/HwSw/logging_bench# perf record -F 99 -g -- python3 custom_logging_benchmark.py --mode my -n 300000 --enabled-checks --handler null -r 5
```

הרצה של השוואה בין שתי הגרסאות על מנת לקבוע את מידת השיפור:

```
root@ubuntu:~/HwSw/logging_bench# python3 compare_logging.py -n 300000 -r 5 --enabled-checks --handler null --formatter message --perf-freq 99
```



3.2. ניתוח ראשוני - Initial analysis

3.20. ניתוח ביצועים

בכדי להבין את הביצועים של ספריית **logging** המובנית בפייתון, ביצענו הרצות של הבנצ'מרק הן בגרסת **STD** (מקורית) והן בגרסה לאחר אופטימיזציות.

תוצאות הבנצ'מרק שהתקבלו:

- **STD** : זמן ממוצע ≈ 3.290 שניות, סטיית תקן ≈ 0.062 שניות.

```
>> perf record -F 99 -g -o perf_std.data -- python3 custom_logging_benchmark.py --mode std -n 300000 -r 5 --enabled-checks --h
andler null --formatter message --out logging_std.json
Couldn't record kernel reference relocation symbol
Symbol resolution may be skewed if relocation was used (e.g. kexec).
Check /proc/kallsyms permission or run as root.
perf_event__synthesize_bpf_events: failed to synthesize bpf images: No such file or directory
Couldn't synthesize bpf events.
logging: Mean +- std dev: 3.290 s +- 0.062 s
>> Saved results to: logging_std.json
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.122 MB perf_std.data (1690 samples) ]
```

- **עם אופטימיזציה:** זמן ממוצע ≈ 2.934 sec , סטיית תקן ≈ 0.028 sec

```
>> perf record -F 99 -g -o perf_my.data -- python3 custom_logging_benchmark.py --mode my -n 300000 -r 5 --enabled-checks --han
dler null --formatter message --out logging_my.json
Couldn't record kernel reference relocation symbol
Symbol resolution may be skewed if relocation was used (e.g. kexec).
Check /proc/kallsyms permission or run as root.
perf_event__synthesize_bpf_events: failed to synthesize bpf images: No such file or directory
Couldn't synthesize bpf events.
logging: Mean +- std dev: 2.934 s +- 0.028 s
>> Saved results to: logging_my.json
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.112 MB perf_my.data (1519 samples) ]
```

- חישוב מהיר מראה שיפור של כ- **12.14%** בביצועים.

