

Udvidelse af Benz med eksternt domænespecifikt sprog

Jens Peter Svensson

*Datalogisk Institut, Københavns Universitet
Universitetsparken 1, 2100 København Ø*

Indhold

1	Introduktion	4
2	Benchmarking	5
2.1	Grundelementer i en benchmarktest	5
2.2	Eksisterende målingsmetoder i Benz	6
2.3	Forslag til nye målingsmetoder i Benz	7
2.4	Faldgruber	8
2.5	Visualisering	10
2.6	Opsamling	12
3	Domænespecifikke sprog (DSL)	12
3.1	Internt DSL	12
3.2	Eksternt DSL	13
3.3	Opsamling	13
4	Refaktorering	14
4.1	Opsplitning i filer	14
4.2	Opdeling af <code>case</code> -klassen	14
4.3	Udtræk af gentaget kode	15
4.4	Opsamling	16
5	Portabilitet	16
5.1	Teknikken	17
5.2	Udførelse	17
5.3	Andre teknikker	18
5.4	Opsamling	18
6	Forøgelse af brugbarheden	19
6.1	Simplere at tilføje nye drivere	19
6.2	Simplere at tilføje nye visualizere	19
6.3	Lettere at ændre driver	19
6.4	Brugen af visualizer	20
6.5	For Windows-brugere	20
6.6	Brugerguider	20
6.7	Opsamling	21
7	Krav til BenzDSL	21
7.1	Designet af BenzDSL	21
7.2	Krav til BenzDSL	21
7.3	Sprogets syntaks	22
7.4	Sprogets design	22
7.5	Opsamling	24
8	Implementeringen af BenzDSL	25
8.1	Fortolkeren	25
8.2	Muligheder med BenzDSL	26

8.3	Konfigurationsfil	27
8.4	Specifikation af BenzDSL	27
8.5	Erklæring af kompilator	29
8.6	Erklæring af strukturer	29
8.7	Erklæring af datatype	31
8.8	Erklæring af testtilfælde	31
8.9	Erklæring af visualisering	32
8.10	Erklæring af driver	32
8.11	Erklæring af benchmarktest	33
8.12	Erklæring af output	34
8.13	Et BenzDSL-program	34
8.14	Opsamling	35
9	Test af Benz DSL	35
9.1	Push_back-test	35
9.2	Midtvejsindsættelsetest	36
9.3	At-test	37
9.4	Unittest	37
9.5	Sorteringsalgoritmetæller	37
9.6	Opsamling	38
10	Konklusion	38
A	DSL-testfiler	41
A.1	Tidsmåling af push_back:	41
A.2	Push_back waste måling	42
A.3	Tidsmåling af midtvejsindsættelse	44
A.4	Kompilatoroptimeringer ved midtvejs indsættelse	45
A.5	Tidsmåling af at	46
A.6	Unittest af vector implementeringer	47
A.7	Sorteringsalgoritme funktionstæller	48
B	Testresultater	49
B.1	Resultat af tidsmåling for push_back	49
B.2	Resultat af push_back waste måling	50
B.3	Resultat af tidsmåling for indsættelse i midten	53
B.4	Resultat af compiler optimeringer	54
B.5	Resultat af kald til at med $n - 1$ på en n vector	55
B.6	Resultat af unittest af vectorer	56
B.7	Resultat af sorteringsalgoritme funktionstæller testen	60

Resumé. Denne rapport beskriver tilblivelsen af fortolkeren BenzDSL, der kan bruges at generere programmer henblik på at foretage benchmarking ved hjælp af applikationen Benz. Programmerne bliver genereret fra det deklarative sprog, der fortolkes af BenzDSL. I rapporten beskrives endvidere, hvordan Benz er gjort mere brugervenligt for både udviklere og brugere.

1. Introduktion

Applikationen Benz giver udviklere mulighed for at skrive benchmarktest i en kombination af C++ og Python-kode. C++ bruges til den del af koden, der foretager benchmarkmålinger (målingskode) for eksempel måling af tid, og den del, der skal måles (testkode) for eksempel indsættelse af elementer i en datastruktur. Python-koden bruges til at kombinere målingskode med testkode for den enkelte benchmarktest og til at erklære, hvordan resultatet af benchmarktesten skal visualiseres.

Formålet med dette projekt var at designe og implementere et domænespecifikt sprog (DSL)¹ til Benz [1] med henblik på at gøre det lettere og hurtigere at implementere benchmarktest. Samtidig var en tilegnelse af viden om DSL og benchmarking en forudsætning for at kunne udføre projektet. Nærværende rapport er det skriftlige resultat af projektet, og applikationen BenzDSL er den fortolker, der er udviklet under projektetforløbet.

I rapporten beskrives benchmarking og domænespecifikke sprog samt det implementerede DSL (BenzDSL); dette bistået af eksempler på konkret brug af BenzDSL.

Efterfølgende er en forklaring af begreber, der bliver brugt i denne rapport:

- **Struktur:** Den kode der skal testes, som regel C++-koden for en klasse eller funktion, f.eks. `std::vector`. Navnet struktur er valgt, fordi dette ikke indikerer hverken en klasse eller en funktion. Dette valg kan skabe problemer for læsere med kendskab til C++, da det ligger op ad C++-konstruktionen ”`struct`”, men på intet tidspunkt i denne rapport vil der blive refereret til denne konstruktion.
- **Test:** Den C++-kode, der foretager en konkret handling med eller på en struktur. Dette kan f.eks. være at sætte et element ind i en datastruktur.
- **Driver:** Det Python-modul, der genererer den kode (som regel i form af C++-kode), der bruges til at foretage målinger af testkoden.
- **Visualizer:** Det Python-modul, der givet resultatet af en benchmarktest foretager en visualisering af dette. Dette kan f.eks. være en graf over tiden, det tager at foretage `push_back` for en instans af `std::vector` med n elementer. Benchmarktesten kan da gentages med forskellige

¹ Domænespecifikt sprog: Et sprog designet til at løse opgaver inden for et specifikt domæne.

værdier n for at danne en graf, hvor hvert punkt svarer til et testtilfælde.

- **BenzDSL:** Bruges om både sprog og fortolkeren og er en sammenkrivning af navnet på det oprindelige program Benz og DSL.

For at implementere BenzDSL var det nødvendigt først at få indsigt i, hvordan Benz virkede. Før dette projekt var den eneste erfaring med Benz, at det ikke kunne afvikles under Windows, hvilket minimerer brugerpotentialet for Benz. Derfor blev det besluttet at gøre det muligt at afvikle Benz under Windows, hvilket indebærer erhvervelse af viden om Benz' opbygning og virkemåde.

Her følger en beskrivelse af de enkelte afsnit i nærværende rapport:

- **Afsnit 2:** En generel beskrivelse af benchmarks, efterfulgt af en beskrivelse af benchmarkmetoder i Benz og forslag til nye.
- **Afsnit 3:** En generel beskrivelse af domænespecifikke sprog.
- **Afsnit 4:** En beskrivelse af den refaktorering af Benz' kildekode, der var nødvendig for at sikre overblik over og genbrug af koden.
- **Afsnit 5:** En beskrivelse af, hvordan Benz' portabilitet blev sikret, hvilket muliggør afvikling under Windows.
- **Afsnit 6:** En beskrivelse af de ændringer, der blev foretaget for at øge brugbarheden af Benz.
- **Afsnit 7:** En beskrivelse af krav til og designet af BenzDSL.
- **Afsnit 8:** En detaljeret beskrivelse af implementeringen af BenzDSL.
- **Afsnit 9:** Eksempler på brug af BenzDSL.

Til denne rapport findes endvidere følgende appendikser:

- **A:** Eksempler på BenzDSL-filer. Dette er bilagt rapporten.
- **B:** Eksempler på resultaterne af afviklingen af BenzDSL-filerne i A. Dette er bilagt rapporten.
- **Kildekode for BenzDSL [18]:** Kildekoden er tilgængelig i elektronisk form på den i litteraturlisten angivne URL.
- **Kildekode for guide til BenzDSL [19]:** Kildekoden er tilgængelig i elektronisk form på den i litteraturlisten angivne URL.

2. Benchmarking

I dette afsnit beskrives først, hvad der hører til benchmarking. Siden beskrives de eksisterende benchmarkmetoder i Benz og forslag til nye benchmarkmetoder.

2.1 Grundelementer i en benchmarktest

Benchmarking af en kode kan bruges til at få indikatorer af kvaliteten for den kode, der bliver benchmarket; en indikator kan f.eks. være køretiden. Hvis

en benchmarktestapplikation kan bruges til at teste flere forskellige koder, så kan resultaterne af at afvikle den med de forskellige koder bruges til at vurdere hvilken af koderne, der er bedst inden for testens rammer. Hvis en kode f.eks. er optimeret for køretid og en anden for hukommelsesbrug, så vil det forventede resultat af benchmarktesten af deres køretid være, at den første kode præsterer bedst, mens det forventes at forholde sig omvendt ved en benchmarktest af deres hukommelsesbrug.

At teste flere koder mod hinanden kaldes for ”motivating comparison” i [14]. Det beskrives, at motivationen til at sammenligne koder med hinanden er et ønske, om at vælge den bedste af de testede koder. I forbindelse med et valg af kode baseret på resultatet af en benchmarktest er det vigtigt at huske, at en kode kan præstere godt i en test men dårligt i en anden. En anden form for motivation er, når den, der foretager testen, har en interesse i den ene af de testede koder. Når en tester har interesse i den testede kode, er motivationen at se, hvor god denne kode er i forhold til anden kode. Et eksempel på denne form for motivation er, når kode fra CPH STL bliver testet mod tilsvarende kode i andre STL-implementeringer (som f.eks. GNU STL eller SGI STL) af udviklerne bag CPH STL.

Den test, der foretages på koderne, kaldes for ”task sample” [14] og skal helst indeholde en opgave indhentet fra virkeligheden. Ved valget af en virkelig opgave som ”task sample” testes noget, der reelt kan ske fremfor at teste rent hypotetiske scenarier. I tilfældet, hvor CPH STL testes, kan opgaven hentes fra eksisterende open source-applikationer, der bruger STL.

Den sidste del af et benchmark er ”performance measures” [14], hvilket er den eller de værdier, som måles under testen. I Benz er det den enkelte driver, der foretager disse målinger, og hver driver kan foretage en eller flere former for performance measures. Det besluttet før benchmarket, hvad der kvalificerer en god og en dårlig værdi. Hvis dette først besluttet efter, kan de målte værdier have en indflydelse på beslutningen. Hvad der er gode og dårlige værdier, skal enten fremgå klart af den valgte driver (f.eks. lav værdi ved køretidsmålinger) eller besluttet af testeren.

2.2 Eksisterende målingsmetoder i Benz

Den eksisterende udgave af Benz havde implementeret drivere til at måle følgende ting i en benchmarktest:

- **Tid:** Hvor lang tid det tager at fuldføre testen. Måling af tid er relevant især for applikationer, hvor køretiden er vigtig. Dette kan f.eks. være i spil, hvor det er vigtigt, at tiden ikke spildes i CPU’en, hvilket kan få spillet til at virke langsomt. En lav tid er altid et bedre resultat end en høj tid, forudsat at resultatet af det testede er korrekt. I figur 5 på side 15 er `cpu_time` et eksempel på en driver, der foretager netop den slags målinger.
- **Branch misprediction ratio:** Hvor ofte programmet foretager et hop til den modsatte branch end den antagne. Med denne måling kan der

vurderes, om programmet foretager uhensigtsmæssigt mange fejlgæt. Denne måling kan bruges til noget, hvis man har en viden om, hvordan et system foretager branch predictions, og om hvordan bedre branch predictions kan opnås ved at ændre i den testede kode. Ratio er et tal, der indikerer, hvor mange af det totale antal gæt, der var forkerte. Et lavt tal er derfor bedre end et højt. I figur 5 på side 15 er `branch_misprediction_ratio` et eksempel på en driver, der foretager netop den slags målinger.

- **Funktionskaldstæller af en specifik funktion:** Hvor mange kald der foretages til en navngiven funktion under testen, hvor det forventes, at denne funktion kaldes af den testede kode. Dette kan f.eks. være antallet af kald til en sammenligningsfunktion foretaget af forskellige sorteringsalgoritmer. I figur 5 på side 15 er `function_call_count` et eksempel på en driver, der foretager netop den slags målinger.
- **Major page fault:** Hvor ofte den eksterne hukommelse, f.eks. harddisken, tilgås under testen for at hente data [9]. Det forventes, at det er den testede kode, der forårsager hukommelsetilgangen. Det er bedst at tilgå den eksterne hukommelse det færrest mulige antal gange, hvorfor et lavt tal er bedre end et højt i denne måling. I figur 5 på side 15 er `execution_mem` et eksempel på en driver, der foretager netop den slags målinger.
- **Minor page fault:** Hvor ofte den ønskede side er i hukommelsen, men denne side ikke fremstår som værende opdateret af hardwaren [9]. Det er selvfølgelig bedre at have et lavt antal sidefejl end et højt. I figur 5 på side 15 er `execution_mem` et eksempel på en driver, der foretager netop den slags målinger.
- **Cache misses:** Hvor mange gange under testen data eller instruktioner var forventet at kunne forefindes i CPU'ens cache, men ikke var der. Hver gang dette sker, skal den forventede instruktion eller data hentes fra en langsommere hukommelsestype, hvorfor en lavere værdi er bedst i denne type måling. I figur 5 på side 15 er `execution_mem` et eksempel på en driver, der foretager netop den slags målinger.

2.3 Forslag til nye målingsmetoder i Benz

I dette afsnit omtales forslag til nye målingsmetoder, der kunne være interessante at have i Benz. Dette afsnit har derfor formentlig størst interesse for fremtidige udviklere af Benz, da det ikke omhandler konkrete implementeringer foretaget i dette projekt, men blot illustrerer ideer.

- **Funktionskaldstæller:** Hvor mange funktionskald der foretages under testen. Denne måling kan bruges til at se, om en kode laver flere eller færre funktionskald end en anden. Det er dog ikke klart, at det er bedre at have færre funktionskald. Et eksempel på en kode med flere funktionskald er nævnt i afsnit 4, hvor en af refaktoreringerne var at lave

nye funktioner og kalde disse. Altså har den refaktorerede kode flere funktionskald. Dette er en generalisering af den før omtalte funktionskaldstæller, der kun talte kald til en specifik funktion.

- **Maksimumhøjden af funktionskaldstakken:** Et mål for, hvor mange funktionskald, en kode maksimalt lægger på stakken. Dette kan være relevant, da det kan tænkes, at højden overstiger den maksimalt tilladte højde for det system, som koden skal køre på.
- **Det maksimale brug af hukommelse:** Et mål for, hvor meget hukommelse en given kode maksimalt bruger. Dette kunne være relevant, da ikke alle systemer har den samme mængde hukommelse, f.eks. har mobiltelefoner mindre hukommelse end PC'er.
- **Størrelsen af den kompilerede løsning:** Et mål for, hvor meget en kompileret kode fylder. Under antagelse af, at benchmarket kun er forskellig i valget af kode, kan denne måling bruges til at vurdere, om en given kode giver et større eller mindre program end en anden. Dette mål kan være relevant for systemer, der har begrænset plads til programmer i hardwaren.
- **Oprettelser af objekter:** Et mål for, hvor mange objekter koden opretter under testen. Dette mål kan være interessant for at se, om en kode opretter flere eller færre objekter i forhold til en anden.

2.4 Faldgruber

Benchmark af en kode skal foretages på alle platforme, koden skal kunne afvikles på. Grunden til dette er, at en kode ikke nødvendigvis opfører sig ens på alle platforme. Et eksempel på kode, der opfører sig forskelligt afhængig af platformen, er nævnt i afsnit 5. Nogle risici ved kun at foretage en benchmarktest på et system er:

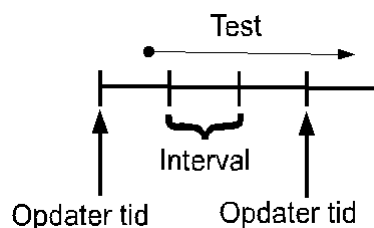
- Hvis testen bruges som grundlag for optimering af koden, så kan en optimering foretaget for den specifikke platform resultere i, at afvikling af koden bliver dårligere på de platforme, der ikke er testet på.
- Hvis testen bruges som grundlag for at foretage et valg mellem koder, så vil valget af den kode, der er bedst på den testede platform, ikke nødvendigvis være valget af den kode, der er bedst på andre platforme.

Når en benchmarktest bruges til at teste forskellige koder, er det vigtigt at undgå de mulige optimeringer, en given kode kan have for netop den pågældende benchmarktest [11]. Ved test af CPH STL-komponenter (som f.eks. `vector` eller `list`) kan den slags optimeringer f.eks. være baseret på datatypen, som gives til komponenten. Disse optimeringer kan lade sig gøre, da C++ tillader partielle specialiseringer. Et eksempel på en partiel specialisering kan bestå i, at komponenten — i tilfælde med kendte datatyper som f.eks. `int` — tager en kopi af hele den blok i hukommelsen, som de berørte data optager, og flytter denne. Den ikke-partielle specialisering vil i stedet,

givet en datatype, flytte elementerne ét ad gangen. Ved at undgå disse optimeringer fås en test af komponenten under dens normale forudsætninger. Hvis ønsket er at vide, hvordan komponenten opfører sig i forhold til en specifik datatype, skal ovennævnte optimeringer ikke forsøges undgået.

Ved benchmarktest af kode er der visse ting, der skal tages højde for [11]; herunder at operativsystemer kan interferere med afviklingen af et program og dermed interferere med koden. Dette sker eksempelvis, når operativsystemet afbryder kørslen af et program for at lade andre programmer køre eller for at reagere på eventuelle brugeres handlinger eller systemhændelser. Varigheden af disse afbrydelser er helt afhængig af, hvad der foretages under afbrydelsen, og afbrydelserne har derfor en effekt på resultatet af en test. Hvis der for eksempel foretages en måling af køretiden for en kode, er det ikke nok at notere tiden før og efter testen, idet denne tid ikke kun vil indeholde tiden, koden reelt brugte CPU'en, men også tiden der forløb under afbrydelsen. For at forebygge dette foreslås det, at en benchmarktest foretages med et minimum af processer kørende, således at den kode, der skal testes, i videst muligt omfang kan afvikles uden afbrydelser. Desuden foreslås det, at der kun udskrives testresultater efter testen er afviklet, da dette i sig selv kan oprette en proces, der kan lægge beslag på CPU-tid [11]. Dette forekommer umiddelbart fornuftigt nok, men hvis koden skal køre på en bestemt platform, må interferensen fra dette system og dets processer anses for at være en del af kodens omgivelser. Ved at ændre i kodens omgivelser for at teste koden, influeres kodens opførsel også. Et simpelt eksempel på dette er, at en kode optimeret til at køre godt, så længe dets data er i CPU'ens cache, vil få en bedre køretid, jo mindre operativsystemet og dermed andre processer interfererer med dets afvikling.

En anden vigtig ting at tage højde for er resolutionen af den målingsteknik, der bruges til at foretage målinger [11]. Hvis den valgte benchmarktest f.eks. foretager målingen af tiden med en resolution, der opdaterer den målte tid i bestemte intervaller, vil denne teknik aldrig kunne måle mere nøjagtigt end størrelsen af disse intervaller.



Figur 1. Målingresolutionsfejl.

Fra figur 1 kan det udledes, at starttiden målt for en given test kan være næsten et helt interval for tidligt, hvis testen begynder lige før, tiden bliver opdateret for det næste interval. Ligeledes kan sluttiden være et helt interval forkert, hvis testen slutter, lige før det næste interval starter. Fejlen for

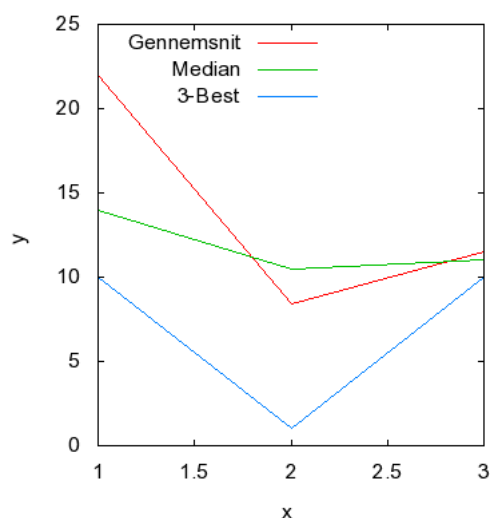
starttiden lægger maksimalt et interval til den reelle tid, og fejlen for sluttiden trækker maksimalt et interval fra den reelle længde. Dermed er den maksimale størrelse for fejlen netop ét interval. Der kan argumenteres for, at tiden kan nå at blive opdateret, efter at testen er slut, men før tiden måles. Sluttiden kan dermed være målt et helt interval for stort. Det antages dog, at årsagen til, at det sker, er, at testen sluttede lige inden intervallet. Hvis en test tager 10 intervaller, kan målingen være 10 % forkert, men hvis den tager 100 intervaller, kan målingen kun være 1 % forkert [10]. En måde at mindske denne fejl på er at øge antallet af intervaller, en test tager. Hvis det ikke er muligt at øge antallet af intervaller, kan testen gentages n gange. Ved at gentage testen øges antallet af intervaller, testen tager at afvikle, mens de overskydende intervaller forbliver det samme. Dog kan der opstå tidsforbrug mellem hver af de gentagne test, f.eks. hvis gentagelsen udføres i form af en løkke, bruges der tid på at rykke fra slutningen af en iteration af løkken til starten af løkken. Det sammenlagte målte resultat divideres med n for at finde den individuelle tests tidsforbrug [11]. En forudsætning for at bruge gentagne test til at gøre en testperiode længere er, at tilstanden for den testede struktur er den samme både før og efter den enkelte test. Hvis tilstanden ikke er den samme før hver gentagelse, er det ikke den samme test, der gentages. Hvis det, der testes, f.eks. er `push_back` på en vektor, så er vektoren efter den enkelte test ikke i den samme tilstand som før testen, idet den har et element mere.

2.5 Visualisering

Den sidste del af en benchmarktest er at visualisere resultatet, idet det er de færreste, der får noget ud af en endeløs række af tal. Til at visualisere resultatet af en benchmarktest kan f.eks. bruges grafer; disse vil typisk blive fremstillet ved at finde gennemsnitsværdien for de afholdte test. For at kunne tegne en linie anvendes forskellige startværdier, der markeres på x -aksen i et koordinatsystem, mens gennemsnitsværdierne for testkørslerne markeres på y -aksen. At beregne gennemsnittet på denne måde har dog den egenskab, at få værdier med en væsentlig afvigelse fra resten, kan trække gennemsnittet i deres retning. I stedet for at bruge gennemsnittet kan medianværdien bruges. Medianværdien har numerisk set halvdelen af målingerne over og under sig og kan beregnes ved først at sortere måleresultaterne og siden finde den værdi, der er i midten af den sorterede talrække. Hvis antallet af målinger er lige, vil medianen være gennemsnittet af de to tal i midten af den sorterede talrække [5]. En anden interessant idé: hvis det er sikkert, at de opnåede målinger altid er højere end det reelle forbrug, og de k bedste målinger ligger inden for en fastsat afvigelse fra hinanden, så må den bedste af de disse målinger være dårligere end det bedste resultatet, en given test kan give. Antagelsen er, at uanset hvilke former for interferens en test er blevet udsat for, så vil denne interferens aldrig have gjort resultatet bedre, end hvis den ikke var forekommet [10]. Forskellen på de tre nævnte visualiseringsteknikker er visualiseret med de tænkte værdier fra tabel 1 i figur 2.

Start værdi	Målinger								Gennemsnit	Median	3-Best
1	11	12	14	14	10	15	50	50	22	14	10
2	2	3	11	12	14	14	10	1	8,38	10,5	1
3	11	12	14	14	10	10	11	10	11,5	11	10

Tabel 1. Tænkte måleresultater.



Figur 2. Visualiseringsteknikker.

På grafen i figur 2 kan det ses, hvordan "Gennemsnit" ved den første måling er høj grundet de to målinger med værdien 50. Disse to målinger har derimod en ringe effekt på "Median" og slet ingen på "3-Best". Ved den anden måling kan ses, hvordan de lave målinger trækker både "Gennemsnit" og "3-Best" ned, mens de har en mindre effekt på "Median". Ved den tredje måling får de tre teknikker næsten samme resultat, da der ikke er nogen værdier, der er væsentligt højere eller lavere end de øvrige. Derfor vil ekstreme værdier have større indflydelse på en vurdering baseret på "Gennemsnit" end en vurdering baseret på "Median", og de vil kun have en indflydelse på en vurdering baseret på "3-Best", hvis de er blandt en af de tre bedste værdier.

Ud over det nævnte er det også vigtigt, at der for en foretaget benchmark-test noteres vigtige informationer om testen ud over resultatet af testen. Disse informationer skal kunne bruges til at genskabe en test og skal blandt andet indeholde informationer om systemet, testen blev afviklet på. Der skal også noteres versionen af det, der blev testet, samt fortolker- eller kompilatorversionen.

2.6 Opsamling

Følgende elementer er en del af en benchmarktest: koden, der skal testes, testen, der tester koden, målingmetoden, der måler testen og visualizeren, der visualiserer resultatet. Det er vigtigt, at det før testen besluttet, hvad der er et godt og et dårligt resultat af testen. Det skal ikke besluttet, efter at en test er afholdt, idet viden om det opnåede resultat kan påvirke denne beslutning. Det er også vigtigt, at testen er fuldkommen, hvilket f.eks. kan tilsikres ved at afvikle den på alle platforme og ved at tage højde for evt. problemer. Et problem kan være, om valget af struktur, test og platform har en effekt på resultatet, hvis der f.eks. anvendes en struktur, der er optimeret i forhold til platform og test. Et andet problem kan opstå ved valget af målingsmetode, hvor der eventuelt skal tages højde for metodens resolution. Et tredje problem kan være, at de opnåede resultater har en u hensigtsmæssig effekt på visualiseringen af dem.

3. Domænespecifikke sprog (DSL)

Formålet med at lave et domænespecifikt sprog er at gøre en opgave simplere for sprogets brugere. Et DSL skal kun kunne udføre opgaver inden for det domæne, som det er tiltænkt. Samtidig skal et DSL gøre en opgave lettere for sine brugere ved at lade dem abstrahere fra, hvordan en opgave normalt skal implementeres i et generelt programmeringssprog [20].

3.1 Internt DSL

Et internt DSL er kendetegnet ved, at det ikke kræver, at der bliver skrevet en fortolker eller kompilator til det, men i stedet kan bruge den, der anvendes i det sprog, som det konkrete DSL er implementeret i.

Et eksempel på et internt DSL kan være at lave et array A med en liste af tal mellem B og C med et interval på D . I f.eks. JavaScript gøres dette normalt på følgende måde:

```

1 var A=new Array();
2 for(var i=B;i<=C;i+=D){
3   A.push(i);
4 }

```

Hvis der skal laves flere af denne type arrays, skal ovenstående kode skrives for hvert array, og hver gang vil det kun være A , B , C og D , der ændres. Dette kan gøres lettere ved at implementere et DSL til at generere de enkelte arrays:

```

1 function talliste(name,from,to,jump){
2   var res=new Array();
3   for(var i=from;i<=to;i+=jump){
4     res.push(i);
5   }

```

```

6  /*
7  Make a variable with the value of name available in global
8  namespace, if name='A' it's possible to write A[0]
9  */
10 window[name]=res;
11 }
12
13 talliste('A',B,C,D);

```

Talliste DSL for JavaScript.

Talliste DSL gør det simplere at lave en liste af tal ved hjælp af en funktion, der tager de fire argumenter, der er nødvendige for at generere et array. Dette er et eksempel på et internt DSL [15], da det er ren JavaScript og dermed kan fortolkes af en JavaScript-fortolker. Det interne DSL er bundet til, hvad der er muligt i det sprog, som det er implementeret i. Dette betyder, at syntaksen for det originale sprog påtvinges DSL'et og dermed dets brugere. Men det betyder også, at funktionaliteter i det originale sprog kan bruges direkte i DSL'et.

3.2 Eksternt DSL

Hvis en bruger af et DSL ikke kender det sprog, som DSL'et er implementeret i, eller ikke er programmeringsvant, kan det tænkes, at en for brugeren mere naturlig måde at erklære en liste af tal på er $A = [B, E..C]$, hvor forskellen på B og E indikerer forskellen på tallene i listen, B og C indikerer da start og stop af tallisten, og D for det interne DSL svarer derfor til $D = B - E$. Da $A = [B, E..C]$ ikke er lovlige JavaScript-syntaks, skal der laves en fortolker.

```

1  function talliste_ext(text){
2      var re=text.match(/([^- ]*) *= *\[[ ]*([0-9]*) *, *([0-9]*) *\.\.
3          *([0-9]*) *\]/);
4      talliste(re[1], parseInt(re[2]), parseInt(re[4]),
5          parseInt(re[3])-parseInt(re[2]));
6  }
7  talliste_ext('A=[B, E..C]');

```

Talliste eksterne DSL for JavaScript.

Ovenstående eksempel er ikke at betragte som et internt DSL, selv om det også kun består af JavaScript. Det bliver betragtet som et eksternt DSL, da det kræver en fortolkning af en tekst og dermed reelt kræver sin egen fortolker. Fordelen ved et eksternt DSL i forhold til et internt DSL er, at sproget ikke er bundet til, hvad der er lovligt i det originale sprog. Denne frihed kan også være en ulempe, fordi sproget er begrænset til netop det, som kan fortolkes af den implementerede fortolker.

3.3 Opsamling

Der er to typer DSL: intern og ekstern. Det interne DSL bliver fortolket af den fortolker, det er skrevet i, mens det eksterne kræver sin egen fortolker.

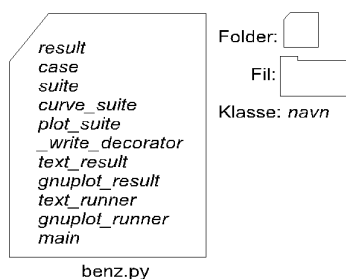
Det interne er dermed lettere at implementere, og det er lettere at få tilgang til funktionaliteter i det originale sprog, men dette betyder også, at det er bundet til, hvad der er lovligt og muligt at skrive i det originale sprog. Det eksterne DSL kræver derimod sin egen fortolker og er dermed ikke bundet af, hvad der er lovligt i det originale sprog. Det eksterne DSL kræver dog som nævnt sin egen fortolker, som skal implementeres.

4. Refaktoring

I dette afsnit beskrives de refaktoreringer, der var nødvendige for at gøre koden bag Benz mere overskuelig og genbrugelig. Før refaktoreringen bestod Benz af en enkelt Python-fil, `benz.py`, der indeholdt 11 klasser fordelt på 1380 linier kode samt nogle C++-filer for de enkelte drivere. Det er kun Python-delen af Benz, der blev refaktoreret.

4.1 Opsplitning i filer

Da det var uvant at arbejde med en opbygning af koden, hvor der var mere end en klasse i en fil, var det første skridt at dele Benz op. Dette blev gjort ved, at de enkelte klasser blev placeret i hver deres fil, idet denne opdeling fandtes mere overskuelig. Figur 3 og figur 4 viser opbygningen før og efter refaktoreringen, hvor `__init__.py`-filer er blevet udeladt for at øge overskueligheden.

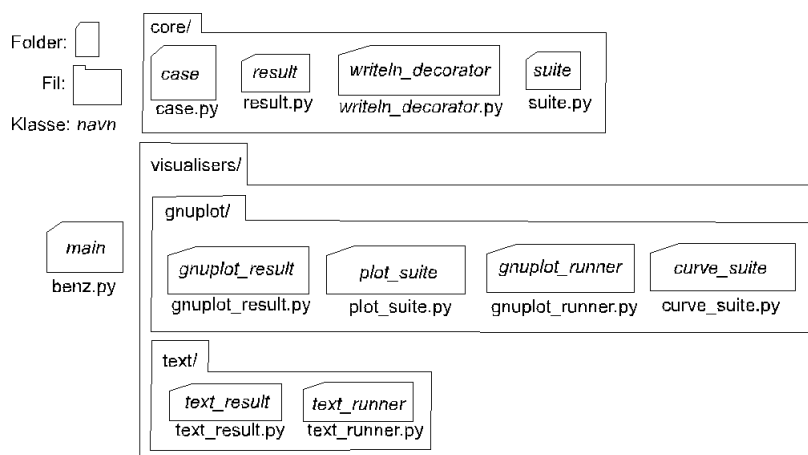


Figur 3. Før refaktoring af klasser til filer.

Selv i en moderne editor, der viser informationslister (lister af klasser og funktioner) i en fil, er det mere overskueligt kun at have én klasse i én fil. Dette skyldes antagelsen, der er baseret på personlig erfaring, om at programøren har tilbøjelighed til at rulle op og ned i den viste kode i stedet for at bruge de tilgængelige informationslister til at navigere med.

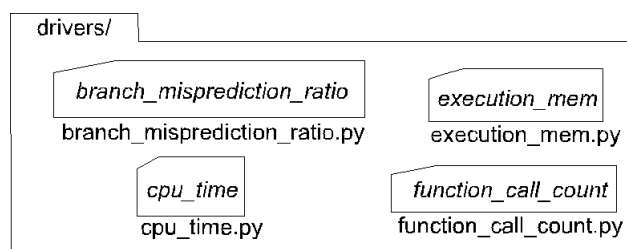
4.2 Opdeling af *case*-klassen

Det andet skridt var en simplificering af `case`-klassen. Denne klasse har ansvaret for at oprette drivere og testforsøg samt afvikle test. Dette gør `case` til en "God Class" [2], og i dette tilfælde var hver driver en funktion



Figur 4. Efter refaktoring af klasser til filer.

i `case`-klassen. For at mindske dette ansvar blev koden til at generere de forskellige drivere taget ud af klassen og lagt i deres egne klasser. Figur 5 viser den nye driverfolder.