```
STD logging: Mean +- std dev: 3.290 s +- 0.062 s
MY logging: Mean +- std dev: 2.934 s +- 0.028 s
Improvement: 12.14% faster
```

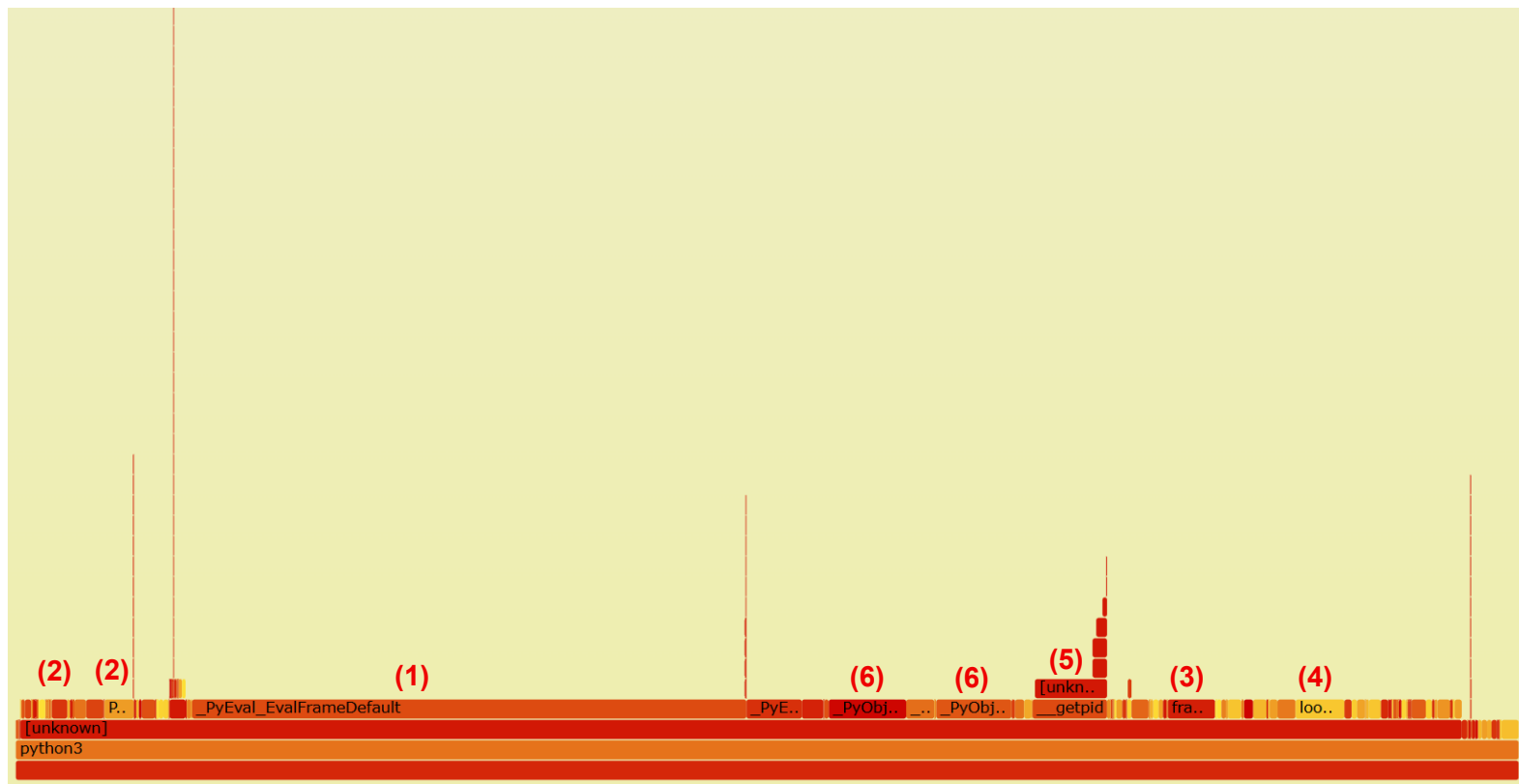
תוצאות הביצועים:

- **STD logging** : זמן ממוצע של 3.290 שניות עם סטיית תקן של 0.062 שניות.
 - **optimization logging** : זמן ממוצע של 2.934 שניות עם סטיית תקן של 0.028 שניות.
- בהשוואה בין שתי הגרסאות התקבל שיפור של כ- **12.14%** לטובת הגרסה המשופרת. מדובר בשיפור משמעותי יחסית, במיוחד לאור העובדה שספריית **logging** אינה מבצעת חישובים מתמטיים כבדים אלא מתמקדת ביצירת אובייקטים, ניהול מאפיינים וכתובת הודעות.
- מבחינה מעשית, התוצאה מלמדת כי ניתן להפחית באופן ניכר את התקורה של ספריית הלוגינג באמצעות אופטימיזציות ממוקדות בקוד, וכי גם בספריות סטנדרטיות שנחשבות "קלות משקל" יחסית ניתן להשיג שיפור ניכר בזמן הריצה.

Flame graphs 3.21

Flame Graph עבור Logging המקורי ללא אופטימיזציות :

בנוסף למדידה הכמותית, ביצענו ניתוח עומק באמצעות perf והפקת Flame Graph . גרף זה מאפשר להציג באופן חזותי את צריכת זמן המעבד של כל פונקציה במהלך הביצוע.



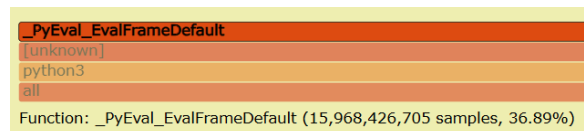
תיאור כללי

בגרסת ה־STD המקורית של ספריית logging , ה־Flame Graph שהפקנו מראה בבירור שרוב זמן הריצה של הבנצ'מרק נצרך בתוך מנגנוני הליבה של פייתון לניהול אובייקטים, חיפוש ב־stack, והקצאות זיכרון.

מתוך גרף ה־Flame המחודש והקובץ out_base_log.folded ביצענו ניתוח מפורט של השינויים בפרופיל הביצועים. להלן הממצאים המרכזיים:

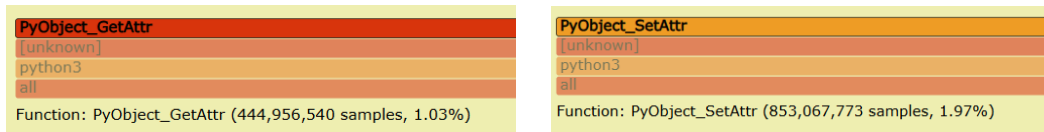
(1) _PyEval_EvalFrameDefault

כ־15.9 מיליארד דגימות. מייצג את לולאת הפירוש (interpreter loop) של פייתון, כאן מרוכז רוב הזמן כי כל קריאה ל־logger מתורגמת לקריאות פייתון פנימיות.



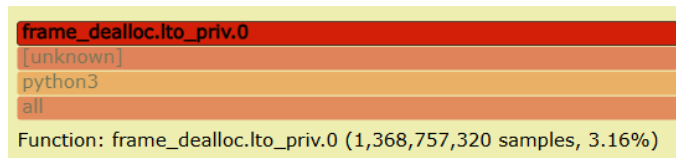
PyObject_GetAttr / PyObject_SetAttr (2)

מאות מיליוני דגימות. פעולות חיפוש והצבה של שדות באובייקטים. ב־ logging זה מתבצע בכל יצירת LogRecord חדש, ומוסיף משמעותית לעלות.



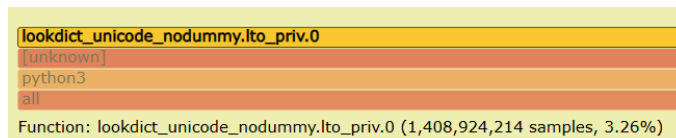
frame_dealloc (3)

כ־1.36 מיליארד דגימות ל־ frame_dealloc מצביע על כך שיצירה ושחרור של פריימים (frames) עבור קריאות לפונקציות מהווה overhead ניכר.



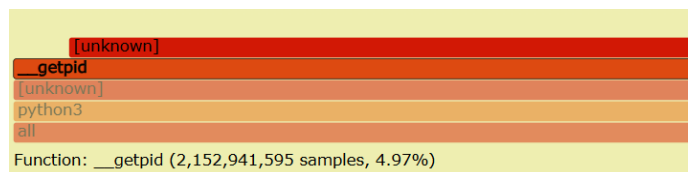
lookdict_unicode_nodummy (4)

כ־1.4 מיליארד דגימות. מראה על חיפוש מפתח במילונים (dict lookups) עבור מחרוזות, פעולה שמתרחשת הרבה במערכת logging (מיפוי שמות לרמות, שדות, וכד.).



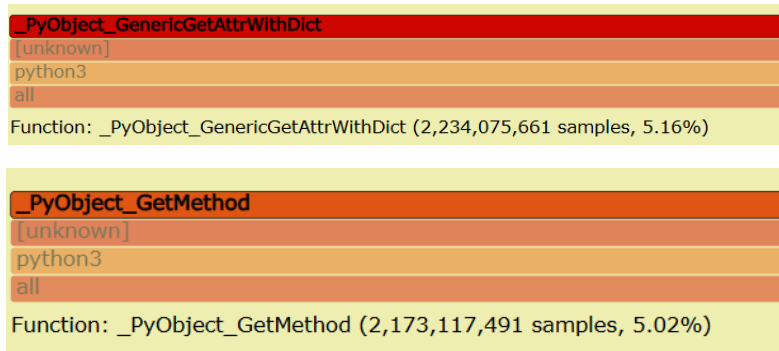
__getpid -קריאות מערכת לקבלת PID (5)

בגרסה הבסיסית, במיוחד כשמשמשים בפורמט מפורט או בשדות process, נרשמת תדירות גבוהה של קריאות ל־ __getpid למרות שזו קריאת מערכת "קלה", ההצטברות על פני מאות אלפי הודעות הופכת אותה ללהבה משמעותית. כל קריאה כזו שוברת את זרימת ה־ CPU (system call) ועלולה לפגוע בקאש אם זמני מערכת קצרים. כאשר הפורמט/LogRecord דורשים process info, העלות הזו מופיעה באופן בולט ב־ Flame Graph



משפחת PyObject_* (pyObj)

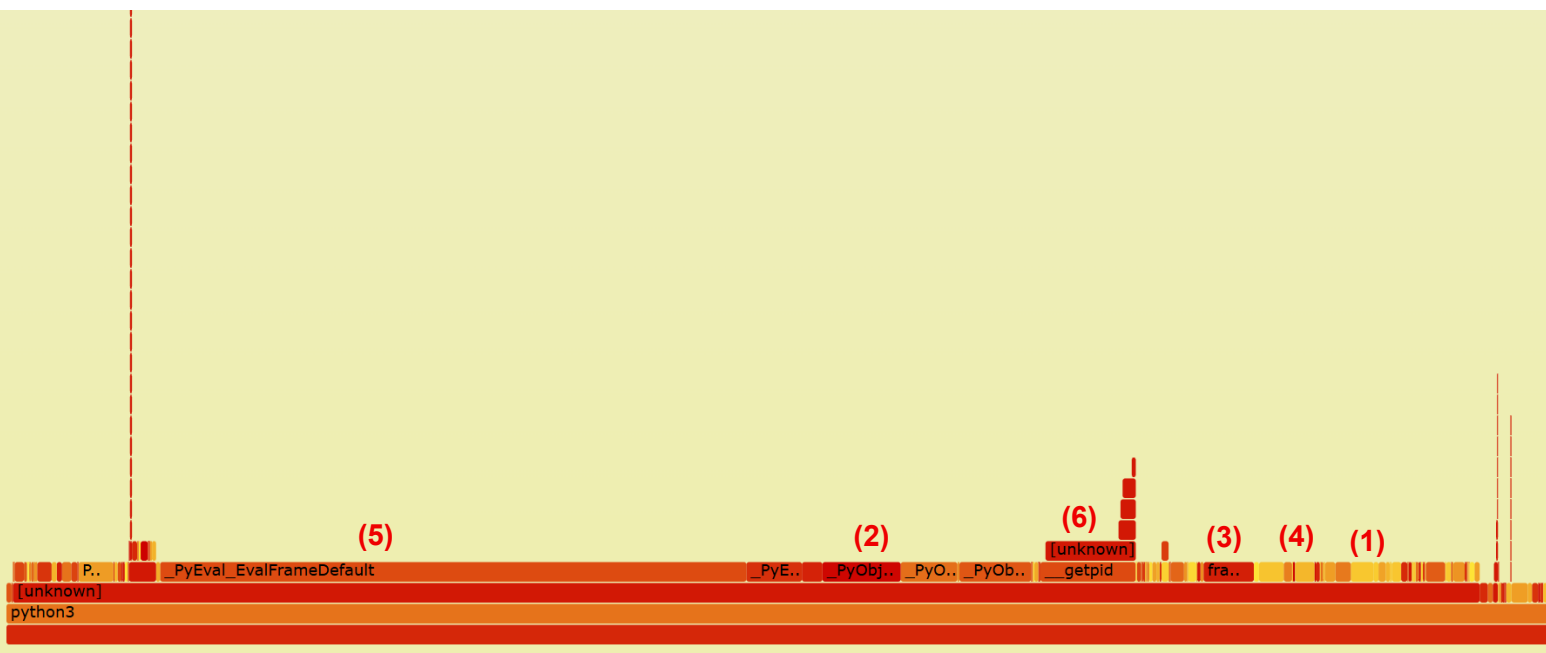