Figur 5. Separeret drivers.

4.3 Udtræk af gentaget kode

Det tredje skridt var at mindske kodeduplikation, hvor kodeduplikation skal forstås i den forstand, at formen af den dupliserede kodelinje er ens fra sted til sted. Disse gentagne kodelinjer blev lagt i funktioner eller klasser, og de steder, hvor pågældende kodelinje befandt sig, blev denne udskiftet med kald til førnævnte funktion eller klasse. De individuelle kodelinjer var af og til forskellige i kraft af variabler, men også i mindre dele af selve koden. Disse forskelle blev implementeret som argumenter til den nye funktion eller klasse. At fjerne kodeduplikation ved at lave nye funktioner/klasser kaldes "Extract Method" i [12] og udføres ikke for at opnå en reduktion af koden, men for at centralisere koden. Centraliseret kode har den fordel, at der kun skal rettes ét sted i koden, hvis den senere skal rettes eller udbygges, lige som

gnu_runner
<pre>class gnuplot_runner(text_runner): def __init__(self, stream = sys.stderr, verbosity = 1): text_runner.__init__(self, stream, verbosity) def run(self, benchmark, result = gnuplot_result): results = <i>gnuplot_result</i>(self.stream, self.verbosity)</pre>
text_runner
<pre>class text_runner: def __init__(self, stream = sys.stderr, verbosity = 1): self.stream = <i>writeln_decorator</i>(stream) self.verbosity = verbosity def run(self, benchmark): results = <i>text_result</i>(self.stream, self.verbosity)</pre>
runner
<pre>class runner: def __init__(self, stream = sys.stderr, verbosity = 1): self.stream = <i>writeln_decorator</i>(stream) self.verbosity = verbosity def run(self, benchmark, <i>visualizer</i>): results = <i>visualizer</i>(self.stream, self.verbosity)</pre>

Tabel 2. Den relevante forskel på `text_runner` og `gnuplot_runner` og den nye `runner` fremhævet i kursiv skrift.

den ved senere brug kan anvendes med et enkelt kald til den nye funktion eller klasse.

Specielt kan nævnes klasserne `gnuplot_runner` og `text_runner`, hvis opgave er at afvikle test og sende resultatet af disse test til klassernes respektive resultatklasser. Den eneste reelle forskel i selve koden i disse klasser var navnet på resultatklassen, der blev oprettet objekter af. Da Python tillader, at en klasse sendes som argument, blev klasserne slået sammen til en enkelt generisk klasse, der får resultatklassen som argument. Forskellen kan ses i koderne i tabel 2.

4.4 Opsamling

Koden til Benz blev lagt ud i flere filer, én fil per klasse. Ved at rykke duplikeret kode ud i nye funktioner og klasser, blev kodeduplikationen minimeret. To klasser — en for hver af de eksisterende visualizere — blev slået sammen til en generisk klasse, hvilket også vil spare en klasse ved fremtidige visualizere, da disse også kan bruge den nye generiske klasse.

5. Portabilitet

I dette afsnit beskrives teknikken, der blev anvendt til at muliggøre afvikling af Benz på Windows XP. Der vil blive beskrevet hvilke ændringer, der

reelt blev foretaget, og endvidere vil en anden teknik til at gøre et program portabelt blive beskrevet.

For at øge anvendeligheden af Benz er det vigtigt, at værktøjet virker på så mange operativsystemer som muligt. Den udgave af Benz, der var tilgængelig ved projektets start, kunne som tidligere nævnt ikke afvikles under Windows, hvilket var en relevant mangel af portabilitet, da det afskar Windows-brugere fra at bruge Benz direkte.

5.1 Teknikken

I forsøget på at tilsikre, at Benz kan afvikles på Windows, anvendtes en iterativ test- og udviklingsstrategi. Denne baserede sig på, at Benz først forsøgtes afviklet og siden på baggrund af de opnåede fejl blev rettet til og derefter atter forsøgt afviklet. Teknikken byggede på den antagelse, at Python-fortolkeren ved hver fejlslagen afvikling af Benz kom med en fejlbesked. Det forventedes endvidere, at fejlbeskeden indeholdt informationer om stedet, hvor fejlen skete samt en tekst, der uddybede fejlen. Disse fejlbeskeder kunne da bruges til at finde og rette det sted i koden, der fejlede.

Denne teknik er fejlbehæftet, da den — for at kunne garantere at et helt program fungerer på en given platform — forudsætter, at fortolkeren fortolker hele programmets kode. Det er næppe sandsynligt, at denne forudsætning er dækkende for kun én afvikling af Benz, men det kan dog antages, at kodekonstruktioner svarende til den, der fejlede, også vil fejle, hvis fortolkeren støder på dem. Hvis denne antagelse holder, kan alle de tilsvarende kodekonstruktioner rettes ved hjælp af den samme rettelse som den, der blev appliceret på det sted, hvor fortolkeren initialt fejlede.

5.2 Udførelse

Da den valgte teknik krævede en afvikling af Benz, skulle der bruges en benchmarktest. Valget af benchmarktest faldt på *"inpluko_time"*, som er en in-place-sammenfletningsalgoritme.

Følgende ikke-portable kodekonstruktioner, der gjorde, at Benz ikke kunne afvikles under Windows, blev fundet og rettet med den tidligere beskrevne teknik:

- `os.uname()` [6]: Ifølge Python-dokumentationen er denne funktion kun understøttet af visse Unix-distributioner og er tiltænkt at skulle bruges til fjernafvikling af benchmarktest på andre maskiner. Funktionen blev udskiftet med `platform.uname()` [7]. Men sidst nævnte returnerede ikke nøjagtigt det samme som `os.uname()`. Dette er dog i praksis ikke et problem, da det kun var indeks 1 af det, `os.uname()` returnerede, der blev brugt, og dette svarer til indeks 2 i det, som `platform.uname()` returnerer.

```
1 >>> import os
2 >>> import platform
```

```

3 >>> print platform.uname()
4 ('Linux', 'brok', '2.6.30-DIKU-r5', '#1_SMP_PREEMPT_Fri_Nov_6_
   16:00:34_CET_2009', 'i686', 'Intel(R) Xeon(TM) CPU 3.20GHz')
5 >>> print os.uname()
6 ('Linux', 'brok', '2.6.30-DIKU-r5', '#1_SMP_PREEMPT_Fri_Nov_6_
   16:00:34_CET_2009', 'i686')

```

Udskrift af de returnerede data fra afvikling af `os.uname()` og `platform.uname()` på en af maskinerne på DIKU.

- **`commands.getstatusoutput(cmd)`** [3]: Denne funktion blev brugt til at afvikle kommandoer på den platform, Benz kørte på. Problemet med anvendelsen af denne funktion i Windows er, at den omkranser kommandoen med "{}", så "`g++ fil.c++`" bliver til "`{g++ fil.c++}`". Denne funktion er listet under Unix-specifikke funktioner i Python-dokumentationen. I stedet kan man anvende en anden funktion [13], der gør det samme, men som ikke sætter "{}" omkring kommandoen, hvis operativsystemet er Windows.

Efter at have foretaget de ovenstående ændringer for hele Benz, kunne Benz afvikles under Windows. Dette er dog ingen garanti for, at alle drivere — hverken deres Python-kode eller C++-kode — kan afvikles under Windows. For at finde ud af dette vil det være nødvendigt udføre udtømmende test af alle drivere.

5.3 Andre teknikker

En anden teknik til at sikre portabilitet er at indkapsle systemspecifik kode i blokke, der kun afvikles på et givent system, mens andre kodeblokke bliver afviklet på andre systemer. Denne teknik er kendt fra blandt andet JavaScript, hvor de forskellige internetbrowsere har gjort funktionaliteter tilgængelige ved hjælp af forskellige teknikker. Derfor skal JavaScript-kode fastslå hvilken browser, der anvendes, og ud fra denne information afvikle koden. Teknikken er også kendt i C/C++, hvor dele af en kode kan indkapsles i præprocessorblokke (*ifdef*). Denne teknik har dog den ulempe, at den skaber dobbelt kode, hvor kun den ene del gælder for et system. Dette betyder, at eventuelle kommende rettelser af koden skal foretages flere steder, hvis rettelser vedrører den systemspecifikke del af koden, og rettelser vel at mærke skal gælde for alle systemer. For hvert system skal rettelser skrives i den systemspecifikke kode, og den skal muligvis skrives uensartet afhængig af det enkelte system.

5.4 Opsamling

Det blev gjort muligt at afvikle Benz under Windows, hvilket udvider brugerpotentialet med alle de udviklere, der foretager benchmarking af kode under Windows. Det er dog ikke alle dele af Benz, der er testet for funktionalitet under Windows. Det vides for eksempel ikke, om funktionaliteten til afvikling af benchmarktest på eksterne maskiner fungerer under Windows.

6. Forøgelse af brugbarheden

I dette afsnit beskrives ændringer af Benz, der søger at gøre Benz mere brugervenlig for dets brugere. De beskrevne ændringer er ikke direkte relateret til BenzDSL, men da BenzDSL bruges til at generere Benz-programmer, gør programmerne brug af de udførte ændringer.

Applikationen Benz har to brugergrupper: Den ene gruppe (G_1) er dem, der udvider Benz med nye funktionaliteter hovedsageligt i form af nye drivere eller visualizere. Den anden gruppe (G_2) er dem, der bruger Benz til at foretage benchmarktest. G_1 er formodentlig en delmængde af G_2 .

6.1 *Simplere at tilføje nye drivere*

For at øge brugbarheden for brugerne i G_1 er følgende tiltag foretaget: Der er lavet en folder til drivere. Benz importerer en klasse fra en Python-fil i denne folder, og denne klasse bliver brugt som driverklassen. Navnet på filen, der skal importeres, gives af benchmarktesten. Dermed er det eneste, der skal gøres for at oprette en ny driver til Benz, at lave en Python-fil med en klasse, hvor både fil og klasse skal have driverens navn, og lægge denne fil i driverfolderen. Ud over kravet til navngivning af fil og klasse er det eneste krav, at klassen skal have en funktion kaldet `generate`. Denne funktion skal generere en C++-kodefil med en `main`-funktion og inklusionsserklæringer af filer nødvendige for driveren og testen. Før denne ændring blev implementeret, skulle der skrives en funktion ind i `case`-klassen for at lave en ny driver. Ændringen øger også brugervenligheden for brugerne i G_2 , hvis de vil installere en ny driver, da det i så fald er tilstrækkeligt at finde implementationen af driveren og lægge denne i driverfolderen.

6.2 *Simplere at tilføje nye visualizere*

En anden ændring foretaget for at øge brugervenligheden af Benz for G_1 er, at visualizere er lagt ud i en folder for sig selv. Hver visualizer er et Python-modul, som skal lægges i folderen. Det er dermed næsten lige så let at skrive en ny visualizer, som det er at skrive en ny driver. Ligesom med drivere er det nok at gøre visualizeren tilgængelig for brugerne i G_2 , der så skal lægge denne i deres visualizerfolder.

Man kan argumentere for, at Benz er blevet gjort mindre brugervenlig for brugerne i G_2 , da det kræver hentning af separate filer, hvis man vil opdatere Benz, men før ændringen ville en opdatering af Benz have krævet, at hele Benz blev hentet og installeret igen. Det forventes, at dette stadig vil være en mulighed, hvor hver ny driver og visualizer vil blive gjort til en del af Benz-installationen.

6.3 *Lettere at ændre driver*

Ændringen af, hvordan Benz indlæser drivere, bevirker, at man i stedet for at skulle skrive navnet på en driver som en fast del af koden i et eksperiment,

angiver navnet som en tekst. Dette betyder, at den samme kode kan bruges af forskellige eksperimenter. Før ændringen kunne brugen af forskellige drivere være opnået ved en `if else`-konstruktion i eksperimentet.

<pre> Nye version: self.driver_file = self.generate_driver('cpu_time') Gamle version: self.driver_file = self.generate_cpu_time_driver() </pre>

Forskellen på den nye og gamle måde at generere en driver i "inpluko_time.py".

6.4 Brugen af visualizer

Den ændring, at visualizere flyttes ud af Benz, gør, at Benz ikke længere kender til alle de klasser, en visualizer indeholder, hvorfor disse ikke automatisk gøres tilgængelige for eksperimenter. Dette betyder, at et eksperiment selv eksplicit skal importere de klasser, det skal bruge fra en given visualizer. Et eksempel på en berørt visualizer er den eksisterende `gnuplot-visualizer`, hvor eksperimenter gør brug af både `plot_suite` og `curve_suite` gennem nedarvning. Dette krav til eksplicit import af klasser gør, at brugere i G_2 skal kende opbygningen af det visualizermodul, som et eksperiment skal bruge.

6.5 For Windows-brugere

En anden ændring er, at Benz nu leder efter foldere med de C++-filer, som en driver skal bruge, i den folder, hvor "benz.py" findes. Dette gøres ved at finde stien til folderen med filen og lede efter navngivne foldere i denne folder. Før denne ændring blev foretaget brugte Benz systemvariablen `PYTHONPATH`. Denne indeholder en liste over foldere, som Python-fortolkeren skal lede i. Benz undersøgte samtlige foldere, der var nævnt i denne systemvariabel, med henblik på at finde den ønskede folder indeholdende filen "benz.py". Ændringen øger brugervengligheden for Windows-brugere, da de ikke længere skal oprette systemvariablen `PYTHONPATH`, lige som brugere af øvrige operativsystemer heller ikke skal rette eller oprette den omtalte systemvariabel.

6.6 Brugerquider

For begge brugergrupper er der skrevet en guide [16], hvilket sigter mod at gøre det lettere at bruge Benz. Guiden er skrevet i en viderebygning af et guideværktøj [17], der gør det let at skrive en guide og siden læse den, da guiden bliver skrevet som ren tekst i en HTML-fil og derefter ligeledes kan læses som HTML-fil. Dette bevirker, at den kan læses i alle browsere, og der anvendes samtidig JavaScript, CSS og HTML til at forbedre visualiseringen og navigationen i guiden set i forhold til guider, der alene er tekstbaserede.

6.7 Opsamling

Det er blevet gjort lettere at udvide Benz, idet nye drivere og visualizere ikke længere kræver ændringer i eksisterende dele af koden men blot fordrer, at nye filer lægges i forudbestemte foldere. Brugen af drivere i Benz er blevet mere fleksibel, da dette ikke længere er knyttet til fast kodede navne men derimod på variable tekststrengene. Det er gjort lettere at bruge Benz, da der ikke længere kræves ændringer i systemvariabler. Endvidere er der skrevet en visuelt orienteret guide til Benz.

7. Krav til BenzDSL

I dette afsnit beskrives krav til funktionaliteter i BenzDSL. Disse krav er dels opstået på baggrund af en analyse af, hvad der hører til benchmarking, fra de eksisterende eksempler i Benz og fra ekspertbrugeren. Afsnittet beskriver også ideen bag designet og syntaksen for BenzDSL.

7.1 Designet af BenzDSL

For at lave et brugbart DSL er det nødvendigt at opnå kendskab til domænet og eventuelt bruge en, der har ekspertkendskab til domænet. I dette projekt er domænet for BenzDSL benchmarking af kode, og formålet med BenzDSL er at generere benchmarktest, der kan afvikles med Benz. Derfor er designet af BenzDSL baseret dels på de tilgængelige testeksempler, der fulgte med Benz, og dels på samråd med en ekspertbruger af Benz.

7.2 Krav til BenzDSL

Ud fra analysen af testeksemplerne, der følger med Benz, skal det være muligt at specificere følgende i BenzDSL:

- Testen der skal afvikles.
- Strukturen som skal testes.
- Testtilfælde som hver kombination af test og struktur skal afvikles med.
- Hvad der skal bruges til at visualisere et testresultat med.
- Hvad der skal måles ved den afviklede test.

Fra ekspertbrugeren er følgende krav til BenzDSL fundet:

- Det skal være muligt at specificere flere strukturer at udføre en benchmarktest på.
- Det skal være muligt at specificere flere benchmarktest i et BenzDSL-program.
- Resultatet af at udføre en benchmarktest med flere strukturer skal udformes som en suite.
- Resultatet af at udføre flere benchmarktest, skal udformes som en suite af suiter.

- Det skal være muligt at visualisere hver benchmarksuite individuelt.
- Det skal være muligt at skrive `gnuplot`-kommandoer direkte, som de bliver skrevet i `gnuplot`-filer.
- Det skal være muligt at specificere kompilatoren og de indstillinger, den skal bruge til at kompilere testen.
- Det skal være muligt at specificere filnavne på de filer, som en afvikling af en test genererer.

Fra processen med at implementere sproget er følgende krav til BenzDSL fundet:

- Det skal være muligt at specificere, at den genererede test skal afvikles med det samme.
- Det skal være muligt at specificere, at de genererede testfiler skal fjernes med det samme.

7.3 Sprogets syntaks

I designet af BenzDSL blev der lagt vægt på, at der skal skrives mindst muligt for at lave en test. Dette er dels, fordi formålet med BenzDSL er at skrive benchmarktest hurtigere, end det er muligt med Benz' kombination af Python og C++, men også under den antagelse, at det vil være lettere at fortolke et lille sprog. Der er også lagt vægt på, at sproget skal være deklarativt [20].

Ideen til syntaksen for sproget er hentet fra INI-filer, hvor hver linie indeholder navnet på et felt og dets værdi. Desuden er syntaksen tilført Pythons regel om, at indrykninger skaber blokke. Uden indrykningsreglen har INI-filer kun to niveauer: et globalt og i en sektion, der er global. Ved at påføre Pythons indrykningsregel gøres det muligt at have flere niveauer. Denne syntaks skaber en struktur, der ligner den, der ofte ses i XML-filer når de læses eller skrives af personer, dog uden de forskellige tegn (f.eks. `<` og `>`) som er en del af XML-syntaksen.

7.4 Sprogets design

Designet af BenzDSL er således, at fortolkeren læser hele BenzDSL-filen, før den begynder at generere det resulterende Benz-program. Ideen med dette er, at det gør rækkefølgen, erklæringer skrives i, ligegyldig og dermed sproget mere frit. Dette svarer til, at funktioner i en C++-klasse kan erklæres i en vilkårlig rækkefølge, og er i modsætning til, at globale C++-funktioner skal erklæres før det tidspunkt, de kaldes. Da rækkefølgen af erklæringer er ligegyldig, vil kun den sidste af én type erklæringer blive brugt, såfremt der kun forventes en erklæring af den pågældende type. Designet af BenzDSL blev foretaget agilt i den forstand, at der løbende blev implementeret egenskaber ind i sproget [20]. En alternativ metode ville være at skrive en sprogspecifikation og derefter implementere denne.

Designet af BenzDSL er endvidere sådan, at et reserveret ord kun er reserveret i den konkrete kontekst, det befinder sig i. Hvis det ikke giver mening, opfattes det reserverede ord som et almindeligt id. Effekten af dette er, at reserverede ord kun minimerer navnerummet for id'er i dele af BenzDSL og ikke i hele BenzDSL, som det f.eks. er tilfældet med det reserverede ord `if` i C++.

Resultatet af en fortolkning af en BenzDSL-fil er en Python-fil og et antal C++-filer. Antallet af testprogrammer afhænger af antallet af erklærede benchmarktest, strukturer og drivere i BenzDSL-filen. Generelt vil der for hver kombination af benchmarktest og struktur kræves en kompilering, men hvis driveren kræver en kompilering for hvert testtilfælde, skal dette ganges med antallet af testtilfælde i testen.

De benchmarktest og strukturer, der bruges, er alle skrevet i C++. Det forudsættes, at det ikke skal være muligt at generere C++-kode fra bunden med BenzDSL. Denne forudsætning bygger på, at det vil være svært at skrive et fornuftigt DSL, der kan dette korrekt. Derfor er BenzDSL i stedet designet således, at strukturer og benchmarktest bliver skrevet i C++-filer.

Det skal også være muligt at erklære den datatype, der skal bruges i en benchmarktest, og den driver, en benchmarktest skal udføres med.

For at kunne generere et benchmarkprogram, skal BenzDSL altså imødekomme alle ovenstående krav. Hvis det antages, at struktur, benchmarktest, datatype og driver konstituerer hver deres modul, er opgaven for BenzDSL at lade brugeren identificere de enkelte moduler og lade BenzDSL kombinere disse til et benchmarkprogram. Ideen med at kombinere flere dokumenter til et enkelt færdigt dokument kendes dels fra diverse HTML-templatesystemer (Smarty [8]), og fra C++, hvor kodefiler kombineres med `#include` og kompileres til et program.

En fordel ved denne teknik er, at den tillader simpelt genbrug af moduler. Når først det er gjort muligt at identificere et struktur-, datatype-, benchmarktest- eller drivermodul, så kan dette modul identificeres på samme måde i alle BenzDSL-programmer. For at dette skal virke, skal strukturen og testen kende navnet på datatypen, og testen skal kende navnet på strukturen. For at øge den mulige genanvendelsesgrad af moduler bygger designet på en antagelse om, at navnet på den brugte datatype altid er "datatype", og navnet på den brugte struktur altid er "structure". At tvinge en type til et navn i C++ er ikke svært; dette gøres med `typedef`:

```
1 typedef int datatype;
```

Definition af type med navnet `datatype` af den originale C++-type `int`.

Hvis den struktur, der ønskes testet, er en funktion, kan ovenstående metode ikke anvendes, da en funktion ikke er en type i C++ men derimod er en instans af en type. For at erklære en funktion som den struktur, der skal testes, kan man gøre én af to ting:

- Kaldet til funktionen kan indkapsles i en klasse, der så kaldes ”structure”.

```

1 class structure{
2     void call(){
3         std::sort();
4     }
5 };

```

Erklæring af ”structure” til at være en klasse med en funktion, der kalder `std::sort`.

- Erklæringen ”`#define`” fra C++ kan bruges til at erklære ”structure” som et pseudonym for funktionens navn. Effekten af dette er, at alle efterfølgende forekomster af ”structure” udskiftes med funktionens navn og dermed kald til funktionen.

```

1 #define structure\
2     std::sort

```

Definer ”structure” til teksten `std::sort`.

Forskellen på de to teknikker er, at ”structure” skal bruges som en type ved indkapsling i en klasse, mens den ved brugen af ”`#define`” skal bruges som navnet på en funktion i et funktionskald. Forskellen på de to kommer i en benchmarktest til udtryk ved, at den første teknik skal igennem en instans af ”structure”-klassen for at foretage det indkapslede funktionskald. Denne teknik kan dermed have en effekt på værdien, der måles i benchmarktesten. Denne vil for eksempel forårsage en længere køretid.

Ideen med at kræve, at en kode har forudbestemte navne, er kendt fra grænseflader i f.eks. Java, hvor en klasse, der implementerer grænseflader, skal implementere et sæt af forudbestemte funktioner. Det er ikke undersøgt, om man ved brugen af ”`typedef`” af typer til et navn ubevidst undgår optimeringer på den pågældende types originale navn.

Med denne teknik undgås det, at man skal skrive benchmarktest, der både kan tage template- og ikke-templateklasser eller -funktioner. Dette undgås, fordi templateklasser og -funktioner bliver specialiseret til en konkret type eller funktion i strukturmodulet. Alternativet ville være, at benchmarktestmodulerne eller BenzDSL havde ansvaret for at specialisere den klasse eller funktion, der blev erklæret i strukturmodulet. Teknikken, der indebærer at bruge faste navne, er kun et forslag, og BenzDSL-fortolkeren skal derfor ikke teste, om dette bliver gjort.

7.5 Opsamling

Det modulære design af BenzDSL gør det let at genbruge strukturer, drivere, test og visualisere i BenzDSL. Genbruget gøres muligt for dets brugere ved at disse skal skrive forholdsvis lidt. Syntaksen for BenzDSL er en kombination

af syntaksen i INI-filer og Python, og dets designet med at fortolke hele filen, før der bliver genereret et Benz-program, gør at erklæringer kan skrives i en vilkårlig rækkefølge. Da syntaksen ligner Pythons, burde det være let for brugerne af Benz at skrive BenzDSL-programmer.

8. Implementeringen af BenzDSL

I dette afsnit gives en beskrivelse af sproget BenzDSL, herunder en beskrivelse af, hvad der er muligt i BenzDSL, og hvordan det skal udformes. Der vil endvidere være en kort beskrivelse af, hvordan BenzDSL-fortolkeren fungerer.

Eksempler på BenzDSL-programmer kan ses i Appendiks A.

8.1 Fortolkeren

Fortolkeren BenzDSL er implementeret på følgende måde: Fortolkeren deler en given tekst op i tokens², hvor hvert token har en tilsvarende Python-klasse. Når et token findes, oprettes der en instans af klassen, som lægges på en tokenliste. Når hele teksten er blevet til en liste af tokeninstanser, kalder fortolkeren en funktion (*eat*) på en instans af en konstruktionsklasse. Sidstnævnte undersøger, om den kan blive til en valid konstruktion på det pågældende sted i tokenlisten. Dette gøres både ved at se på de enkelte tokens i tokenlisten og ved at se, om mindre konstruktioner kan oprettes på stedet i tokenlisten. Hvis en konstruktion kan være på det pågældende sted i tokenlisten, rykker instansen pointeren i tokenlisten frem til det første token, der ikke er en del af konstruktionen. Den første konstruktion er selve programmet. Illustreret med konstruktionen `"if (COND) { STMTS }"` fra C++ ville instansen foretage følgende skridt for at genkende if-konstruktionen::

1. Undersøge om tokeninstanserne for `if` og `(` er de næste tokens i listen — og hvis de er — rykkes pointeren to tokens frem i tokenlisten.
2. Oprette en instans af `COND`-konstruktionen og kalde dennes `"eat"`-funktion. Hvis `COND` er valid, rykker denne pointeren frem i tokenlisten.
3. Undersøge om tokeninstanserne for `)` og `{` er de næste tokens i listen — og hvis de er — rykkes pointeren to tokens frem i tokenlisten.
4. Oprette en instans af `STMTS`-konstruktionen og kalde dennes `"eat"`-funktion. Hvis `STMTS` er valid, rykker denne pointeren frem i tokenlisten.
5. Undersøge om tokeninstansen for `}` er det næste token i listen — og hvis det er — rykkes pointeren et token frem i tokenlisten.

Hvis instansen finder, at et af ovenstående skridt ikke er muligt, rykkes pointeren tilbage til startpositionen, og instansen returnerer besked om, at konstruktionen ikke er valid.

² Et token er et stykke tekst, der er blevet genkendt med et regulært udtryk.

8.2 Muligheder med BenzDSL

I dette afsnit vil mulighederne med BenzDSL kort blive beskrevet. Det er i BenzDSL muligt at skrive en suite af benchmarktest i en tekstfil, der følger syntaksreglerne for BenzDSL.

I en BenzDSL-fil er det muligt at erklære alle ting, der er beskrevet i nedenstående liste:

- Det er muligt at erklære de enkelte benchmarktest på et globalt niveau.
- Det er muligt at erklære den datatype, der skal testes med. Denne kan erklæres på et globalt niveau for alle benchmarktest og individuelt for den enkelte benchmarktest. En individuel erklæring har præcedens over en global, og en global erklæring har præcedens over den forudbestemte type, som er `int`.
- Det er muligt at erklære de strukturer, der skal testes. Disse kan erklæres på globalt niveau for alle benchmarktest og individuelt for den enkelte benchmarktest. Individuelt og globalt erklærede strukturer slås sammen og bruges alle for den enkelte benchmarktest.
- Det er muligt at erklære den kompilator, der skal bruges til at kompilere med. Denne kan erklæres på globalt niveau for alle benchmarktest, individuelt for den enkelte benchmarktest og individuelt for den enkelte struktur. En kompilator erklæret for en struktur har præcedens over en, der erklæret for en benchmarktest. En kompilator erklæret for en benchmarktest har præcedens over en, der er erklæret globalt, og en kompilator erklæret globalt har præcedens over den, der er forudbestemt af Benz.
- Det er muligt at erklære den driver, der skal foretage målinger af benchmarktesten. Denne kan erklæres på globalt niveau for alle benchmarktest og individuelt for den enkelte. En individuel erklæring har præcedens over en global.
- Det er muligt at erklære den visualizer, der skal bruges til at visualisere et resultat. Denne kan erklæres på globalt niveau for alle benchmarktest og individuelt for den enkelte benchmarktest. En individuel erklæring har præcedens over en global.
- Det er muligt at erklære de testtilfælde, der skal bruges i en benchmarktest, som heltal. Disse kan erklæres på globalt niveau for alle benchmarktest og individuelt for den enkelte benchmarktest. Individuelt og globalt erklærede testtilfælde slås sammen og bruges alle for den enkelte benchmarktest.
- Det er muligt at erklære, hvor de genererede testfiler skal lægges, afvikle og fjerne dem igen. Dette kan gøres på globalt niveau.
- Det forventes, at benchmarktest- og strukturfiler i C++ bliver skrevet på en sådan måde, at de er mest genbrugelige. For at muliggøre dette gælder, at koden i en benchmarktest-fil kan antage, at en struktur kaldes "structure" og en datatype kaldes "datatype", og dermed kan bruge disse bestemte navne.

- Det er muligt for den individuelle benchmarktest at sætte værdier i `case`-klassen, som kan bruges af både driver og visualizer. Disse værdier er også tilgængelige for den aktive visualizer i dennes `commands`-erklæring samt i en benchmarktestfil beregnet på tekstudskiftninger skrevet på Python-formen³.
- Det er muligt for den individuelle drivererklæring at erklære værdier, der bliver tilgængelige i `case`-klassen og dermed tilgængelige for driveren under afviklingen af en test.

Når en beskrivelse i ovenstående liste erklærer præcedens over en anden, vil den erklæring med højest præcedens blive anvendt. Dette svarer til de gængse scopingregler i programmeringssprog (som f.eks. C++). Findes to variabler med samme navn, da vil den variabel, hvis scope er tættest på det scope i koden, programpointeren er ved, 'dække' over den variabel, hvis scope er længere væk.

8.3 Konfigurationsfil

Den implementerede BenzDSL-fortolker bruger en konfigurationsfil "configuration.py" til at konfigurere sig selv. Denne fil indeholder følgende konfigurationsopsætninger:

- Erklæring af stien til folderen med strukturer, som er sat til "Structures".
- Erklæring af stien til folderen med datatyper, som er sat til "Datatypes".
- Erklæring af stien til folderen med testskabeloner, som er sat til "Benchmarks".
- Erklæring af filendelsen på filerne i førnævnte foldere. Denne er sat til "C++".

I de efterfølgende sektioner vil det antages, at konfigurationsopsætningen er som beskrevet i ovenstående.

8.4 Specifikation af BenzDSL

Denne sektion indeholder en specifikation af sproget BenzDSL skrevet i notationsformen EBNF [4].

```

1 int = ? det regulæudtryk [0-9]+ identificere en int, en int kan
   også bruges som en id terminal?;
2
3 id = ? det regulæudtryk [^\s]+ identificere et id ?;
4
5 string = ? det regulæudtryk ('|")(?P<string>.*?(?='|"))('|")
   identificere en streng ?;
6
```

³ Python kan udskifte "%(navn)s" i en tekststreng med værdien af "navn" i en datastruktur.

```

7 | indent = ?En indent terminal startes med en ny linie \n efterfulgt af
   | to mellemrum "  " per scopelevel efterfulgt af et tegn, der ikke
   | er whitespace [^\s]. Scopelevel starter på 0 og stiger med 1 for,
   | hver indent terminal til den nuværende terminal. Dermed skal
   | globale erklæringer ikke have noget mellemrum foran sig og som er
   | undererklæring til en anden erklæring skal have to mellemrum mere
   | foran sig end den anden erklæring?;
8
9 | anything = int
10 |           | string
11 |           | id
12
13 | compiler = "compiler" , id , {indent , anything};
14
15 | datatype = "datatype" , id;
16
17 | namedparameter = indent , id , (id | string);
18
19 | driver = "driver" , id , {namedparameter};
20
21 | remotelyoption = indent ,( "connection"
22 |                       | "transfer"
23 |                       | "computer" ), id;
24
25 | remotely = ?Ordet "remotely" plus de 3 mulige
26 |           remotelyoption terminaler?;
27
28 | id_list = id , {id};
29
30 | title = "title" , id;
31
32 | filename = "filename" , id;
33
34 | template = "template" , id;
35
36 | includepath = indent , string;
37
38 | includepaths = "include" , "paths" , {includepath};
39
40 | structureoptions = indent , ( compiler
41 |                             | title
42 |                             | includepaths );
43
44 | structure = indent , id_list , {structureoptions};
45
46 | structures = "structures" , {structure};
47
48 | benchmarkstatement = indent , ( structures
49 |                             | template
50 |                             | datatype
51 |                             | driver
52 |                             | visualizer
53 |                             | filename
54 |                             | compiler
55 |                             | cases
56 |                             | namedparameter);
57
58 | benchmark = ?benchmarkstatement-terminalen med template skal findes ,
   | terminalen:"benchmark" , {benchmarkstatement} ?;
59
60 | list_tuple = ("["|"{" ) , int , [","] , int , (".." , ["."]), int ,
   | ("]"|"}") );
61
62 | cases = "cases" , {indent , ( list_tuple
63 |                             | int ) };

```

```

64 |
65 | gnuplotcommands = indent , anything;
66 |
67 | unittestcommands = indent , "output" , string;
68 |
69 | visualizercommands = "commands" , ?Terminal gnuplotcommands hvis
    |   visualizer id er 'gnuplot', terminal unittestcommands hvis
    |   visualizer id 'unittest'?;
70 |
71 | visualizeroptions = indent , ?Hvis id terminalen is visualizer er
    |   'gnuplot' eller 'unittest' kan terminalen visualizercommands
    |   bruges?;
72 |
73 | visualizer = "visualizer" , id , {visualizeroptions};
74 |
75 | outputoptions = indent , ( "execute"
76 |   | "cleanup"
77 |   | "path" , string
78 |   | "name" , id );
79 |
80 | output = "output" , {outputoptions};
81 |
82 | program = { cases
83 |   | compiler
84 |   | datatype
85 |   | driver
86 |   | output
87 |   | remotely
88 |   | structures
89 |   | visualizer };

```

8.5 Erklæring af kompilator

Det er muligt at erklære den kompilator, der skal bruges til at kompilere testene med, samt at erklære indstillinger til denne. En erklæring af kompilator starter med ordet "compiler" efterfulgt af navnet på den pågældende kompilator. Indstillingerne til kompilatoren erklæres i de efterfølgende linier ved at skrive dem på den form, de skal bruges af kompilatoren. Hvis en erklæring af kompilator er mulig i efterfølgende forklaringer af erklæring, vil dette blive indikeret med ordet **COMPILER**. Erklæring af kompilator er altid valgfri, og hvis denne ikke findes, anvender Benz den forudbestemte GNU-kompilator.

```

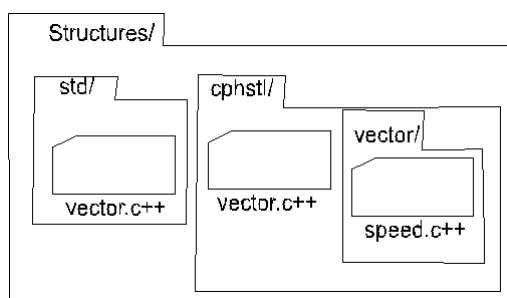
compiler g++
-O3

```

Erklæring af g++-kompilatoren med option -O3.