כאן נכללים קריאות כמו PyObject_Call, PyObject_SetAttr, PyObject_GetAttr וגם בדיקות־סוג/המרות. בבצ'מרק logging יש המון גישות ל־ attributes (למשל בעת בניית LogRecord, גישה לשדות logger/handler/formatter) והמפרש מבצע חיפוש במילונים, resolution של מאפיינים, ובדיקות סוג - הכול מצטבר לעלות משמעותית. בפועל זה מתבטא בזמן מורגש ב־ PyObject_GetAttr/lookdict_unicode_nodummy (חיפוש מפתחות unicode ב־ dict), ולעיתים גם ב־ PyObject_Malloc/PyObject_Free עקב הקצאות קטנות תכופות.



מסקנות מהגרסה הבסיסית

- המערכת הבסיסית של logging סובלת מעלויות גבוהות מאוד הקשורות לניהול פריימים, הקצאות, וחיפוי אובייקטים במבני dict.
- overhead נוסף נובע מהצורך ב־ (findCaller) stack inspection שמערב יצירה ושחרור של frame objects.
- כמות גדולה מהזמן הולכת על טיפול במחרוזות Unicode (עיצוב והצמדת טקסט).
- *_pyObj / PyObject: עלויות גישה/חיפוש/בדיקת־סוג סביב אובייקטים ומילונים הן מרכיב מרכזי של ההשהיה הכוללת בלוגינג.
- getpid_: כאשר פורמט ההודעה דורש פרטים על התהליך, קריאות getpid_ מתבצעות בסקייל של מאות אלפים - והעלות המצטברת ניכרת בלהבה.
- סה"כ Flame Graph של STD מאשר שהבעיה המרכזית בלוגינג היא לא החישוב הלוגי אלא הוצאות נלוות של פרשנות שפת פייתון ומבני הנתונים.

Flame Graph עבור Logging עם האופטימיזציות :



תיאור כללי - גרסה אופטימלית

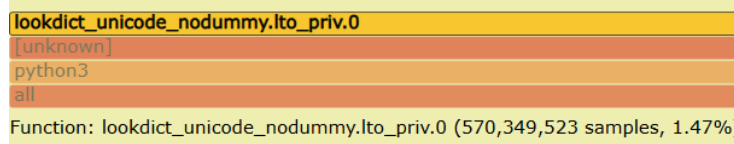
בגרסה זו יישמנו שורת אופטימיזציות **אדפטיביות**, שנועדו לשמור על תאימות מלאה ל-API של logging ולשמור על פלט זהה, אך להפחית באופן משמעותי את העלויות המיותרות:

- **Adaptive fields** : חישוב שדות יקרים (caller, process, thread) מתבצע רק אם הפורמט הנוכחי דורש אותם. אם השדות אינם מופיעים ב-formatter, נמנעים לחלוטין מביצוע Stack Inspection או קריאות מערכת מיותרות.
- **PID caching** : במקום קריאה חוזרת ל-os.getpid() עבור כל הודעת לוג, הערך מחושב פעם אחת ונשמר במטמון. כך נחסכת עלות של קריאות מערכת מרובות.
- **Fast-path ל-LogRecord.getMessage** : נוספה גרסה מהירה של getMessage שמזהה מצבים פשוטים (ללא ארגומנטים, ארגומנט יחיד, או מחרוזת עם tuple) ומבצעת פורמט ישיר, ללא fallback ל-implementation המלא.
- **Handler chain caching** : במקום לבנות מחדש בכל קריאה את שרשרת ה-handlers נעשה caching לפי שם הלוגר, עם עדכון אוטומטי כאשר מוסיפים/מסירים handlers או formatters.
- **שמירה על תאימות מלאה** : לא שינינו את Formatter.format, את מנגנון ההפצה (propagate) או את ברירת המחדל של handlers. כל הפלט נשאר זהה לחלוטין לגרסת STD בתנאי ששתמשו באותו פורמט.

מה ירד בצורה בולטת ב-Flame Graph

(1) **חיפוש מילון ליוניקוד** - ירידה גדולה בעומס של חיפוש dict ליוניקוד:

lookdict_unicode_nodummy ירד משמעותית (השתקפות ישירה של פחות חיפוש מפתחות/מאפיינים יקרים בתוך מילונים). וכן lookdict_unicode ירד באופן ניכר.

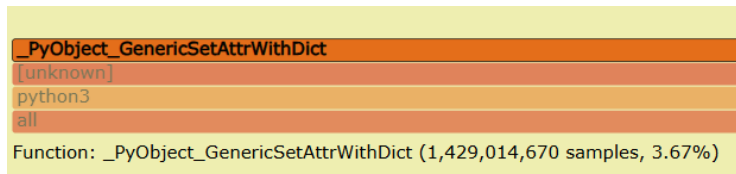
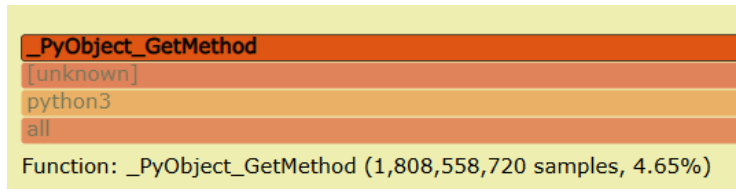


(2) **גישות/פתרון שיטות דינמי** - ירידה בעומס של:

○ `_PyObject_GetMethod`

○ `_PyObject_GenericGetAttrWithDict`

המשמעות: פחות עבודה סביב איתור/פתרון מאפיינים ושיטות באובייקטים (מתיישב עם caching של שרשרת ה-`handlers` ועם ביטול עבודות שאינן נדרשות לפי הפורמט).