8.6 Erklæring af strukturer

Hvis hver strukturfil lægges i en folder i filsystemet, kan BenzDSL finde denne fil ved at få en liste bestående af navnene på de forskellige foldere efterfulgt af filens navn.



Figur 6. Eksempel på en mulig opbygning af Structures-folderen.

Hvis det antages, at BenzDSL kigger i "Structures"-folderen vist på figur 6, så kan fortolkeren bruge listen ("std","vector") af id'er til at finde filen "Structures/std/vector.c++" og en anden liste ("cphstl","vector","speed") til at finde filen "Structures/cphstl/vector/speed.c++". Sidstnævnte kan antages at indeholde en specialisering af CPH STL's vector optimeret for hastighed. Dermed skal brugerne af BenzDSL ikke bekymre sig om C++-kode for hver struktur, eller om der skal anvendes / eller \ som separator mellem foldere (Windows og Unix bruger hver sin).

Visse strukturer kræver, at den anvendte kompilator får information om, i hvilke foldere den skal lede efter ekstra kodefiler. Et eksempel er "*g++-Ic:/cphstl/...*" hvor g++-kompilatoren får informationer om, at den skal lede i folderen "cphstl" på disken "c". Det er derfor nødvendigt, at man kan erklære disse stier for hver struktur. Det første forslag var at bruge absolutte stier til dette, men ekspertbrugerne gjorde opmærksom på, at brugen af absolutte stier bevirker, at den enkelte test i så fald kun kan bruges på maskiner, hvor stien passer. I stedet for kan syntaksen "*\$(CPHSTL)/...*" bruges, hvor det antages, at CPHSTL er inkluderet i operativsystemets systemvariabler. Denne syntaks kendes fra makefiler. Erklæringen af strukturen *cphstlvector* kan se ud som følger:

```

cphstl vector
include paths
  "$(CPHSTL)/Source/Vector/Code"
  "$(CPHSTL)/Source/Iterator/Code"
  "$(CPHSTL)/Source/Type/Code"
title "cphstl vector"
COMPILER
  
```

Erklæring af CPH STL-vector.

For at undgå, at brugerne skal skrive inklusionsstier for alle løsninger i samtlige BenzDSL-filer, er det gjort muligt at erklære inklusionsstierne i en blok-kommentar i starten af den enkelte strukturs C++-fil.

```

/*
  -I$(CPHSTL)/Source/Vector/Code
  -I$(CPHSTL)/Source/Iterator/Code
  
```

```
-I$(CPHSTL)/Source/Type/Code
*/
```

Toppen af `cphstl/vector.c++` filen.

Disse stier bliver brugt, såfremt der ikke specifikt erklæres nogen i BenzDSL-filen.

Da det var et krav, at det er muligt at erklære flere strukturer, skal disse erklæres i en blok, der indledes med ordet "structures".

```
structures
cphstl vector
include paths
  "$(CPHSTL)/Source/Vector/Code"
  "$(CPHSTL)/Source/Iterator/Code"
  "$(CPHSTL)/Source/Type/Code"
COMPILEER
std vector
COMPILEER
```

Erklæring af strukturerne *cphstlvector* og *stdvector*.

Der kan også erklæres en titel og en kompilator for den enkelte struktur. Titlen bruges af den valgte visualizer til at identificere resultatet for den enkelte struktur i en visualisering af resultatet af testen. Hvis kompilatorerklæringen udelades, bruges den kompilator, der vælges af den pågældende test. Hvis en erklæring af en strukturblok er mulig i efterfølgende forklaringer af erklæring, indikeres dette med **STRUCTURES**.

8.7 Erklæring af datatype

Som med strukturer kan datatypefiler (C++) lægges i en folder "Datatypes". Navnet på filen bruges i BenzDSL til at erklære brugen af den enkelte datatype. De mest basale typer som `int` og `float` er ikke i denne folder, men deres respektive typedefs skrives direkte af BenzDSL-fortolkeren til det genererede testprogram. For at erklære en datatype bruges ordet "datatype" efterfulgt af typens navn. Denne erklæring er altid valgfri. Hvis ingen erklæring findes, bruger Benz den forudbestemte datatype `int`. Hvis en erklæring af datatype er mulig i efterfølgende forklaringer af erklæring, indikeres dette med **DATATYPE**.

```
datatype int
```

Erklæring af datatypen `int`.

8.8 Erklæring af testtilfælde

Testtilfælde kan kun erklæres som heltal. Hvis en given test skal anvende andre værdier end heltal, må den selv generere dette ud fra det tal, den får. Da det antages, at der skal erklæres flere testtilfælde, kan dette gøres med tallisteerklæring enten på formen `[INT, INT .. INT]` eller

$\{INT\ INT \dots INT\}$. Der er ingen forskel på de to former ud over den visuelle, og det er endvidere er muligt at erklære flere tallister. Alle testtilfælde erklæres inde i en blok, der indledes med ordet "cases". Hvis alle *INT* i en tallisteerklæring er ens, er det kun et testtilfælde. Hvis de derimod er forskellige, er det en liste af testtilfælde, hvor det første og sidste *INT* afgrænser listen (begge inklusive), og forskellen på det første og andet er intervallet mellem de enkelte testtilfælde i listen. Hvis en erklæring af testtilfælde er mulig i efterfølgende forklaringer af erklæringer, indikeres dette med **CASES**.

```
cases
{1 2...3}
{5 5...5}
```

Erklæring af testtilfældene 1, 2, 3 og 5.

8.9 Erklæring af visualisering

For at erklære, hvordan en benchmarktest skal visualiseres, er det nok at skrive id'et på visualiseringsteknikken, der skal bruges. Visse visualiseringsteknikker accepterer eller forventer ekstra værdier; disse skrives under undererklæringen **commands**. Hvordan **commands** bliver fortolket, afhænger af den valgte visualiseringsteknik. Hvis den valgte teknik for eksempel er **gnuplot**, bliver den tekst, der er skrevet under **commands**, accepteret, som den er, og skrevet til **gnuplot**-filen. Det er dog muligt at skrive tekstudskiftninger på Python-formen; disse vil blive forsøgt udskiftet med informationer hentet fra den enkelte benchmarktest. Ordet "visualizer" starter en visualiseringserklæring og efterfølges af navnet på teknikken, der skal bruges. Hvis en erklæring af visualizer er mulig i efterfølgende forklaringer af erklæringer, indikeres dette med **VISUALIZER**.

```
visualizer gnuplot
  commands
  set title 'Test'
```

Erklæring af visualizeren **gnuplot** og **commands** "set title 'Test'".

8.10 Erklæring af driver

For at erklære en driver er det nok at skrive ordet "driver" efterfulgt af dennes navn. Der kan gives værdier til den enkelte driver, hvilket kan gøres ved at skrive værdiens navn efterfulgt af dens værdi. De erklærede værdier vil være tilgængelige i den instans af **case**-klassen, der opretter en driver ved afvikling af en test; dette i dennes dictionary. Hvis en erklæring af driver er mulig i efterfølgende forklaringer af erklæringer, indikeres dette med **DRIVER**.

```
driver cpu_time
  time_unit ns
  dual_exist 1
```

Erklæring af driveren "cpu_time" og værdierne "time_unit" og "dual_exist".

8.11 Erklæring af benchmarktest

En benchmarktest erklæres med ordet "benchmark" og skal have et unikt navn, hvilket udføres med undererklæringen `name` efterfulgt af navnet på testen. Det er vigtigt, at navnet er unikt for hver benchmarktest i en BenzDSL-fil, da det bruges i navngivningen af den korresponderende C++-fil. Der er en undererklæring `template`, der skal efterfølges af navnet på den testtemplate, der skal bruges. Værdien tildelt `template` skal være navnet på en C++-fil i folderen "Benchmarks". Der kan erklæres en visualizer, strukturer, driver, datatype, testtilfælde og kompilator for den enkelte benchmarktest. Dette gøres ved at erklære disse som tidligere beskrevet — dog indrykket, således at de er under den pågældende benchmarktestserklæring.

```
benchmark
  name push_back
  template N_Repeat
  DATATYPE
  STRUCTURES
  COMPILER
  CASES
  DRIVER
  VISUALIZER
  call push_back
  plot_title "Timing sequential pop back"
  xlabel "0 to N calls"
  ylabel "Time in ms"
  structures
    std vector
      title "Std:Vector"
    std list
      title "Std:list"
  filename push_back
```

Erklæring af benchmarktesten "push_back".

Erklæring af datatype, driver, kompilator og visualizer har præcedens over tilsvarende globale erklæring af samme art. Erklæring af cases og structures lægges til globale erklæring af samme art. Det er også muligt at erklære andre værdier, hvilket gøres ved at skrive værdiens navn efterfulgt af dens værdi. Disse værdier vil være tilgængelige i `case`-klassens dictionary, når Benz programmet bliver afviklet. Værdierne bliver også brugt i tekstudskiftninger af `gnuplot`-visualizerens `commands`-erklæring. Navnet på den fil, som resultatet af benchmarktesten skal skrives til, kan erklæres ved at skrive ordet `filename` efterfulgt af navnet på filen uden filendelsen. Filendelsen skal ikke skrives, da denne er afhængig af den valgte visualizer, f.eks. er filendelsen for `gnuplot`-visualizeren ".gp". Hvis en erklæring af en

benchmarktest er mulig i efterfølgende forklaringer af erklæringer, indikeres dette med **BENCHMARK**.

8.12 Erklæring af output

Ordet "output" starter output-erklæringen, der accepterer følgende under-erklæringer:

- Erklæring af stien til folderen, hvor de genererede C++-filer og Python-filen skal placeres. Denne erklæres med **path** efterfulgt af stien i en tekststreng.
- Erklæring af navnet på den genererede Python-fil foretages med **name** efterfulgt af navnet. Hvis denne erklæring udelades, bliver filens navn "test.py".
- Hvis det genererede Benz-program ønskes afviklet med det samme, kan dette gøres ved at skrive **execute**-erklæringen. Hvis erklæringen udelades, bliver programmet ikke automatisk afviklet.
- Hvis det genererede program ønskes fjernet, kan dette gøres ved at skrive **cleanup**-erklæringen. Denne erklæring bruges til at fjerne programmet efter afvikling og bør derfor kun bruges sammen med **execute**-erklæringen. Det giver ikke mening at bruge denne erklæring uden **execute**, men det er muligt.

Hvis en erklæring af output er mulig i efterfølgende forklaringer af erklæringer, indikeres dette med **OUTPUT**.

```
output
  path ""
  name test.py
  execute
  cleanup
```

Erklæring af output. Den genererede Python-fil bliver sat til at skulle hedde "test.py", og det genererede Benz-program bliver sat til at blive afviklet og slettet.

8.13 Et BenzDSL-program

Et BenzDSL-program skrives i en tekstfil og består af de ovenstående erklæringer. For alle erklæringer bortset fra **BENCHMARK** gælder det, at programmet bør have ingen eller netop én af dem. Der kan dog skrives flere, men da er det kun den sidste af de skrevne, der bliver brugt. For **BENCHMARK**-erklæringer forventes det, at der er én eller flere. Hvis der ikke er nogen, medfører dette fejlmeddelelser til brugeren.

```
DATATYPE
COMPILER
DRIVER
STRUCTURES
```

```

CASES
BENCHMARK
.
.
.
BENCHMARK
VISUALIZER
OUTPUT

```

Erklæring af et BenzDSL-program.

8.14 Opsamling

Med de beskrevne erklæringskonstruktioner er det muligt at skrive en BenzDSL-fil, der henter og kombinerer C++-koder til en eller flere filer, når BenzDSL-filen bliver eksekveret. C++-koderne består både af den eller de strukturer, der testes, den eller de benchmarktest, der skal udføres med strukturerne, samt den eller de drivere, der skal måle benchmarktestene. Det er muligt at angive forskellige testtilfælde for hver benchmarktest, lige som det er muligt at angive, hvordan resultatet af en test skal visualiseres. Det er endvidere muligt at få det genererede Benz-program afviklet og slettet med det samme, så det kun er resultatet af benchmarktestene, der er tilbage. Slutteligt er det muligt at bestemme, hvilken kompilator der skal bruges, og hvilke indstillinger, denne skal have.

9. Test af Benz DSL

I dette afsnit gives der eksempler på BenzDSL-filer samt resultatet af at afvikle de benchmarktest, der er indeholdt i disse filer. Til projektet blev der udviklet en driver "win_cpu_time", som blev brugt i de efterfølgende test, der har med tidsmålinger at gøre. Grunden til, at denne driver blev brugt og ikke den eksisterende "cpu_time", var, at den eksisterende driver var for lang tid om at afvikle en test.

Den nye driver implementerer ideen om, at den bedste tid i en mængde af ens test må være den tid, der havde mindst interferens som beskrevet i afsnit 2.

9.1 Push_back-test

Denne test måler tiden, det tager at udføre `push_back`-operationen på en vektor, der allerede har n elementer. Hvert testtilfælde er antallet af elementer, og for hvert tilfælde bliver testen gentaget 100 gange. BenzDSL-koden for dette kan ses i appendiks A.1 og resultatet af testen er grafen i appendiks B.1. På grafen kan det ses, hvor hurtige de enkelte implementeringer er om at sætte et element ind i vektoren, og effekten af implementeringens vækstpolitik (growth policy) kan ses i form af skarpe stigninger i køretiden.

Der blev også skrevet en driver "as.is". Denne driver tager resultatet af den enkelte test og konverterer dette til en Python-værdi. Den nuværende

udgave af driveren accepterer kun decimalværdier og kan f.eks. bruges til at visualisere den spildplads, der findes i en vektor efter hvert enkelt `push_back`-kald. Spildpladsen kan beregnes ved hjælp af følgende formel: $\frac{B-A}{B}$, hvor A er antallet af elementer i vektoren, og B er vektorens kapacitet — det vil sige antallet af elementer, der er plads til i vektoren. En BenzDSL-fil, der bruger denne driver, kan ses i appendiks A.2, og resultatet af testen kan ses i appendiks B.2. Til spildpladstesten hæfter der sig den kommentar, at GNU STL-vektoren har en spildplads på 50 % hele tiden. En mulig tolkning af dette er, at implementeringen, når den bliver instantieret med tallet n , laver en vektor med plads til to gange n elementer. Dette er ikke en fejl, men det er et interessant implementeringsvalg, at GNU STL-vektoren automatisk oprettes med dobbelt så meget plads, som brugeren udbeder sig.

De ovennævnte to test afprøver, hvorvidt det i BenzDSL er muligt at:

- Erklære en global datatype.
- Erklære en global driver.
- Erklære værdier for den globale driver.
- Erklære en benchmarktest.
- Erklære flere strukturer globalt.
- Erklære flere strukturer for benchmarktest.
- Erklære flere benchmarktest i samme BenzDSL-fil.
- Erklære titler for både globale og lokale strukturer.
- Erklære en gnuplot-visualizer.
- Erklære cases globalt.
- Erklære cases lokalt for benchmarktesten.
- Bruge værdier fra benchmarktesten og strukturer i visualizeren.
- Erklære output.
- Erklære en global kompilator.
- Bruge inklusionsstier skrevet i strukturfilerne.
- Erklære værdier i en benchmarktest, der kan bruges af driveren.

9.2 Midtvejsindsættelsetest

Denne test måler tiden, det tager at indsætte m elementer i en vektor med n elementer, hvor n er en værdi, der afhænger af hvert enkelt testtilfælde, og m er defineret som en fast værdi i drivererklæringen i BenzDSL-filen (appendiks A.3). Som ved `push_back`-testen gentages hvert testtilfælde 10 gange, og kun den mindste af de resulterende værdier bruges. Resultatet kan ses på grafen i appendiks B.3. Et eksempel på test af samme struktur, men med forskellige kompilatorindstillinger blev lavet med midtvejsindsættelse. BenzDSL-filen for dette kan ses i appendiks A.4 og resultatet i appendiks B.4.

Disse to test afprøver, hvorvidt det i BenzDSL er muligt at:

- Erklære inklusionsstier for en struktur i BenzDSL-filen, der så bruges i stedet for dem, der findes i strukturfilen.
- Erklære individuelle kompilatoropsætninger for den enkelte struktur.

9.3 *At-test*

Denne test måler tiden, det tager at tilgå et element i en vektor ved brug af funktionen `at`. Hvert testtilfælde er antallet af elementer i vektoren, og indekset, der slås op, er det sidste element i denne vektor. Hvert testtilfælde er blevet gentaget 1000 gange. BenzDSL-filen for denne test kan ses i appendiks A.5 og resultatet i appendiks B.5. Resultatet kan ved første øjekast virke noget uklart, men her er det vigtigt at lægge mærke til, hvor små udsvingene faktisk er.

Denne test afprøver ikke funktionalitet i BenzDSL, der ikke allerede er afprøvet i de foregående test.

9.4 *Unittest*

Unittest anses normalt ikke for at være en benchmarktest, men vejlederen til dette projekt rejste spørgsmålet om, hvad den reelle forskel på de to former for test er. Konklusionen på dette spørgsmål er, at en unittest godt kan anses for at være et benchmark af, hvordan den testede struktur klarer testen. Ud fra denne konklusion blev der lavet en driver, der læser resultatet af en test som et antal linier, hvor hver linie er resultatet af en del af testen. Hver linie starter med enten 1 (bestået) eller 0 (ikke-bestået) efterfulgt af en unik tekst, der beskriver og identificerer den pågældende del af testen. Visualiseringen af resultatet tæller på basis af testtilfælde bestået og ikke-bestået sammen for hver del af testen og giver et procentuelt svar for beståelsen af den enkelte del. Dette blev testet på forskellige vektorimplementeringer i CPH STL. BenzDSL-filen for dette findes i appendiks A.6, og resultatet kan ses i appendiks B.6.

Denne test verificerer, at unittest-driveren og -visualizeren kan bruges i BenzDSL. Da unittest består af en ny visualizer og driver blev det også her testet, hvor let Benz kunne udvides med en visualizer og driver.

9.5 *Sorteringsalgoritmetæller*

Denne test tæller antallet af kald, der foretages til en sammenligningsfunktion for forskellige sorteringsalgoritmer. Hvert testtilfælde n er antallet af elementer i den vektor, der skal sorteres. De forskellige algoritmer er alle globale funktioner, og deres strukturfiler indeholder derfor `#define`-erklæringer i stedet for `typedef`-erklæringer som beskrevet tidligere. BenzDSL-filen for denne test kan findes i appendiks A.7 og resultatet i appendiks B.7.

9.6 Opsamling

Hele sproget BenzDSL er blevet testet — med undtagelse af muligheden for fjernafvikling — og virkede, som det var forventet. Som effekt af at teste BenzDSL, blev der lavet en ny tidsmålingsdriver, der var hurtigere end den eksisterende. Der blev også testet, hvor let det var, at udvide Benz med nye drivere og visualizere; dette fandtes forholdsvis let. Det var til gengæld mere krævende at udvide BenzDSL med en ny visualizer, da dette krævede ændringer af fortolkeren i den del, der fortolker visualizer-konstruktionen af sproget.

10. Konklusion

Der er blevet udviklet et DSL kaldet BenzDSL, som kan bruges til at skrive benchmarktest i. Et af resultaterne af dette projekt er, at det blev lettere at udvide applikationen Benz med nye drivere og visualizere, da disse ikke længere skal flettes ind i eksisterende kode, men bare lægges i en folder. Det er relativt let at skrive en benchmarktest i BenzDSL, lige som det er gjort enkelt at gøre testskabeloner, strukturskabeloner og datatyper tilgængelige for BenzDSL, idet dette blot kræver at deres filer lægges i de rigtige foldere.

Det er til gengæld ikke så let at udvide BenzDSL til at fortolke nye typer af visualiseringer. Grunden til dette er, at den Python-kode, som hver visualisering kræver i den genererede Python-fil, er skrevet direkte ind i BenzDSL-fortolkeren, og denne kode er individuel for hver enkelt visualizer. Derfor kræver en udvidelse med en ny visualizer ændringer i fortolkeren. Dette kan formentlig gøres bedre ved at lægge koderne i visualizer-folderen, men det vil stadig kræve ændringer i BenzDSL-fortolkeren, da hver visualiseringsteknik tolker sin `commands`-undererklæring forskelligt. Enhver driver, som er tilgængelig for Benz, kan til gengæld også bruges i BenzDSL; dette under forudsætning af, at den enkelte driver har en funktion kaldet "output" tilgængelig, der konverterer resultatet af en benchmarktest til Python-værdier.

Sproget BenzDSL er statisk i den forstand, at det ikke kan udvides med nye funktionaliteter ved at skrive noget i selve sproget. For at udvide sproget med nye funktionaliteter skal der rettes i fortolkeren af sproget. Store dele af fortolkeren er skrevet i en enkelt fil på trods af, at det kan være kompliceret at overskue filer med flere klasser. Grundet opbygningen af fortolkeren — især med visse autogenerated klasser — er det ikke muligt at skrive alle klasserne i fortolkeren i hver deres fil.

Det faktum, at fortolkeren er skrevet således, at den læser hele filen før, den handler, gør det også lettere at skrive en test, da dette betyder, at der ikke er nogen forudbestemt rækkefølge for, hvorledes elementer skal erklæres. Det implementerede sprog er basalt set en form for datafil og kunne principielt være skrevet som f.eks. en XML-fil, hvilket dog ville have krævet, at en bruger af sproget skulle skrive XML med alle dets ekstra tegn og regler.

Man kan sætte spørgsmålstegn ved, hvorfor der i implementeringen af BenzDSL ikke blev anvendt syntaks fra et eksisterende sprog. Svaret på dette er dels, at dette ville kræve af brugerne, at de skulle anvende alle for det pågældende sprog specifikke ekstra tegn og syntaksregler. Endnu vigtigere er det dog, at man undgår forudantagelser om funktionaliteterne i sproget BenzDSL baseret på det sprog, det måtte ligne.

Det er blevet gjort muligt at afvikle Benz under Windows. Dette øger portabiliteten af Benz med et ekstra operativsystem, og dermed øges Benz' mulige brugerbase.

Det implementerede sprog BenzDSL og dets fortolker er blevet testet. Dette er gjort ved at skrive og afvikle benchmarktest skrevet med BenzDSL. Testen viste at implementeringen af sproget fungerede, som det var forventede. Dog mangler, der at blive testet, om det er muligt at skrive benchmarktest til fjernafvikling i BenzDSL.

Sproget BenzDSL skriver felter og værdier direkte ind i en klasse, der nedarver fra `case`-klassen. Det er dermed muligt at overskrive vigtige funktioner eller felter i den originale `case`-klasse, hvorfor en fremtidig videreudvikling af fortolkeren bør derfor søge at fjerne dette. Det vil dog formentlig kræve ændringer i eksisterende drivere og visualizere, der leder efter værdier i `case`-klassens variabler.

Litteratur

- [1] CPH STL, Benz, Website accessible at <http://www.cphstl.dk/WWW/tools.html> (2009).
- [2] SourceMaking, The blob, Worldwide Web Document (2009). Available at <http://sourcemaking.com/antipatterns/the-blob>.
- [3] Python, `commands.getstatusoutput(cmd)`, Website accessible at <http://docs.python.org/library/commands.html#commands.getstatusoutput> (2009).
- [4] Wikipedia, Extended backus-naur form, Website accessible at http://en.wikipedia.org/wiki/Extended_BackusNaur_Form (2009).
- [5] Wikipedia, Median, Website accessible at <http://en.wikipedia.org/wiki/Median> (2009).
- [6] Python, `os.uname()`, Website accessible at <http://docs.python.org/library/os.html#os.uname> (2009).
- [7] Python, `platform.uname()`, Website accessible at <http://docs.python.org/library/platform.html#platform.uname> (2009).
- [8] Smarty, Smarty, Website accessible at <http://www.smarty.net> (2009).
- [9] Wikipedia, Page fault, Website accessible at http://en.wikipedia.org/wiki/Page_fault (2010).
- [10] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, Prentice Hall (2010). (<http://www.csapp.cs.cmu.edu/public/ch9-preview.pdf>.)
- [11] R. M. Clapp, L. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultze, Toward real-time performance benchmarks for Ada, *Communications of the ACM* **29**, 8 (1986), 760–778.
- [12] S. McConnell, *Code Complete*, 2 Edition, Microsoft Press, Redmond, Washington (2003).
- [13] A. Reedick, Windows friendly `commands.getstatusoutput`, Worldwide Web Document (2008). Available at <http://mail.python.org/pipermail/python-win32/2008-January/006606.html>.
- [14] S. E. Sim, S. Easterbrook, and R. C. Holt, Using benchmarking to advance research:

- A challenge to software engineering, *Proceedings of the 25th International Conference on Software Engineering* (2003).
- [15] V. Subramaniam, Creating DSLs in Java, part 3: Internal and external DSLs, Worldwide Web Document (2008). Available at <http://www.javaworld.com/javaworld/jw-08-2008/jw-08-dsls-in-java-3.html>.
 - [16] J. P. Svensson, The complete guide to Benz, Worldwide Web Document (2009). Available at <http://cphstl.dk/Tool/BenzDSL/Guide/guide.html>.
 - [17] J. P. Svensson, Doc: Dokumentationsværktøj for CPH STL, CPH STL rapport **2010-1**, CPH STL (2009). Available at <http://cphstl.dk/Tool/Doc/Report/report.pdf>.
 - [18] J. P. Svensson, Kildekode for BenzDSL, Worldwide Web Document (2009). Available at <http://cphstl.dk/Tool/BenzDSL/Report/kildekodeBenzDSL.pdf>.
 - [19] J. P. Svensson, Kildekode for guide til BenzDSL, Worldwide Web Document (2009). Available at <http://cphstl.dk/Tool/BenzDSL/Report/kildekodeGuide.pdf>.
 - [20] E. Visser, WebDSL: A case study in domain-specific language engineering, Technical report, Delft University of Technology (2008).

A. DSL-testfiler

A.1 Tidsmåling af push_back:

```

1
2 datatype int
3
4 driver win_cpu_time
5   time_unit ns
6   runs 100
7
8 cases
9   {10 11 .. 2000}
10
11 structures
12   std vector
13   title "std::vector"
14
15 benchmark
16   name test_push_back
17   template nth_push_back
18   structures
19     cphstl vector lap
20     title "cphstl::lap_vector"
21     cphstl vector hat
22     title "cphstl::hat_vector"
23     cphstl vector safe_da
24     title "cphstl::safe_da_vector"
25     cphstl vector esafe_da
26     title "cphstl::esafe_da_vector"
27   plot_title "Timing_n_vector_push_back_call"
28   filename pushback
29
30
31
32 visualizer gnuplot
33   commands
34     set encoding iso_8859_1
35     set output "%(name)s.png"
36     set terminal png size 2000,800
37     set key vert left
38     set title '%(plot_title)s'
39     set data style lines
40     set xlabel 'Nth_call'
41     set ylabel 'Time_in_ns'
42
43 compiler g++
44   -O3
45
46 output
47   execute
48   cleanup

```

A.2 Push_back waste maling

```

1
2 datatype int
3
4 driver as_is
5   result_type float
6
7
8 cases
9   {10 11 .. 2000}
10
11 benchmark
12   name test_push_back_waste_std
13   template nth_base
14   structures
15     std_vector
16     title "std::vector"
17   plot_title "Waste_on_GNU_STL_vector_push_back_call"
18   primal "container.push_back(datatype());"
19   verify "container.size()==n+1;"
20   result_type "float"
21   result "(float)(container.capacity()-container.size())_/_
22     (float)container.capacity()"
23   filename waste_std_vector
24
25 benchmark
26   name test_push_back_waste_safe_da
27   template nth_base
28   structures
29     cphstl_vector_safe_da
30     title "cphstl::safe_da_vector"
31   plot_title "Waste_with_safe_da_vektor_push_back_call"
32   primal "container.push_back(datatype());"
33   verify "container.size()==n+1;"
34   result_type "float"
35   result "(float)(container.capacity()-container.size())_/_
36     (float)container.capacity()"
37   filename waste_safe_da
38
39 benchmark
40   name test_push_back_waste_esafe_da
41   template nth_base
42   structures
43     cphstl_vector_esafe_da
44     title "cphstl::esafe_da_vector"
45   plot_title "Waste_with_esafe_da_vector_push_back_call"
46   primal "container.push_back(datatype());"
47   verify "container.size()==n+1;"
48   result_type "float"
49   result "(float)(container.capacity()-container.size())_/_
50     (float)container.capacity()"
51   filename waste_esafe_da
52
53 benchmark
54   name test_push_back_waste_hat
55   template nth_base
56   structures
57     cphstl_vector_hat
58     title "cphstl::hat_vector"
59   plot_title "Waste_with_hat_vector_push_back_call"
60   primal "container.push_back(datatype());"
61   verify "container.size()==n+1;"

```

```
61 result_type "float"
62 result "(float)(container.capacity()--container.size())_/_
    (float)container.capacity()"/>
63 filename waste_hat
64
65
66 benchmark
67 name test_push_back_waste_lap
68 template nth_base
69 structures
70     cphstl vector lap
71     title "cphstl::lap_vector"
72 plot_title "Waste_with_lap_vector_push_back_call"
73 primal "container.push_back(datatype());"
74 verify "container.size()==n+1;"
75 result_type "float"
76 result "(float)(container.capacity()--container.size())_/_
    (float)container.capacity()"/>
77 filename waste_lap
78
79
80 visualizer gnuplot
81 commands
82     set encoding iso_8859_1
83     set output "%(filename)s.png"
84     set terminal png size 2000,300
85     set key vert left
86     set title '%(plot_title)s'
87     set data style lines
88     set xlabel 'Nth_call'
89     set ylabel 'Waste_%%'
90
91 output
92     execute
93     cleanup
```

A.3 Tidsmåling af midtvejsindsættelse

```

1
2 datatype int
3
4 driver win_cpu_time
5   time_unit ns
6   runs 10
7
8 cases
9   {10 10000 .. 1000000}
10
11
12 benchmark
13   name test_midway_insertion
14   template midway_insertion
15   structures
16     cphstl vector safe_da
17     title "cphstl::_safe_da_vector"
18     cphstl vector esafe_da
19     title "cphstl::_esafe_da_vector"
20     cphstl vector lap
21     title "cphstl::_lap_vector"
22     cphstl vector hat
23     title "cphstl::_hat_vector"
24     cphstl vector
25     include paths
26       "$(CPHSTL)/Source/Type/Code"
27       "$(CPHSTL)/Source/Iterator/Code"
28       "$(CPHSTL)/Source/Vector/Code"
29     title "cphstl::vector"
30     plot_title "Timing_midway_insertion_vector_10000_elements"
31     filename midway_insertion
32     M 100000
33
34
35
36 visualizer gnuplot
37   commands
38     set encoding iso_8859_1
39     set output "%(name)s.png"
40     set terminal png size 2000,800
41     set key vert left
42     set title '%(plot_title)s'
43     set data style lines
44     set xlabel 'Container_size'
45     set ylabel 'Time_in_ns'
46
47 output
48   execute
49   cleanup

```

A.4 Kompilatoroptimeringer ved midtvejs indsættelse

```

1
2 datatype int
3
4 driver win_cpu_time
5     time_unit ns
6     runs 10
7
8
9 cases
10 {10 100000 .. 10000000}
11
12
13 benchmark
14     name test_midway_insertion_compiler
15     template midway_insertion
16     structures
17         cphstl vector lap
18             title "cphstl::lap_vector, g++-O3"
19             compiler g++
20             -O3
21         cphstl vector lap
22             title "cphstl::lap_vector, g++-O2"
23             compiler g++
24             -O2
25         cphstl vector lap
26             title "cphstl::lap_vector, g++-O1"
27             compiler g++
28             -O1
29         cphstl vector hat
30             title "cphstl::hat_vector, g++-o3"
31             compiler g++
32             -O3
33         cphstl vector hat
34             title "cphstl::hat_vector, g++-O2"
35             compiler g++
36             -O2
37         cphstl vector hat
38             title "CPHSTL::hat_vector, g++-O1"
39             compiler g++
40             -O1
41     plot_title "Timing_midway_insertion_vector_10000_elements, _with_
42         compiler_optimisations"
43     M 100000
44     filename midway_insertion_optimisations
45
46
47 visualizer gnuplot
48     commands
49         set encoding iso_8859_1
50         set output "%(name)s.png"
51         set terminal png size 2000,800
52         set key vert left
53         set title '%(plot_title)s'
54         set data style lines
55         set xlabel 'Container_size'
56         set ylabel 'Time_in_ns'
57
58 output
59     execute
60     cleanup

```

A.5 Tidsmåling af at

```

1
2 datatype int
3
4 driver win_cpu_time
5   time_unit ns
6   runs 1000
7
8 cases
9   {1 2...2000}
10
11 benchmark
12   name test_sda_at
13   template nth_at
14   structures
15     std vector
16     title "std::vector"
17     cphstl vector safe_da
18     title "cphstl::_safe_da_vector"
19     cphstl vector esafe_da
20     title "cphstl::_esafe_da_vector"
21     cphstl vector
22     include paths
23       "$(CPHSTL)/Source/Type/Code"
24       "$(CPHSTL)/Source/Iterator/Code"
25       "$(CPHSTL)/Source/Vector/Code"
26     title "cphstl::_vector"
27     cphstl vector hat
28     title "cphstl::_hat_vector"
29     cphstl vector lap
30     title "cphstl::_lap_vector"
31   plot_title "Timing_n_vector_at_calls"
32   filename at
33
34
35 visualizer gnuplot
36   commands
37     set encoding iso_8859_1
38     set output "%(name)s.png"
39     set terminal png size 2000,800
40     set key out vert right
41     set title '%(plot_title)s'
42     set data style lines
43     set xlabel "Nth_call"
44     set ylabel "Time_in_ns"
45
46 output
47   execute
48   cleanup

```

A.6 Unittest af vector implementeringer

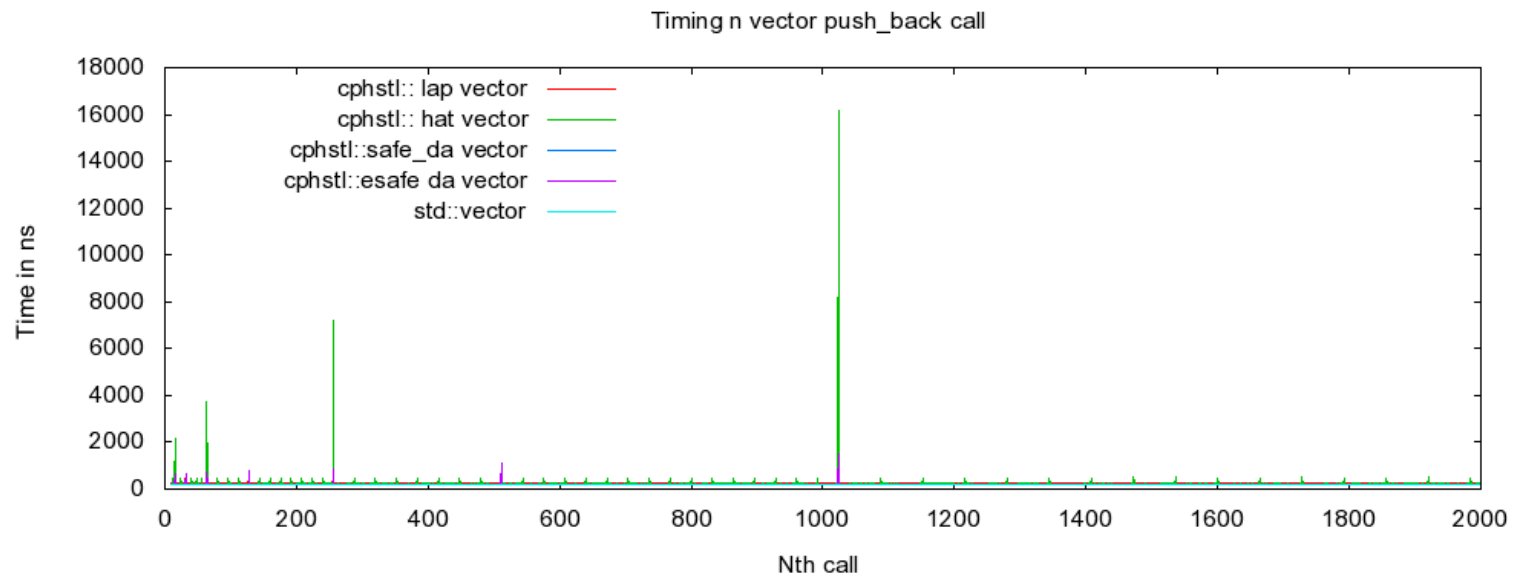
```
1 datatype int
2
3 driver unittest
4
5 structures
6   cphstl vector
7
8   std vector
9
10  cphstl vector safe_da
11
12  cphstl vector esafe_da
13
14  cphstl vector hat
15
16  cphstl vector lap
17
18
19 benchmark
20   name StandardVector
21   template unittest_vector
22   cases
23     1
24   filename unittestVector
25
26 visualizer unittest
27   commands
28     set output '%(filename)s.txt '
```

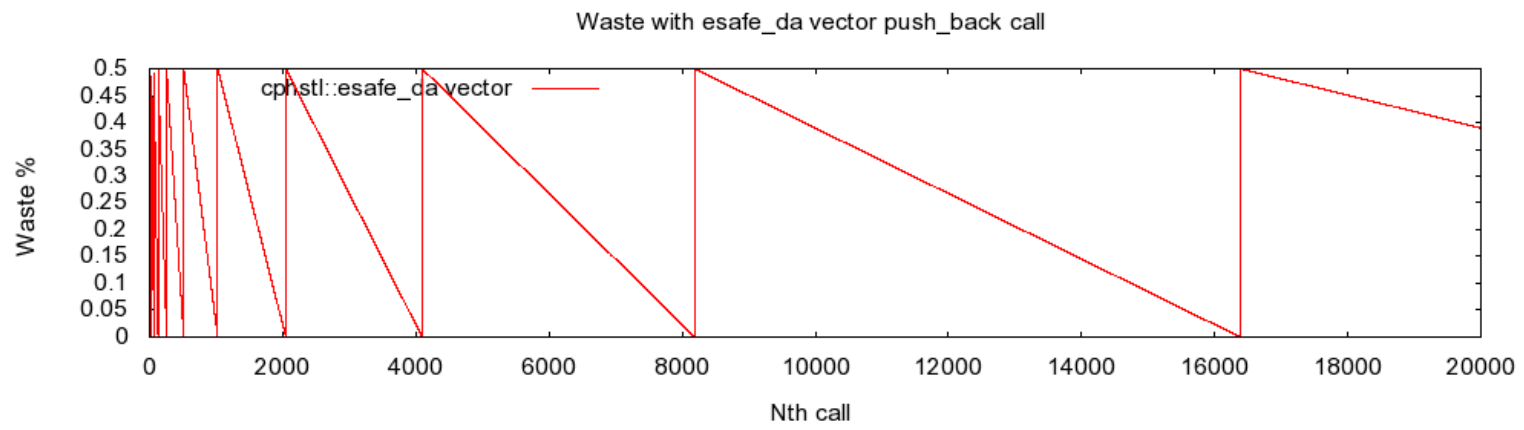
A.7 Sorteringsalgoritme funktionstæller

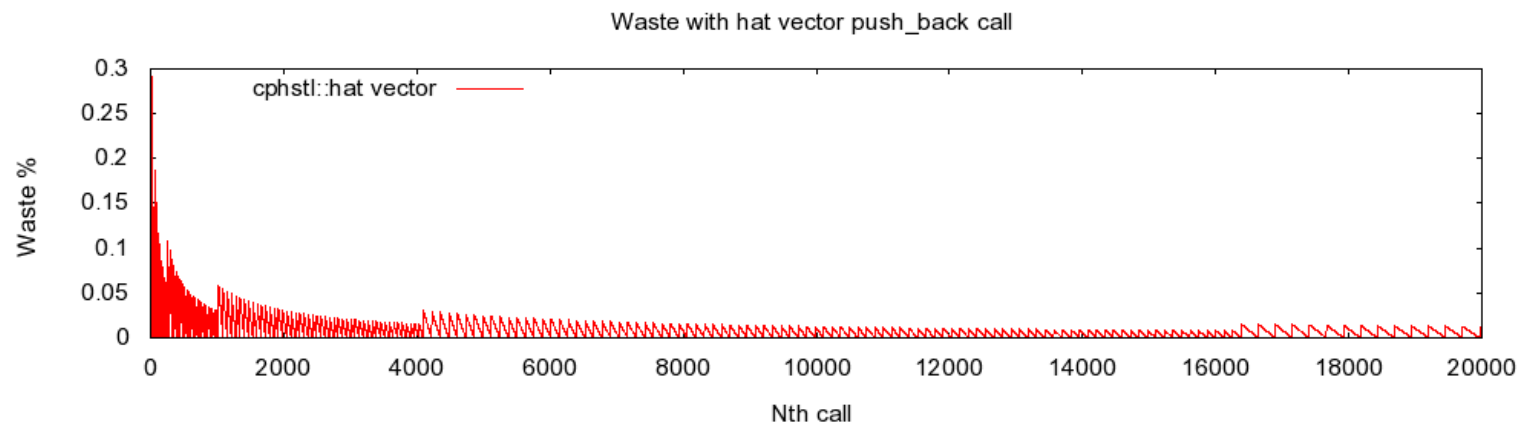
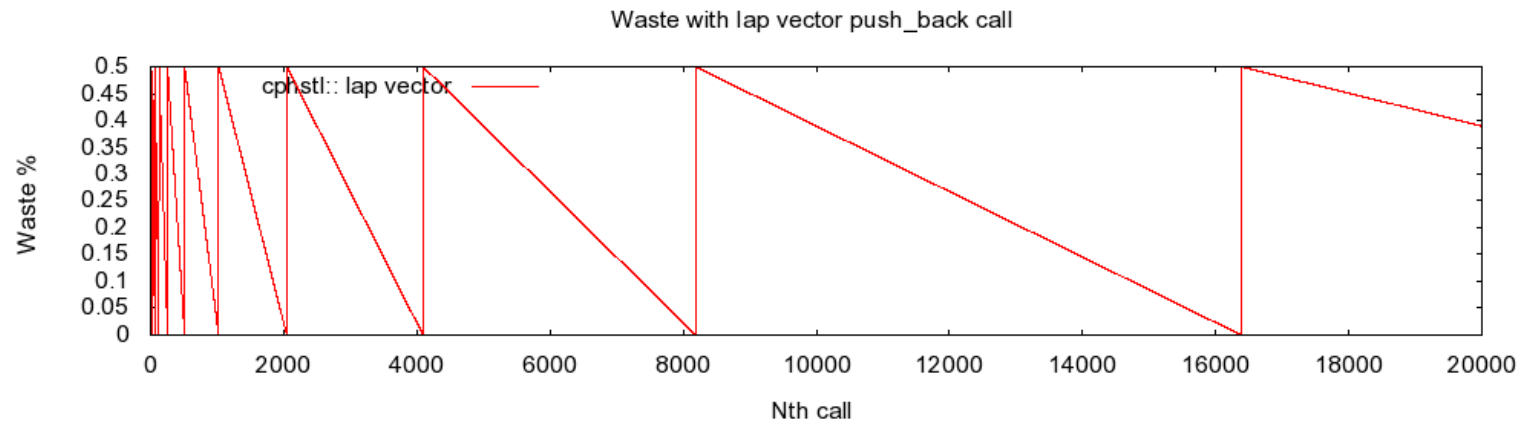
```
1 datatype int
2
3 driver function_call_count
4
5 cases
6   {10 15 .. 100}
7
8 structures
9   sorts bubble
10    title 'bubblesort'
11   sorts quick
12    title 'quicksort'
13   sorts selection
14    title 'selectsort'
15   sorts std
16    title 'std::sort'
17
18 benchmark
19   name test_std_sort
20   template comparisons
21   counted_function comp
22   plot_title "Comparison_calls_in_sort_algorithms."
23   filename sort_count
24
25 visualizer gnuplot
26   commands
27     set encoding iso_8859_1
28     set output "%(name)s.png"
29     set terminal png size 2000,800
30     set key vert left
31     set title '%(plot_title)s'
32     set data style lines
33     set xlabel 'Container_size'
34     set ylabel 'Count_calls_to_compare_function'
35
36 output
37   execute
38   cleanup
```

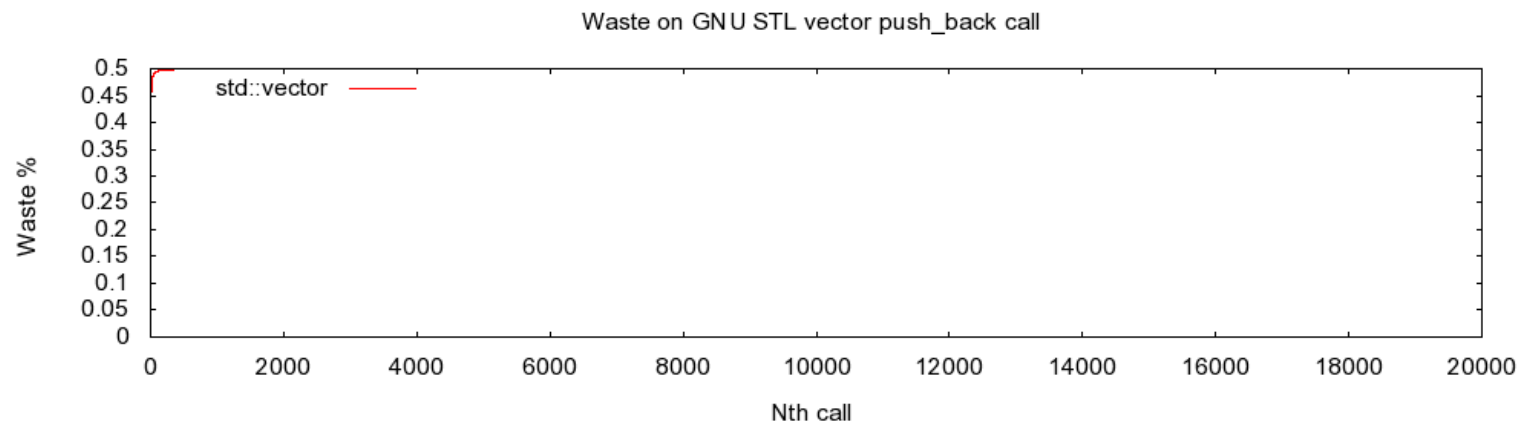
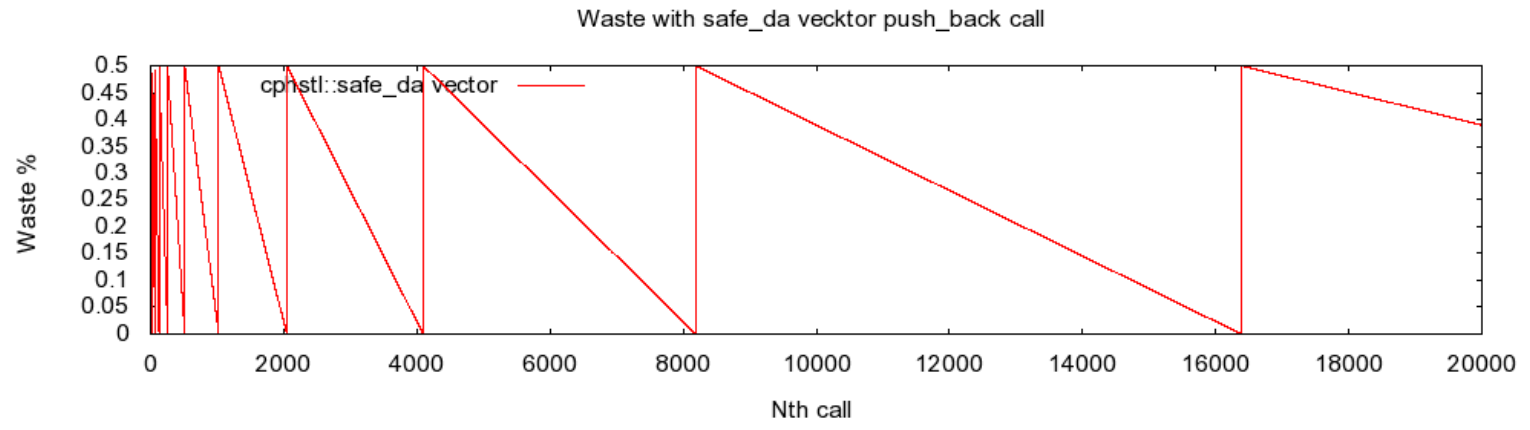
B. Testresultater

B.1 Resultat af tidsmåling for push_back

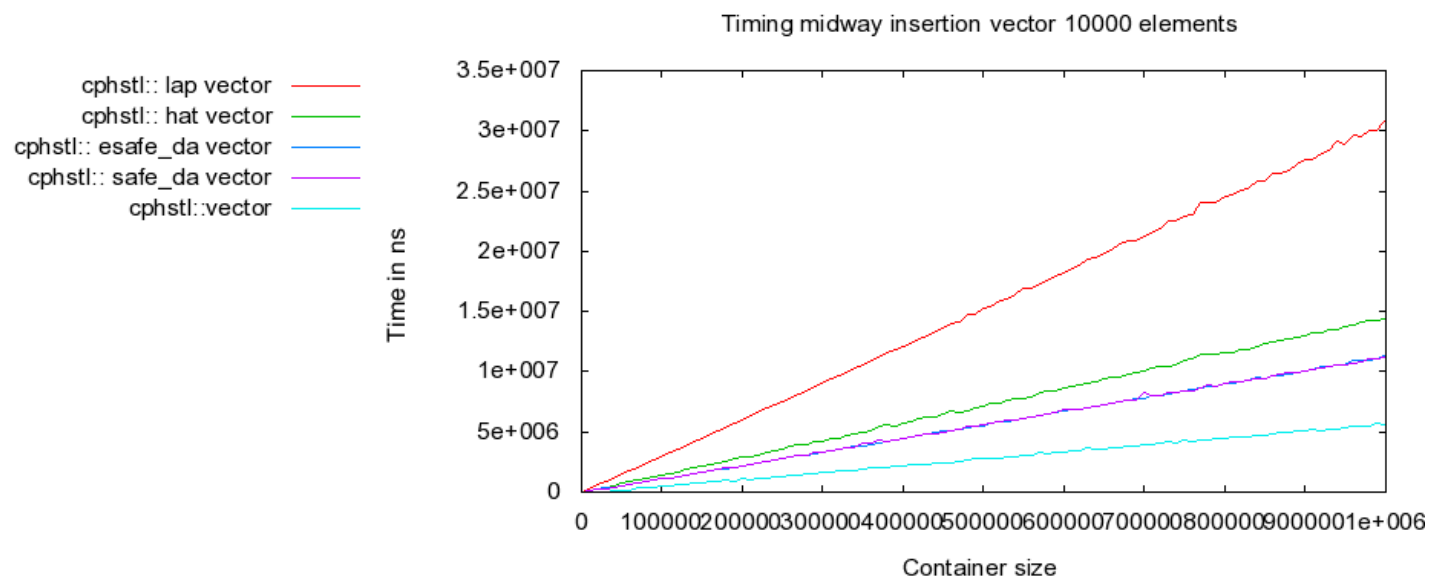


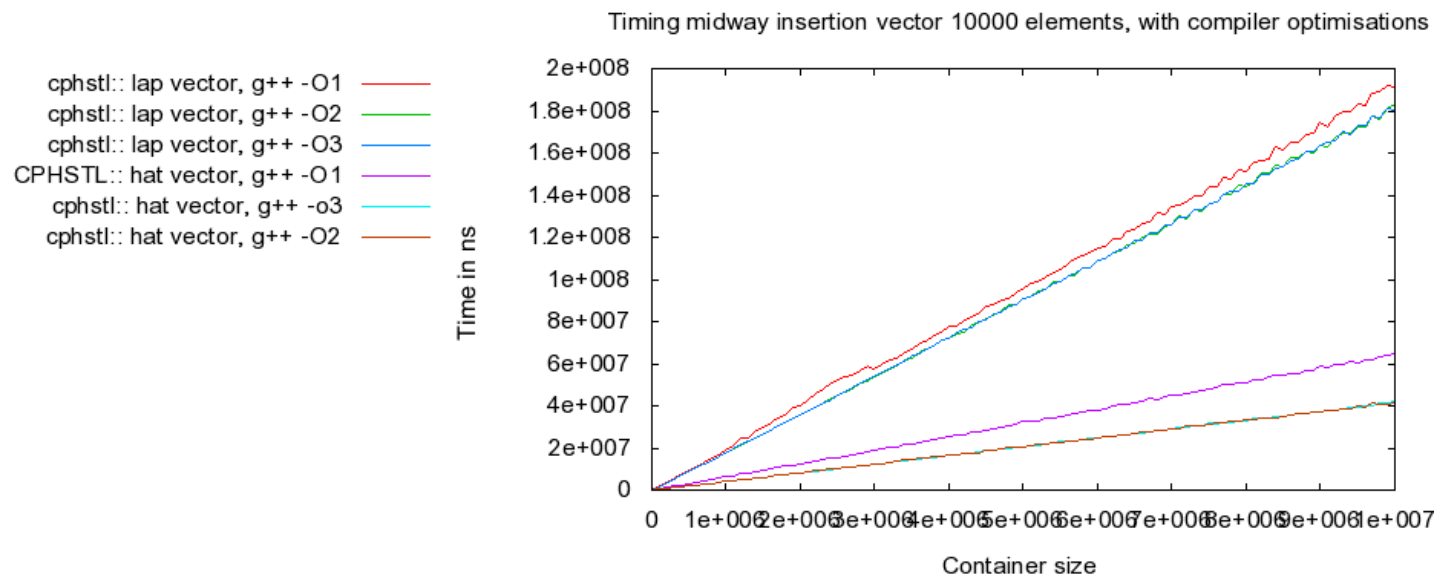




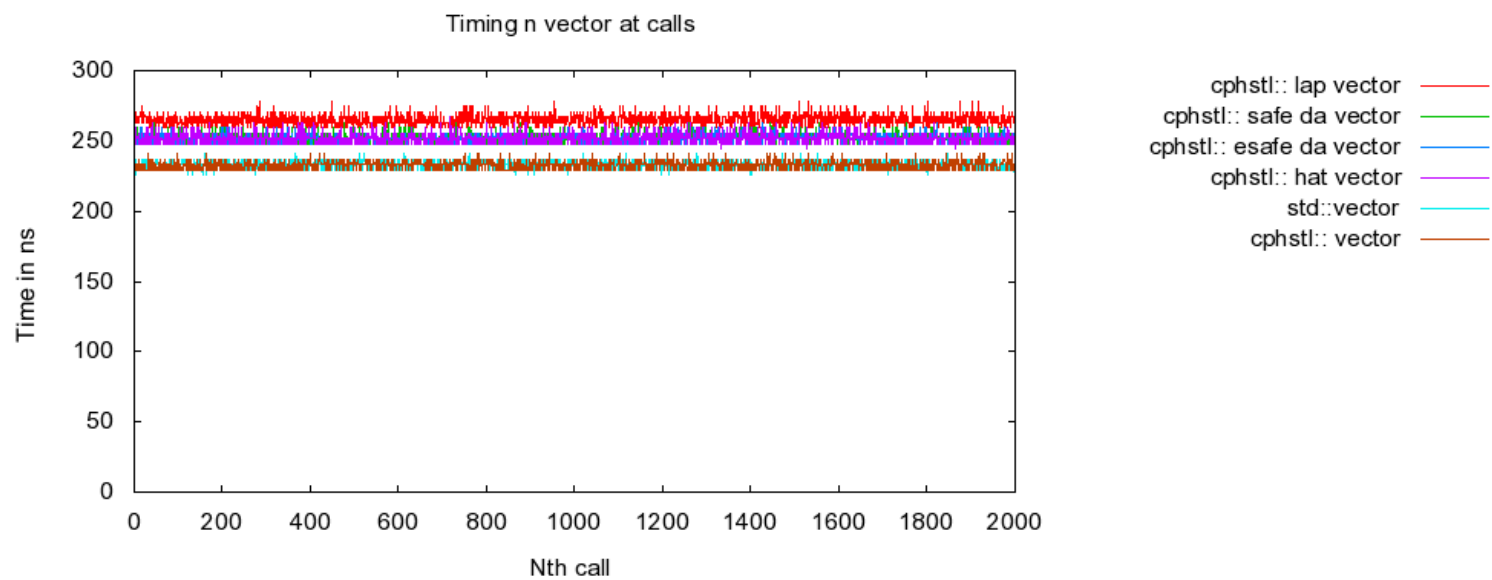


B.3 Resultat af tidsmåling for indsættelse i midten





B.5 Resultat af kald til at med $n - 1$ på en n vector



B.6 Resultat af unittest af vectorer

```

1  --main...dsl_unittest_plot
2  cphstl::vector
3  100.0% Empty Constructor.
4  100.0% Construct with empty allocator.
5  100.0% Construct with size argument.
6  100.0% Construct with size argument and value.
7  100.0% Construct with size argument,value and allocator.
8  100.0% Copy constructor.
9  100.0% Construct iterator range.
10 100.0% Construct iterator range and allocator.
11 100.0% Assignment.
12 100.0% Reserve.
13 100.0% Resize.
14 100.0% Resize with value.
15 100.0% Begin.
16 100.0% End.
17 100.0% Const begin.
18 100.0% Const end.
19 100.0% Rbegin.
20 100.0% Rend.
21 100.0% Const rbegin.
22 100.0% Const rend.
23 100.0% Push back.
24 100.0% Pop back.
25 100.0% Assign iterator range.
26 100.0% Assign size and value.
27 100.0% Insert before iterator value.
28 100.0% Insert before iterator size and value.
29 100.0% Insert iterator range.
30 100.0% Insert array range.
31 100.0% Erase.
32 100.0% Erase Iterator range.
33 100.0% Clear.
34 100.0% Swap structure.swap().
35 100.0% Global Swap swap(structure,structure).
36 100.0% Equality ==
37 100.0% In equality !=
38 100.0% Less than <
39 100.0% Greater than >
40 100.0% Less than or equal<=
41 100.0% Greater than or equal >=
42  std::vector
43 100.0% Empty Constructor.
44 100.0% Construct with empty allocator.
45 100.0% Construct with size argument.
46 100.0% Construct with size argument and value.
47 100.0% Construct with size argument,value and allocator.
48 100.0% Copy constructor.
49 100.0% Construct iterator range.
50 100.0% Construct iterator range and allocator.
51 100.0% Assignment.
52 100.0% Reserve.
53 100.0% Resize.
54 100.0% Resize with value.
55 100.0% Begin.
56 100.0% End.
57 100.0% Const begin.
58 100.0% Const end.
59 100.0% Rbegin.
60 100.0% Rend.
61 100.0% Const rbegin.
62 100.0% Const rend.
63 100.0% Push back.

```

```

64 | 100.0% Pop back.
65 | 100.0% Assign iterator range.
66 | 100.0% Assign size and value.
67 | 100.0% Insert before iterator value.
68 | 100.0% Insert before iterator size and value.
69 | 100.0% Insert iterator range.
70 | 100.0% Insert array range.
71 | 100.0% Erase.
72 | 100.0% Erase Iterator range.
73 | 100.0% Clear.
74 | 100.0% Swap structure.swap().
75 | 100.0% Global Swap swap(structure , structure).
76 | 100.0% Equality ==
77 | 100.0% In equality !=
78 | 100.0% Less than <
79 | 100.0% Greater than >
80 | 100.0% Less than or equal<=
81 | 100.0% Greater than or equal >=
82 |     cphstl:: safe_da vector
83 | 100.0% Empty Constructor.
84 | 100.0% Construct with empty allocator.
85 | 100.0% Construct with size argument.
86 | 100.0% Construct with size argument and value.
87 | 100.0% Construct with size argument, value and allocator.
88 | 100.0% Copy constructor.
89 | 100.0% Construct iterator range.
90 | 100.0% Construct iterator range and allocator.
91 | 100.0% Assignment.
92 | 100.0% Reserve.
93 | 100.0% Resize.
94 | 100.0% Resize with value.
95 | 100.0% Begin.
96 | 100.0% End.
97 | 100.0% Const begin.
98 | 100.0% Const end.
99 | 100.0% Rbegin.
100 | 100.0% Rend.
101 | 100.0% Const rbegin.
102 | 100.0% Const rend.
103 | 100.0% Push back.
104 | 100.0% Pop back.
105 | 100.0% Assign iterator range.
106 | 100.0% Assign size and value.
107 | 100.0% Insert before iterator value.
108 | 100.0% Insert before iterator size and value.
109 | 100.0% Insert iterator range.
110 | 100.0% Insert array range.
111 | 100.0% Erase.
112 | 100.0% Erase Iterator range.
113 | 100.0% Clear.
114 | 100.0% Swap structure.swap().
115 | 100.0% Global Swap swap(structure , structure).
116 | 100.0% Equality ==
117 | 100.0% In equality !=
118 | 100.0% Less than <
119 | 100.0% Greater than >
120 | 100.0% Less than or equal<=
121 | 100.0% Greater than or equal >=
122 |     cphstl:: esafe_da vector
123 | 100.0% Empty Constructor.
124 | 100.0% Construct with empty allocator.
125 | 100.0% Construct with size argument.
126 | 100.0% Construct with size argument and value.
127 | 100.0% Construct with size argument, value and allocator.
128 | 100.0% Copy constructor.

```

```
129 | 100.0% Construct iterator range.
130 | 100.0% Construct iterator range and allocator.
131 | 100.0% Assignment.
132 | 100.0% Reserve.
133 | 100.0% Resize.
134 | 100.0% Resize with value.
135 | 100.0% Begin.
136 | 100.0% End.
137 | 100.0% Const begin.
138 | 100.0% Const end.
139 | 100.0% Rbegin.
140 | 100.0% Rend.
141 | 100.0% Const rbegin.
142 | 100.0% Const rend.
143 | 100.0% Push back.
144 | 100.0% Pop back.
145 | 100.0% Assign iterator range.
146 | 100.0% Assign size and value.
147 | 100.0% Insert before iterator value.
148 | 100.0% Insert before iterator size and value.
149 | 100.0% Insert iterator range.
150 | 100.0% Insert array range.
151 | 100.0% Erase.
152 | 100.0% Erase Iterator range.
153 | 100.0% Clear.
154 | 100.0% Swap structure.swap().
155 | 100.0% Global Swap swap(structure,structure).
156 | 100.0% Equality ==
157 | 100.0% In equality !=
158 | 100.0% Less than <
159 | 100.0% Greater than >
160 | 100.0% Less than or equal <=
161 | 100.0% Greater than or equal >=
162 | cphstl:: hat vector
163 | 100.0% Empty Constructor.
164 | 100.0% Construct with empty allocator.
165 | 100.0% Construct with size argument.
166 | 100.0% Construct with size argument and value.
167 | 100.0% Construct with size argument,value and allocator.
168 | 100.0% Copy constructor.
169 | 100.0% Construct iterator range.
170 | 100.0% Construct iterator range and allocator.
171 | 100.0% Assignment.
172 | 100.0% Reserve.
173 | 100.0% Resize.
174 | 100.0% Resize with value.
175 | 100.0% Begin.
176 | 100.0% End.
177 | 100.0% Const begin.
178 | 100.0% Const end.
179 | 100.0% Rbegin.
180 | 100.0% Rend.
181 | 100.0% Const rbegin.
182 | 100.0% Const rend.
183 | 100.0% Push back.
184 | 100.0% Pop back.
185 | 100.0% Assign iterator range.
186 | 100.0% Assign size and value.
187 | 100.0% Insert before iterator value.
188 | 100.0% Insert before iterator size and value.
189 | 100.0% Insert iterator range.
190 | 100.0% Insert array range.
191 | 100.0% Erase.
192 | 100.0% Erase Iterator range.
193 | 100.0% Clear.
```

```
194 | 100.0% Swap structure.swap().
195 | 100.0% Global Swap swap(structure , structure).
196 | 100.0% Equality ==
197 | 100.0% In equality !=
198 | 100.0% Less than <
199 | 100.0% Greater then >
200 | 100.0% Less than or equal<=
201 | 100.0% Greater then or equal >=
202 |     cphstl:: lap vector
203 | 100.0% Empty Constructor.
204 | 100.0% Construct with empty allocator.
205 | 100.0% Construct with size argument.
206 | 100.0% Construct with size argument and value.
207 | 100.0% Construct with size argument, value and allocator.
208 | 100.0% Copy constructor.
209 | 100.0% Construct iterator range.
210 | 100.0% Construct iterator range and allocator.
211 | 100.0% Assignment.
212 | 100.0% Reserve.
213 | 100.0% Resize.
214 | 100.0% Resize with value.
215 | 100.0% Begin.
216 | 100.0% End.
217 | 100.0% Const begin.
218 | 100.0% Const end.
219 | 100.0% Rbegin.
220 | 100.0% Rend.
221 | 100.0% Const rbegin.
222 | 100.0% Const rend.
223 | 100.0% Push back.
224 | 100.0% Pop back.
225 | 100.0% Assign iterator range.
226 | 100.0% Assign size and value.
227 | 100.0% Insert before iterator value.
228 | 100.0% Insert before iterator size and value.
229 | 100.0% Insert iterator range.
230 | 100.0% Insert array range.
231 | 100.0% Erase.
232 | 100.0% Erase Iterator range.
233 | 100.0% Clear.
234 | 100.0% Swap structure.swap().
235 | 100.0% Global Swap swap(structure , structure).
236 | 100.0% Equality ==
237 | 100.0% In equality !=
238 | 100.0% Less than <
239 | 100.0% Greater then >
240 | 100.0% Less than or equal<=
241 | 100.0% Greater then or equal >=
```

B.7 Resultat af sorteringsalgoritme funktionstæller testen