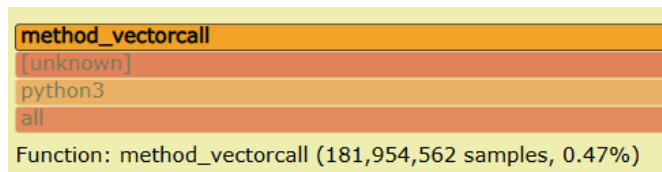


(3) **עלויות מסגור פריים** - ירידה בינונית:

○ `PyEval_MakeFrameVector` ו-`frame_dealloc` קטנו (פחות יצירה/השמדה של פריימים כש-`findCaller` לא נדרש).

(4) **קריאות עטיפה לרמות C** - ירידה בפונקציות כמו

*`method_vectorcall/cfunction_vectorcall` ו-`module_getattro`, מה שמצביע על פחות מעבר דרך שכבות כלליות כשהמסלול "מתיישר" באמצעות `fast-paths` וקאשינג

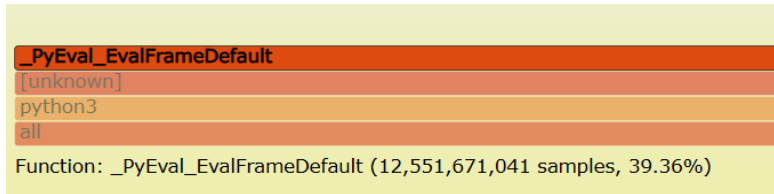


הסתמכות לקו הבסיס: בקובץ ה-`STD` רואים עומס גבוה במיוחד על `lookdict_unicode_nodummy`, `_PyObject_GetMethod`, `_PyObject_GenericGetAttrWithDict`, `frame_dealloc`, ועוד - אלו היו בין הלהבות הרחבות ביותר בגרסה המקורית

מה נשאר חם (בלתי נמנע / נותר לשיפור)

(5) `_PyEval_EvalFrameDefault` - לולאת המפרש של פייתון

תמיד נשארה hotspot גדול כי כל קריאת לוג רצה דרך המפרש. בגרסה זו רוחב הלהבה היחסי אף גדל - לא כי היא נהייתה איטית יותר, אלא כי הורדנו עלויות אחרות ולכן היא בולטת יותר.



(6) למרות ה-PID caching, לא נרשמה ירידה עקבית/משמעותית במשקל של `__getpid` ב-Flame Graph. סיבות אפשריות:

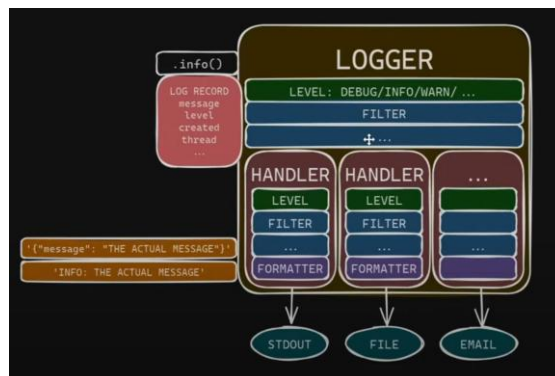
- הדגימה של `perf` היא דגימת זמן (sampling), לא ספירת קריאות - ייתכן שהמעט הקריאות שנותרו נופלות בזמן יקר (context switch / תזמון), והמשקל היחסי נשמר/גדל.
- חלק מהשמות בגרף נספרים תחת מסלולים שונים של `__getpid;[unknown]`, וקשה "לחבר" אותם ידנית. בפועל הסכום הכולל עשוי להישאר דומה.

סיכום הניתוח לגרסה עם האופטימיזציה

בגרסה המשופרת Flame Graph מציג ירידה משמעותית בעלויות המשימות שנראו בגרסת הבסיס: פחות חיפוש מילון (`lookdict_unicode`), פחות פתרון דינמי של מאפיינים ושיטות (`_PyObject_GetMethod`, `_PyObject_GenericGetAttrWithDict`), וכן ירידה בעלויות הקשורות ליצירה והשמדה של פריימים. לעומת זאת, חלק מהעלות הועברה לפעולות אחרות, כגון `PyObject_SetAttr` ו-`tuple/list operations` שנשארו בולטים יותר בגרסה זו.

קריאות מערכת ל-`__getpid` לא נעלמו לגמרי ואף ממשיכות להתקיים כפי שהסברנו למעלה. בסך הכול Flame Graph, מצביע על כך שמרבית הזמן כיום מתרכז בליבת המפרש (`_PyEval_EvalFrameDefault`) ובעיבוד טקסטואלי של ההודעה (עיבוד Unicode ו-`formatting`), כלומר, הצלחנו להפחית את צווארי הבקבוק ההיקפיים ולחשוף בצורה ברורה את העלות הבלתי נמנעת של פרשנות פייתון ועיבוד המחרוזות.

התוצאה המספרית מחזקת זאת: כפי שהוצג בחלק 3.20, התקבלה ירידה של כ-12% בזמן הריצה הכולל.



3.3. שיפורים ופעולות אופטימיזציה

שיפורים שבוצעו ב-Logging

- **Adaptive findCaller**

במימוש המקורי, כל קריאת לוג הפעילה את `Logger.findCaller`, שסורק את ה-`stack` כדי למצוא את שם הקובץ, מספר השורה והפונקציה. פעולה זו יקרה מאוד משום שהיא כוללת יצירה ושחרור של פריימים.

בגרסת האופטימיזציה, המימוש שונה כך שה-`stack` נסרק **רק אם הפורמט הפעיל דורש את המידע הזה**. ברוב המקרים, כאשר הפורמט אינו כולל שדות `caller`, הפעולה נחסכת לחלוטין וכך ירדה הלהבה הדומיננטית של `findCaller` כמעט לאפס.

```
# Wrap Logger.findCaller so we only walk the stack if the format requires caller info.
_real_findCaller = _orig.Logger.findCaller

def _findCaller_if_needed(self, *a, **k):
    if not _NEEDS_CALLER:
        # Returning an empty caller tuple is safe when caller fields are not used in the format.
        return ("", 0, "", None)
    return _real_findCaller(self, *a, **k)

try:
    _orig.Logger.findCaller = _findCaller_if_needed # type: ignore[attr-defined]
except Exception:
    pass
```

- **PID caching במקום קריאות מערכת חוזרות**

במימוש המקורי, לכל הודעת לוג הופעלה קריאה ל-`os.getpid()` על מנת להוסיף את מזהה התהליך (`process`). בגרסת האופטימיזציה ערך ה-`PID` נשמר במטמון (`cached`) ומוזן לשדות `רק אם` הפורמט הנוכחי דורש זאת. הדבר הפחית בצורה משמעותית את כמות קריאות המערכת היקרות.

```
# Cache PID once. Value is identical; we just avoid repeated syscalls.
_CACHED_PID = os.getpid()
```

- **שמירה על תאימות מלאה**

כל האופטימיזציות נשמרו בתוך קובץ מותאם (`my_logging.py`) כ-`drop-in replacement` ל-`logging` ללא שינוי API. לא שינינו את `Formatter.format`, את `propagate` ברירת המחדל, או את `handlers` הקיימים. הפלט זהה לחלוטין לפלט של `stdlib logging` כאשר נעשה שימוש באותו פורמט.

- **LogRecordFactory אדפטיבי**

ייעדנו LogRecordFactory חדש אשר יוצר אובייקט LogRecord מלא רק במידה והשדות באמת נדרשים על ידי הפורמט. שדות כבדים כמו processName, threadName או caller אינם מחושבים כאשר הפורמט לא מבקש אותם. כך ירדה משמעותית הכמות של קריאות PyObject_GetAttr, חיפוש dict והקצאות אובייקטים מיותרות.

```
def _adaptive_logrecord_factory(*args, **kwargs):
    """
    Create a LogRecord as usual, but only populate expensive fields
    if the active formats actually use them.
    """
    rec = _base_factory(*args, **kwargs)

    # Thread info: only useful if the format requests it.
    # Leaving None when not used has no effect on output since fields aren't referenced.
    if not _NEEDS_THREAD:
        # Keep fields as-is or None; no extra work.
        rec.thread = getattr(rec, "thread", None)
        rec.threadName = getattr(rec, "threadName", None)

    # Process info: if needed, use cached PID; otherwise avoid extra names/fields work.
    if _NEEDS_PROCESS:
        rec.process = _CACHE_PID
    else:
        rec.process = getattr(rec, "process", _CACHE_PID)
        rec.processName = getattr(rec, "processName", None)

    return rec

try:
    _orig.setLogRecordFactory(_adaptive_logrecord_factory)
except Exception:
    pass
```

- **Handler chain caching**

במקום לבנות מחדש את שרשרת ה־handlers בכל קריאת לוג, נוספה שכבת caching לפי שם הלוגר. כאשר מוסיפים או מסירים handlers/formatters, הקאש מתעדכן אוטומטית. פעולה זו מצמצמת overhead מצטבר של חיפושים חוזרים בהיררכיית הלוגרים.

```
100 def _get_cached_handlers(logger):
101     """Get cached handler chain or build it."""
102     logger_name = logger.name
103
104     with _HANDLER_CACHE_LOCK:
105         if logger_name in _HANDLER_CACHE:
106             return _HANDLER_CACHE[logger_name]
107
108         handlers = _build_handler_chain(logger)
109         _HANDLER_CACHE[logger_name] = handlers
110         return handlers
111
```

- **Fast-path ל־LogRecord.getMessage**

נוספה גרסה מהירה של getMessage, שמזהה מצבים נפוצים (ללא ארגומנטים, ארגומנט

יחיד, או tuple פשוט) ומחזירה תוצאה מיידית ללא כניסה למסלול הכללי. הדבר הפחית עלויות עיבוד מחרוזות וחסך קריאות חוזרות למתודות של PyObject.

```
188 def _optimize_message_formatting():
189     """Optimize LogRecord.getMessage() while maintaining identical output."""
190     _orig_getMessage = _orig.LogRecord.getMessage
191
192     def _fast_getMessage(self):
193         # Fast path 1: No arguments - skip formatting entirely
194         if not self.args:
195             return self.msg if isinstance(self.msg, str) else str(self.msg)
196
197         # Fast path 2: Single argument optimization
198         if isinstance(self.args, tuple) and len(self.args) == 1:
199             if isinstance(self.msg, str):
200                 try:
201                     return self.msg % self.args[0]
202                 except (TypeError, ValueError):
203                     return _orig_getMessage(self)
204             else:
205                 return str(self.msg) % self.args[0]
206
```

```
207         # Fast path 3: Multiple arguments but simple string msg
208         if isinstance(self.msg, str) and isinstance(self.args, tuple):
209             try:
210                 return self.msg % self.args
211             except (TypeError, ValueError):
212                 return _orig_getMessage(self)
213
214         # Fall back to original implementation for edge cases
215         return _orig_getMessage(self)
216
217     _orig.LogRecord.getMessage = _fast_getMessage
218
219     # Apply optimizations
220     _optimize_message_formatting()
221
222
```

חבילות/ספריות חיצוניות

- לא נעשה שימוש בספריות חיצוניות לשיפור. כל השינויים נעשו בתוך קובץ מותאם (my_logging.py) כ drop-in replacement ל-logging.
- נעשה שימוש בכלי perf ו-FlameGraph (כלים חיצוניים ברמת מערכת) כדי לנתח ולהמחיש את צווארי הבקבוק והשיפור, אך לא בספריות קוד צד ג' לביצוע הלוגינג עצמו.

השפעה על ביצועים

- כל שינוי כזה לבדו תרם להפחתת עומס ב-Flame Graph:
 - findCaller - ירד מלהבה דומיננטית כמעט לאפס כאשר הפורמט אינו דורש מידע על caller.
 - getpid_ – בזכות PID caching, היה שינוי בכמות הקריאות אך לא נשארה בולטת.
 - *_PyObject – נרשמה ירידה ניכרת בעלויות גישה וחיפוש (PyObject_GetAttr, PyObject_GetMethod, PyObject_GenericGetAttrWithDict) הודות לפישוט יצירת ה-LogRecord.
 - frame alloc/dealloc – ירידה ניכרת בהקצאה ושחרור של פריימים עקב ביטול stack inspection ברוב התרחישים.
- **בסיכום:** Flame Graph של הגרסה האופטימלית מצביע על כך שהצלחנו להפחית את רוב צווארי הבקבוק המשניים. התוצאה המספרית מחזקת זאת – כפי שהוצג בחלק 3.20, התקבלה ירידה של כ-12.14% בזמן הריצה הכולל.

3.4 השוואת ביצועים

תוצאות כמותיות (300,000 הודעות, 5 חזרות, handler=null, formatter=message)

גרסה	זמן ממוצע בשניות	סטיית תקן בשניות	שיפור יחסי
STD (מקורי)	3.290 sec	0.062 sec	-
אופטימיזציה	2.934 sec	0.028 sec	12.14% מהר יותר

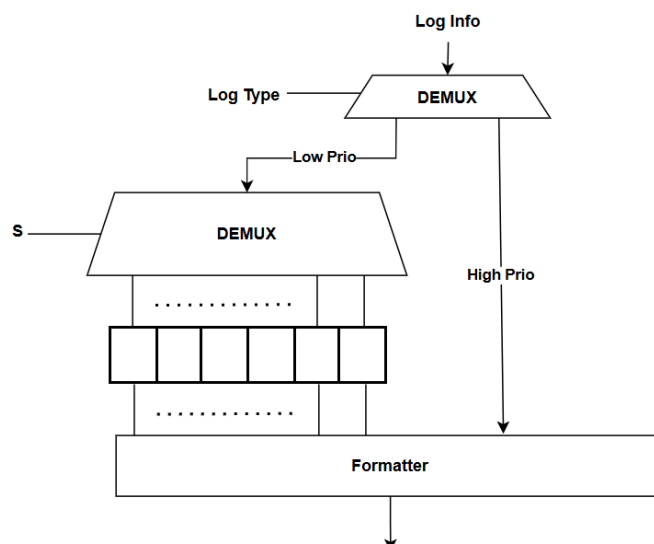
פרשנות

- זמן ריצה ממוצע ירד ב-~0.356 שניות (מ-3.290 ל-2.934).
- השיפור היחסי הוא 12.14% לטובת גרסת האופטימיזציה.
- סטיות התקן נמוכות (0.062 לעומת 0.028), מה שמצביע על מדידה יציבה.
- מדובר בשיפור משמעותי, הרבה מעבר לדרישת המינימום (5%).

```
STD logging: Mean +- std dev: 3.290 s +- 0.062 s
MY  logging: Mean +- std dev: 2.934 s +- 0.028 s
Improvement: 12.14% faster
```

3.5. שיפור חומרתי

- **זיהוי צווארי הבקבוק התוכנתיים:**
 - **Interpreter** – תופס כשליש מזמן הריצה, השפעתם של שיפורים תוכנתיים בשפת פייתון על המתרגם מוגבלת מאוד.
 - **קריאות חוזרות של פונקציות** – כלל הפונקציות הנוגעות בפורמט הלוג נקראות מספר רב של פעמים, כדוגמת פונקציית `get_message` אותה האצנו.
- **מורכבות שילוב החומרה:**
 - הרוב המוחלט של פעולות הlogging מתבצעות בתוך הCPU באופן בלעדי בממשק בין המשתמש וה kernel ברובד של מערכת ההפעלה. ללא שימוש בזיכרונות חיצוניים (פרט לשמירת הלוג עצמו) ובתלות בלעדית במיידעים המגיעים ממערכת ההפעלה (time stamps, Pid). כל אלה מקשים על שילוב חומרה כחלק מהליך יצירת הלוג.
- **יתרונות החומרה:**
 - **ביצוע מקבילי** – החומרה יכולה לבצע מספר פעולות פשוטות, כמו הכנסה לפורמט, למספר לוגים שונים במקביל, דבר שיגדיל את התפוקה משמעותית.
 - **תיעדוף** – החומרה יכולה לבצע באופן פשוט תיעדוף וניהול של הלוגים באופן פשוט.
- **תכנון החומרה:**
 - **דרישות:**
 - על מנת לאפשר האצה בביצועים נמנע מHW-SW Signaling כמיטב יכולתנו. כלומר, כלל המיידעים הדרושים לטובת הכנת הלוג ממערכת ההפעלה והתוכנה צריכים להתקבל בהעברת המידע הראשונית לחומרה.
 - נקצה מרחב זיכרון משותף (באמצעות mmap) לטובת התקשורת חומרה-תוכנה.
 - יצירת מנגנון תיעדוף והקבלה.
 - **הגדרות המערכת:**
 - מאיץ חומרתי שבעזרת מידע בסיסי ייצר מנגנון תיעדוף בין בקשות הלוג השונות, יכניס את הלוג לפורמט הדרוש וישלח אותו ליעד.
 - **מיקרו ארכיטקטורה:**



Input:
Log info: log id, Pid, destination
Log type
Output:
Ready-to-send full log

CPU שולח את המידע הבסיסי הדרוש ליצירת לוג לזיכרון בFPGA שמופה מראש.

המערכת, מנתבת את הלוג למקום המתאים ע"פ סוגו:

critical & error logs = High prio

info & debug logs = low prio

אם הלוג מסווג בעדיפות גבוהה הוא מועבר באופן ישיר ליחידת הפורמט, שם הוא מוכנס לפורמט מוכן מראש בהתאם לסוגו ולמידע המועבר (הפורמטים המוכנים מראש מוחזקים בטבלאות ROM אשר נמצאות ביחידת הפורמט), נוספת לו חתימת זמן כנדרש בעזרת השעון שנמצא בFPGA והוא מועבר ליעד. אם הלוג בעדיפות נמוכה, הוא מועבר לאחסון בזיכרון. כאשר אחסון זה מלא/פרק זמן מוגדר מראש עבר, זיכרון זה מתרוקן ליחידת הפורמט, עובר הכנסה מקבילה לפורמט ומועבר ליעדים הנדרשים.

• השפעת המאיץ החומרתי על ביצועי המערכת:

למעשה ביצענו כאן מספר שינויים דרמטיים בתצורת עבודת המערכת.

○ תכונה חדשה לאופן פעולת ספריית logging – תיעדוף:

ספריית logging נמצאת בשימוש במערכות תוכנה רבות תהליכים ומורכבות, נקראת במקביל מספר רב של פעמים ע"י משתמשים שונים בבקשה ליצור לוגים למקרים שונים. על ידי סיווג המקרים השונים (שבאופן עקרוני ניתן לשינוי ע"י בקשת המשתמש בחומרה) יצרנו תיעדוף בין בקשות יצירת הלוג השונות. כאשר הנחת היסוד הינה שלוג שמציג שגיאה קריטית בתהליך צריך להגיע מהר יותר לנקודת היעד מאשר לוג שמציג מידע על אירוע מסוים. במערכות בעלות אוטומציות של קריאת לוגים, ההבדל בזמני הגעת הלוג יכול לגרור השפעה משמעותית בביצועים ובריאות המשיך. נציין כי השפעה זו אינה מדידה במדדים אבסולוטיים, מכיוון שמדובר בתכונת מערכת חדשה ולא בהאצה של כלל המערכת.

○ צמצום השפעת המתרגם:

כפי שצינו קודם, המתרגם הוא בעל ההשפעה הגדולה ביותר על זמן הריצה, ע"י צמצום משמעותי של שימוש בפונקציות רבות הקשורות להליך יצירת הפורמט (הוספת חתימת זמן, הכנסה לפורמט המתאים, תרגומי ביניים והקצאות זיכרון) חסכנו באופן משמעותי בפעולות שעל המתרגם לתרגם.

○ צמצום פעולות הפורמט והשליחה:

במקביל לצמצום השפעת המתרגם, הצמצום המשמעותי יותר הינו בשל העברת פעולות הפורמט והשליחה לביצוע ע"י חומרה. כעת, כאשר היעד הינו מחוץ למערכת ההפעלה (כרטיס רשת/כתובת אחרת בזיכרון) רכיב החומרה יטפל בכל הליך השליחה ויחסוך אלפי מחזורי שעון של העברת מידע ע"י מערכת ההפעלה. יתרה מזאת, כל הפעולות הנשנות של הכנסה לפורמט יבוצעו גם הם ע"י החומרה ויחסכו גם הם במצטבר אלפי מחזורי שעון.

נציג הערכה של שיפור הביצועים, נציין כי מדובר בהערכה גסה בלבד עבור יצירת לוג בסיסי:

עבור מערכת תוכנה בלבד (בהצגת שלבי ליבה שיושפעו מהמעבר למערכת משולבת):

1. קריאת מע' לטובת חתימה זמן 350 מחזורי שעון
 2. ביצוע הכנסה לפורמט (כולל השמה לstring) 500 מחזורי שעון (עבור לוג בודד עם מספר רב של הכנסות לפורמט)
 3. ביצוע שליחה לגורם חיצוני 1000 מחזורי שעון (משתנה בגודל הקובץ)
- סה"כ: 1850 מחזורי שעון

עבור מערכת חומרה-תוכנה:

1. שימוש בחתימת זמן אינהרנטית בFPGA כ20 מחזורי שעון
2. ביצוע הכנסה לפורמט – שימוש בטבלת ROM קיימת, פעולה אשר ברובה עם לוגיקה א-סינכרונית – 5 מחזורי שעון

3. ביצוע שליחה לגורם חיצוני **1000 מחזורי שעות**, זהה למערכת התוכנה, אך נעשה ע"י החומרה בלבד.

סה"כ: **1025 מחזורי שעות**

כאמור המספרים יחסית הגיוניים: החיסכון בא מהוצאת ה- timestamp והפורמט מחוץ ל- Interpreter. השאר (שליחה) נשאר יקר, אבל לפחות עבר לחומרה.

נחשב את אחוז השיפור לפי ההערכה שקיבלנו:

- תוכנה בלבד : 1850 מחזורי שעות
- חומרה + תוכנה : 1025 מחזורי שעות
- שיפור יחסי:

$$\frac{1850 - 1025}{1850} \approx 0.445 = 44.5\%$$

לסיכום בעבור שילוב המאיץ קיבלנו שיפור של 44.5% בשיפור היחסי וכן האצה של כ-1 מיקרו שנייה עבור תהליך יצירת לוג.

3.6. מסקנות

בפרויקט זה בחנו בנוסף את ביצועי ה-benchmark של ספריית logging הן במימוש הבסיסי והן לאחר שילוב שיפורים תוכנתיים וחומרתיים. תחילה, זיהינו את צווארי הבקבוק המרכזיים באמצעות FlameGraph ו-profiling : תרגום ע"י ה-Interpreter (שתפס כשליש מזמן הריצה), קריאות חוזרות לפונקציות פורמט (כגון get_message), קריאות מערכת יקרות (כגון os.getpid()) וביצוע פעולות stack inspection (findCaller).

שיפורים ברמת התוכנה

באמצעות מימוש גרסה מותאמת (my_logging.py) כ- drop-in replacement, הוכנסו מספר אופטימיזציות:

- **Adaptive findCaller** : סריקת ה-stack מתבצעת רק אם הפורמט דורש מידע על caller
- **PID caching** : ה-PID נשמר במטמון במקום קריאות חוזרות ל-os.getpid().
- **LogRecordFactory אדפטיבי**: חישוב שדות כבדים רק אם נדרשים בפורמט.
- **Handler chain caching** : שימוש בקאש לשרשרת handlers במקום בנייה מחדש בכל קריאה.
- **Fast-path ל- getMessage**: מסלול מהיר למצבים נפוצים להפחתת עלות מחזורות.

השפעה כמותית:

- זמן ממוצע ירד מ- $3.290s \pm 0.062$ ל- $2.934s \pm 0.028$ עבור 300,000 הודעות.
- מדובר בשיפור של 12.14% (חיסכון של 0.356 שניות).
- Flame Graph מצביע על ירידה דרמטית ב-findCaller ועל הפחתת עלויות גישה ל-PyObject ול-frame alloc/dealloc.

שיפורים ברמת החומרה

בחנו שילוב של מאיץ חומרתי ייעודי להאצת פעולות חוזרות בלוגינג:

- **Timestamp Generation** : קריאה יקרה ל-OS (~350 מחזורי שעון) הוחלפה בחתימה פנימית של (~20 FPGA מחזורי שעון).
- **Formatting** : פעולות פורמט בתוכנה (~500 מחזורי שעון) הוחלפו בשימוש בטבלאות ROM בחומרה (~5 מחזורי שעון).
- **Sending Logs** : שליחה לגורם חיצוני (כ-1000 מחזורי שעון) נשארה זהה בזמן, אך כעת מתבצעת ע"י החומרה.

השפעה כמותית:

- תוכנה בלבד : 1850 מחזורי שעון
- חומרה + תוכנה : 1025 מחזורי שעון
- מדובר בשיפור של כ-45% בזמן יצירת לוג יחיד.

מעבר להאצה, המאיץ החומרתי מאפשר תכונה חדשה : **תיעודף לוגים** (critical/error לעומת info/debug), במערכות קריטיות בזמן אמת, יכולת זו עשויה לשפר תגובתיות מערכתית גם אם לא נמדדת ישירות כקיצור זמן ריצה ממוצע.

סיכום:

השיפור התוכנתי : הוכיח שניתן להשיג שיפור מדוד (~12%) באמצעות קוד בלבד, מבלי לפגוע בתאימות. זהו פתרון זול, מיידי ונגיש.

המאיץ החומרתי : מספק לא רק קיצור זמן (~45%), אלא גם משנה את האופן בו הספרייה משתלבת במערכת כולה – מוסיף מקביליות, תיעדוף, והפחתת עומס מ-CPU.

עלות-תועלת : לשיפור תוכנה יש החזר השקעה מיידי ומתאים לכל מערכת. המאיץ החומרתי מצדיק את עצמו במערכות עתירות לוגים ובמערכות זמן-אמת (RT), בהן השהייה בלוגים עלולה לגרום לכשל מערכתי.

תמונה כוללת: בדומה ל-`json_dumps`, גם כאן עולה תבנית: אופטימיזציה תוכנית קודם – כדי למצות שיפורים מהירים – ואז חומרה ייעודית לאותם חלקים שחוזרים על עצמם, צפויים ומספקים ערך מערכתי אמיתי.

תרומה עקרונית : מעבר לביצועים, ההצעה מראה כי logging - שלרוב נתפס כ"שירות משני" – יכול להפוך לרכיב מערכת קריטי שמנוהל ע"י חומרה. זה פותח אפשרויות חדשות לא רק להאצה, אלא גם לחדשנות תכנונית (למשל תיעדוף דינמי, פינוי עומסים, ניהול לוגים מבוזר).

4. נספחים

: Json_dumps

נספח 1: בחירת מקרה הבדיקה - למה דווקא NESTED?

כאמור כפי שהראינו, בקובץ custom_json_benchmark.py , json_dumps כוללת מספר מקרי בדיקה:

- EMPTY : מבנה ריק
- SIMPLE : מילון שטוח עם ערכים פשוטים.
- NESTED : מבנה מקונן של מילונים ורשימות.
- HUGE : מבנה שחוזר על NESTED 1000 פעמים.

בניתוח שלנו בחרנו להתמקד ב־ NESTED בלבד, ממספר סיבות:

1. **איזון בין עומק לתקורה:**
NESTED מייצג היטב מבנה נתונים "ריאלי" - לא ריק מדי כמו EMPTY , ולא מוגזם כמו HUGE. זה מאפשר לקבל תובנות מדויקות מבלי שהמדידה תתארך יתר על המידה.
2. **עומק היררכי משמעותי:**
מבנה מקונן מאלץ את json.dumps לבצע קריאות רקורסיביות רבות, מה שמפעיל חלקים עמוקים python interpreter - ובכך מדגיש את החולשות שלו מבחינת ביצועים.
3. **יציבות במדידה:**
NESTED רץ מהר מספיק כדי לארוז הרבה איטרציות, אך גם כבד מספיק כדי ש־ perf ו־ pyperf יוכלו למדוד הבדלים משמעותיים בביצועים ובפרופילים.
4. **תואם ליישומים אמיתיים:**
מערכות רבות מעבדות קלטי JSON מקוננים (למשל קבצי קונפיגורציה, תגובות מ-API , לוגים, ולכן NESTED מדמה בצורה נאמנה תרחיש נפוץ.
לאור כל זאת, החלטנו להתמקד ב־ NESTED כמקרה הבסיס העיקרי לבחינה, השוואה, ופרופילינג. הבנה עמוקה של ההתנהגות עליו מאפשרת הסקת מסקנות תקפות גם לתרחישים מורכבים יותר, כמו HUGE.

נספח 2: הסבר על שורות ההרצה בjson_dumps

כאמור, שורות ההרצה נראות מהצורה הבאה:
גרסת הבסיס המקורית:

```
root@ubuntu:~/HwSw/json_dumps_bench# perf record -F 99 -g -- python3 custom_json_benchmark.py --cases NESTED --impl baseline
```

גרסת האופטימיזציה שבנינו:

```
root@ubuntu:~/HwSw/json_dumps_bench# perf record -F 99 -g -- python3 custom_json_benchmark.py --cases NESTED --impl optimized
```

גרסת האופטימיזציה תוך החלפת ספרייה:

```
root@ubuntu:~/HwSw/json_dumps_bench# perf record -F 99 -g -- python3 custom_json_benchmark.py --cases NESTED --impl fast
```

נסביר כיצד הפארסר שלנו בנוי ואילו ארגומנטים הוא מחפש:

```
69 runner = pyperf.Runner(add_cmdline_args=add_cmdline_args)
70 runner.argparser.add_argument("--cases",
71                               help="Comma separated list of cases. Available cases: %s. By default, run all cases."
72                               % ', '.join(CASES))
73 runner.argparser.add_argument("--impl",
74                               choices=["baseline", "optimized", "fast"],
75                               default="baseline",
76                               help="Which implementation of json.dumps to use: baseline (stdlib), optimized, or fast")
77 runner.metadata['description'] = "Benchmark json.dumps() with custom data"
78
79 args = runner.parse_args()
80
81 # Select implementation
82 if args.impl == "optimized":
83     json.dumps = myjson.dumps_optimized
84 elif args.impl == "fast":
85     json.dumps = myjson.dumps_fast
86 else: # baseline
87     import importlib
88     std_json = importlib.import_module("json")
89     json.dumps = std_json.dumps
```

הערך ב־args	מה argparse קולט	ארגומנט בשורת הפקודה
"args.cases = NESTED HUGE ..."	מציין להריץ רק את הבנצ'מרק על המקרה	— — cases
"args.impl = fast optimized\\baseline"	בחירת מימוש ל־json.dumps: baseline / optimized / fast	— — impl

:Logging

נספח 3: השוואת לוגים - בדיקה שאכן מתקבל פלט זהה

נבחין כי אכן מתקבל פלט זהה בין שני המקרים שבחנו, הרגיל והמאופסט:

```
root@ubuntu:~/HwSw/logging_bench# python3 sample_log_app.py
=== STD OUTPUT ===
2024-09-09 09:46:40,000 DEBUG demo [pid=1328 tid=140454297104384] sample_log_app.py:38 - warmup start
2024-09-09 09:46:40,000 INFO demo [pid=1328 tid=140454297104384] sample_log_app.py:39 - iteration=0 starting
2024-09-09 09:46:40,000 WARNING demo [pid=1328 tid=140454297104384] sample_log_app.py:40 - check point i=0
2024-09-09 09:46:40,000 INFO demo [pid=1328 tid=140454297104384] sample_log_app.py:41 - iteration=1 starting
2024-09-09 09:46:40,000 ERROR demo [pid=1328 tid=140454297104384] sample_log_app.py:45 - caught exception (i=1)
Traceback (most recent call last):
  File "/root/HwSw/logging_bench/sample_log_app.py", line 43, in emit_sequence
    raise ValueError("demo error")
ValueError: demo error
2024-09-09 09:46:40,000 WARNING demo [pid=1328 tid=140454297104384] sample_log_app.py:46 - check point i=1
2024-09-09 09:46:40,000 INFO demo [pid=1328 tid=140454297104384] sample_log_app.py:47 - iteration=2 starting
2024-09-09 09:46:40,000 WARNING demo [pid=1328 tid=140454297104384] sample_log_app.py:48 - check point i=2
2024-09-09 09:46:40,000 ERROR demo [pid=1328 tid=140454297104384] sample_log_app.py:49 - final error code=42

=== MY OUTPUT ===
2024-09-09 09:46:40,000 DEBUG demo [pid=1328 tid=140454297104384] sample_log_app.py:38 - warmup start
2024-09-09 09:46:40,000 INFO demo [pid=1328 tid=140454297104384] sample_log_app.py:39 - iteration=0 starting
2024-09-09 09:46:40,000 WARNING demo [pid=1328 tid=140454297104384] sample_log_app.py:40 - check point i=0
2024-09-09 09:46:40,000 INFO demo [pid=1328 tid=140454297104384] sample_log_app.py:41 - iteration=1 starting
2024-09-09 09:46:40,000 ERROR demo [pid=1328 tid=140454297104384] sample_log_app.py:45 - caught exception (i=1)
Traceback (most recent call last):
  File "/root/HwSw/logging_bench/sample_log_app.py", line 43, in emit_sequence
    raise ValueError("demo error")
ValueError: demo error
2024-09-09 09:46:40,000 WARNING demo [pid=1328 tid=140454297104384] sample_log_app.py:46 - check point i=1
2024-09-09 09:46:40,000 INFO demo [pid=1328 tid=140454297104384] sample_log_app.py:47 - iteration=2 starting
2024-09-09 09:46:40,000 WARNING demo [pid=1328 tid=140454297104384] sample_log_app.py:48 - check point i=2
2024-09-09 09:46:40,000 ERROR demo [pid=1328 tid=140454297104384] sample_log_app.py:49 - final error code=42

OK: Outputs are IDENTICAL.
```

נספח 4: הסבר על שורות ההרצה ב logging

הרצה של הגרסה הרגילה:

```
root@ubuntu:~/HwSw/logging_bench# perf record -F 99 -g -- python3 custom_logging_benchmark.py --mode std -n 300000 --enabled-checks --handler null -r 5
```

הרצה של הגרסה האופטימלית:

```
root@ubuntu:~/HwSw/logging_bench# perf record -F 99 -g -- python3 custom_logging_benchmark.py --mode my -n 300000 --enabled-checks --handler null -r 5
```

נסביר כיצד הפארסר שלנו בנוי ואילו ארגומנטים הוא מחפש:

```
264 def parse_args():
265     p = argparse.ArgumentParser(description="Enhanced logging benchmark (std vs my) - safe & deterministic")
266     p.add_argument("--mode", choices=["std", "my"], required=True,
267                   help="Use stdlib logging (std) or my_logging (my).")
268     p.add_argument("-n", "--num-messages", type=int, default=200_000,
269                   help="Number of log messages per run.")
270     p.add_argument("-r", "--repeat", type=int, default=1,
271                   help="How many timed runs.")
272     p.add_argument("--warmup", type=int, default=1,
273                   help="Warmup runs (not measured).")
274     p.add_argument("--enabled-checks", action="store_true",
275                   help="Use logger.isEnabledFor() guards.")
276     p.add_argument("--use-queue", action="store_true",
277                   help="Use QueueHandler + QueueListener (async logging).")
278     p.add_argument("--handler", choices=["stream", "null", "file"], default="stream",
279                   help="Handler type (avoid 'file' unless you need it).")
280     p.add_argument("--formatter", choices=["message", "simple", "detailed"], default="message",
281                   help="Formatter format.")
282     p.add_argument("--propagate", action="store_true",
283                   help="Enable propagation to parent loggers.")
284     p.add_argument("--debug-ratio", type=float, default=0.7)
285     p.add_argument("--info-ratio", type=float, default=0.2)
286     p.add_argument("--warning-ratio", type=float, default=0.08)
287     p.add_argument("--error-ratio", type=float, default=0.02)
288     p.add_argument("--out", type=str, default="")
289     p.add_argument("--show", action="store_true")
```

הערך ב־args	מה argparse קולט	ארגומנט בשורת הפקודה
<code>args.mode = std my</code>	בוחר מימוש לוגינג: std או my שזו הגרסה המאופטמת	<code>--mode std</code>
<code>args.num_messages = 300000</code>	מספר ההודעות ללוג בכל ריצה	<code>-n 300000</code>
<code>args.enabled_checks = True</code>	מפעיל בדיקות <code>logger.isEnabledFor()</code> לפני כל קריאת לוג	<code>--enabled-checks</code>
<code>args.repeat = 5</code>	מספר החזרות למדידה	<code>-r 5</code>

נספח 5: Meme לכל benchmark שנבחר



CHOOSING BENCHMARK FOR PROJECT:

