Sammendrag

Dette dokumentet beskriver flyttingen av BSPlab fra *Microsoft Visual Studio*-utviklingsplattformen til *GCC* på Linux og *MinGW* på Windows. BSPlab er et simuleringsverktøy som gjør det mulig å simulere BSP-programmer på en rekke forskjellige parallelle arkitekturer. Flyttingen ble gjort for å gjøre BSPlab tilgjengelig på ikke-proprietære utviklingsplattformer.

Dokumentet starter med å beskrive målsetningene og motivasjonen for prosjektet. Deretter gis noe bakgrunnsinformasjon til BSP og BSPlab. Etter dette kommer hoveddelen av dokumentet som beskriver hvordan vi først planla å gjøre denne flyttingen og så en steg for steg beskrivelse av prosessen. Det vises også hvordan vi implementerte installasjonsverktøy for begge plattformene og testing av sluttproduktet.

I tillegg utforsker vi muligheten for å benytte BSPlab fra Java. Dette er av interesse fordi det muliggjør at brukere som ikke har kompetanse på C/C++-programmering kan nyttiggjøre seg BSPlab.

Til slutt i dokumentet er det skrevet en bruksanvisning for installasjon av den nye BSPlab-pakken og en veiledning på hvordan man kan benytte og utvide BSPlab. De to siste delene er skrevet på engelsk fordi de bør være tilgjengelige for brukere fra hele verden.

Forord

Dette dokumentet representerer den endelig rapporten for diplomprosjektet, våren 2005 ved NTNU (Norges Tekniske-naturvitenskaplige universitet), IDI (Institutt for datateknikk og informasjonsvitenskap). Prosjektet har blitt gjennomført i løpet av 20 uker av Erik Østby og Torje Lundereng.

Vi vil benytte anledningen til å takke vår veileder, Lasse Natvig, for uvurdelig støtte og oppfølging gjennom hele prosjektet. Hans glød for prosjektet og holdning har vært til stor inspirasjon. Vi håper at BSPlab-prosjektet vil fortsette å leve videre også etter at vår jobb med det nå er ferdig.

Trondheim, torsdag 16. juni 2005.

Erik Østby
Torje Lundereng

Innhold

1 Innledning **1**
1.1 Målsetting . . . 1
1.2 Motivasjon . . . 1
1.3 Definisjoner . . . 2
1.4 Oversikt . . . 2

2 Bakgrunn **5**
2.1 BSP . . . 5
2.1.1 Supersteps og synkronisering . . . 6
2.1.2 BSP-parameterene . . . 6
2.1.3 BSP-funksjonene . . . 7
2.2 BSPlab . . . 7
2.3 C++Sim . . . 7

3 Fremgangsplan **9**
3.1 Gjennomgang og oppdatering av opprinnelig kode . . . 9
3.1.1 Valg av automake-verktøy . . . 10
3.2 Flytting til Linux/GCC og Windows . . . 11
3.3 Automatisk installasjon . . . 11
3.3.1 Automatisk installasjon på Windows . . . 11
3.3.2 Automatisk installasjon på Linux . . . 11
3.4 Testing . . . 12
3.5 Java-bindinger . . . 13
3.5.1 Krav til rammeverk for Java-bindinger . . . 13

4 Gjennomgang og oppdatering av opprinnelig kode **15**
4.1 Kompilering av BSPlab under Microsoft Visual Studio .Net . . . 15
4.2 Testing av BSPlab under Microsoft Visual Studio .Net . . . 16
4.2.1 Modifikasjoner til de opprinnelige batch-skriptene . . . 16
4.2.2 Kjøring av de nye batch-skriptene . . . 19
4.3 Enkle endringer av koden . . . 20
4.3.1 Benytte innebygd støtte for bolske verdier i kompilatoren . . . 20
4.3.2 Oppdatere include statements . . . 21

4.4 Oppgradering av C++Sim . . . 23
4.4.1 Nødvendige modifikasjoner til C++Sim . . . 23
4.4.2 Videre testing med ny C++Sim . . . 24

5 Flytting av BSPlab til ikke-proprietære plattformer 25
5.1 Valg av gratis kompilator for Windows . . . 25
5.1.1 Cygwin . . . 25
5.1.2 MinGW . . . 26
5.1.3 Endelig valg av kompilator . . . 26
5.2 Valg av automake verktøy . . . 26
5.2.1 Innledende undersøkelser . . . 27
5.2.2 Beskrivelse av automake-alternativene . . . 28
5.2.3 Evaluering av automake-verktøy . . . 31
5.3 Oppsett av CMake . . . 33
5.3.1 BSPlabs opprinnelige byggestruktur . . . 33
5.3.2 C++Sim . . . 33
5.3.3 BSPlab . . . 35
5.4 C++Sim på GCC 3.3 . . . 36
5.4.1 Feil bruk av friend-nøkkelordet . . . 36
5.4.2 Bytte fra gammel til ny "stream"-standard . . . 37
5.4.3 Bruk av den innbygde Boolean-typen . . . 38
5.5 Endringer i BSPlab til plattformuavhengige alternativer . . . 38
5.5.1 Bruk av __min og __max . . . 39
5.6 Funksjonalitet som måtte implementeres forskjellig på Windows og Linux . 39
5.6.1 Egen random-funksjon . . . 39
5.6.2 BSP_Platform_flushall() . . . 40
5.6.3 BSP_Platform_sleep(int msec) . . . 40
5.6.4 BSP_Platform_filelength(FILE* p) . . . 41
5.6.5 Plattformuavhengig timing-kode . . . 42
5.7 Feil i koden . . . 45
5.7.1 NOW-arkitekturen . . . 46
5.7.2 Viktigheten av virtuelle destruktorer . . . 47
5.8 Gjenstående problem på Linux . . . 49
5.9 BSPlab på MinGW . . . 50
5.9.1 MinGW MSYS . . . 50
5.9.2 BSPlab . . . 51
5.9.3 C++Sim . . . 51
5.10 Nye BSPlab på Visual Studio . . . 53
5.10.1 Eldre utgaver av Visual Studio . . . 53

6 Automatisk installasjon 55
6.1 Automatisk installasjon på Windows . . . 55
6.1.1 Inno Setup Compiler . . . 55
6.2 Automatisk installasjon på Linux . . . 58

6.2.1 Oversikt over installasjonsverktøy 58
6.2.2 Installasjonsprogrammet *bsplab_install* 60

7 Testing av nye BSPlab 63
7.1 Test av maks antall prosessorer 63
7.1.1 Maks antall prosessorer på Windows 64
7.1.2 Maks antall prosessorer på Linux 66
7.1.3 Stack-størrelsen i parameterfilen 69
7.2 Sammenlikning av tidsforbruk på Windows og Linux 74
7.2.1 Sanntidsprioritering 75
7.2.2 Testing av tidsforbruket 77
7.2.3 Andre testprogrammer 83

8 Java-bindinger 85
8.1 Java Native Interface 85
8.1.1 Kall til C++-funksjoner fra Java 85
8.1.2 Kall til Java-metoder fra C++ 88
8.2 Løsningsskisser 90
8.2.1 Krav til rammeverk for Java-bindinger 90
8.2.2 Løsningsskisse for Krav 2 90
8.2.3 Løsningsskisse 1 for Krav 1 og 3 91
8.2.4 Utprøvning av konseptene til Løsningsskisse 1 92
8.2.5 Løsningsskisse 2 for Krav 1 og 3 93
8.2.6 Utprøvning av Løsningsskisse 2 94
8.2.7 Raffinering av løsningsskissene 97
8.3 Flytting av Linux-varianten til Windows 102
8.3.1 Bruk av interface-klasse for å løse *back-linking* problemet 102
8.3.2 Finpuss av endelig løsning 105
8.4 Endelig resultat 106
8.4.1 Bruk av den endelige løsningen 107

9 Installation Howto 109
9.1 Windows installation 109
9.1.1 Default BSPlab installation with Dev-C++ and MinGW 110
9.1.2 Custom BSPlab installation (for user with i.e. Visual Studio .NET) 113
9.2 Linux installation 116
9.2.1 Systemwide BSPlab installation 116
9.2.2 User BSPlab installation 117

10 BSPlab Tutorial 119
10.1 How to run programs with BSPlab 119
10.1.1 The *Hello World* BSP test program 119
10.1.2 Running your own BSP programs in the simulator 120
10.1.3 Making your own Dev-C++ projects run in the simulator 121

10.1.4 The *Parameters.dat* file 122
10.2 Customizing BSPlab 122
10.2.1 Making a new distribution of BSPlab 123
10.2.2 BSPlab structure and CMake setup 123
10.2.3 Making the BSPlab distribution library 126
10.2.4 Customizing BSPlab for dummies 126
10.2.5 Running tests on the customized BSPlab simulator 129

11 Konklusjon **131**
11.1 Erfaringer 131
11.1.1 Erfaringer med programvarepakker 131
11.2 Videre arbeid 133
11.3 Konklusjon 133

A Installasjons-CD **135**
A.1 Installasjonsprogram for Windows 135
A.1.1 Automatisk oppstart av installasjonsprogrammet 135
A.2 Installasjonsprogram for Linux 136
A.3 HTML dokumentasjon 136
A.4 Referansefiler som ikke skal være en del av installasjons-CDen 136
A.4.1 Diplomrapporten 136
A.4.2 Skript-filen for installasjonsprogrammet 137
A.4.3 Forskjellige versjoner av BSPlab 137

B Kjente problemer med BSPlab **139**

C Testkode **141**
C.1 MergeSort 141
C.2 Parameterfilen 146

Bibliografi **151**

Kapittel 1

Innledning

1.1 Målsetting

Målet med oppgaven er å gjøre det mulig å benytte BSPlab med andre utviklingsverktøy enn *Microsoft Visual Studio*. Spesifikt er det ønskelig å gjøre BSPlab tilgjengelig på en gratis kompilator for Windows og for *GCC* på Linux. Det er også å foretrekke at BSPlab fortsatt vil fungere på nyere versjoner av *Microsoft Visual Studio*, slik at prosjekter som er utviklet her vil fortsette å fungere som de skal med få eller ingen modifikasjoner. Det er ønskelig å enkelt kunne installere BSPlab på alle plattformer via installasjons-programmer. I tillegg er det satt som mål å undersøke mulighetene for at BSPlab skal kunne benyttes fra andre programmeringsspråk enn C/C++.

1.2 Motivasjon

BSPlib har en enkel oppbygning med svært få funksjonskall i forhold til andre biblioteker for parallelle systemer (se seksjon 2.1.3 for mer om dette). Derfor er det ideelt å benytte i sammenheng med undervisning. BSPlab gjør det mulig å teste ut forskjellige algoritmer og programmer på en rekke parallelle arkitekturer og sammenlikne ytelsen. Tidligere har BSPlab kun vært tilgjengelig på den proprietære utviklingsplattformen *Microsoft Visual Studio 4.0*. Dette gjør det upraktisk å benytte BSPlab i forbindelse med undervisning siden studentene må ha tilgang til proprietær programvare for å nyttiggjøre seg av BSPlab. Motivasjonen for oppgaven er derfor å gjøre BSPlab tilgjengelig for utviklingsverktøy og plattformer som ikke er proprietære. Dette vil gjøre det mulig for alle som er interessert i å benytte seg av BSPlab å gjøre dette uten å måtte investere i dyr programvare. I tillegg vil det være en fordel dersom BSPlab kan benyttes fra andre språk enn kun C/C++. Slik folk som har erfaring fra andre programmeringsspråk også nyttiggjøre seg av BSPlab uten og måtte sette seg inn i C/C++.

1.3 Definisjoner

Her følger en beskrivelse av ofte brukte ord og uttrykk i denne rapporten:

GCC *GNU C(++) Compiler*. Den mest brukte C(++) kompilatoren på Linux.

MinGW *Minimalistisk GNU for Windows* [MinGW, 2005]. MinGW omfatter i utgangspunktet en stor del Unix-verktøy som har blitt portet til Windows, hvorav det mest kjente verktøyet nok er GCC. Denne Windows-porten av GCC omtales derfor ofte som MinGW, en praksis vi også vil bruke i denne rapporten.

pthreads *POSIX Threads*. Trådpakke som er vanlig å benytte på Unix. Dette er den foretrukne trådpakken å benytte på Linux.

automake-verktøy Programvare som benyttes til å konfigurere og gjøre klart et prosjekt til å bli kompilert og linket opp. Slike verktøy er svært nyttige for å enkelt kunne bygge et helt prosjekt med en enkel kommando.

BSPlib Biblioteket BSPlab er laget for å kunne simulere programmer til [BSPlib, 2005]. Nyeste versjon er 1.14, BSPlab er implementert til å støtte versjon 0.72 alpha.

Microsoft Visual Studio Proprietær utviklingsverktøy fra Microsoft. BSPlab er implementert til å benyttes med dette utviklingsverktøyet.

STL *Standard Template Library* Biblioteker som følger med alle nyere kompilatorer som inneholder en rekke praktiske klasser og funksjoner til å benytte med C++.

1.4 Oversikt

Kapittel 2 - Bakgrunn

I dette kapittelet blir det gitt bakgrunnsinformasjon for prosjektet. Først går vi gjennom hva BSP og BSPlab er, og hvordan det kan benyttes. Deretter introduserer vi C++Sim som er simuleringsmotoren BSPlab benytter.

Kapittel 3 - Fremgangsplan

Her går vi i kronologisk rekkefølge gjennom vår plan for hvordan vi skal klare å utføre oppgaven på en best mulig måte. Vi starter med å diskutere hvordan vi bør gå frem for å gjøre selve flyttingen og hvordan vi bør legge opp installasjon på både Windows og Linux. Til slutt introduserer vi hvordan vi ser for oss å lage bindinger mellom BSPlab og Java.

Kapittel 4 - Gjennomgang og oppdatering av opprinnelig kode

Dette kapittelet tar for seg hvordan vi gikk frem for å oppdatere BSPlab til å kjøre stabilt på *Microsoft Visual Studio .Net*. Deretter viser vi hvordan vi satte opp et testoppsett for å ha mulighet til å teste BSPlab gjennom hele flytteprosessen. Til slutt går vi gjennom oppgraderingen av C++Sim til nyeste versjon.

Kapittel 5 - Flytting av BSPlab til ikke-propiretære plattformer

I dette kapittelet viser vi hvordan vi fikk flyttet BSPlab fra *Microsoft Visual Studio* til *GCC* på Linux og *MinGW* på Windows. Det vises også hvordan vi gikk frem for å finne et godt *automake-verktøy* til å støtte byggeprosessen på begge plattformene.

Kapittel 6 - Automatisk installasjon

Her viser vi hvordan vi bestemte oss for å gjøre installasjonen av BSPlab på både Windows og Linux. Først går vi gjennom hvordan vi lagde en installasjons-*wizard* for Windows. Deretter viser vi hvordan vi lagde et installasjonsprogram for Linux som ikke bruker et grafisk brukergrensesnitt.

Kapittel 7 - Testing av nye BSPlab

I dette kapittelet viser vi først hvordan vi testet og klarte å maksimere antallet prosessorer BSPlab er i stand til å simulere på både Windows og Linux. Deretter gjør vi en test av tidsforbruket på begge plattformene. Det vises også hvordan vi gjorde det mulig å kjøre BSPlab med sanntidsprioritering i operativsystemet for å maksimere nøyaktigheten til tidsmålingene.

Kapittel 8 - Java-bindinger

Her går vi gjennom hvordan vi gikk frem for å finne en løsning for å benytte BSPlab fra Java. Det lages til slutt et *proof-of-concept* som viser med et enkelt eksempel hvordan dette er mulig.

Kapittel 9 - Installation Howto

En installasjonsveiledning for hvordan man legger inn BSPlab på både Windows og Linux. Kapittelet er skrevet på engelsk fordi det kan være av interesse for brukere av BSPlab fra hele verden.

Kapittel 10 - BSPlab Tutorial

En introduksjon til hvordan man kan benytte BSPlab til å simulere BSP-programmer. Det vises også hvordan man skal gå frem for å gjøre egne modifikasjoner og utvidelser til selve BSPlab. Kapittelet er skrevet på engelsk fordi det kan være av interesse for brukere av BSPlab fra hele verden.

Kapittel 11 - Konklusjon

En oppsummering av prosjektet med våre egne erfaringer og videre arbeid som kan gjøres på BSPlab.

Tillegg A - Installasjons-CD

En oversikt over hva som finnes på CDen som følger med rapporten.

Tillegg B - Kjente problemer med BSPlab

En oversikt over problemer med BSPlab som er identifisert, men ikke fikset ved prosjektets slutt.

Tillegg C - Testkode

Koden og parameterfilen som ble brukt til å kjøre testene i kapittel 7.

Kapittel 2

Bakgrunn

I dette kapittelet vil det bli gitt bakgrunnsinformasjon for prosjektet. Vi har valgt å ikke fokusere så mye på alle detaljene rundt hvordan BSP fungerer, men heller på de delene som er relevant for hvorfor dette prosjektet kan være nyttig. En svært god introduksjon til BSP gis i den originale BSPlab-rapporten [Uthus and Dybdahl, 1997]. For lesere som er interessert i en god innføring i BSP vil vi anbefale å lese kapittel 2 og 3 i denne rapporten. Vi gir alikevel en kort innføring for helhet.

2.1 BSP

Ved programmering på parallelle arkitekturer er det et vanlig problem at det er vanskelig å skrive algoritmer som utnytter forskjellige parallelle plattformer godt. Ofte løses dette ved å lage en abstraksjon mellom selve programmet og den parallelle arkitekturen det kjører på. For å kunne gjøre en slik abstraksjon trenger man en modell som generelt beskriver parallelle plattformer. En slik modell, kalt *the Bulk Synchronous Parallel model* (BSP), ble foreslått i 1990 av Valiant [Valiant, 1990]. Modellen antar at en parallell datamaskin består av følgende deler:

- **Prosessorer**. Hver enkelt prosessor har lokalt minne.
- **Kommunikasjonssystem.** Dette gjør det mulig for prosessorer å aksessere ikke-lokalt minne, for eksempel minnet til en annen prosessor.
- **Synkroniseringsmekanisme.** Gjør det mulig å garantere at alle prosessorer har nådd et bestemt punkt i programmet før noen prosessor fortsetter videre. Denne mekanismen trenger ikke være en del av den parallelle datamaskinen men kan implementeres i software ved å benytte kommunikasjonssystemet.

2.1.1 Supersteps og synkronisering

Et BSP-program er delt inn i ett eller flere *supersteps* som er splittet opp av *barriere synkroniseringer*. Et *superstep* kan bestå av:

- En eller flere beregninger, der hver prosessor kun bruker lokale variabler.
- Sending av meldinger fra hver prosessor til andre prosessorer.
- En *barriere synkronisering*.

På slutten av et *superstep* blir mottatt ekstern data tilgjengelig som lokale data på hver prosessor i neste *superstep*.

Barriere synkroniserings-mekanismen sørger for at alle prosessorer har kommet like langt i programmet. Dette kan føre til at noen prosessorer må vente på andre før de kan jobbe videre. Hvor effektivt maskinen klarer å implementere *barriere synkroniserings*-mekanismen har stor innflytelse på hvor effektivt maskinen klarer å kjøre BSP-programmer som er delt opp i mange *supersteps*. Denne implementasjonen kan gjøres både i hardware og software.

2.1.2 BSP-parameterene

BSP-modellen har fire parametere, p, s, l og g, som karakteriserer en spesifikk parallell datamaskin. Disse parameterene sier noe om ytelsen til maskinen og kan benyttes til å forutsi hvordan en konkret algoritme vil kjøre på den. [McColl, 1990] definerer de fire parameterene på følgende måte:

- **p - antall prosessorer.** Som oftest er l og g en funksjon av denne parameteren.
- **s - prosessorhastigheten.** Et mål for antall *time steps* per sekund prosessorene kan utføre.
- **l - nettverksforsinkelsen.** Et mål for det minimale antallet *time steps* mellom etterfølgende synkroniseringer.
- **g - global communication ratio.** Denne beregnes som det totale antallet operasjoner utført av alle prosessorene på et sekund delt på det totale antallet *ord* som er levert av kommunikasjonsnettverket på et sekund.

Parameterene p, l og g spenner ut et rom der enhver BSP-maskin befinner seg i et punkt, mens parameteren s skalerer de tre andre parameterene. For en mer detaljert beskrivelse av BSP og disse parameterene, se [Uthus and Dybdahl, 1997].

2.1.3 BSP-funksjonene

I BSP-standarden som er definert i *BSP Worldwide Standard Library* [BSP Worldwide, 2005] finnes det rundt 20 BSP-funksjoner som benyttes til å styre BSP-programmet. Dette er et svært lavt antall funksjoner sammenliknet med andre bibliotek for parallell programmering. Til sammenlikning har for eksempel det populære *Message Passing Interface*-biblioteket (MPI) rundt 125 funksjoner. Dette gjør BSP interessant for undervisning fordi studentene ikke er nødt til å sette seg inn i et stort antall funksjoner for å nyttegjøre seg av BSP.

2.2 BSPlab

BSPlab ble utviklet av Ivan Uthus og Haakon Dybdahl i deres diplomoppgave ved NTNU i 1997 [Uthus and Dybdahl, 1997]. BSPlab er en simulator som gjør det mulig å teste ytelsen til et BSP-program på en rekke forskjellige plattformer. Dette gjør BSPlab interessant både ved utvikling av BSP-programmer og til undervisning der studentene får mulighet til å se hvordan forskjellige plattformer påvirker ytelsen til programmet.

I årene etter at BSPlab ble utviklet er det blitt kjørt flere prosjektoppgaver basert på dette arbeidet. Blant annet ble det i 1998 gjort en grundig gjennomgang og testing av BSPlab av Eirik Lilleaas [Lilleaas, 1998] og i 2004 ble det gjort en flytting av BSPlab til Linux [Sund, 2004]. Spesielt oppgaven der BSPlab ble flyttet til Linux var svært aktuell i forhold til vårt arbeid. Denne oppgaven nådde bare delvis de målene som var satt, men den gjorde flere observasjoner som var nyttige for oss.

2.3 C++Sim

BSPlab er bygget oppå en simuleringspakke som heter C++Sim [C++SIM, 1997]. Denne pakken benytter tråder til å holde orden på forskjellige prosesser. Selve simulasjonen kjører ikke i parallell, det er til enhver tid bare én tråd som kjører. Grunnen til at C++Sim benytter tråder er rett og slett at det fungerer greit som et verktøy for å holde orden på prosesser og deres variabler. C++Sim støtter i utgangspunktet en rekke plattformer og trådpakker. I starten av prosjektet fikk vi inntrykk av at valg av trådpakke kunne være et sentralt punkt i flyttingen av BSPlab til andre plattformer, men dette viste seg å ikke være tilfelle. Grunnen til dette er at C++Sim er satt opp til å benytte *NT-threads* dersom det blir kompilert på Windows og *pthreads* dersom det blir kompilert på Linux. Vi så ingen grunn til å endre dette oppsettet.

Kapittel 3

Fremgangsplan

For å nå målsettingene som er satt for oppgaven trenger vi en plan for hvordan vi skal gå frem. I dette kapittelet vil vi i kronologisk rekkefølge gå gjennom vår plan for hvordan vi ser for oss at vi kan komme frem til et best mulig resultat. For hver av seksjonene i dette kapittelet finnes det et kapittel senere i rapporten som beskriver hvordan vi faktisk gikk frem og hva resultatet ble.

3.1 Gjennomgang og oppdatering av opprinnelig kode

Koden til BSPlab ble skrevet i 1997 for *Microsoft Visual Studio 4.0*. Denne kompilatoren er etter dagens standard utdatert og hadde blandt annet ikke skikkelig støtte for *Standard Template Library* (STL). En konsekvens av dette er at BSPlab benytter en tredjeparts implementasjon av STL. Det er ønskelig for oss å kvitte oss med dette biblioteket og heller benytte standardbiblioteket som følger med alle dagens kompilatorer. I tillegg bør koden gjennomgås for å forsikre oss om at det benyttes korrekt C++ i forhold til dagens standarer. Dette anses som nødvendig for å gjøre det enklest mulig å vedlikeholde koden i fremtiden.

BSPlab benytter C++Sim som motor for simuleringen. Nyeste versjon av dette biblioteket er 1.7.4 [C++SIM, 1997]. Dette er en litt nyere versjon enn versjon 1.6 som opprinnelig fulgte med BSPlab. I varianten av C++Sim som fulgte med BSPlab er det gjort en del endringer for å fikse feil. Det er ønskelig å benytte oss av nyeste versjon av C++Sim ettersom denne inneholder noen feilrettinger som kanskje kan ha positiv effekt for BSPlab. Det blir derfor nødvendig for oss å identifisere de problemene som ble fikset i C++Sim-varianten som følger med BSPlab og å gå gjennom versjon 1.7.4 for å sjekke om det er nødvendig med tilsvarende endringer der.

For å forsikre oss om at vi ikke gjør noen feil når vi gjør disse endringene vil vi med jevne mellomrom kjøre gjennom testene som følger med BSPlab for å sjekke at disse fortsatt fungerer som de skal. Vi vil derfor på dette stadiet konsentrere oss om å få BSPlab til å kjøre på *Microsoft Visual Studio .Net*. I kapittel 4 går vi gjennom hvordan vi gjennomførte

oppdateringen av den opprinnelige koden.

3.1.1 Valg av automake-verktøy

For å kompilere og linke et såpass stort prosjekt som BSPlab, med alle tilhørende komponenter, er det en stor fordel å ha et automatisk oppsett der man ved hjelp av en enkel kommando kan bygge hele prosjektet. Det finnes flere verktøy man kan benytte seg av for å få til dette. Slike verktøy kalles typisk for *automake-verktøy* fordi de automatiserer prosessen å "make" (bygge) prosjektet. Vi bør finne og velge et automake-verktøy som gjør det enkelt å få kompilert opp all kode på riktig måte på alle plattformer. Det vil si at det skal være enkelt å legge til eller fjerne kodefiler på en uniform måte. Det er også viktig at verkøyet er gratis i og med at litt av poenget med dette prosjektet er å frigjøre BSPlab fra proprietær programvare. I seksjon 5.2 går vi gjennom hvordan vi gjorde valget av automake-verktøyet.

Et automake-verktøy fungerer vanligvis på den måten at det er uavhengig av plattform og kompilator. Klassiske *Makefiler* som brukes for å beskrive hvilke filer som skal kompileres med hvilken kompilator vil typisk bare fungere på den maskinen hvor filen ble skrevet. Problemet er at navnet på kompilatoren fort kan skifte fra f.eks. *gcc* til *gcc-3.3* til og med om man holder seg på samme type operativsystem. Dette gjelder også navnene på bibliotekene man linker mot. I tillegg er det andre hensyn som må tas. Kanskje må man linke mot et ekstra bibliotek på en spesiell plattform eller kanskje er det viktig å vite om plattformen er *big-endian* eller *little-endian*. Overordnet fungerer de fleste automake-verktøy på den måten at man skriver en uavhengig byggefil som beskriver hvilke filer som skal kompileres. I tillegg kan en spesifiere en rekke sjekker som skal gjøres på èn bestemt, eller flere plattformer. Når man så kjører automake-verktøyet med utgangspunkt i denne uavhengige byggefilen vil det generere en datamaskin- og plattformavhengig *Makefile*. Automake-verktøyet vil selv prøve å finne ut (med mindre man spesifiserer noe annet selv) navnet på kompilatoren, navnet på forskjellige bibliotek som programmet er avhengig av å linkes mot, og f.eks. om mål-plattformen er *big-endian* eller *little-endian*.

De tingene vi legger mest vekt på ved valg av et automake-verktøy vil være:

1. Hvor god støtten er for Linux og Windows.
2. Hvor godt verktøyet fungerer med forskjellige *Integrated Development Environments* på Linux og Windows.
3. Hvor enkelt det er å lage et prosjekt med verktøyet.
4. Hvor enkelt det er å vedlikeholde et prosjekt med verktøyet.
5. Hvor godt verktøyet er dokumentert.

3.2 Flytting til Linux/GCC og Windows

Neste steg vil være å få BSPlab til å kjøre stabilt på GCC-kompilatoren for Linux. Til dette arbeidet vil vi dra nytte av erfaringene som ble gjort i et tidligere prosjekt som tok for seg å flytte BSPlab til Linux [Sund, 2004]. I dette prosjektet ble det rapportert om en rekke problemer både med å få C++Sim til å kjøre på nyere versjoner av GCC og med å få BSPlab til å kjøre stabilt. Disse problemene vil vi måtte se nærmere på og løse på en tilfredstillende måte. Til slutt vil vi flytte den nye versjonen av BSPlab tilbake på en gratis kompilator for Windows. I kapittel 5 går vi gjennom hvordan vi gikk frem for å løse flyttingen til de nye plattformene.

3.3 Automatisk installasjon

Det er viktig at brukeren ikke skal trenge å ha kunnskap om hvordan alle delene av BS-Plab henger sammen. Et mål for prosjektet er at brukeren enkelt skal kunne installere de nødvendige komponentene for å ta i bruk BSPlab uansett hvilken plattform brukeren benytter.

3.3.1 Automatisk installasjon på Windows

Vi må lage et installasjonsprogram for Windows som helst bør være på en standard form som brukere av Windows er vant med. Alle utviklingsverktøy som trengs for å kjøre og kompilere BSPlab-programmer skal følge med vår distribusjon av BSPlab. På denne måten blir alt klart til bruk etter installasjon uten krav om at brukeren må kjøpe eller hente ned egne programmer.

Installasjon på Windows bør følgende egenskaper:

1. Installasjonsprogrammet skal enkelt kunne startes av brukeren.
2. Programmet skal være utformet som en *Wizard* der brukeren enkelt klikker seg gjennom noen dialoger for å installere BSPlab.
3. Det skal være satt et standardvalg til det mest vanlige oppsettet slik at de fleste brukere ikke trenger gjøre noen endringer på hvilke komponenter som er valgt.

I seksjon 6.1 går vi gjennom hvordan vi laget en god løsning for installasjon av BSPlab på Windows.

3.3.2 Automatisk installasjon på Linux

Ettersom installasjoner på Linux/Unix typisk ikke gjøres på samme måte som på Windows, vil vi ikke gå inn for å bruke samme grensesnitt for installasjon på de forskjellige plattfor-

mene. På Linux er det vanlig at installasjonen gjøres forskjellig avhengig av om brukeren er *root* (administrator) eller en vanlig bruker. Installasjonsprogrammene bruker også generelt å være tekstbasert slik at man skal kunne installere programmet på et system som ikke har grafiske komponenter installert. Vi velger å holde oss til Unix-standarden og planlegger å lage et tekstbasert installasjonsprogram som støtter å installeres globalt på systemet hvis man er root, eller på brukerens hjemmeområde dersom man er en vanlig bruker.

På Linux er det upraktisk å sende med ferdigkompilerte bibliotek. Grunnen til dette er at mange Linux-installasjoner har helt forskjellige versjoner av GCC installert. Et bibliotek kompilert opp og linket mot en versjon av GCC-bibliotekene vil ikke nødvendigvis fungere så bra mot en annen versjon. Heldigvis kommer de fleste Linux-distribusjoner med utviklingsverktøyene ferdig installert (og om de ikke skulle være installert, er det enkelt for brukerne å legge de inn). Således kan vi enkelt kompilere opp bibliotekene uten å ha med utviklingsverktøy for Linux med vår distribusjon av BSPlab.

Installasjonsprogrammet bør i utgangspunktet sjekke systemet for følgende før installasjonen:

1. Sjekke hvilken bruker som kjører installasjonen.
2. Sjekke om utviklingsverktøy er installert på systemet.
3. Sjekke om automake-verktøyet som trengs er installert på systemet.

Punkt *1* brukes til å bestemme hvilken type installasjon som skal utføres. Hvis brukeren er *root* installeres programvaren globalt, ellers installeres den bare på brukerens hjemmeområde. Punkt *2* avgjør om vi kan kompilere opp bibliotekene som trengs. Til slutt, i punkt *3* avgjøres det om vi må installere automake-verkøyet fra vår distribusjon av BSPlab. Vi stiller ikke krav til at brukeren må ha dette verkøyet installert fra før på systemet sitt, ettersom det ikke er sikkert at vi kommer til å bruke Unix sitt tradisjonelle automake-verktøy.

3.4 Testing

Det er viktig å få testet BSPlab godt på de nye plattformene. Ettersom oppgaven har konsentrert seg om å få BSPlab til å kjøre på en gratis kompilator på Windows og GCC på Linux vil vi fokusere testingen på disse to plattformene. Vi vil teste om resultatene er like og om tidsforbruket til BSP-programmer er likt på begge plattformer. Vi vil forsøke å finne andre BSP-programmer enn de som er en del av testprogrammene som fulgte med BSPlab for å teste disse. Til slutt vil vi forsøke å teste hvor store systemer BSPlab er i stand til å simulere ved å se hvor mange prosessorer BSPlab maksimalt klarer å benytte. I kapittel 7 viser vi hvordan vi gikk frem for å finne gode tester og hva resultatene ble.

3.5 Java-bindinger

Som en utvidelse til oppgaven ble vi bedt om å undersøke muligheten for å få laget et grensesnitt som gjør det mulig å bruke BSPlab i Java. Dette er interessant ettersom Java er et populært programmeringsspråk å benytte i forbindelse med undervisning.

Ordet bindinger i programmeringsspråksammenheng gir blant annet uttrykk for et språks mulighet til å laste et annet språks funksjoner for så å kalle disse. I dette tilfellet vil det altså bli snakk om å laste og kalle C++-funksjoner. Ettersom C++ blir kompilert om til maskinkode er det en smal sak å laste disse i minnet på datamaskinen og få kalt funksjonene fra Java. Utfordringene ligger i å få konvertert Java sine datatyper til datatyper som kan brukes i C++-funksjonen som skal bli kalt.

Siden Java-programmer ikke kompileres til maskinkode slik som C++-programmer, kan ofte programmer kodet i Java være en del treigere. Da *SUN Microsystems* laget Java hadde de dette i bakhodet og laget et eget grensesnitt for å gjøre det enklere for programmere å laste C/C++-biblioteker, slik at maskinkompilerte funksjoner kan bli kalt for å gjøre ressurskrevende algoritmer. Dette grensesnittet kalles for *Java Native Interface* (JNI) [JNI, 2005] For en kort innføring i JNI-grensesnittet se kapittel 8.1.

3.5.1 Krav til rammeverk for Java-bindinger

Vi stiller følgende krav til rammeverket:

1. Bør ikke kreve noen forandringer på selve BSPlab.
2. Grensesnittet mot BSP-funksjonene bør ligne mest mulig på C-grensesnittet.
3. Det bør være ukomplisert å få kjørt BSP-simulering av Java-programmer.
4. Må kunne kjøre både på Windows og Linux.

Vi stiller krav *1* for å holde vedlikeholdbarheten til BSPlab høyest mulig. Hvis en løsning for Java-bindinger krever forandringer på selve BSPlab, blir man nødt til å vedlikeholde to varianter av biblioteket. Dette kan ende opp med at feil blir rettet i den ene varianten, men ikke i den andre. For å forhindre et slikt utfall kreves det at løsningen for Java-bindinger bruker det samme C++-biblioteket som også brukes til C/C++-simuleringen.

Årsaken til krav *2* er at det er ønskelig med et uniformt grensesnitt selv når man introduserer en API til et nytt programmeringsspråk. Dette fordi det er en fordel om man kan programmere C/C++ BSP-programmer hvis man først har lært seg Java-varianten eller motsatt. Syntaktisk vil nok ikke dette kravet skape store problemer ettersom Java ligner syntaktisk ganske mye på C/C++. BSP-funksjonene vil ikke kunne bli kalt med nøyaktig samme semantikk som i C/C++ fordi Java ikke har støtte for brukerdefinerte funksjoner i det globale navnerommet (alle brukerdefinerte funksjoner må være knyttet til en klasse) [Flanagan, 1996].

Krav *3* skal gjøre det mulig for en Java-bruker å ta i bruk BSP-simuleringen uten noen spesiell teknisk kompetanse utover Java-programmering. Vi ser for oss at den beste løsningen er at brukeren programmerer og kompilerer Java-programmet sitt ved hjelp av samme verktøy som han er vant med. Deretter kan det kjøres i simulatoren ved hjelp av én enkel kommando.

Siden selve oppgaven for BSPlab i utgangspunktet kort sagt var å få den til å kjøre på Windows òg Linux, må det være et poeng at Java-bindingene også kan kjøres på begge disse plattformene. Krav *4* skal altså gjenspeile at det bare skal være ett kodetre (ett for både Windows og Linux, ikke ett for hver) for Java-bindingene slik at vedlikeholdbarheten ikke blir redusert.

Kapittel 4

Gjennomgang og oppdatering av opprinnelig kode

Før vi begynte å gjøre noen endringer på den opprinnelige koden til BSPlab var det viktig å få kjørt gjennom testene for å se hva som fungerte i utgangspunktet. Siden vi ikke hadde tilgang til *Microsoft Visual Studio 4.0*, som den opprinnelige koden var skrevet for, måtte vi forsøke å få BSPlab til å kompilere på den versjonen vi hadde tilgang til; *Microsoft Visual Studio .Net.*

4.1 Kompilering av BSPlab under Microsoft Visual Studio .Net

Vi lot VS.Net konvertere den gamle prosjektfilen fra VS 4.0 slik at vi kunne åpne prosjektet. Vi fikk mange *deprecated* advarsler fra kompilatoren første gang vi prøvde å kompilere. Disse advarslene blir ikke gitt fordi noe er direkte galt i koden, men benyttes av kompilatoren til å informere om at headerfilene som blir inkludert er gått ut på dato og at det ikke er noen garanti for at de kommer til å fungere i fremtiden. All koden kompilerte korrekt til tross for disse advarslene, men vi fikk problemer da vi skulle linke programmet. Problemet skyltes en *undefined reference* til funksjonen `set_new_handler` i tredjeparts-implementasjonen av STL. Denne funksjonen benyttes til å sette en funksjon som skal kalles dersom operativsystemet ikke er i stand til å allokere mer minne til programmet. Istedet for å gå løs på å fjerne tredjeparts-implementasjonen av STL valgte vi å løse problemet ved å forsøke å finne en annen funksjon som var tilgjengelig med samme funksjonalitet som den originale. Vi søkte i *Microsoft Software Developer Network* databasen [MSDN, 2005] og fant en funksjon som skal benyttes istedet for den utgåtte `set_new_handler`. Vi endret koden til å benytte den korrekte funksjonen som heter `_set_new_handler`. Etter denne endringen linket programmet korrekt og vi fikk en kjørbar versjon av BSPlab.

4.2 Testing av BSPlab under Microsoft Visual Studio .Net

Neste skritt var å teste om versjonen av BSPlab vi nå hadde fått til å kompilere og linke ville fungere på testprogrammene slik vi hadde fått dokumentert at det gjorde under VS 4.0. Det fulgte med to *batch-skript* som skulle automatisere kjøringen av disse testene. Det viste seg at disse skriptene ikke fungerte så godt under Windows XP. Syntaksen som ble brukt i skriptene var ikke kompatible med syntaksen Windows XP benytter. Vi stod nå ovenfor valget om vi skulle prøve å fikse skriptene slik at vi kunne bruke de eller om vi skulle forsøke å lage nye skripts i et annet, mer fleksibelt språk, slik det ble gjort i [Sund, 2004]. Siden oppgavene vi skulle ha skriptene til å utføre er relativt enkle, og Windows XP sitt batch-språk er kraftig nok til å utføre dem, valgte vi å kun skrive om de opprinnelige skriptene. Vi så ikke noe poeng i å dra inn en annen skript-pakke i prosjektet kun til dette formålet.

4.2.1 Modifikasjoner til de opprinnelige batch-skriptene

Batch-skriptene var skrevet for 4DOS [4DOS, 2005]. Scriptspråket som er tilgjengelig på 4DOS har en annen syntaks enn det Windows XP støtter, så det ble nødvendig å skrive om skriptene litt for å få de til å fungere for oss.
Helt konkret var det tre ting som måtte endres på når det gjelder syntaks:

1. I skriptet benyttes variabler uten å escape %-tegnet. Dette fungerer ikke under Windows XP og løsningen ble å bytte ut % med %% der slike variabler benyttes.

2. For-løkker som skal iterere over innholdet i en fil var satt opp på formen “for %f (@file)”. Dette fungerte ikke under Windows XP, men samme funksjonalitet ble gitt av å bytte ut med “for /F %%f (file)”.

3. Windows XP støtter ikke alias og unalias så vi byttet ut alle stedene dette ble brukt med en tilsvarende kommando.

Det var i utgangspunktet to skript-filer som ble benyttet til å kjøre tester; *BSPMake.bat* og *BSPTest.bat*. Oppgaven til disse skriptene var å kjøre alle kombinasjoner av ferdig oppsatte parameterfiler og testprogrammer. Vi tok utgangspunkt i de kombinasjonene av parametere som ble brukt i testoppsettet i [Lilleaas, 1998]. Her var det satt opp 25 ulike oppsett av parametere som alle ble testet. Vi gikk gjennom dette oppsettet og lagde 25 forskjellige parameterfiler som hver inneholdt et oppsett som var identisk med det som ble gjort i [Lilleaas, 1998].

BSPMake.bat

BSPMake.bat tok som argument lokasjonen til bsp-programmet som skulle kjøres, kompilerte dette, kjørte det og om mulig sjekket at resultatet var som forventet opp mot en

sjekkfil. Vi la til en liten utvidelse til dette skriptet som fikk det til å legge alle differansefilene mellom resultatet av hver kjøring og sjekkfila på én plass, slik at det ble enkelt å identifisere de testene som ikke kjørte korrekt. I listing 4.1 ser vi det originale skriptet. Syntaksen her fungerte mot Windows XP, så vi trengte bare endre litt på hvor skriptet skriver ut differansefilene sine. I listing 4.2 ser vi hvordan vårt nye skript ble.

Listing 4.1: Originale BSPMake.bat

```
del Debug\bspsim.exe
set INCLUDE=C:\MSDEV\INCLUDE
set LIB=C:\MSDEV\LIB

del Debug\bspmain.*
copy %1.cpp bspmain.cpp
copy %1.dat parameters.dat

nmake /f bspmake.mak

copy %1.out %1.old

Debug\BSPSim.exe > %1.out

rem if a check file exists, do a check for the output.
if exist %1.chk fc %1.out %1.chk >> differences

echo %1 Finished !!
```

Listing 4.2: Nye BSPMake.bat

```
@echo off
del Debug\bspsim.exe

del Debug\bspmain.*
copy TestCode\%1\bspmain.cpp bspmain.cpp
copy parameters\%2.pam parameters.dat

nmake /f bspmake.mak

rem copy %1.out %1.old

Debug\BSPSim.exe > TestCode\%1\bspmain.out

rem if a check file exists, do a check for the output.
if exist TestCode\%1\bspmain.chk fc
        TestCode\%1\bspmain.out
        TestCode\%1\bspmain.chk > diffs\%1.%2.diff

echo %1 Finished !!
```

BSPTest.bat

BSPTest.bat ble benyttet til å kjøre alle testprogrammene som en batchjobb som gikk helt til alle testene var ferdige. Syntaksen på den originale *BSPTest.bat* fungerte ikke på Windows XP, så vi ble nødt til å sette oss inn i batch-syntaks på Windows XP og skrive om. Vi kan se hvordan det originale skriptet var satt opp i listing 4.3. Filen *TESTFILES.TXT* inneholder her en liste over alle testprogrammene som skal kjøres.

Listing 4.3: Originale BSPTest.bat

```
unalias copy

del differences

for %f in (@TESTFILES.TXT) do call BSPMake.bat %f

alias copy = *copy /r
```

Vi kom frem til at den enkleste måten å få funksjonaliteten vi ønsket var å dele opp skriptet i to filer, der den ene kaller den andre for hver parameterfil som skal testes. Vi lagde et nytt skript vi kalte *BSPTestAll.bat* som kaller *BSPTest.bat* som tar som argument nummeret på parameterfilen som skal kjøres og kjører denne på alle testprogrammene. Vi kan se de to skriptene i listing 4.4 og 4.5.

Listing 4.4: Nye BSPTest.bat

```
@echo off

for /F %%f in (TESTFILES.TXT)
    do call BSPMake.bat %%f %1
```

Listing 4.5: BSPTestAll.bat

```
@echo off

for %%f in (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
       18 19 20 21 22 23 24 25) do call BSPTest.bat %%f
```

4.2.2 Kjøring av de nye batch-skriptene

Etter å ha gjort de nødvendige modifikasjonene av skriptene var vi i stand til å kunne kjøre gjennom de 25 parameterfilene på alle testprogrammene med en eneste kommando. Vi kunne nå se resultatet fra hver gjennomkjøring i en egen fil. Dette gjorde oss i stand til å senere kunne identifisere dersom noen av testene slutter å fungere etter at vi hadde gjort endringer i koden. Denne funksjonaliteten er uhyre viktig hjelpemiddel for å kunne ende opp med et stabilt og godt produkt til slutt. Vi satte opp filen *TestFiles.txt* med listen over alle testprogrammene som skal kjøres. Denne ses i listing 4.6.

Listing 4.6: TestFiles.txt, liste over alle testene

```
bsplib_000
bsplib_001
bsplib_002
bsplib_003
bsplib_004
bsplib_005
bsplib_006
bsplib_007
bsplib_008
bsplib_009
bsplib_010
bsplib_011
bsplib_014
bsplib_015
HelloWorld
LargeScan
MergeSort
QuickSort
```

Denne listen er tatt fra testene som ble kjørt i [Lilleaas, 1998]. Ut ifra den rapporten kunne vi forvente at test nummer 8 og *QuickSort* ikke vil fungere. Gjennomkjøringen av testene bekreftet forventningene våre, alle tester unntatt 8 og *QuickSort* kjørte igjennom uten problemer. Vi konkluderte med at den versjonen av BSPlab som nå kjører på *Microsoft Visual Studio .Net* fungerer tilsvarende den originale versjonen. Vi hadde nå ett oppsett der vi, for hver endring vi gjør i BSPlab-koden, kan sjekke at alt fortsatt fungerer som før med disse testene.

4.3 Enkle endringer av koden

I første omgang ønsket vi å få ryddet opp i en del "gammeldags" C++-kode vi hadde sett i kodefilene og starte klargjøringen for andre plattformer enn VS.Net. Vi tok tak i ting som ga *deprecated* advarsler, ting som vi mener kan gjøres bedre på en annen måte og ting som åpenbart ikke ville fungere under Linux.

4.3.1 Benytte innebygd støtte for bolske verdier i kompilatoren

De nye C++-kompilatorene det er aktuelt for oss å flytte BSPlab til, har støtte for typen `bool`. Dette var ikke tilfellet på tidspunktet BSPlab ble skrevet. Det har ingenting å si i forhold til funksjonaliteten til programmet om man benytter den innebygde støtten i kompilatoren eller ikke, men vi ser for oss at det vil gjøre det enklere å vedlikeholde koden i fremtiden dersom man gjør det. Først fjernet vi bruken av `std::boolVal` til å representere typen til bolske verdier og erstattet denne med typen `bool`. Deretter fjernet vi bruken av headerfilen *Include/Common/Boolean.h* fra C++Sim som inneholdt typedefinisjoner av typene `Boolean`, `TRUE` og `FALSE`. Vi erstattet så disse med de tilsvarende innebygde typene `bool`, `true` og `false`.

Hver gang vi nå kompilerte prosjektet, oppdaget kompilatoren og ga advarsel om en del plasser der funksjoner for eksempel tar inn integer-verdier, men får sendt inn en bolsk verdi eller liknende. Denne rapporteringen gjorde det mulig for oss å enkelt gå inn i disse funksjonene og se hva som var tenkt og dermed kunne avverge eventuelle feil som kanskje ikke var blitt oppdaget da de innebygde typene for bolske verdier ikke var i bruk. En advarsel som gikk igjen mange steder var at det går utover ytelsen til programmet når en funksjon som returnerer en bolsk verdi returnerer en integer-verdi i koden. Listing 4.7 viser et eksempel på hvordan man kan bli kvitt denne advarselen.

Listing 4.7: Hvordan bli kvitt advarsel om at feil type returneres

```
bool foo(int i)
{
    return i; //Denne vil gi en advarsel om at
              //det returneres en int som bool
}

bool foo(int i)
{
    return i != 0; //Denne gir ingen advarsel
}
```

Vi fant noen variabler i koden som logisk kun tar verdiene true eller false, men som var av typen int. Dette gjaldt variablene `m_iWaitForMess` og `m_iWaitForSync` i *netmicroprocessor.h* og variabelen `get` i structen `_tagMemOpInfo`. Vi gjorde om typen til disse til `bool` for å gjøre koden mer forståelig. Etter denne opprenskningen i koden kjørte vi gjennom alle testene på nytt og alt så ut til å fungere som det skulle. at Vi valgte å legge inn nyeste versjon, 1.7.4. Det var rapportert på C++Sim sin hjemmeside [C++SIM, 1997]

4.3.2 Oppdatere include statements

I C++ er det vanlig å inkludere headerfiler i kodefilene. Disse headerfilene kan blant annet inneholde ting som skal deles mellom objektfiler og gjør det mulig for kode i forskjellige objektfiler å identifisere elementer inne i andre objektfiler. Det er vanlig å benytte såkalte *include guards* som skal hindre at en headerfil blir inkludert mer enn én gang i samme objektfil. Disse *include guardene* er det vanlig å sette opp i starten av hver headerfil slik at kompilatoren lar være å inkludere noe kode fra headerfilen dersom denne allerede har vært inkludert en annen plass. Listing 4.8 viser et eksempel på en enkel headerfil med *include guards.*

Listing 4.8: En enkel headerfil med include guards

```
//foo.h
#ifndef INCLUDED_FOO_H_
#define INCLUDED_FOO_H_

class foo {

};

#endif //INCLUDED_FOO_H_
```

Denne koden setter en tekststreng, i dette tilfellet strengen *INCLUDED_FOO_H*, til å være

definert. Den sørger også for at dersom strengen er definert i starten av fila så vil ikke noe av innholdet i headerfila bli gitt videre til kompilatoren.

I bsplab koden var dette gjort litt annerledes. Alle headerfiler inneholdt *include guards*, men det var i tillegg satt opp guards rundt mange plasser headerfiler ble inkludert. Listing 4.9 viser et eksempel på hvordan dette var gjort.

Listing 4.9: Include guards rundt plassen headerfilen blir inkludert

```
//foo.cc
#ifndef INCLUDED_FOO_H_
#define INCLUDED_FOO_H_
   #include <foo.h>
#endif
```

Dette er en litt merkelig måte å inkludere headerfiler på, og er ikke nødvendig ettersom *foo.h* allerede inneholder *include guards*. Det er mulig at dette ble gjort på grunn av problemer med en gammel kompilator, men det er hvertfall ikke nødvendig å ha med nå. Vi gikk derfor gjennom all koden og fjernet *include guards* rundt plasser der headerfiler ble inkludert.

For å gjøre koden klar til å kompilere på plattformer som er *case sensitive* endret vi i samme slengen alle tegn i alle headerfiler til små bokstaver. Tidligere var disse en blanding av store og små bokstaver uten noen konsistent praksis. Vi endret også alle filnavn på kodefiler i prosjektet til å bare inneholde små bokstaver.

Det er også mulig å inkludere headerfiler som ligger i andre kataloger enn filen som blir kompilert. Måten dette gjøres på er at relativ eller absolutt sti til filen blir brukt i include statementet. I den opprinnelige koden ble det noen plasser benyttet *backslash*, til å skille kataloger fra hverandre mens det andre plasser ble benyttet *slash*. Begge deler fungerer på Windows, men kun *slash* fungerer på Linux, så vi endret alle include statements til kun å benytte *slash*.

Begrens avhengighetene mellom kodefilene

Det er god praksis i C++ å forsøke å inkludere så få andre headerfiler som mulig i en headerfil. Grunnen til dette er at det er ønskelig med så lite avhengigheter som mulig mellom kodefiler. Det er flere grunner til at dette er god praksis. Kanskje den viktigste grunnen rent praktisk er at du må kompilere om igjen mindre kode jo mindre avhengigheter mellom kodefiler det er når det gjøres en endring i en headerfil. Det er to teknikker som er typisk å benytte for å ha færrest mulig inkluderinger i headerfiler. Den første er ganske åpenbar; dersom du bare trenger innholdet i den headerfila du vil inkludere i kodefiler, så skal denne ikke inkluderes i noen headerfil. Den andre teknikken er å benytte seg av noe som kalles *forward-deklarasjon*. Å *forward-deklarere* noe betyr at du forteller kompilatoren at en type finnes, men du beskriver ikke hvordan typen er definert. Det er i mange tilfeller nok for kompilatoren å vite at en type eksisterer, uten at den trenger å vite nøyaktig hvordan

definisjonen av typen er.

I BSPlab var det benyttet noe *forward declaring*, men dette kunne med fordel vært benyttet i større grad. Et større problem var at ganske mange headerfiler var inkludert i headerfiler som enten ikke benyttes i det hele tatt eller kun benyttes i kodefilene. Dette fører til at når du gjør små endringer i enkelte filer så må nesten hele prosjektet rekompileres. Vi gikk igjennom headerfilene og fikk ryddet opp i mange unødvendige inkluderinger.

Benytt korrekte headerfiler

En rekke funksjoner benyttes i C++ som egentlig kommer fra standard C. BSPlab benytter seg av flere av disse og de finnes deklarert i ulike headerfiler. Et problem er at BSPlab som blir kompilert som C++ inkluderer headerfiler som egentlig er ment å kompileres med en C-kompilator. Dette har gått bra, men det er ingen garanti for at det vil gjøre det i fremtiden. Løsningen ble å endre alle disse inkluderingene fra C-headerfilene til de korrekte C++-headerfilene. Listing 4.10 viser endringene vi gjorde.

Listing 4.10: Endringene til korrekte headerfiler

```
#include <assert.h> -> #include <cassert>
#include <cmath.h> -> #include <cmath>
#include <stdarg.h> -> #include <cstdarg>
#include <stdio.h> -> #include <cstdio>
#include <time.h> -> #include <ctime>
```

Dette har ingen praktisk verdi utover at det er den korrekte måten å gjøre ting på og at det på den måten hjelper til med å sikre at BSPlab vil fungere mot nye kompilatorer i fremtiden.

Vi kjørte gjennom alle testene på nytt og alt fungerte like godt som før.

4.4 Oppgradering av C++Sim

I den opprinnelige rapporten [Uthus and Dybdahl, 1997] beskreve de en del problemer med C++Sim. Disse gikk primært på problemer med *deadlocks* og at enkelte tråder noen ganger ikke terminerte som de skulle. Den opprinnelige BSPlab kjørte med C++Sim versjon 1.6. Vi valgte å legge inn nyeste versjon, 1.7.4, siden det var rapportert at enkelte problemer skulle være fikset i denne versjonen [C++SIM, 1997].

4.4.1 Nødvendige modifikasjoner til C++Sim

På grunn av et problem der operativsystemet i enkelte tilfeller ikke klarer å avslutte en tråd når det blir bedt om det hadde C++Sim som fulgte med BSPlab fått lagt til noe

tilleggsfunksjonalitet. Dette var i form av en funksjon, `WaitForTermination`, som sjekker om tråden til en prosess har terminert og returnerer en status til avsenderen. Måten BSPlab benytter dette på er å sjekke om tråden til en prosess er avsluttet før den forsøker å frigjøre minnet til prosessen. Vi var klar over at det var en viss mulighet for at dette problemet nå kanskje ikke lenger var aktuelt på grunn av at dette var en nyere versjon av C++Sim, men vi valgte å i første omgang implementere denne funksjonen og heller se nærmere på dette senere.

Problemer med deadlocks

Etter at vi hadde byttet ut C++Sim fikk vi problemer med at simuleringen noen ganger bare hang seg opp. Vi var forberedt på at dette problemet kunne oppstå etter å ha lest den opprinnelige BSPlab-rapporten [Uthus and Dybdahl, 1997] og Erik Sund sin rapport [Sund, 2004], der problemet beskrives. Feilen skyldes at under visse omstendigheter kan programmet forsøke å terminere en tråd to ganger, noe som resulterer i at hele C++Sim henger. Vi valgte å løse deadlock-problemet på samme måte som i den opprinnelige BSPlab ved å legge inn et flagg som hindrer en tråd i å bli terminert to ganger.

4.4.2 Videre testing med ny C++Sim

Etter vi hadde identifisert og laget en løsning på deadlock-problemet viste det seg at det dukket opp enda et problem under testing. I noen tilfeller stoppet simuleringen med en *divide by zero*-feil. Etter en del debugging viste det seg at denne feilen kom fra BSPlab og ikke C++Sim. Den oppstår når programmet forsøker å ta modulo med 0 i koden i *oneport.h*. I koden ser det ut som om den forventer å få resultatet 0 av denne beregningen, og ikke en programkrasj. Det er derfor en mulighet at det på gamle kompilatorer gir svar 0 hvis man tar modulo med 0, men vi fant ingen plass dette er nevnt i den dokumentasjonen vi har fått tak i om kompilatorene. Vi fant ingen sammenheng mellom at denne feilen ble oppdaget nå og oppgraderingen av C++Sim. Det virker som om den bare oppstår en gang i blant når one port modellen benyttes, så det er en god mulighet for at den har vært der hele tiden under VS.Net. Ettersom det ser ut til at koden forventer å få resultatet 0 av denne beregningen la vi inn en sjekk om ganske enkelt gir resultat 0 istedet for å forsøke å gjøre modulo operasjonen. Etter denne endringen kjørte alle testene som normalt igjen. Vi var nå sikre på at vi hadde en oppdatert versjon av BSPlab som kjører stabilt med *Visual Studion .Net.*

Kapittel 5

Flytting av BSPlab til ikke-proprietære plattformer

Etter å ha fått BSPlab til å kjøre stabilt på *Microsoft Visual Studio .Net* var vi klare til å starte på hoveddelen av prosjektet. Først fant vi frem til hvilken kompilator vi skulle benytte på Windows. Deretter bestemte vi oss for et automake-verktøy for begge plattformene. Til slutt gikk vi i detalj gjennom hvordan vi fikk BSPlab til å fungere stabilt på Linux og den nye kompilatoren for Windows.

5.1 Valg av gratis kompilator for Windows

Siden Windows tross alt er en mer vanlig plattform en Linux, var det også ønskelig å kjøre BSPlab på Windows. I oppgaveteksten er det foreslått å flytte BSPlab til bl.a. Cygwin [Cygwin, 2005]. Vi vil her gjøre rede for hvordan vi valgte utviklingsverktøy for Windows. Ettersom vi har drevet en del med kryss-plattform utvikling fra før kjente vi allerede til en del av alternativene for gratis kompilatorer til Windows.

5.1.1 Cygwin

Cygwin [Cygwin, 2005] er i hovedsak et Unix-miljø for Windows som er utviklet av Linux-leverandøren *Red Hat*. Cygwin sin kompilator gir mulighet til å kompilere opp program på Windows som kan benytte Unix/Linux C-funksjoner som i utgangspunktet ikke er portable. Prisen man må betale for dette er at programmet må linke mot et spesielt Cygwin bibliotek. Dette biblioteket kan av lisens-grunner ikke linkes statisk inn i den kjørebare filen, men må sendes med som tredjepartsprogramvare.

5.1.2 MinGW

Minimalistisk GNU for Windows [MinGW, 2005] har i utgangspunktet mye av den samme funksjonaliteten som Cygwin. MinGW er et sett av Unix-verktøy portet til å kunne kjøre på Windows. Et av de viktigste verktøyene i dette settet er GCC. Hovedforskjellen mellom MinGW og Cygwin er at MinGW har valgt å fjerne bruk av ikke portable C-funksjoner fra de verktøyene som er portet. Dermed kan man bruke MinGW porten av GCC til å kompilere opp Windows-programmer uten at programmet blir avhengig av tredjepartsbibliotek som ikke distribueres med Windows.

Ettersom MinGW GCC bare er en kompilator, kan den være vanskelig å bruke for brukere som er vant med *Microsoft Visual Studios Integrated Development Enviroment* (IDE). Det finnes et gratis IDE for MinGW GCC som heter Dev-C++ [Dev-C++, 2005]. Ved bruk av MinGW og Dev-C++ vil vi kunne distribuere et helt gratis utviklingsmiljø sammen med BSPlab. Begge disse verktøyene er lisensert under *GNU General Public License* [GPL, 2005]. Kort fortalt gir lisensen en fri rett til å bruke verktøyene i enhver form uten at noen har krav på kompensasjon. Hvis man derimot gjør endringer på kildekoden og distribuerer et program som er under GPL-lisens, kreves det at man også gjør hele den endrede kildekoden tilgjengelig for hvem som helst. Det er også slik at hvis man kopierer kode fra et GPL-prosjekt inn i eget prosjekt er man nødt til å lisensere det egne prosjektet som GPL eller kompatibel lisens. Dette betyr i realiteten at man er nødt til å frigi *all* kode for det egne prosjektet hvis man har valgt å bruke en eneste kode-linje fra et GPL-prosjekt. Siden vi ikke har noen planer om å endre kildekoden til verktøyene, har ikke dette noen betydning for oss utover at vi helt fritt kan distribuere binærversjonene av MinGW og Dev-C++ både gratis og kommersielt.

5.1.3 Endelig valg av kompilator

Vi ser ingen fordeler med å bruke Cygwin fremfor MinGW. Ettersom BSPlab i utgangspunktet er laget og kompilerer på Windows, er det lite trolig at vi vil trenge C funksjoner som ikke er portable når vi skal flytte det tilbake til Windows. Da det er ønskelig at BSPlab også fortsatt skal kunne kompilere på Visual Studio, er det sannsynlig å tro at Cygwin vil medføre flere ulemper enn fordeler. Vi valgte derfor å bruke MinGW som gratis kompilator på Windows, og Dev-C++ som IDE.

5.2 Valg av automake verktøy

Som beskrevet i kapittel 3.1.1 er det viktig å finne et enkelt og gratis automake-verktøy som fungerer både på Windows og Linux. For å finne det riktige verktøyet gjorde vi en del undersøkelser på kryss-plattform automake-verktøy.

5.2.1 Innledende undersøkelser

Vi valgte å undersøke hvilke automake-verktøy større kryss-plattform produkter bruker, samt å se etter slike verktøy i større "opensource" prosjektdatabaser slik som *Freshmeat* [Freshmeat, 2005] og *Sourceforge* [SourceForge, 2005].

QMake

Qt [Qt, 2005] er et anerkjent kryss-plattform GUI-verktøy som brukes av mange aktører, blant annet Adobe og *European Space Agency* (ESA). Qt er også rammeverket som brukes i et av de største desktop-systemene på Unix, KDE [KDE, 2005]. Det er det norske selskapet Trolltech som står bak produktet.

Qt inkluderer et automake-verktøy som heter QMake. Dette verktøyet fungerer på mange plattformer. Dette inkluderer "embedded systems", da dette er et av Trolltech/Qt sine største satningsområder. Selv om QMake er et godt utprøvd verktøy med mye funksjonalitet er det umulig for oss å bruke da Qt for Windows er proprietært. Qt er bl.a. lisensert under GPL for alle andre plattformer enn Windows, og det ryktes at den nyeste versjonen, Qt 4.0, som er under utvikling også skal lisenseres under GPL for Windows. Dette er nok for sent til at vi kunne vurdere å bruke QMake som automake-verktøy.

GNU Autotools

Vi kjente godt til *GNU Autotools* fra før ettersom vi har brukt dette verktøyet daglig i mange år. *GNU Autotools* brukes av store Unix-prosjekter som KDE [KDE, 2005], GNOME [GNOME, 2005] og Apache HTTP Server [Apache HTTP Server, 2005]. En god del KDE-programmer har blitt flyttet til også å kjøre på Windows v.h.a. *GNU Autotools*. Apache HTTP Server kjører på veldig mange plattformer inklusive Windows, men på Windows bruker Apache en egen prosjekt-fil og ikke Autotools slik som på resten av plattformene.

IMake

IMake er verktøyet som originale C++Sim bruker. Verktøyet er opprinnelig laget for XFree86, den mest brukte serveren for grafisk grensesnitt på Unix. Selv om XFree86 er et Unix-prosjekt, fungerer IMake også på Windows.

CMake

Vi fant frem til CMake [CMake, 2005] gjennom både Freshmeat og Sourceforge. Ingen prosjekter vi kjenner til bruker dette systemet, men ut fra den informasjonen vi kunne finne på internett virket dette verktøyet veldig lovende. Mens de andre automake-verktøyene vi har sett på gjerne har hatt hovedfokus på Unix-system, og Windows støtte har kommet

som en "tilleggs-hack", har CMake tatt hensyn til Windows-plattformen fra starten av, uten at dette nødvendigvis har gått utover støtten for Unix.

Egne scriptfiler

En del prosjekter som for eksempel OpenSSL [OpenSSL, 2005] bruker egne script-filer for å gjøre bygge-konfigureringen. Siden script-språket til Windows ikke kan kalles særlig kraftfullt, og heller ikke er særlig plattform-uavhengig, kreves det at brukeren installerer et ekstra script-språk på Windows. I OpenSSL-tilfellet er det Perl som brukes.

Resultat av innledende undersøkelser

Det viste seg at svært få kryss-plattform prosjekter bruker noen spesiell form for automake-verktøy på Windows, mens på alle Unix-plattformer brukes det samme automake-verktøyet. De prosjektene som bruker automake-verktøy på Unix, har vanligvis bare en ferdig konfigurert prosjektfil for Visual Studio eller MinGW på Windows. Grunnen til dette kan nok være at det til nå har vært få automake-verktøy med støtte for Windows. Det har derfor vært enklere bare å ha egne prosjektfiler på denne plattformen selv om dette kan føre til inkonsistens mellom prosjektfilene og/eller plattformene.

Etter de innledende undersøkelsene sto vi igjen med følgende alternativer:

GNU Autotools Det mest brukte automake-verktøyet på Linux.

IMake IMake brukes av XFree86 og på den originale versjon av C++Sim.

CMake CMake [CMake, 2005] er nyere enn de to ovennevnte verktøyene og har som mål å være mindre komplisert å bruke enn *GNU Autotools*.

5.2.2 Beskrivelse av automake-alternativene

Etter grovutvalget av automake-verktøyene gikk vi de gjenstående alternativene litt nærmere i sømmene for å finne det mest optimale verktøyet for BSPlab-prosjektet.

GNU Autotools

GNU Autotools er det mest bruke automake-verktøyet på Linux. Veldig mange programmer utviklet på Linux bruker *GNU Autotools* på en eller annen måte. *GNU Autotools* er en fellesbetegnelse for en rekke programmer, først og fremst *GNU Automake* [GNU Automake, 2005], *GNU Autoconf* [GNU Autoconf, 2005] og *GNU Libtool* [GNU Libtool, 2005]. Hvert av disse programmene benyttes i forskjellige steg i byggeprosessen. Verktøyene kan brukes til å gjøre programmet ditt kompilerbart på en stor del av de plattformer som eksisterer idag inkludert *embedded*-systemer. Programmene inneholder

en rekke innebygde sjekker man kan bruke for å undersøke om den plattformen systemet blir forsøkt kompilert på tilfredsstiller de kravene programmet som skal kompileres har. For eksempel kan verktøyene sjekke om man er på et *big-endian* eller *little-endian* system, og sette definisjoner deretter som koden som skal kompileres kan sjekke. Verktøyene kan også undersøke om alle avhengigheter, for eksempel tredjeparts biblioteker, er installert på systemet i riktig versjon.

GNU Autotools kan gjøre det meste du ønsker deg i bygge-prosessen din, noe som også gjenspeiles i manualene til hvert av de tre programmene *GNU Autotools* består av. Selv om systemet er allsidig, beskyldes det for å ha en altfor lang læringskurve. Bare det å sette opp et enkelt *HelloWorld*-prosjekt med *GNU Autotools* kan ta flere timer første gangen, og da er man ikke nødvendigvis klar over hvordan systemet er satt opp etter at man er ferdig. Endringer til prosjektet kan derfor ta lang tid for de som ikke bruker verktøyet til daglig. Systemet beskyldes også for å ha uforklarlige tilbakemeldinger på feil som kan oppstå [McCall, 2003], noe som kan gjøre bygge-prosessen enda mer tidkrevende. Det er også et problem at nye versjoner av verktøyet ikke nødvendigvis er bakoverkompatibel, slik at man kan få problemer med å konfigurere og kompilere eldre prosjekter som bruker *GNU Autotools*.

Selv om det finnes en rekke programmer som setter opp og forandrer *Autotools*-miljøet ditt på Linux, er disse programmene mangelvare på Windows. Programmene er også sjelden installert som standard på Unix-distribusjoner.

GNU Autotools er først og fremst ment til å brukes på *POSIX*-kompatible systemer, det vil i første rekke si de fleste Unix-varianter, men ikke Windows. Det er likevel mulig å bruke systemet på Windows, men dette krever at man installerer en rekke tredjepartsprogrammer, først og fremst *ActivePerl* [ActivePerl, 2005] og *MSYS* [MSYS, 2005], hvor sistnevnte gir en Unix-stil kommandolinje med tilhørende verktøy i Windows.

GNU Autotools er støttet av en rekke IDEer på Linux, blant annet har den populære IDEen *KDevelop* [KDevelop, 2005] så god støtte for Autotools at en rekke av kompleksitetsproblemene ikke merkes dersom man kun administrerer kode-prosjektene sine fra *KDevelop*. Ettersom *GNU Autotools* krever et Unix-miljø for å fungere skikkelig på Windows er det heller ikke støttet av noen IDEer i Windows.

Da *GNU Autotools* er så å si standard automake-verktøy på Linux er det et godt dokumentert verktøy, men størrelsen på den offisielle dokumentasjonen gjenspeiler også hvor omfattende og komplekst verktøyet er.

IMake

IMake brukes først og fremst av de store X-server-prosjektene for Unix, *XFree86* og *X.org*. IMake er i likhet med *GNU Autotools* et gammelt og veletablert verktøy. Dessverre plages også dette verktøyet med at det kan bli altfor komplekst. IMake er heller ikke på langt nær like godt dokumentert som *GNU Autotools*, noe som gjør det lite appellerende for nye utviklere.

CMake

CMake er et automake-verktøy av nyere dato. Da dette prosjektet ble startet var målet å lage et automake-system som var enkelt og raskt å lære. CMake skal gjøre det enkelt å sette opp et kryss-plattform bygge-miljø uten alle kompleksitetsproblemene som plager *GNU Autotools* og *IMake*. Systemet benytter seg av tekstfiler med enkle kommandoer for å sette opp bygge-miljøet.

I motsetning til *GNU Autotools* trengs bare programmet CMake for å konfigurere og bygge et prosjekt. Når det gjelder kompatibilitet med forskjellige IDEer bruker CMake omvendt filosofi i forhold til de fleste andre automake-verktøy. I stedet for å kreve at IDEene har støtte for verktøyet og kan gjøre forandringer på konfigurasjonsfilene for bygge-miljøet, fungerer CMake slik at det genererer prosjektfiler for den valgte IDEen ut fra CMake-filene for dette prosjektet. Dette gjør at man ofte kan bruke sin favoritt-IDE for å programmere på et CMake-prosjekt selv om IDEen ikke har støtte for CMake. På Windows kan man for eksempel generere makefiler for *nmake* hvis man vil bruke Visual Studio-kompilatoren fra kommandolinjen, eller man kan lage en Visual Studio prosjektfil hvis man ønsker å bygge prosjektet i Visual Studio IDEen. Det er også mulig å generere Unix type makefiler på Windows som blant annet kan brukes av MinGW. I utgangspunktet støtter CMake å generere prosjektfiler for *NMake*, *Borland*, *Unix Makefiles*, *KDevelop* og en rekke versjoner av *Visual Studio*.

CMake konfigureres ved å lage filer ved navn *CMakeLists.txt* i alle områder som har kildekode, samt alle områder som er underordnet disse. Dvs at dersom du har et prosjekt *helloworld* hvor kildekoden ligger i området *src* under *helloworld*, så trenger du en *CMakeLists.txt*-fil både i området *helloworld* og underområdet *src*. Har du derimot enda et område under *helloworld* som ikke innholder kildekode, trenger du ikke å lage en CMake-fil der.

For *helloworld*-eksempelprosjektet vårt får vi en *CMakeLists.txt* for rot-området vårt som vist i listing 5.1. I linje 2 definerer vi navnet på prosjektet. Dette gjøres blant annet for at vi skal kunne hente ut en del egenskaper tilhørende prosjektet via variabler som blir navngitt ut fra prosjektnavnet. Dersom vi skulle trenge å vite hva absolutt sti til *helloworld*-prosjektet vårt er, kan denne for eksempel hentes ut fra variabelen `HELLOWORLD_SOURCE_DIR`. I linje 4 forteller vi CMake hvilke andre områder direkte under rot-området som må traverseres for *CMakeLists.txt*-filer. I *src* har vi en *CMakeLists.txt* fil som vist i listing 5.2. Det eneste vi gjør i denne filen er å fortelle CMake at vi vil ha en kjørbar fil ved navn *helloworld* og at den kjørbare filen kan fremskaffes ved å kompilere kilekodefilen *helloworld.cc*.

Listing 5.1: CMakeLists.txt i rot-området til “helloworld”

```
1 # The project name
2 PROJECT(HELLOWORLD)
3 # Subdirs to build
4 SUBDIRS(src)
```

Listing 5.2: CMakeLists.txt i src-området til "helloworld"

```
ADD_EXECUTABLE( helloworld  helloworld.cc )
```

Når så *CMakeLists.txt* filene for *helloworld*-prosjektet er ferdigskrevet, trenger vi bare å stille oss i rot-området og skrive "cmake .". CMake vil da se etter om operativsystemet har installert en brukbar kompilator. Dersom den finner en kompilator vil den generere kompilator- og plattformavhengige bygge-filer. På Linux vil det bli bygget *Unix Makefiles* med mindre noe annet er spesifisert. På Windows avhenger det av hvilken kompilator som er installert hva slags filer som blir generert. Har man for eksempel bare Visual Studio installert, vil det bli generert en Visual Studio prosjektfil som så kan åpnes i Visual Studios IDE og kompileres derfra. CMake tar seg også av andre forskjeller mellom plattformene slik som at den kjørbare filen får navnet *helloworld* på Linux, mens på Windows blir den hetende *helloworld.exe*.

CMake er på langt nær like godt dokumentert som *GNU Autotools*, men ettersom kompleksitetsnivået til CMake også er langt lavere finner man det meste man trenger på hjemmesiden til CMake [CMake, 2005]. Dersom man skulle trenge mer avansert eller spesiell funksjonalitet er det veldig mye god hjelp å få på mailing-listen til CMake.

5.2.3 Evaluering av automake-verktøy

I dette avsnittet vil vi sette opp en tabell for evaluering av automake-verktøyene. I tabellen vil vi gradere de forskjellige egenskapene vi legger mest vekt på som spesifisert i seksjon 3.1.1. Graderingene vil være *Lav* (L), *Middels* (M) eller *Høy* (H). Hver gradering tilegnes henholdsvis 1, 2 og 3 poeng, slik at verktøyet tilslutt vil få en poengsum som tilsvarer summen av graderingene for egenskapene.

For enkelhets skyld oppsummerer vi kravene en gang til:

1. Hvor god støtten er for Linux og Windows.
2. Hvordan godt verktøyet fungerer med forskjellige IDEer på Linux og Windows.
3. Hvor enkelt det er å lage et prosjekt med verktøyet.
4. Hvor enkelt det er å vedlikeholde et prosjekt med verktøyet.
5. Hvor godt verktøyet er dokumentert.

Verktøy	**1**	**2**	**3**	**4**	**5**	**Poengsum**
Autotools	L	L	M	M	H	9
IMake	M	L	L	L	L	6
CMake	H	M	H	H	M	13

Tabell 5.1: Oversikt over automake-verktøy

Autotools

Autotools får en *lav* gradering på de to første kravene som følge av at støtten på Windows krever et Unix-miljø som ikke nødvendigvis er spesielt brukervennlig for brukere som aldri har brukt Unix. Det eksisterer heller ikke noen IDEer med Autotools-støtte på Windows. Verktøyet får *middels* på krav *3* og *4* bare fordi vanskelighetsgraden på disse punktene reduseres betraktelig av tilgjengelige verktøy. Denne graden er dog svak da slike verktøy ikke er tilgjengelig på vanlige Windows-plattformer. Det eneste punktet Autotools får toppkarakter på er punkt *5*, dokumentasjon. Dokumentasjonen til Autotools er av ypperste klasse, hver eneste liten funksjonalitet er beskrevet i detalj. Størrelsen på dokumentasjonen kan dog virke avskrekkende på mange.

IMake

IMake får *middels* på krav *1* ettersom støtten for Windows ikke er overveldende, men likevel bedre enn Autotools sin. Krav *2* til *4* får karakteren *lav* ettersom det er få IDEer som støtter IMake på begge plattformer og IMake sliter med de samme problemer med kompleksitet som Autotools gjør. Verktøyet har også lite dokumentasjon og får således karakteren *lav* på krav *5*.

CMake

CMake har veldig god støtte for Linux og Windows. Verktøyet gjør det enkelt å lage et nytt kodeprosjekt samt å vedlikeholde det. CMake får derfor karakteren *høy* på kravene *1*, *3* og *4*. CMake støtter mange IDEer både på Linux og Windows, men trekkes ned til karakteren *middels* på krav *2* da det ikke støtter å generere prosjektfiler for Dev-C++ som er gratis-IDEen vi sender med BSPlab. Dette er forsåvidt ikke et kjempestort problem ettersom det ikke er mening at gjennomsnittbrukeren av BSPlab skal gjøre forandringer på BSPlab-pakken, men Dev-C++-støtte hadde likevel ikke vært å forakte. Dokumentasjonen til CMake er som påpekt i seksjon 5.2 ikke så omfattende som Autotools sin, men god nok til å få karakteren *middels* på krav *5*.

Endelig valg av automake-verktøy

Etter å ha testet de forskjellige automake-verktøyene på begge plattformer satt vi igjen med en følelse av at ett verktøy utmerket seg spesielt fremfor de to andre. Denne følelsen stemmer helt overens med sluttpoengsummen som kommer frem i tabell 5.1. Med andre ord tilsier både vår egen utprøvning og andres erfaringer hentet fra ulike fora på internett at vi bør bruke CMake som automake-verktøy for dette prosjektet.

5.3 Oppsett av CMake

Det første vi måtte gjøre når vi skulle flytte BSPlab fra Windows til Linux var å få satt opp automake-verktøyet. For at vi i det hele tatt skulle få kompilert BSPlab trengte vi et ferdig oppsatt bygge-miljø.

5.3.1 BSPlabs opprinnelige byggestruktur

BSPlab var opprinnelig bygd opp som ett eneste program. Det vil si at alle kodefiler, inklusive C++Sim og BSP-programmet som skulle simuleres, lå i samme kodetre og kompilerte til en kjørbar *.exe*-fil. Vi fant denne strukturen langt fra optimal. En bruker av simuleringsprogrammet måtte åpne et prosjekt hvor alle kildefilene til både BSPlab og C++Sim vistes for å i det hele tatt få kompilert BSP-programmet som skulle simuleres. Dette kan mildt sagt virke forvirrende for noen som ikke trenger å ha noe forhold til BSPlab utover at det simulerer kode i BSP-programmer. Vi fant det derfor naturlig å forandre strukturen på BSPlab og C++Sim slik at de kunne brukes mer på den måten BSPlib benyttes.

Den nye strukturen til BSPlab innebærer at BSP-programmet til brukeren fortsatt blir et selvstendig program, men i steden for å linke mot et BSP-bibliotek linker man mot BSPlab- og C++Sim-bibliotekene. Dette medfører at brukeren strengt tatt ikke trenger å ha noe forhold til BSPlab-kildekoden ettersom han bare trenger tilgang på binære biblioteksfiler for å få simulert programmet sitt i BSPlab. Vi mener at dette er en mye mer ryddig og effektiv struktur enn den opprinnelige strukturen.

5.3.2 C++Sim

Vi valgte like godt å bytte automake-verktøyet til C++Sim fra IMake til CMake. Grunnen til dette er at det er mer effektivt for oss å bruke samme verktøy på C++Sim som BSPlab, men også at IMake ikke fungerte uten videre med GCC 3 (j.fr. seksjon 5.4).

C++Sim for BSPlab

Det viste seg at selv om C++Sim opprinnelig ble kompilert direkte inn i BSPlab, var C++Sim egentlig tenkt å brukes som bibliotek. C++Sim består av en rekke bibliotek, *C++Sim*, *Event*, *Stat*, *Common* og *SimSet*, hvorav BSPlab bare bruker *C++Sim*, *Common* og *Event*. For å gjøre hele BSPlab-pakken så enkel som mulig å vedlikeholde valgte vi å fjerne de delene som ikke brukes av BSPlab.

CMake konfigurasjonsfiler

Å få satt opp C++Sim til å kompilere ved hjelp av CMake viste seg å være en relativt enkel jobb. Bare ved bruk av de få kommandoene beskrevet i seksjon 5.2.2 kommer man

langt. I listing 5.3 ser vi *CMakeLists.txt* filen på roten til C++Sim. Kommandoen `SET` i linje 4 brukes til å be CMake legge alle bibliotek-filer som blir generert i *lib*-området under C++Sim-roten. Vi synes dette er ryddigere enn å ha bibliotekfilene spredt utover flere områder. I linje 5 brukes kommandoen `ADD_DEFINITIONS`. Denne kommandoen brukes for å gi kompilatoren en rekke preprosessor-definisjoner. De definisjonene som er listet opp er kopiert fra IMake konfigurasjonen til C++Sim og trengs for å få kompilert C++Sim riktig på Linux. I linje 14 brukes kommandoen `SUBDIRS` for å fortelle CMake hvor den skal lete etter flere *CMakeLists.txt*-filer.

Listing 5.3: *CMakeLists.txt* i rot-området til C++Sim

```
1  # The projects name is C++SIM
2  PROJECT(CPPSIM)
3  # Nake sure all library files are created in CPPSIM-ROOT/lib
4  SET(LIBRARY_OUTPUT_PATH ${CPPSIM_BINARY_DIR}/lib)
5  ADD_DEFINITIONS(-DPTHREAD_DRAFT_LINUX -DPOSIX_THREAD
6                  -DHAVE_BOOLEAN_TYPE -DHAVE_WCHAR_TYPE
7                  -DHAVE_LONG_LONG -DGCC_STATIC_INIT_BUG
8                  -DMEMFNS_IN_STRING_H -DNEED_MALLOC_T
9                  -DHAVE_MSGCONTROL -D__LINUX__
10                 -DSYSV -D_CONSTVALUE="" -D_CONSTVALUE2=""
11                 -D_NORETURN="" til -D_NORETURN2=""
12                 -DProcessList_Queue)
13 # Subdirs to build
14 SUBDIRS(Common ClassLib Event)
```

Hvert av områdene *Common*, *ClassLib* og *Event* inneholder en enkel *CMakeLists.txt*-fil med kommandoen `SUBDIRS(src)`.

I listing 5.4 ser vi innholdet av *CMakeLists.txt*-filen på området *ClassLib/src*. Linje 2 forteller CMake at den må be kompilatoren lete etter header-filer i områdene *Include* under rot-området til C++Sim. Den ber også kompilatoren lete på området *../include* relativt i forhold til hvor *CMakeLists.txt*-filen befinner seg. Header-filene som befinner seg i disse områdene er nødvendig for at kodefilene skal kunne kompilere. I steden for å bruke kommandoen `ADD_EXECUTABLE` (beskrevet i seksjon 5.2.2) som gir oss en kjørbar fil, bruker vi kommandoen `ADD_LIBRARY` i linje 5. Denne kommandoen har samme syntax og fungerer som `ADD_EXECUTABLE`, men gir oss en bibliotekfil i steden for en kjørbar fil. *CMakeLists.txt*-filene i områdene *Common/src* og *Event/src* ser likedan ut, bortsett fra at navnet på biblioteket og kildefilene i linje 5 er forskjellig. Bibliotek-navnene er henholdsvis *Common* og *Event*.

Listing 5.4: *CMakeLists.txt* i *ClassLib/src*-området til C++Sim

```
1 # Set include path
2 INCLUDE_DIRECTORIES(${CPPSIM_SOURCE_DIR}/Include ../include)
3
4 # Make library C++SIM
5 ADD_LIBRARY(C++SIM Process.cc Random.cc thread.cc
6                    ProcessList.cc ProcessCons.cc
7                    ProcessIterator.cc nt_thread.cc)
```

5.3.3 BSPlab

I listing 5.5 ser vi *CMakeLists.txt*-filen i roten til BSPlab. Denne filen er tilnærmet lik den tilsvarende filen for C++Sim. I linje 5 ber vi CMake legge bibliotekfilene i *lib* området under BSPlab-roten, og i linje 7 forteller vi CMake at den finner flere konfigurasjonsfiler i *src*-området.

Listing 5.5: *CMakeLists.txt* i rot-området til BSPlab

```
1 # The projects name is BSPlab
2 PROJECT(BSPLAB)
3 # Make sure all library files are
4 # created in BSPlab-ROOT/lib
5 SET(LIBRARY_OUTPUT_PATH ${BSPLAB_BINARY_DIR}/lib)
6 # Subdirs to build
7 SUBDIRS(src)
```

Listing 5.6 viser en forkortet versjon av *CMakeLists.txt*-filen i *src*-området til BSPlab. I linje 1 ber vi CMake fortelle kompilatoren at den skal lete etter header-filer i *Include*-området til C++Sim, og i området CMake-filen befinner seg i. Vi antar her at C++Sim befinner seg på samme område som *bsplab*-området og at området er kalt *cppsim*. I linje 3 bruker vi kommandoen `LINK_DIRECTORIES` for å fortelle linkeren hvor den skal lete etter bibliotek som vi ber den linke med. Vi ser at vi bare ber den lete i C++Sim sitt *lib*-område. Vi ber den ikke lete i BSPlab sitt *lib*-område fordi CMake selv finner alle bibliotek som er definert internt i prosjektet.

Listing 5.6: *CMakeLists.txt* i *src*-området til BSPlab

```
1  INCLUDE_DIRECTORIES(
2                 ${BSPLAB_SOURCE_DIR}/../cppsim/Include .)
3  LINK_DIRECTORIES(${BSPLAB_SOURCE_DIR}/../cppsim/lib)
4  ADD_LIBRARY(bsplab
5  ./aleksstring/aleksstring.cpp
6  ./bus/dmacontroller.cpp
7  ./bus/processorbus.cpp
8  ./bus/busbspmachine.cpp
9  ./bspmachine.cpp
10 ./bus.cpp
11 ./main.cpp
12 [...])
13
14 ADD_EXECUTABLE(bspmain bspmain.cpp)
15 TARGET_LINK_LIBRARIES(bspmain bsplab C++SIM
16                       Common Event pthread)
```

5.4 C++Sim på GCC 3.3

C++Sim er skrevet for å fungere på en rekke plattformer, blant annet Linux med GCC. Dessverre ble utviklingen av C++Sim avsluttet i 1997, noe som har hatt konsekvenser for kompatibiliteten med dagens kompilatorer. C++Sim var skrevet for GCC 2 og kompilerer ikke helt uten videre på GCC 3. Vi var på forhånd klar over dette ettersom det var nevnt i [Sund, 2004], der det blir foreslått at både konfigurasjonssystemet og syntaktiske forskjeller mellom GCC-versjonene skaper problemer. Vi har allerede løst det første problemet ved å benytte CMake til all konfigurering. Problemet med syntaktiske forskjeller mellom GCC-versjonene måtte løses ved å gjøre noen små endringer i koden til C++Sim.

5.4.1 Feil bruk av friend-nøkkelordet

Det første problemet vi fikk var at syntaksen som ble brukt for å deklarere en `friend`-klasse ikke fungerte i GCC 3.3. At en klasse deklareres som `friend` av en annen klasse betyr at klassen gis spesielle rettigheter til å aksessere `private` og `protected` variabler og funksjoner i en annen klasse som ellers ikke ville vært tilgjengelige. Endringen vi måtte gjøre kan ses i listing 5.7.

Listing 5.7: Hvordan en *friend*-klasse må deklareres i GCC 3.3

```
//orginal C++Sim kode
class bar {
  friend foo; //kompilerer ikke på GCC 3.3,
              //gir syntaktisk feil
};


//ny C++Sim kode, modifisert for å kompilere på GCC 3.3
class bar {
  friend class foo; //strengen ''class'' må forekomme
                    //foran navnet på friend-klassen
};
```

5.4.2 Bytte fra gammel til ny "stream"-standard

Det største problemet viste seg å oppstå på grunn av at GCC 3.3 har helt kuttet ut den gamle C++-standarden for *streams*, eller strømmer, og gått over til den nye standarden som baserer seg på *templates*. I utgangspunktet skal den nye standarden være bakoverkompatibel, men det viser seg ikke alltid å stemme helt.

Først måtte vi forandre alle referanser til klassen `istrstream` i *Common/src/Debug.cc* til det nye navnet `istringstream`.

Det viste seg så at ostream var forward-deklarert som vist i listing 5.8 i filene *Common/Uid.h*, *Common/Debug.h*, *Common/Filtsbuf.h*, *Common/Error.h*, *ClassLib/Process.h* og *ClassLib/thread.h*. Dette blir feil ettersom `ostream` er definert ved hjelp av *templates* i den nye standarden. I STL er alle typer strøm-klasser definert med *templates* som tar inn datatypen som skal lagres i strømmen, `_CharT`, og egenskapene til denne datatypen, `_Traits`. Vi ser denne deklarasjonen av `basic_ostream` i linje 2 til 3 i listing 5.9. Den mest brukte datatypen i strømmer er *bytes*, eller `char`, som det heter i C++. For at programmerere skal slippe å skrive `basic_ostream<char, char_traits<char> >` hver gang de skal bruke en ut-strøm med denne datatypen, og for bakoverkompatibilitet, er denne typen strøm typedefinert som `ostream`. Linje 5 i listing 5.9 viser denne typedefineringen.

Til tross for at forward-deklarasjon av klasser er nyttig (j.fr. 4.3.2), finner det merkelig at klasser som er deklarert i standard-headerfilene til C++ blir forward-deklarert. Disse header-filene vil bare forandre seg dersom man legger inn en ny versjon av kompilatoren, og vi ser derfor ingen spart kompileringstid som følge av denne forward-deklareringen. For å slippe å begynne med *templates* forward-deklareringer valgte vi heller å inkludere riktig header-fil som er *ostream* i dette tilfellet. Legg merke til at filnavnet ikke ender i *.h*. Dette er fordi at alle standard C++-headerfiler som følger den nye standarden ikke skal slutte i *.h*, mens de som følger den gamle standarden slutter i *.h*. Derfor vil man på kompilatorer som fortsatt støtter den gamle standarden finne den gamle deklarasjonen av `ostream` i *ostream.h*. Det er også en annen viktig forskjell mellom de gamle og de nye header-filene. De

nye header-filene befinner seg i navnerommet `std` som vi ser av linje 1 i listing 5.9. Så i tillegg til å bytte ut forward-deklarasjonen av `ostream` med *ostream* header-filen, måtte vi også bytte ut alle referanser til `ostream` i for eksempel *Common/Debug.h* med `std::ostream`. I alle implementasjonsfilene valgte vi i steden å legge til `using namespace std;` i starten av filene. Dette gjør at vi kan bruke alle klasser og funksjoner som ligger i `std`-navnerommet i resten av filen uten å skrive `std::` foran. Grunnen til at vi ikke gjorde det samme i header-filene er fordi at dette ville kunne ha forkludret navnerommet. Alle filer som hadde inkludert en slik header-fil ville også automatisk få tilgang til `std`-navnerommet selv om dette kanskje ikke er ønskelig.

Listing 5.8: Original forward-deklarering av ostream i C++Sim

```
class ostream;
```

Listing 5.9: Deklarering av ostream i den nye C++-standarden

```
1 namespace std {
2 template<typename _CharT, typename _Traits>
3 class basic_ostream;
4
5 typedef basic_ostream<char, char_traits<char> > ostream;
6 }
```

5.4.3 Bruk av den innbygde Boolean-typen

Den siste forandringen vi gjorde var strengt tatt ikke nødvendig, men ble gjort på grunn av ryddighet. Som mye gammel C++-kode bruker C++Sim egne typedefinisjoner for "boolean" typer. I *C++Sim/Include/Common/Boolean.h* var `unsigned short` typedefinert som `Boolean`. I tillegg var en konstant `Boolean` variabel ved navn `TRUE` satt lik 1, og en annen `FALSE` satt til 0. Vi fjernet likegodt hele filen og forandret alle referanser til disse til de innebygde variantene `bool`, `true` og `false`.

5.5 Endringer i BSPlab til plattformuavhengige alternativer

BSPlab benyttet seg av en del C++-elementer som kun er tilgjengelige på Windows. Noen av disse har tilsvarende funksjonskall som er plattformuavhengige og en del av C++-standardbibliotekene. Disse trengte vi bare endre til å benytte de plattformuavhengige alternativene og kunne la koden være lik på Linux og Windows.

5.5.1 Bruk av __min og __max

BSPlab benytter to makroer som kun eksisterer på Windows til å sammenlikne to tall og finne det høyeste eller laveste. Disse heter `__min` og `__max`. Det finnes helt tilsvarende funksjoner som en del av *Standard Template Library* (STL) som heter `std::min` og `std::max`. Disse finnes i *algorithm* headerfilen. Vi byttet ut alle plassene disse makroene ble brukt med den tilsvarende STL-varianten. Denne endringen påvirker dermed både Linux- og Windows-koden, men funksjonaliteten blir den samme og skal fungere likt på begge plattformer.

5.6 Funksjonalitet som måtte implementeres forskjellig på Windows og Linux

Det viste seg at enkelte funksjoner BSPlab er avhengig av ikke har noe plattformuavhengig alternativ. Vi valgte derfor, i likhet med løsningen som ble valgt i [Sund, 2004], å sette opp koden slik at forskjellig kode ble kompilert avhengig av plattform. Vi abstraherte dette ut av den originale BSPlab-koden ved å opprette to filer; *platform.h* og *platform.cpp* som inneholdt all den plattformavhengige koden.

5.6.1 Egen random-funksjon

`random` er en funksjon som benyttes til å få ut et pseudorandom tall. BSPlab har definert sin egen random-funksjon. På Linux er det en random-funksjon som også heter `random` og som er en del av standardbibliotekene. GCC gir feilmelding om at det nå er flere definisjoner av denne siden den nå eksisterer to steder. Det er ønskelig at samme random-funksjon benyttes på både Windows og Linux. Dette gjør det mulig å kjøre samme BSP-program på begge plattformene og kunne forvente at de gir samme resultat. Løsningen ble å endre navn på funksjonen i BSPlab til `BSP_random` istedet for `random`. Vi ser hvordan denne ble i listing 5.10. Det står oppgitt i BSPlab koden at disse funksjonene er tatt fra `rand` og `seed` på Windows. Vi flyttet også denne koden fra *networkadapter.cpp* til *platform.cpp* ettersom denne funksjonaliteten er noe som ikke bare er nyttig for NOW-arkitekturen. Det ble nødvendig å deklarere en variabel *BSP_RAND_MAX* istedet for *RAND_MAX* ettersom variabelen som blir brukt til å representere den maksimale verdien `BSP_random` returnerer heter *RAND_MAX* på begge plattformer, men har forskjellig verdi. På Windows har *RAND_MAX* verdien 32767, mens på Linux har den verdien 2147483647. Vi definerte *BSP_RAND_MAX* til å ha verdien 32767 ettersom `BSP_random` benytter denne verdien og byttet ut *RAND_MAX* med *BSP_RAND_MAX* der den ble brukt i *networkadapter.cpp*.

Listing 5.10: BSP_random() i platform.cpp

```
static long curr_rand = 1L;

int BSP_random()
{
  return( ((curr_rand = curr_rand * 214013L + 2531011L)
                                    >> 16) & 0x7fff );
}

void my_seed(unsigned int seed)
{
  curr_rand = seed;
}
```

5.6.2 BSP_Platform_flushall()

BSPlab benytter i `BSPMachine`-klassen en funksjon for å tømme alle buffere til alle åpne kanaler. I den originale BSPlab-koden gjøres dette med funksjonen `_flushall`. Denne funksjonen er spesifikk for Windows-plattformen og en funksjon med samme navn eksisterer ikke på Linux. Det finnes en funksjon i standardbibliotekene på Linux som gjør samme nytten. Denne er deklarert i headerfilen *cstdio* og heter `fflush`. Dersom `fflush` kalles med parameteren *NULL* tømmer den bufferene til alle åpne kanaler. Vi deklarerte funksjonen `BSP_Platform_flushall` i *platform.h* og vi kan se hvordan implementasjonen i *platform.cpp* ble i listing 5.11.

Listing 5.11: BSP_Platform_flushall() i platform.cpp

```
#ifdef _WIN32
  void BSP_Platform_flushall() {_flushall();}
#else
  void BSP_Platform_flushall() {fflush(NULL);}
#endif
```

Nå var det bare å bytte ut plassene `_flushall`-funksjonen ble brukt med den nye funksjonen `BSP_Platform_flushall` for å få samme funksjonalitet på Windows og Linux.

5.6.3 BSP_Platform_sleep(int msec)

For å få programmet til å vente et visst antall millisekunder benyttes funksjonen `Sleep`. Denne funksjonen finnes ikke på Linux. Det finnes en funksjon som heter `sleep` (liten forbokstav i motsetning til den som benyttes i BSPlab på Windows), men denne tar inn antall sekunder den skal vente, ikke antall millisekunder. På Linux har man et alternativ

som heter `nanosleep` som tar inn en struktur som beskriver hvor mange nanosekunder funksjonen skal vente. Løsningen for å få en funksjon med det samme grensesnittet på både Windows og Linux ble å deklarere `BSP_Platform_sleep` i *platform.h*. Vi kan se hvordan implementasjonen i *platform.cpp* ble i listing 5.12.

Listing 5.12: BSP_Platform_sleep() i platform.cpp

```
#ifdef _WIN32
  void BSP_Platform_sleep(int msec) {Sleep(msec);}
#else
  #include <memory>

  void BSP_Platform_sleep(int msec)
  {
    timespec spec;
    memset(&spec, 0, sizeof(timespec));
    spec.tv_nsec = msec * 1000000;
    nanosleep(&spec, NULL);
  }
#endif
```

Siden funksjonen på Linux tar inn antallet nanosekunder programmet skal vente, må verdien `BSP_Platform_sleep` tar inn multipliseres med én million. Headerfilen *memory* må inkluderes fordi `memset` benyttes til å nulle ut strukturen som sendes til nanosleep før det settes noen verdier i den. Nå var det bare å bytte ut de plassene der `Sleep` ble benyttet med den nye funksjonen `BSP_Platform_sleep`.

5.6.4 BSP_Platform_filelength(FILE* p)

BSPlab benytter en funksjon som heter `filelength` til å sjekke størrelsen til en fil. Denne funksjonen eksisterer kun på Windows. På Linux finnes det en funksjon som heter `fstat`. Denne henter ut en rekke informasjon om en fil, blant annet størrelsen. Vi deklarerte funksjonen `BSP_Platform_filelength` i *platform.h*. Implementasjonen i *platform.cpp* kan ses i listing 5.13.

Listing 5.13: BSP_Platform_filelength() i platform.cpp

```
#ifdef _WIN32
  #include <io.h>
  int BSP_Platform_filelength(FILE* p)
  {
    return filelength(_fileno(p));
  }
#else
  #include <sys/stat.h>
  int BSP_Platform_filelength(FILE* p)
  {
    struct stat buf;
    fstat(fileno(p), &buf);
    return buf.st_size;
  }
#endif
```

På Windows fungerer `filelength` som før. På Linux fylles en datastruktur med informasjon om fila som så størrelsen hentes ut fra. Vi byttet ut `filelength` med `BSP_Platform_filelength` i BSPlab for å gjøre koden plattformuavhengig.

5.6.5 Plattformuavhengig timing-kode

BSPlab benytter seg av tidtakning for å kontrollere hvor lang tid et BSP-funksjonskall tar. Dette gjøres ved at BSPlab måler et tidspunkt når en BSP-funksjon kalles og tar differansen mot et tidspunkt når funksjonen er ferdig utført. På Windows brukte BSPlab funksjonen `QueryPerformanceCounter` for å måle disse tidspunktene. Denne funksjonen er ikke tilgjengelig på Linux, så vi valgte å bruke `gettimeofday` istedet. Datastrukturene disse funksjonene benytter er forskjellige, så vi valgte å løse dette på samme måte som i [Sund, 2004] ved å deklarere en plattformavhengig type, *BSPTime*. Denne kan ses i listing 5.14.

Listing 5.14: BSPTime i platform.h

```
#ifdef _WIN32
  #include <windows.h>

  struct BSPTime {
    LARGE_INTEGER time;
    LARGE_INTEGER freq;
  };

#else
  #include <sys/time.h>
  #include <time.h>
  struct BSPTime {
    timeval time;
  };
#endif
```

Vi hadde nå en plattformuavhengig type vi kunne bruke til å uttrykke tid. Det som gjenstod var å definere to metoder som benytter denne datatypen. Én metode for å starte tidtakningen på et *BSPTime*-objekt og én metode for å finne ut hvor lang tid som var gått siden tidtakningen startet på det objektet. Vi deklarerte metodene `BSP_Timing_Init` og `BSP_Timing_End` i *platform.h* og implementerte de i *platform.cpp*. Listing 5.15 viser implementasjonen av `BSP_Timing_Init` og listing 5.16 viser `BSP_Timing_End`.

Listing 5.15: BSP_Timing_Init() i platform.cpp

```
#ifdef _WIN32

bool BSP_Timing_Init(BSPTime& time)
{
  static LARGE_INTEGER freq;
  static bool first = true;
  if(first) {
    QueryPerformanceFrequency(&freq);
    first = false;
  }
  time.freq = freq;
  return QueryPerformanceCounter(&time.time);
}

#else

bool BSP_Timing_Init(BSPTime& time)
{
  return !gettimeofday(&time.time, NULL);
}

#endif
```

Her settes *BSPTime*-objektet som sendes inn til funksjonen til en verdi som representerer tidspunktet metoden kalles. Denne verdien vil senere bli brukt til å beregne hvor lang til som har gått siden denne metoden ble kalt.

Listing 5.16: BSP_Timing_End() i platform.cpp

```
#ifdef _WIN32

double BSP_Timing_End(BSPTime& starttime)
{
  LARGE_INTEGER end;
  QueryPerformanceCounter(&end);
  return (static_cast<double>(end.QuadPart)
        - static_cast<double>(starttime.time.QuadPart))
        / static_cast<double>(starttime.freq.QuadPart);
}

#else

double BSP_Timing_End(BSPTime& starttime)
{
  double t1, t2;
  BSPTime end;
  gettimeofday(&end.time, NULL);
  t1 = static_cast<double>(starttime.time.tv_sec)
     + (static_cast<double>(starttime.time.tv_usec)
     / 1000000);
  t2 = static_cast<double>(end.time.tv_sec)
     + (static_cast<double>(end.time.tv_usec)
     / 1000000);
  return t2 - t1;
}

#endif
```

Denne metoden tar inn et objekt som på forhånd har vært initialisert av `BSP_Timing_Init` og returnerer en verdi som representerer tiden som har gått siden objektet ble initialisert.

Med disse to metodene kunne vi bytte ut koden som gjør tidsberegningene i BSPlab med vår plattformuavhengige versjon.

5.7 Feil i koden

Det er slik at når man bytter kompilator og plattform så kan det dukke opp feil som tidligere ikke har vært synlige. Dette skyldes at ulike kompilatorer vil generere kode som oppfører seg litt forskjellig og ulike plattformer vil sette opp minnet der programmet kjører på forskjellige måter. Dette kan resultere i at feil som tidligere ikke har blitt oppdaget, fordi de tilfeldigvis ikke har fått noen konsekvens, resulterer i at hele programmet krasjer eller ikke oppfører seg korrekt. Dette skjedde, ikke helt uventet, når vi begynte å kjøre tester på

Linux.

5.7.1 NOW-arkitekturen

Som beskrevet i [Sund, 2004] fungerte ikke NOW-arkitekturen på Linux uten videre og det er ikke beskrevet noen forklaring eller løsning på problemet. Å kjøre simulasjoner på NOW resulterte i at programmet noen ganger krasjet med en *segmentation fault* eller *aborted* feilmelding fra operativsystemet. En feil av denne typen skyldes som oftest at programmet ikke håndterer minnet riktig og at operativsystemet dermed ber det avslutte. Det at feilen bare oppstod en gang i blant, og tilsynelatende ganske tilfeldig, fikk oss til å mistenke at det dreide seg om en *race condition* mellom tråder. Denne typen feil oppstår sporadisk ettersom et program som bruker tråder, slik BSPlab gjør, ikke kjører gjennom på nøyaktig samme måte hver gang. Mistanken vår ble styrket da det viste seg at programmet aldri krasjet når vi kjørte det i en debugger. I debuggeren kjører alt mye saktere enn når programmet kjører normalt, noe som typisk fører til at tilstanden som får programmet til å krasje på grunn av interaksjon mellom forskjellige tråder, ikke oppstår. Denne typen feil kan være vanskelig å løse og krever ofte en fullstendig analyse av interaksjonen mellom trådene i programmet for og utbedres. Heldigvis viste det seg at våre mistanker om en *race condition* ikke stemte og at feilen kun skyltes feil håndtering av minnet uten at interaksjon mellom tråder var med på å påvirke den.

Utbedring av feilen i NOW-arkitekturen

NOWBSPMachine-klassen tar seg av implementasjonen av NOW-arkitekturen. Denne klassen arver fra BusBSPMachine. Feilen som fikk NOW-maskina til å kjøre ustabilt på Linux skyldes at en peker blir slettet to ganger i NOWBSPMachine-klassen. Deler av BusBSPMachine-klassen kan ses i listing 5.17.

Listing 5.17: Deler av BusBSPMachine-klassen

```
class BusBSPMachine : public BSPMachine {
// Attributes
protected:
  Bus* m_pBus; // The bus that transfers the data
};
```

Her deklareres en peker til et objekt av typen *Bus*. Denne pekeren blir så slettet i destruktoren til BusBSPMachine som ses i listing 5.18

Listing 5.18: BusBSPMachine-destruktoren

```
BusBSPMachine::~BusBSPMachine()
{
  // Remove and delete the bus itself
  if (m_pBus != NULL) {
    delete m_pBus;
  }
}
```

Denne destruktoren sørger for at minnet blir ryddet opp dersom *m_pBus*-pekeren har blitt tilordnet et objekt. Dette hindrer en minnelekasje som ellers kunne oppstått. Problemet oppstår i NOWBSPMachine-klassen sin destruktor som kan ses i listing 5.19

Listing 5.19: NOWBSPMachine-destruktoren

```
NOWBSPMachine::~NOWBSPMachine()
{
  if (m_pNoise != NULL) {
    m_pNoise->WaitForTermination();
    delete m_pNoise;
  }

  if (m_pBus != NULL) {
    delete m_pBus;
  }
}
```

Her ser vi at *m_pBus* blir slettet også i denne destruktoren. Det vil si at dersom denne pekeren har blitt initialisert til noe, vil programmet forsøke å slette den to ganger. Dette er ikke gyldig å gjøre i C++ og fører til at programmets oppførsel nå er udefinert. Feil av typen der C++-standarden ikke definerer hva som skjer dersom noe gjøres er gode kandidater til å bli oppdaget når koden blir portet over til en ny kompilator eller plattform. Denne feilen var ikke blitt oppdaget på Windows fordi kompilatoren her tydeligvis genererer kode som i dette tilfellet ikke fikk programmet til å krasje eller oppføre seg ukorrekt. Når vi byttet kompilator og plattform resulterte feilen i at det nå ble generert kode som i enkelte tilfeller krasjer programmet når den kjører. Å utbedre problemet ble enkelt gjort ved å fjerne koden som sletter pekeren i NOWBSPMachine destruktoren.

5.7.2 Viktigheten av virtuelle destruktorer

C++ har dessverre en del latente fallgruver som det er lett å gå i dersom man ikke passer nøye på å gjøre ting på riktig måte. En av disse dukker opp når det benyttes arving og det som kalles for *polymorphism*. *Polymorphism* er en teknikk der man utnytter at en klasse

kan arve egenskapene til en annen klasse og samtidig, dersom den ønsker det, reimplementere deler av funksjonaliteten som arves. Dette er en svært nyttig teknikk og benyttes ofte flittig i god objektorientert kode [Webopedia, 2005]. I C++ er det svært viktig å ikke glemme å deklarere destruktoren til en klasse som virtuell dersom klassen skal benyttes i kode som utnytter *polymorphism*. En tommelfinger-regel er at dersom en klasse inneholder noen virtuelle funksjoner, så skal også destruktoren gjøres virtuell for sikkerhetsskyld [Meyers, 1997]. Grunnen til dette er at dersom en peker til en arvet polymorphet klasse slettes, er det i følge C++ standarden udefinert hva som skjer dersom destruktoren ikke er virtuell. Det som i praksis typisk skjer er at kun destruktoren til den arvede klassen kalles, men ikke til klassen selv, slik at man kan miste minne.

Mangel på virtuelle destruktorer i BSPlab

BSPlab benytter mye arving og *polymorphism* sammen med C++Sim for å på en elegant måte støtte mange forskjellige parallelle arkitekturer innenfor det samme rammeverket. Det viste seg at destruktoren til bspmachine-klassen ikke var virtuelle selv om den måtte være det for at minnet skulle bli ryddet opp korrekt. Dette er en relativt alvorlig feil ettersom alle de forskjellige arkitektur-klassene arver fra denne klassen, noe som kan ha forårsaket minnelekkasje i alle arkitekturene. Vi løste dette ved ganske enkelt å gjøre destruktoren i bspmachine-klassen virtuell. Det interessante var at når vi etterpå kjørte tester, krasjet konsekvent network-arkitekturen når *all port*-parameteren var satt.

Retting av feilen

Når vi gjorde destruktoren til bspmachine-klassen virtuell forårsaket dette at en arkitektur som tidligere hadde fungert nå begynte å krasje. Vi fant ut grunnen til dette etter en del debugging. Det viste seg at når destruktoren til network-maskina nå ble kalt, satte dette i gang en kjedeeffekt som førte til at destruktoren til portmodel-klassen også ble kalt. Det hadde den ikke blitt tidligere. Destruktoren til portmodel kan ses i listing 5.20

Listing 5.20: PortModel-destruktoren

```
PortModel::~PortModel()
{
  delete m_pInChan;
  delete m_pOutChan;
}
```

Her slettes to pekere til objekter av typen NetworkLink. Det er flere ting som er skrevet på en litt uheldig måte her som forårsaker at dette krasjer. For det første blir ikke disse pekerene satt til verdien NULL i konstruktoren til PortModel. For det andre blir de heller ikke satt til noen verdi i AllPort konstruktoren. Ettersom hverken PortModel-klassen eller en subklasse setter disse pekerene, vil dette krasje. Det er nettopp det som skjer i AllPort klassen som

arver fra PortModel. Ettersom AllPort konstruktoren aldri setter disse pekerene til noen bestemt verdi, peker de til tilfeldig minne når de blir slettet og dermed krasjer programmet. Dersom pekerene hadde hatt verdien NULL ville ingenting galt skjedd ettersom C++-standarden spesifiserer at å slette en peker som har verdien NULL ikke gjør noe som helst [Meyers, 1997]. Vi løste denne feilen ved å legge til i PortModel sin konstruktor (listing 5.21) at `m_pInChan` og `m_pOutChan` tar verdien NULL i utgangspunktet.

Listing 5.21: PortModel-konstruktoren

```
PortModel::PortModel() :
            m_nInportNo(0), m_nOutportNo(0),
            m_pInChan(NULL), m_pOutChan(NULL) {};
```

Det står i en kommentar i portmodel.cpp at det ikke er meningen at det skal lages objekter av typen *PortModel*, men at denne klassen kun skal arves og spesialiseres av subklasser. En slik klasse kalles i objektorientert sammenheng for en *abstrakt* klasse. I C++ kan en klasse spesifiseres til å være abstrakt ved at minst en av metodene i klassen er virtuell og satt lik 0. Vi valgte å gjøre PortModel-klassen abstrakt ved å sette alle funksjonene det er meningen at skal reimplementeres lik 0. Se listing 5.22 for hvordan dette ser ut.

Listing 5.22: Deler av PortModel-klassen

```
class PortModel {
  virtual int Busy(int iNode) = 0;
  virtual void ReserveInport (int iNodeId) = 0;
  virtual void ReserveOutport(int iNodeId,
                              int iRoutingModel) = 0;

  virtual void ReleaseInport(int iNodeId) = 0;
  virtual void ReleaseOutport(int iNodeId) = 0;
}
```

Dette ble gjort fordi klassen skal være abstrakt og C++ støtter å spesifisere dette. Det vil dermed gå enda klarer frem for programmerere som skal vedlikeholde koden i fremtiden hva som er intensjonen og forhindre at noen forsøker å instansiere PortModel direkte, ettersom kompilatoren da vil gi en feilmelding.

5.8 Gjenstående problem på Linux

Under utviklingen av BSPlab i 1997 ble det oppdaget et problem der operativsystemet i enkelte tilfeller ikke frigjorde en tråd umiddelbart når det ble bedt om det. Dette resulterte i at BSPlab krasjet når et *Entity*-objekt ble slettet mens tråden i det fortsatt kjører. Dette

ble løst i den originale BSPlab ved at en funksjon ble lagt til i C++Sim som sjekker om tråden i objektet fortsatt kjører. På den måten kunne man la være å slette objektet dersom tråden ikke hadde terminert ennå. Måten dette ble gjort på i Windows var gjennom et kall til funksjonen `WaitForSingleObject` som sjekker om tråden i objektet fortsatt kjører. Denne funksjonen er kun tilgjengelig på Windows og det finnes ikke noe alternativ til den på Linux som gir samme funksjonalitet [Hundhammer, 2005]. Samme problem oppstår på Linux som på Windows; noen ganger terminerer ikke tråden umiddelbart når den får beskjed om det. Hvorfor det er slik er usikkert. Det er mulig at det dreier seg om at en ressurs er delt mellom flere tråder inne i C++Sim, slik at tråden ikke kan avsluttes med en gang den får beskjed om det, men dette er bare spekulasjoner. Når C++Sim lager tråder på Linux får ikke disse nødvendigvis en egen *process id* (pid), slik at programmet har ingen måte å sjekke om tråden faktisk er terminert ennå eller ikke. Dette er samme konklusjon som ble nådd i [Sund, 2004]. Etter at hverken egne analyser eller leting etter en løsning i *pthreads*-dokumentasjonen ga resultater, bestemte vi oss for å akseptere en minnelekkasje i BSPlab på Linux for å få stabilitet. Implementasjonen av BSPlab på Linux lar være å slette *Entity*-objekter for å unngå krasjer og ustabiliteter som følge av at tråden i objektet ikke rekker avslutte for objektet slettes. Dette fører til at BSPlab på Linux lekker minne. Om denne minnelekasjen er så alvorlig at den fører til at det ikke kan kjøres store simulasjoner på Linux er uvisst, men i følge testingen som er gjort i kapittel 7 er dette hvertfall ingen fatal feil.

5.9 BSPlab på MinGW

Etter at vi hadde BSPlab opp å kjøre på Linux var vi kommet frem til den siste fasen av flyttingen. Nå skulle vi få BSPlab til å kompilere og kjøre i MinGW. Ettersom MinGW i prinsippet er akkurat den samme kompilatoren som blir brukt på Linux, burde ikke dette by på for mange problemer. Noe som er et typisk problem for "flytting" mellom disse kompilatorene er bruk av Unix-spesifikke funksjoner. Selv om MinGW er GCC for Windows finner man ikke nødvendigvis alle de samme funksjonene i MinGW som på Linux ettersom noen av disse ikke trenger å være støttet av operativsystemet. Da vi allerede hadde fått BSPlab opp å kjøre i nyeste versjon av *Visual Studio* før vi begynte flyttingen til Linux, og også tenkt nøye igjennom eventuelle plattform-avhengigheter under flyttingen, viste dette seg ikke å være noe stort problem.

5.9.1 MinGW MSYS

For å få brukt automake-verktøyet vårt sammen med MinGW var vi nødt til å ha et minimalt Unix-miljø. Vi valgte å bruke MSYS [MSYS, 2005], et lett og enkelt Unix-miljø for Windows som også blir distribuert av MinGW organisasjonen. Når CMake genererer "Unix Makefiles", som er den type bygge-filer MinGWs bygge-program *make* kan bruke, legger CMake også inn en del avhengighetssjekker i bygge-filen som krever noen Unix shell-programmer. For enklest å kunne bruke CMake sammen med MinGW må vi derfor bruke

MSYS på Windows.

5.9.2 BSPlab

Etter at vi hadde fått installert MSYS trengte vi bare å bruke automake-verktøyet vårt, CMake, til å lage bygge-filer for oss. Nå viste det seg hvor praktisk det er å ha et plattform-uavhengig automake-verktøy. Det eneste som måtte forandres i prosjektfilene til CMake var at den ikke skal linke BSPlab mot et trådbibliotek dersom plattformen er Windows. Dette skyldes at på Windows ligger trådfunksjonene i standard-bibliotekfilene alle Windows-programmer blir linket mot. På linje 2 i listing 5.23 ser vi hvilke biblioteker testprogrammer som følger med BSPlab er satt opp til å linke mot. Utover BSPlab- og C++-Sim-bibliotekene ser vi *pthread* som er et trådbibliotek for Unix. Vi løste problemet med å sette inn en if-løkke i konfigurasjonsfilen til CMake som ber den om ikke å linke mot pthread dersom plattformen er Windows. I linje 2 til 6 i listing 5.24 ser vi at variablen `LINK_LIBRARIES` ikke blir satt til å inkludere biblioteket *pthread* dersom plattformen er Windows. I linje 7 blir BSPlab-eksempelprogrammet satt til å linke mot bibliotekene i denne variabelen. Med denne lille forandringen kompilerte BSPlab uten problemer.

Listing 5.23: Forandring i BSPlibs CMakeLists.txt

```
1 ADD_EXECUTABLE(bspmain bspmain.cpp)
2 TARGET_LINK_LIBRARIES(bspmain bsplab C++SIM
3                       Common Event pthread)
```

Listing 5.24: Forandring i BSPlibs CMakeLists.txt

```
1 ADD_EXECUTABLE(bspmain bspmain.cpp)
2 IF( WIN32 )
3     SET(LINK_LIBRARIES bsplab C++SIM Common Event)
4 ELSE( WIN32 )
5     SET(LINK_LIBRARIES bsplab C++SIM Common Event pthread)
6 ENDIF( WIN32 )
7 TARGET_LINK_LIBRARIES(bspmain ${LINK_LIBRARIES})
```

5.9.3 C++Sim

Heller ikke C++Sim trengte forandringer i koden for å kompilere i MinGW. Det trengtes også her bare små forandringer i CMake prosjektfilene. Ettersom C++Sim er laget med veldig mange plattformer og kompilatorer i tankene, finnes det mange C++ preprosessor definisjoner man kan sette avhengig av hvilen "oppførsel" man vil ha. På Linux hadde vi satt en rekke definisjoner for å få kompilert C++Sim på ønsket måte, utdrag av *CMakeLists.txt*-filen på Linux vises i listing 5.25. På Windows trengtes bare én av disse definisjonene, så

vi forandret CMake-filen som vist i listing 5.26.

Listing 5.25: C++Sims CMakeLists.txt på roten

```
ADD_DEFINITIONS(-DPTHREAD_DRAFT_LINUX -DPOSIX_THREAD
  -DHAVE_BOOLEAN_TYPE -DHAVE_WCHAR_TYPE -DHAVE_LONG_LONG
  -DGCC_STATIC_INIT_BUG -DMEMFNS_IN_STRING_H
  -DNEED_MALLOC_T -DHAVE_MSGCONTROL -D__LINUX__ -DSYSV
  -D_CONSTVALUE="" -D_CONSTVALUE2="" -D_NORETURN=""
  -D_NORETURN2="" -DProcessList_Queue)
```

Listing 5.26: C++Sims nye CMakeLists.txt på roten

```
IF( WIN32 )
    ADD_DEFINITIONS(-DProcessList_Queue)
ELSE( WIN32 )
    ADD_DEFINITIONS(-DPTHREAD_DRAFT_LINUX -DPOSIX_THREAD
      -DHAVE_BOOLEAN_TYPE -DHAVE_WCHAR_TYPE -DHAVE_LONG_LONG
      -DGCC_STATIC_INIT_BUG -DMEMFNS_IN_STRING_H
      -DNEED_MALLOC_T -DHAVE_MSGCONTROL -D__LINUX__ -DSYSV
      -D_CONSTVALUE="" -D_CONSTVALUE2="" -D_NORETURN=""
      -D_NORETURN2="" -DProcessList_Queue)
ENDIF( WIN32 )
```

Forsikre kompilering av riktig tråd-kode

En vanlig måte å implementere bruk av forskjellig tråd-kode på ville vært å ha en `Thread`-klasse med virtuelle metoder, og så implementere den koden som var forskjellig mellom trådpakkene i for eksempel en `NTThread`- og `PThread`-klasse som arvet fra `Thread`. Deretter kunne man ha instansiert riktig trådklasse ved hjelp av eksempelvis preprosessordefinisjoner i koden.

I C++Sim har de derimot valgt å ikke bruke arv. Alle `Thread`-metoder som er felles for trådpakkene er definert i filen *thread.cc*, mens metodene som er spesielle for tråd-pakken blir definert i for eksempel *posix_thread.cc* og *nt_thread.cc* for henholdsvis Posix- og NT-tråder. Det er derfor viktig at ikke flere enn én av de spesielle tråd-pakke-filene blir forsøkt linket inn i BSPlab-biblioteket ettersom dette vil gi *multiple definitions*-feil. I listing 5.27 ser vi et utsnitt av den opprinnelige *CMakeLists.txt* som ligger i *ClassLib/src* under C++Sim-området. For å få CMake til å bruke *nt_thread.cc* i stedet for *posix_thread.cc* på Windows, brukte vi som vist i listing 5.28 samme strategi som tidligere.

Hvis BSPlab senere skal utvides til å kjøre på plattformer som har flere trådpakker tilgjengelige burde kanskje CMake-filen utvides til å spørre brukeren om hvilken trådpakke han vil bruke.

Listing 5.27: C++Sims CMakeLists.txt på underområde

```
ADD_LIBRARY(C++SIM Process.cc Random.cc thread.cc
                ProcessList.cc ProcessCons.cc ProcessIterator.cc
                posix_thread.cc)
```

Listing 5.28: C++Sims CMakeLists.txt på underområde

```
IF( WIN32 )
    SET(LIBCPPSIM_SOURCE_FILES Process.cc Random.cc
        thread.cc ProcessList.cc ProcessCons.cc
        ProcessIterator.cc nt_thread.cc)
ELSE( WIN32 )
    SET(LIBCPPSIM_SOURCE_FILES Process.cc Random.cc
        thread.cc ProcessList.cc ProcessCons.cc
        ProcessIterator.cc posix_thread.cc)
ENDIF( WIN32 )
ADD_LIBRARY(C++SIM ${LIBCPPSIM_SOURCE_FILES})
```

5.10 Nye BSPlab på Visual Studio

Selv om det ikke var en del av oppgaven at BSPlab fortsatt skulle fungere på Visual Studio, valgte vi å forsikre oss om at det fortsatt kompilerer og kjører etter alle tilpasningene våre var gjort. Vi mener det er viktig at selv om installasjonsprogrammet ikke skal støtte direkte installasjon mot Visual Studio, så bør ekspertbrukere som bruker Visual Studio fortsatt ha mulighet til å bruke BSPlab.

Å kompilere den nye versjonen av C++Sim og BSPlab viste seg å være helt uproblematisk. Igjen var det CMake som gjorde mye av jobben for oss. Ved hjelp av CMake genererte vi Visual Studio prosjektfiler for C++Sim og BSPlab, som vi så åpnet i Visual Studio og kompilerte. Ingen forandringer av prosjektfilene til CMake var nødvendig utover de vi allerede hadde gjort for å kompilere BSPlab i MinGW.

5.10.1 Eldre utgaver av Visual Studio

Vi har ikke fått tilgang til eldre versjoner av Visual Studio for å teste om BSPlab fungerer der, men fra tidligere erfaring finner vi det lite sannsynlig at BSPlab fungerer på Visual Studio 6.0 eller eldre. Grunnen til dette er at disse versjonene av Visual Studio ikke er fullstendig kompatible med nyere versjoner av C++-standarden, spesielt *Standard Template Library* (STL) standarden for *streams*. Vår reviderte utgave av C++Sim er forandret til å bruke denne nyeste standarden av flere grunner, hvorav den viktigste er at den nyeste versjonen av GCC ikke er bakoverkompatibel og ikke kompilerer originale C++Sim. En

annen grunn er at vi ønsker å gjøre hele BSPlab-pakken så vedlikeholdsvennlig som mulig, noe den ikke blir ved å benytte kode som etterhver kanskje ikke vil fungere på nyere kompilatorer. Vi vil tro at de som ønsker å bruke BSPlab i Visual Studio i stedet for i det gratis utviklermiljøet vi distribuerer med BSPlab, har tilgang på nyere versjoner av Visual Studio. Utover det vil vi påpeke at Visual Studio ikke er "offisielt støttet" som utviklerverktøy for BSPlab.

Kapittel 6

Automatisk installasjon

Dette kapittelet vil beskrive hvordan vi gikk frem for å lage installasjonsprogrammer for Windows og Linux. Ettersom disse plattformene er svært forskjellig med tanke på hvordan brukere er vant til å installere programmer på de, valgte vi å lage to forskjellige løsninger, én for Windows og én for Linux. Windows-varianten forsøker å være så lik som mulig installasjonsprogrammene til typiske Windows-programmer, slik at det blir så enkelt som mulig for Windows-brukere å installere BSPlab. På Linux valgte vi å gå for en løsning som ikke krever et grafisk brukergrensesnitt og som tilbyr den fleksibiliteten Linux-brukere er vant til å ha i installasjonsprogrammer.

6.1 Automatisk installasjon på Windows

Det finnes en rekke programmer man kan benytte for å lage installasjonsprogrammer for Windows. Vi hadde fra før lite erfaring med dette, så vi søkte på internett etter programmer vi kunne benytte for å lage et slikt program. Vi fant først et program fra Microsoft som heter *Visual Studio Installer* [Visual Studio Installer, 2005]. Dette hadde all funksjonaliteten vi trengte, men det krever en lisens for å benyttes. Vi bestemte oss for å forkaste dette alternativet på grunn av kravet om lisens. Det neste programmet vi fant var *Inno Setup Compiler* [Inno Setup Compiler, 2005]. Dette programmet så ut til å ha all funksjonaliteten vi ville ha bruk for og er i tillegg gratis å benytte. Ettersom oppgaven som skal løses er relativt enkel, og dette programmet så ut til å ha det som trengs for å løse den, bestemte vi oss for å ikke lete videre etter flere alternativer.

6.1.1 Inno Setup Compiler

Å bruke dette programmet viste seg å være overraskende enkelt. Det tok ikke lang tid før vi hadde en test som installerte noen filer og som gjorde det mulig å avinstallere filene igjen. Ettersom vi ønsket at vår installasjonspakke for Windows skulle inneholde alt man trenger

for å benytte seg av BSPlab trengte vi litt mer avansert funksjonalitet enn kun å kopiere inn filer.

Installasjon av Dev-Cpp med MinGW

Vi ønsket at installasjonsprogrammet skulle ha mulighet til å installere *Dev-Cpp* med *MinGW* for brukeren automatisk dersom dette er ønskelig. Ettersom hovedplattformen på Windows er *MinGW* mener vi det er viktig at dette verktøyet følger med BSPlab på Windows og at det enkelt lar seg installere. Dersom brukeren har installert *Dev-Cpp* fra før skal han kunne velge bort at vårt installasjonsprogram legger det inn.

Installasjon av CMake og MSYS

Vi har lagt mindre vekt på å ha et ferdig opplegg for *Microsoft Visual Studio .Net* (VS.Net) ettersom målet med oppgaven var å få flyttet BSPlab vekk fra denne plattformen. Vi har alikevel valgt å legge til rette for at de som ønsker å benytte BSPlab på VS.Net skal kunne gjøre dette med vår versjon uten problemer. Måten vi har lagt opp dette på er at vi sender CMake og MSYS med i installasjonspakken vår, slik at brukeren selv kan generere prosjektfiler til VS.Net. Se tutorial i seksjon 9.1.2 for mer informasjon om hvordan man benytter CMake til å generere prosjektfiler for VS.Net.

Skriptfilen til Inno Setup Compiler

Inno Setup Compiler benytter seg av et skript som skrives med en bestemt syntaks for å generere et installasjonsprogram bestående av én fil. Deler av skriptet vi lagde kan ses i listing 6.1. Mindre viktige deler av skriptet er utelatt av plasshensyn.

Listing 6.1: Inno Setup Compiler skriptet for å installere BSPlab

```
[Tasks]
Name: devcpp; Description: Install DevCpp and MinGW
Name: cmake; Description: Install CMake
            (for Visual Studio project file generation);
            Flags: unchecked
Name: msys; Description: Install MSYS (needed if you want
            to rebuild BSPlab or C++SIM);
            Flags: unchecked

[Files]
Source: bsplab\*; DestDir: {app}\bsplab; Components: main;
     Flags: ignoreversion createallsubdirs recursesubdirs
Source: bspprog\*; DestDir: {app}\bspprog; Components: main;
     Flags: ignoreversion createallsubdirs recursesubdirs
Source: cppsim\*; DestDir: {app}\cppsim; Components: main;
     Flags: ignoreversion createallsubdirs recursesubdirs
Source: jbspsim\*; DestDir: {app}\jbspsim; Components: main;
     Flags: ignoreversion createallsubdirs recursesubdirs
Source: distclean; DestDir: {app}; Components: main;
     Flags: ignoreversion
Source: devcpp4990setup.exe; DestDir: {tmp};
     Flags: ignoreversion nocompression
Source: CMSetup206.exe; DestDir: {tmp};
     Flags: ignoreversion nocompression
Source: MSYS-1.0.10.exe; DestDir: {tmp};
     Flags: ignoreversion nocompression

[Types]
Name: full; Description: Full installation; Flags: iscustom

[Components]
Name: main; Description: Main BspLab Files;
           Types: full;

[Run]
Filename: {tmp}\devcpp4990setup.exe; Tasks: devcpp
Filename: {tmp}\CMSetup206.exe; Tasks: cmake
Filename: {tmp}\MSYS-1.0.10.exe; Tasks: msys
```

Dette skriptet fungerer rimelig rett frem. Først defineres tre *tasks*, å installere *Dev-Cpp*, *CMake* og *MSYS*. Disse er oppgaver brukeren kan aktivere eller deaktivere under installasjonen. Deretter velges hvilke filer som skal være med i installasjonspakka. Her velges BSPlab-filene pluss installasjonsprogrammene til *Dev-Cpp*, *CMake* og *MSYS*. Resten av skriptet definerer at BSPlab skal installeres og i tillegg at de to ekstrakomponentene skal installeres dersom disse ble valgt. Dette skriptet kjøres så gjennom *Inno Setup Compiler* og

det genereres en ferdig installasjonspakke som skal fungere på alle versjoner av Windows.

6.2 Automatisk installasjon på Linux

Det første vi gjorde når vi skulle lage et installasjonsprogram for Linux var å se om vi fant noen verktøy for å lage installasjonsprogrammer. Linux er en allsidig plattform, og det finnes knapt noen standard måte å installere programmer på alà Windows sine installerings wizards. Vi lette likevel etter et verktøy som kunne tilfredsstille våre krav. De verktøyene vi fant som hadde størst potensiale er beskrevet i neste avsnitt.

6.2.1 Oversikt over installasjonsverktøy

I tabell 6.1 har vi listet opp installasjonsverktøy for Linux og deres egenskaper. Egenskapene er nummerert som følger:

1. Ikke proprietært (gratis)
2. Kan gjøre systemsjekker
3. Støtter kompilering
4. Støtter alle Linux distribusjoner
5. Tekstbasert grensesnitt
6. Grafisk grensesnitt
7. Kan gjøre forskjellig installasjon avhengig av bruker

Egenskap Program	**1**	**2**	**3**	**4**	**5**	**6**	**7**
RPM	J	D	J	N	J	D	N
loki_setup	J	N	N	J	J	J	J
autopackage	J	J	J	N	J	J	N
BitRock InstallBuilder	N	J	J	J	J	J	J

Tabell 6.1: Oversikt over installasjonsverktøy

RPM

Redhat Package Manager [RPM, 2005] brukes av mange Linux-distribusjoner for å holde orden på installerte pakker på systemet. RPM støtter å sjekke om nødvendige pakker allerede er installert på systemet, men ettersom navnekonvensjonen på installerte pakker er

forskjellig fra distribusjon til distribusjon, er dessverre ikke systemsjekkene distribusjonsuavhengig. Det vil si at RPM-pakker laget på en Linux-distribusjon ikke vil installeres uten problemer på en annen Linux-distribusjon. I utgangspunktet er RPM-programmet tekstbasert, men det finnes tredjeparts programmer som gir RPM et grafisk grensesnitt. Ettersom å holde orden på installerte systempakker i utgangspunktet gjøres av *root* på Unix, har ikke RPM noen spesiell støtte for å gjøre forskjellige installasjoner avhengig av bruker.

autopackage

autopackage [autopackage, 2005] er et pakke-administrasjonsprogram som prøver å rette opp problemet RPM har med at pakkene er distribusjonsavhengig. Til tross for dette er dessverre autopackage installert på veldig få Linux-distribusjoner, noe som gjør denne løsningen langt fra universal. autopackage har heller ikke noen spesiell støtte for forskjellig installasjon avhengig av bruker.

loki_setup

loki_setup [lokisetup, 2005] er et installasjonsprogram som er brukt for å installere mange forskjellige spill som er gitt ut for Linux, eller blitt flyttet fra Windows til Linux. Installasjonsprogrammet oppfyller derfor alle krav om at det skal være distribusjonsuavhengig, men ettersom det i første rekke er ment for å installere ferdigkompilerte spill, støtter ikke programmet i utgangspunktet å kompilere opp det som skal installeres.

BitRock InstallBuilder

BitRock InstallBuilder [Bitrock Installbuilder, 2005] var det eneste installverktøyet vi fant for Linux som hadde flesteparten av egenskapene vi trengte. Dessverre er dette programmet proprietært, og kunne derfor ikke brukes til å installere BSPlab på Linux.

Valg av installasjonsverktøy

De verkøyene vi nå har beskrevet ville sikkert fungert for vanlige programmer, men ettersom vi skulle gjøre en del systemsjekker samt kompilere opp biblioteker (j.fr. 3.3.2), ser vi at ingen av disse verktøyene tilfredsstiller våre krav.

Da vi ikke fant noen verktøy for å lage installasjonsprogrammet, bestemte vi oss for å skrive hele programmet selv. Vi kunne selvsagt valgt å legge til den funksjonaliteten vi trengte til et av de eksisterende programmene, men fra vårt ståsted virket dette mindre effektivt. Shell script-språkene på Unix/Linux er godt utviklede programmeringsspråk. Slike Shell-script trenger ikke kompileres, og kjører uten problemer på de fleste Unix-systemer. Mange nytteprogrammer og installasjonprogrammer for Unix er skrevet i shell script-språk. Vi

valgte derfor å skrive vårt Linux-installasjonsprogram i script-språket til *Bourne Again Shell* (BASH) [BASH, 2005].

6.2.2 Installasjonsprogrammet *bsplab_install*

bsplab_install er et enkelt installasjonsprogram skrevet i BASH. Programmet har intet grafisk grensesnitt, men benytter seg i stedet av et tekstbasert grensesnitt; noe som fortsatt er veldig vanlig på Unix-plattformer. Programmet oppfyller alle krav spesifisert i kapittel 3.3.2.

Sjekke hvilken bruker som kjører installasjonen

Ettersom installasjonen skal gjøres forskjellig avhengig av om det er en vanlig bruker eller administrator (root) som kjører installasjonsprogrammet, er programmet nødt til å finne ut hvilken bruker som startet det. Dette gjøres ved å sjekke miljøvariabelen UID som vist i listing 6.2. Denne variabelen er alltid satt til den tallverdien brukeren representerer i operativsystemet. Brukeren *root* har alltid brukerid 0, så dersom UID er 0 er det administrator som har startet programmet og vi skal utføre en systeminstallasjon.

Listing 6.2: Shell-kode for å sjekke om brukeren er root

```
# if user is root, set global install to true
if [ "$UID" -eq "0" ] ; then
    GLOBAL_INSTALL=1;
fi
```

Sjekke om utviklingsverktøy er installert på systemet

Da det er nærmest umulig å sende med binærversjoner av BSPlab for etthvert Linux-system, er vi nødt til å kompilere opp programmet på den maskinen hvor BSPlab skal installeres. For at dette skal være mulig må systemet være utstyrt med de nødvendige utviklingsverktøy. Det er viktig at installasjonsprogrammet sjekker om systemet har de utviklingsverktøy som kreves, og opplyser brukeren om situasjonen hvis dette ikke er tilfellet.

Programmene *bsplab_install* trenger er *make* og kompilatoren *g++*. For å sjekke om disse er installert på systemet tar installasjonsprogrammet ganske enkelt bare og ser etter om disse programmene ligger i programstien til operativsystemet. Det er slik at hvis disse utviklingsverktøyene er installert på et Linux-system skal de også ligge i programstien til operativsystemet. Koden som gjør denne sjekken vises i listing 6.3. Vi ser også at dersom programmet ikke finner ett av utviklingsverktøyene vil brukeren få beskjed om hva som ikke ble funnet.

Listing 6.3: Shell-kode for å sjekke om utviklingsverktøyene er installert

```
detect_makeutils()
{
    echo -n "Detecting compiler: ";
    GCC=`which g++ 2> /dev/null`;
    if [ "$GCC" == "" ] ; then
        fatal_error "not found, aborting...";
    fi
    echo $GCC;

    echo -n "Detecting make utility: ";
    MAKE=`which make 2> /dev/null`;
    if [ "$MAKE" == "" ] ; then
        fatal_error "not found, aborting...";
    fi
    echo $MAKE;
}
```

Sjekke om automake-verktøyet som trengs er installert på systemet

For å få kompilert BSPlab trenger vi å få generert ett bygge-miljø ved hjelp av automake-verktøyet vårt. Vi valgte å bruke *CMake* som automake-verktøy som beskrevet i seksjon 5.2. Ettersom *CMake* ikke per dags dato er et standard utviklingsverktøy på Linux-systemer, vil *bsplab_install* selv installere *CMake* dersom det ikke finnes på systemet. Sjekken for *CMake* gjøres på samme måte som sjekken for utviklingsverktøy, bortsett fra at i dette tilfellet blir ikke installasjonen avbrutt dersom *CMake* ikke finnes på systemet.

Kompilering og installasjon

Etter at installasjonsprogrammet har gjort den nødvendige analysen av systemet BSPlab skal installeres på, blir brukeren spurt om hvor han vil at BSPlab skal installeres. Hvis brukeren ikke har noen innvendinger blir BSPlab installert under *bsp* på hjemmeområdet hans. Dersom det er snakk om en systeminstallasjon av administrator er det vanlig på Unix at filene blir installert på sine respektive områder under */usr/local*. Det vil si at binærfiler blir installert i */usr/local/bin*, header-filer i */usr/local/include* og bibliotek-filer i */usr/local/lib*. Selve prefiksen */usr/local* kan også overstyres av administrator ved installasjon.

Hele BSPlab-pakken (BSPlab, C++Sim og et eksempel BSP-program) ligger i en komprimert tar-fil, noe som er standard for komprimerte arkiver på Unix. CMake ligger i en egen tar-fil ettersom dette verktøyet ikke tilhører BSPlab-pakken. Dette gjør det også enkelt å bytte ut CMake-versjonen uten å forandre på BSPlab-pakken. Ved brukerinstallasjon blir BSPlab, og eventuelt CMake, pakket opp til området brukeren spesifiserte, og deretter kompilert opp på samme plass. Dersom man utfører en systeminstallasjon vil BSPlab bli pakket

opp til temporærområdet */tmp* og kompileringen utført her. Når BSPlab er ferdigkompilert vil de nødvendige filene (bibliotek- og header-filer) bli kopiert til sine respektive systemområder som beskrevet i forrige avsnitt. CMake vil også bli installert og gjort tilgjengelig for alle brukere dersom det ikke allerede er installert.

Kapittel 7

Testing av nye BSPlab

Under flyttingen av BSPlab har vi konsentrert oss om å kjøre gjennom testene som fulgte med den originale BSPlab. Dette har gjort det mulig for oss å til en viss grad kvalitetssikre arbeidet som er gjort underveis. Nå har vi kommet så langt at vi vil gå videre med testingen utover det som er beskrevet i seksjon 4.2. Vi vil konsentrere oss om å teste BSPlab under GCC på Linux og MinGW på Windows. Først vil vi kjøre noen tester for å finne ut hvor mange prosessorer BSPlab er i stand til å simulere. Testen blir i praksis en sjekk av hvor mange tråder og virtuelt minne operativsystemene er i stand til å gi BSPlab. Deretter vil vi teste BSPlab på et program med forskjellige antall prosessorer og sammenlikne tidsforbruket på Windows og Linux. Til slutt vil vi forsøke å benytte BSPlab på noen programmer av nyere dato vi henter ned fra internett for å se om BSPlab er i stand til å kjøre andre programmer enn de vi har testet så langt.

7.1 Test av maks antall prosessorer

Det er av interesse å finne ut hvor store parallelle systemer BSPlab er i stand til å simulere. Vi lagde et oppsett for å teste dette både på Windows og Linux. For å kun se på hvor mange prosessorer BSPlab klarer å simulere satte vi opp en enkel *Null*-maskin der *NumberOfProcessors*-parameteren bestemmer antall prosessorer. Som testprogram brukte vi en modifisert versjon av *Hello World*-programmet som fulgte med BSPlab. Programmet vi kjørte kan ses i listing 7.1. Dette programmet skriver ut id'en til prosessoren som kjører det og kaller *bsp_sync()* for å vente på de andre prosessorene.

Listing 7.1: Enkelt testprogram

```
void bsp_main(int argc, char **argv)
{
    bsp_begin(bsp_nprocs());
    printf("pid: %d\n", bsp_pid());
    bsp_sync();
    bsp_end();
}
```

7.1.1 Maks antall prosessorer på Windows

Når antall prosessorer oversteg 1014 fikk vi beskjed om at operativsystemet ikke ville gi oss flere tråder. Etter å ha gått igjennom C++Sim-koden i *nt_thread.cc* som inneholder implementasjonen av klassen `Thread` for Windows, fant vi ut at forfatterene av BSPlab hadde lagt til en sjekk for om operativsystemet nekter å lage en ny tråd, og i så fall skrives feilmeldingen vi fikk ut og simuleringen avsluttes. Denne koden vises i listing 7.2. Som vi ser av listingen er ikke feilmeldingen så mye å bli klok av ettersom vi vil få akkurat den samme feilmeldingen uansett hva som feiler i kallet til `CreateThread`. For å bedre kunne forstå hva som etterhvert går feil i kallet til `CreateThread`, valgte vi å legge til utskrift av den faktiske feilmeldingen fra operativsystemet som vist i listing 7.3.

Listing 7.2: Utdrag av nt_thread.cc

```
_data->thrHandle =
CreateThread(NULL,
             0,
             (LPTHREAD_START_ROUTINE) ThreadData::Execute,
             p1,
             0,
             &_data->mid);

//Added by Haakon Dybdahl to terminate the simulation
//if the o.s. does not give us another thread
if(_data->thrHandle == NULL) {
        error_stream
        << WARNING
        << "Trying to spawn another thread but didn't get
           any from the O.S., terminating.."
        << endl;
        Thread::Exit(0);
}
```

Listing 7.3: Forbedret feilutskrift i nt_thread.cc

```
if(_data->thrHandle == NULL) {
        // Get OS error message and print it to the user
        LPVOID lpMsgBuf;
        DWORD dw = GetLastError();

        FormatMessage(
                FORMAT_MESSAGE_ALLOCATE_BUFFER |
                FORMAT_MESSAGE_FROM_SYSTEM,
                NULL,
                dw,
                MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                (LPTSTR) &lpMsgBuf,
                0, NULL );

        error_stream
        << WARNING
        << "Trying to spawn another thread but didn't get
           any from the O.S., terminating.."
        << endl;

        error_stream
        << WARNING
        << "Error Message: "
        << (LPSTR)lpMsgBuf
        << endl;
        LocalFree(lpMsgBuf);
        Thread::Exit(0);
}
```

Feilmeldingen fra operativsystemet fortalte oss at Windows ikke klarer å lage flere tråder fordi den går tom for minne. Problemet ligger i at Windows i utgangspunktet allokerer 2MB stack til hver eneste tråd som blir laget. [CreateThread, 2005] Ettersom alle versjoner av Windows (inklusive Windows XP) har en virtuelt minne grense på 2GB, vil man fort få problemer med minne når man overgår 1000 tråder i BSPlab.

Funksjonen `CreateThread` [CreateThread, 2005] gir deg muligheten til å sette stack-størrelsen på den tråden du skal lage, men ikke helt uten problematikk. Stack-størrelsen til en tråd i Windows er delt i *reservert minne* og *initially committed memory*. [Win32 Thread Stack, 2005]. *Reservert minne* representerer den totale stack-størrelsen reservert i virtuelt minne, og er således begrenset av det virtuelle adresserommet. *Initially committed memory* brukes ikke før dette minnet faktisk blir referert, men legger likevel beslag på antall ledige sider for systemets totale *commit* grense. Denne grensen er satt av totalt tilgjengelig minne på systemet som er det fysiske minne pluss størrelsen på swap-filen. Parameter nummer to til `CreateThread` forteller hvor mye *initially committed memory*-stack tråden skal startes med, men dette forandrer ikke trådens *reserverte stack-minne*; det blir

fortsatt satt til standardverdien. Resultatet er at vi likevel går tom for virtuelt minne, selv om vi i realiteten ikke har brukt en brøkdel av det faktiske minnet.

Det finnes en løsning på problemet for nyere versjoner av Windows. For å sette størrelsen på trådens *reserverte stack*-minne må man sende inn flagget `STACK_SIZE_PARAM_IS_A_RESERVATION` som sjette parameter til `CreateThread`. Da vil parameter nummer to representere *reservert minne* istedet for *initially committed memory*. Det eneste aberet ved denne løsningen er at i følge Microsoft støttes ikke dette flagget på Windows 95/98/ME og Windows 2000/NT. Per dags dato vil derfor denne løsningen kun fungere på Windows 2003 og Windows XP.

Ved å spesifisere reservert stack-størrelse greide vi å kjøre over 16000 prosessorer på testprogrammet i listing 7.1 uten å gå tom for virtuelt minne. Stack-størrelsen var da satt til 100KB. I realiteten vil derfor antall prosessorer det er mulig å simulere avhenge av hvor stor stack programmet som skal simuleres trenger. Uten å ta hensyn til hvilke andre prosesser som kjøres på operativsystemet kan maksimum antall prosessorer beregnes ganske rundt som $n_{max} = \frac{2GB}{stack_size}$. Det hjelper selvsagt ikke bare at programmet ikke krever større stack enn f.eks. 100KB, forandring i stack-størrelsen må gjøres i C++Sim som så må rekompileres. Det hadde vært mer gunstig om brukeren selv kunne overstyre den vanlige stack-størrelsen uten å rekompilere C++Sim. Vi valgte derfor å se nærmere på å gjøre det mulig å overstyre programmets stack-størrelse i BSPlab sin parameterfil (j.fr. seksjon 7.1.3).

7.1.2 Maks antall prosessorer på Linux

Ved antall prosessorer høyere enn rundt 1040 hang BSPlab før simuleringen kom igang på Linux. Dette skyldtes at det ikke var noen kode for å håndtere feil ved generering av tråder i C++Sim på Linux. Vi bestemte oss for å legge til kode i C++Sim i fila *posix_thread.cc* for å støtte å håndtere feil ved generering av tråder. I listing 7.4 er et utdrag av koden som viser hvordan denne feilhåndteringen ble lagt til.

Listing 7.4: Utdrag av posix_thread.cc

```
int create_result = 0;
#ifndef PTHREAD_DRAFT_HPUX
  create_result = pthread_create(&_data->_thread,
                                 &_data->_attr,
                                 ThreadData::Execute, p1);
#else
  create_result = pthread_create(&_data->_thread,
                                 _data->_attr,
                                 ThreadData::Execute, p1);
  if(create_result == 0) {
    pthread_setprio(_data->_thread, MaxPriority);
  }
#endif
  if(create_result != 0) {
    error_stream
    << WARNING
    << "Error creating new thread; "
    << strerror(create_result)
    << " ("
    << create_result
    << ")"
    << endl;
    Thread::Exit(1);
  }
```

Denne feilhåndteringskoden sjekker status etter at en ny tråd ble forsøkt opprettet. Dersom dette feilet skriver den ut feilmeldingen som kommer fra operativsystemet, inkludert feilmeldingskoden, og avslutter hele programmet. Dette er en grei feilhåndtering siden BSPlab i dette tilfellet ikke ville simulert med det valgte antallet prosessorer og simuleringen dermed ville blitt feil dersom man fortsatte.

Når vi nå startet simuleringen på Linux med over 1040 tråder fikk vi feilmelding om at operativsystemet ikke er i stand til å allokere mer minne. Grunnen til dette er at hver tråd får tildelt en stack-størrelse på ca 2MB. Ettersom Linux normalt har 2GB virtuelt minne tilgjengelig for programmet, skyldes denne feilen at operativsystemet rett og slett går tom for minne fordi alt minnet blir allokert bort som stack til alle de nye trådene [Gorman, 2004]. Med *pthreads* er det mulig å spesifisere hva stack-størrelsen til den nye tråden skal være før du lager den. Det er sannsynlig å tro at de fleste BSP-programmer ikke har behov for så mye som 2MB stack-plass, men det er vanskelig å si nøyaktig hvor mye som trengs ettersom dette vil kunne variere fra program til program.

Vi testet å sette ned stack-størrelsen til 300kB og det ble nå mulig å simulere over 4000 prosessorer. Ettersom det ikke er kjent hvor stor stack-størrelse et BSP-program har bruk for på forhånd bør det være mulig for brukeren å sette dette i parameterfilen uten å måtte rekompilere C++Sim. Se seksjon 7.1.3 for hvordan vi gikk frem for å få til dette.

Ved verdier noe særlig høyere enn 4000 fikk vi feilmelding fra operativsystemet om at det ikke er i stand til å gi programmet flere tråder. Etter en del undersøkelser av hva dette kunne skyldes fant vi ut at programmet har en begrensning fra operativsystemet på hvor mange prosesser det får lov til å starte uten å ha spesielle rettigheter. Denne begrensningen lå på 4091 i utgangspunktet på vår distribusjon av Linux (SuSE 9.3). Dette fant vi ut ved å legge inn litt testkode i BSPlab som benyttet `getrlimit`-funksjonen til å hente ut denne verdien.

Fjerning av maks subprosess-begrensningen

Vi bestemte oss for å legge inn en sjekk som undersøker om programmet kjører som superbruker og i så fall fjerne begrensingen som hindrer programmet i å starte mer enn 4000 tråder. Koden som gjør dette kan ses i listing 7.5 og blir kalt i starten av `main`-metoden i fila *main.cpp*.

Listing 7.5: Fjerning av subprosessbegrensningen i platform.cpp

```
void BSP_Platform_Init()
{
  if(geteuid() == 0) {
    rlimit limit;
    memset(&limit, 0, sizeof(rlimit));
    limit.rlim_cur = RLIM_INFINITY;
    limit.rlim_max = RLIM_INFINITY;
    if(setrlimit(RLIMIT_NPROC, &limit) != 0) {
      cerr
      << "Error setting NPROC limit; "
      << strerror(errno)
      << endl;
      exit(1);
    }
  }
}
```

Etter at denne begrensningen var fjernet hadde vi mulighet til å simulere over 4000 prosessorer så lenge programmet ble startet av en superbruker. Vi fant ikke noen måte å få fjernet denne begrensingen dersom programmet ble kjørt som en vanlig bruker.

Det neste maksantallet tråder vi kom til lå på litt over 8000. Igjen fikk vi den samme meldingen om at operativsystemet ikke var i stand til å gi programmet flere tråder. Denne gangen skyldes det verdien for *threads-max* som er satt i operativsystemet. Verdien på denne kan leses ut ved å kjøre kommandoen "*cat /proc/sys/kernel/threads-max*" fra et *shell*. I vår Linux-distribusjon (SuSE 9.3) var *threads-max* satt til 8192. Variabelen kan settes direkte av en superbruker ved å utføre kommandoen "*echo nnnn > /proc/sys/kernel/threads-max*", der *nnnn* representerer det maksimale antallet tråder et program er tillatt å opprette. Vi

testet å sette denne til 16000 og testet en simulering med BSPlab med 15000 tråder og det fungerte som det skulle. Konklusjonen her er at en vanlig bruker er begrenset til å kunne simulere et maks antall prosessorer som er gitt av en begrensning i operativsystemet. Denne begrensningen varierer høyst sannsynlig mellom distribusjoner. En superbruker kan i utgangspunktet simulere så mange prosessorer han vil, så lenge programmet ikke går tom for minne.

7.1.3 Stack-størrelsen i parameterfilen

Stack-størrelsen til trådene som blir kjørt i BSPlab gjør det umulig å kjøre noe særlig mer enn 1000 tråder i simuleringen både på Windows og Linux. I hver tråd som lages av BSPlab blir BSP-programmet som skal simuleres kjørt. I de fleste tilfeller vil ikke dette BSP-programmet trenge en stack-størrelse på 2MB slik som blir allokert til det dersom noe annet ikke er spesifisert. Vi ønsket derfor å gjøre brukeren av BSPlab i stand til å overstyre programmets stack-størrelse uten å gjøre noen forandringer i kildekoden til BSPlab eller C++Sim (det er ikke forventet at en BSPlab-bruker i det hele tatt skal kunne kompilere opp BSPlab-koden).

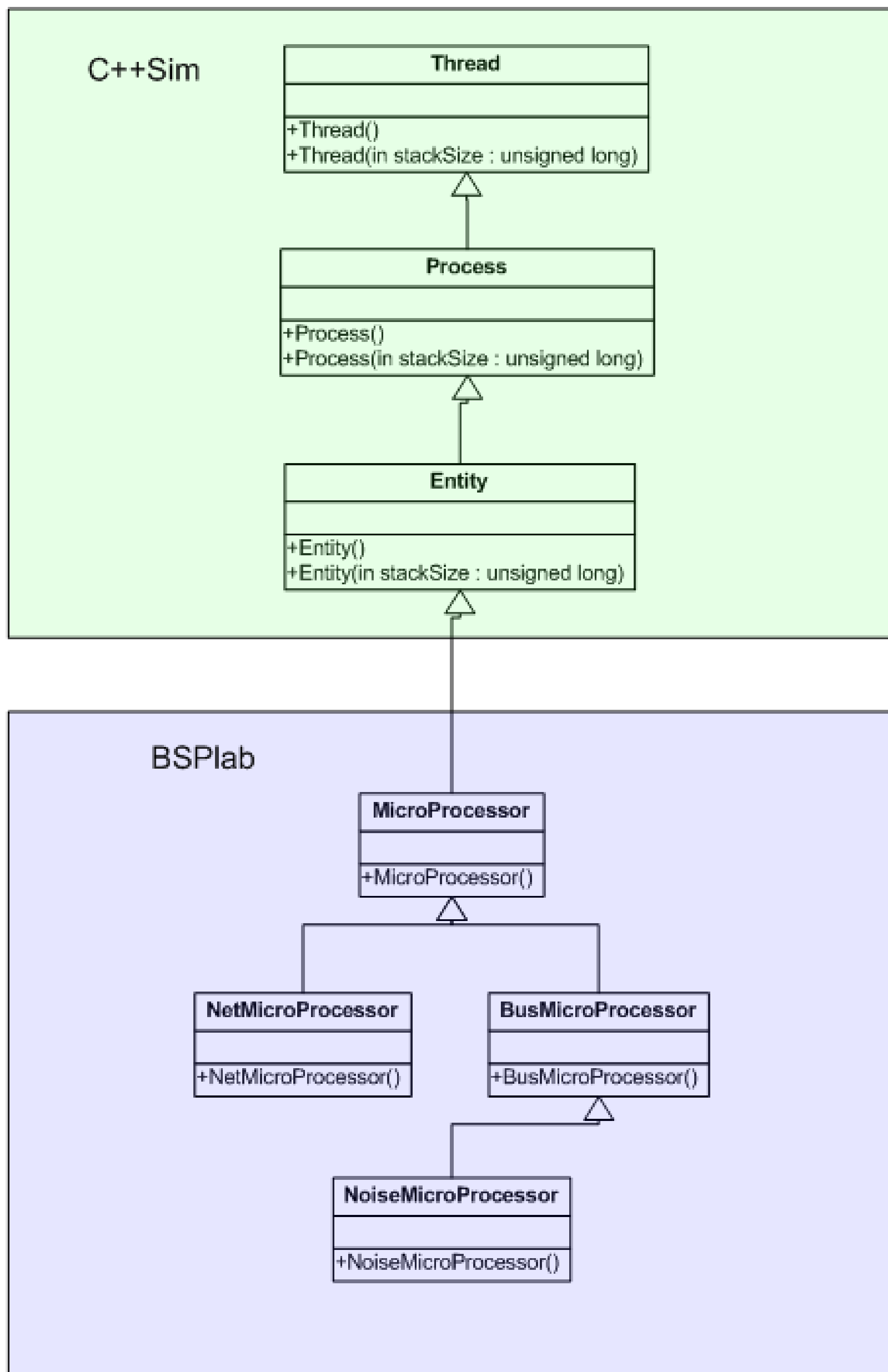
I følge dokumentasjonen til Microsoft [Win32 Thread Stack, 2005] kan trådenes *reservert minne* stack-størrelse settes i hodet til den kjørbare filen. Dette er ikke en god løsning da det også her kreves at koden kompileres på nytt med nye parametre til kompilatoren for at stacken blir satt til en ny størrelse. Et annet og enda større problem med denne løsningen er at stack-størrelsen vil bli satt for hele filen (også BSPlab) og ikke bare trådene. Det er sannsynlig å tro at i mange tilfeller krever BSPlab en større stack enn BSP-programmene som skal simuleres. Man ville derfor ikke få utnyttet stack-størrelsen på en slik måte at man får bruke et optimalt antall prosessorer.

En bedre og mer dynamisk løsning vil være at brukeren kan overstyre den “fabrikkinnstilte” størrelsen på stacken via en parameter i BSPlab sin parameterfil. Vi fryktet i utgangspunktet at en slik forandring ville kreve større omskrivninger av C++Sim, men det viste seg at C++Sim er designet på en slik måte at man skal kunne sette stack-størrelsen. Klassen `Thread` representerer en tråd i C++Sim. Denne klassen har to konstruktorer, en som lager en tråd med den “fabrikkinnstilte” stack-størrelsen, og en som tar inn som parameter hvor stor trådens stack skal være.

Forandringer på MicroProcessor

I BSPlab kjøres BSP-programmet som skal simuleres i en tråd. Denne tråden representerer en type mikroprosessor. For å skjønne hvordan det hele henger sammen med C++Sim vises et lite utdrag av avhengighetene mellom BSPlab og C++Sim i figur 7.1. Her ser vi også hvordan man har mulighet til å sende inn stack-størrelsen til en tråd til `Thread`-klassen, samt alle klasser som arver fra `Thread`. Figuren viser at det ikke trengs noen forandringer til C++Sim for å få lagt til et stack-parameter i parameterfilen. Prosessorklassene i BSPlab trenger derimot en ekstra konstruktor som tar inn stack-størrelsen som parameter. Listing

7.6 viser definisjonen av den ekstra konstruktoren for `MicroProcessor`. I linje 4 ser vi at parameteren for stack-størrelsen `stackSize` sendes videre til konstruktoren til `Entity`. `MicroProcessor` brukes for å simulere prosessoren i *Null*- og *Simple*-arkitekturen. For de andre arkitekturene brukes `NetMicroProcessor` og `BusMicroProcessor`. Vi ga alle disse en ekstra konstruktor som tar stack-størrelsen som parameter og sender denne videre til super-klassen.



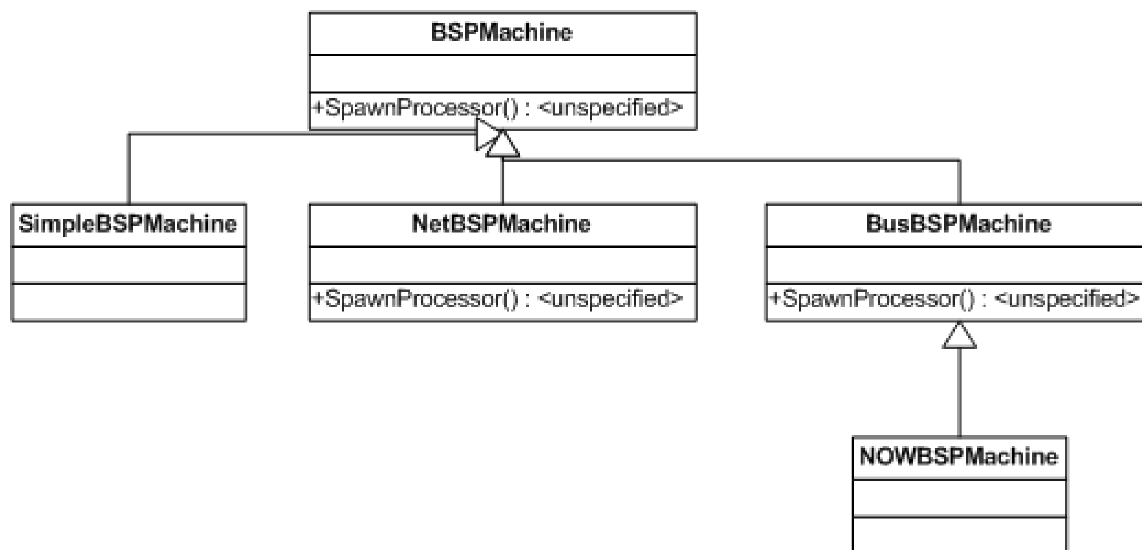
Figur 7.1: UML-utsnitt for C++Sim og BSPlab

Listing 7.6: MicroProcesssor-konstruktor med stack-størrelse-parameter

```
1  MicroProcessor::MicroProcessor(int prosNr,
2                                 bspProgramType fPtr,
3                                 unsigned long stackSize)
4  : Entity(stackSize)
5  {
6      m_iPID = prosNr;
7      m_pCodeToRun = fPtr;
8      m_pCodeToRunInit = NULL;
9      m_dblSuperstepProbeTime = 0;
10     m_bIsInSync = false;
11 }
```

Forandringer på BSPMachine

Etter at alle prosessorene hadde fått mulighet til å ta inn stack-størrelsen i konstruktoren, var vi nødt til å få BSPlab til å ta i bruk denne konstruktoren hvis nøkkelordet `StackSize` er spesifisert i parameterfilen. I BSPlab er det klassen `BSPMachine` eller klasser som arver fra denne som tar seg av å instansiere mikroprosessor-klassene. I figur 7.2 ser vi en forenklet UML av `BSPMachine` og alle klasser som arver fra denne. Det er metoden `SpawnProcessor` som tar seg av å instansiere en mikroprosessor. I `BSPMachine` sitt tilfelle dreier dette seg om klassen `MicroProcessor`. Vi ser at `SpawnProcessor` er definert som `virtual` slik at klasser som arver `BSPMachine` skal kunne ta over metoden og instansiere sin egen type prosessor. De klassene som gjør dette har metoden `SpawnProcessor` tegnet inn i UMLen i figur 7.2.



Figur 7.2: UML for BSPMachine

Vi la til en variabel `m_stackSize` til klassen `BSPMachine`, denne variabelen blir satt i konstruktoren til `BSPMachine` som vist i listing 7.7. Dersom *StackSize* ikke er satt i parameterfilen vil variabelen `m_stackSize` automatisk bli satt til 0 av `ParameterFile::ReadInt`. Grunnen til at vi velger å lese inn denne parameteren i konstruktoren til `BSPMachine` har med ytelse å gjøre. På denne måten vil parameteren kun bli lest inn en gang i løpet av hele programmets levetid. Hadde vi derimot lest inn denne variabelen i `SpawnProcessor`, ville `ParameterFile::ReadInt` blitt kalt hver gang BSPlab skal lage en ny prosessor. Ettersom `ParameterFile` åpner parameterfilen fra disk, og leter gjennom filen etter parameteret hver eneste gang den blir kalt, kunne dette hatt innvirkning på ytelsen ved bruk av mange prosessorer.

Listing 7.7: Utdrag av konstruktoren til BSPMachine

```
BSPMachine::BSPMachine()
{
    [...]
    // Thread stack size
    m_stackSize = ParameterFile::ReadInt("StackSize",
                                         0,
                                         false);
    [...]
}
```

`SpawnProcessor`-metodene i `BSPMachine` måtte også forandres til å kalle riktig `MicroProcessor`-konstruktor avhengig av om *StackSize* er satt eller ikke. I listing 7.8 ser vi den opprinnelige koden for `SpawnProcessor`-metoden. I den nye koden vist i listing 7.9 ser vi at den opprinnelige `MicroProcessor`-konstruktoren blir kalt dersom `m_stackSize` er 0 (dvs ikke satt), mens den nye konstruktoren som tar inn stack-størrelsen som siste parameter blir kalt dersom brukeren selv har satt en grense på stack-størrelsen. Den samme forandringen måtte også gjøres i alle klasser som arvet `BSPMachine` og gjenimplementerte `SpawnProcessor`-metodene. Som vi ser av figur 7.2 gjelder dette `NetBSPMachine` og `BusBSPMachine`.

Listing 7.8: Opprinnelige BSPMachine::SpawnProcessor

```
BSPMachine::SpawnProcessor(int nr, bspProgramType fPtr)
{
    MicroProcessor *FirstProc = new MicroProcessor(nr, fPtr);
    FirstProc->Activate();
    return FirstProc;
}
```

Listing 7.9: Nye BSPMachine::SpawnProcessor

```
BSPMachine::SpawnProcessor(int nr, bspProgramType fPtr)
{
    MicroProcessor *FirstProc =
          m_stackSize
          ? new MicroProcessor(nr, fPtr, m_stackSize)
          : new MicroProcessor(nr, fPtr);
    FirstProc->Activate();
    return FirstProc;
}
```

Andre forandringer

For arkitekturene Network, Bus og NOW måtte vi gjøre noen ekstra forandringer.

Arkitekturen Network lager like mange `MessageProcess`-objekter som prosessorer for hvert kall til `bsp_sync`. Ettersom `MessageProcess` også arver fra `Entity` og derfor lager en trå, ble vi nødt til å gi `MessageProcess` en ny konstruktor, samt å gjøre lignende forandringer som tidligere alle plasser hvor et nytt `MessageProcess`-objekt ble instansiert i klassen `NetBSPMachine`.

Arkitekturen Bus bruker klassen `DMAController`, som arver fra `BusController`, som igjen arver fra `Entity`. I tillegg bruker arkitekturen NOW klassen `NetworkAdapter` som også arver fra `BusController`. Vi måtte også her legge til nye konstruktorer og forandre litt på instansieringskoden.

Resultat

Resultatet av disse forandringene er at brukeren nå kan bruke parameteren *StackSize* i *Parameters.dat*-filen for å spesifisere nøyaktig hvor mange bytes han vil at operativsystemet skal sette av til stack for BSP-programmet. For nærmere forklaring på bruken av denne parameteren, se seksjon 10.1.4.

7.2 Sammenlikning av tidsforbruk på Windows og Linux

En viktig indikasjon på om flyttingen til Linux og MinGW har gått bra er at resultatene man får av simuleringer på de to platformene er sammenliknbare. For å teste dette vil vi benytte programmet *MergeSort* på 256k (262144) tall og kjøre det på forskjellig antall prosessorer. Dette vil vi gjøre på alle de fem arkitekturene med et fast oppsett på hver arkitektur på både Windows og Linux på nøyaktig samme maskinvare. Resultatene vi vil sammenlikne kommer fra den automatiske tidtakningen i BSPlab. Vi vil ta tiden på alle simuleringene på begge platformer og forhåpentligvis få resultater som ligger veldig nær hverandre. Den

automatiske tidtakningen til BSPlab måler hvor mange sekunder hver enkelt prosessor har brukt frem til et gitt tidspunkt. Vi vil måle tiden fra rett før simuleringen starter til rett etter den er ferdig. For å få en så pålitelig tidtakning som mulig trenger vi sanntidsprioritering av BSPlab på begge plattformene. Det står beskrevet i [Uthus and Dybdahl, 1997] hvordan BSPlab kan kjøres med sanntidsprioritering fra kommandolinjen i Windows, men ettersom vi også vil trenge dette på Linux, og vi ikke vet noen måte å få til dette på fra et shell, vil vi legge til en parameter for det i parameterfila. Etter at dette er gjort vil vi få mest mulig nøyaktige målinger av tidsforbruket på både Windows og Linux.

7.2.1 Sanntidsprioritering

I både Windows og Linux er det støtte for å gi et program ekstra prioritering slik at det hele tiden får førsteprioritet til å kjøre i operativsystemet. Den høyeste gruppen av slik prioritering er *sanntidsprioritering*. Det er ønskelig at BSPlab skal ha mulighet til å be om å få sanntidsprioritering i tilfeller der det er kritisk at måling av tidsforbruket er så nøyaktig som mulig. Ved normal prioritering vil operativsystemet foreta seg en rekke andre ting og la andre programmer få tilgang på maskinressurser mens BSPlab kjører. Dette fører til at tidtakningen i BSPlab blir unøyaktig ettersom ekstra tid blir brukt på å kjøre andre programmer. Ettersom vi her skal teste nettopp tidsforbruket til et BSP-program er det kritisk at dette blir så nøyaktig som mulig. Ved sanntidsprioritering får BSPlab maksimalt med ressurser fra operativsystemet og vi oppnår at tidtakningen blir så nøyaktig som mulig.

Vi har valgt å legge inn et eget parameter i parameterfilen som slår sanntidsprioritering av og på. Dette parameteret heter *RealTime* og blir sjekket i starten av `main`-funksjonen. Det kalles en platformavhengig funksjon som er implementert i *platform.cpp* dersom *RealTime* er satt. Hvordan dette ser ut kan ses i listing 7.10.

Listing 7.10: Aktivering av sanntidsprioritering i main.cpp

```
if(ParameterFile::ReadBoolean("RealTime", false)) {
  BSP_Set_RealTime_Priority();
}
```

Det sjekkes om *RealTime*-parameteren er satt i parameterfilen og sanntidsprioritering aktiveres dersom den er det.

Sanntidsprioritering på Linux

Under Linux kreves det at man er superbruker for å få lov til å sette prioriteringen til et program til sanntid. Dette skyldes at det er mulig å låse hele operativsystemet dersom et sanntidsprogram havner i en uendelig løkke og ingen andre programmer har prioritet til å kunne avbryte det. Det vil si at sanntidsprioritering av BSPlab under Linux kun blir

tilgjengelig dersom man er superbruker. Hvis man forsøker å sette på sanntidsprioritering uten å være superbruker vil det bli gitt en feilmelding som indikerer at dette ikke er mulig.

Det holder å sette sanntidsprioritering på hovedprosessen ettersom alle tråder som blir opprettet av denne automatisk arver prioriteringsgruppen [Unixhelp, 2005]. Listing 7.11 viser hvordan denne koden ble i *platform.cpp*.

Listing 7.11: Støtte for sanntidsprioritering på Linux i platform.cpp

```
#include <sched.h>
void BSP_Set_RealTime_Priority() {
  sched_param p;
  memset(&p, 0, sizeof(sched_param));
  p.sched_priority = 10;

  if(sched_setscheduler(0, SCHED_FIFO, &p) != 0) {
    cerr << "Error setting real-time scheduling; "
    << strerror(errno)
    << endl;
    exit(1);
  }
}
```

Her settes ganske enkelt prioriteringsgruppen til sanntid (*SCHED_FIFO*) med et kall til `sched_setscheduler`-funksjonen. Dersom dette kallet feiler betyr det antakeligvis at brukeren ikke er superbruker. En feilmelding skrives ut og simuleringen avsluttes.

Sanntidsprioritering på Windows

På Windows er det i likhet med Linux nok å sette sanntidsprioritering på hovedprosessen. Alle nye tråder arver denne prioriteringsklassen når de opprettes. Listing 7.12 viser hvordan koden i *platform.cpp* ble på Windows.

Listing 7.12: Støtte for sanntidsprioritering på Windows i platform.cpp

```
void BSP_Set_RealTime_Priority()
{
  if(SetPriorityClass(_data->thrHandle,
                      REALTIME_PRIORITY_CLASS) == 0)
  { //error occured
    LPVOID lpMsgBuf;
    DWORD dw = GetLastError();

    FormatMessage(
      FORMAT_MESSAGE_ALLOCATE_BUFFER |
      FORMAT_MESSAGE_FROM_SYSTEM,
      NULL,
      dw,
      MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
      (LPTSTR) &lpMsgBuf,
      0, NULL );

    cerr << "Unable to set real-time priority; "
    << (LPSTR)lpMsgBuf
    << endl;
    LocalFree(lpMsgBuf);
    exit(1);
  }
}
```

Denne er i funksjonalitet helt lik det som ble gjort på Linux. Eneste forskjellen er navnet på funksjonen som blir kalt og hvordan feilmeldingen hentes ut dersom noe går galt.

7.2.2 Testing av tidsforbruket

Vi benyttet en dedikert maskin med både Windows XP og Gentoo Linux installert [Gentoo, 2005]. Med dedikert mener vi at maskinen kun ble benyttet til å kjøre simuleringer, den kjørte minst mulig andre prosesser i bakgrunnen samtidig. Maskinen hadde 1GB RAM, en 2.8GHz Pentium 4 prosessor og 2GB swap-størrelse på både Windows og Linux. Som test satte vi opp å sortere 256k tall med *MergeSort* på 2, 4, 8, 16, 32, 64, 128 og 256 prosessorer på Windows og Linux. Disse skulle kjøres gjennom på hver av de fem arkitekturene, Null, NOW, Network, Bus og Simple. Hver simulering ble kjørt tre ganger og snittverdien vises sammen med resultatene. Valget av antall tall å sortere og antallet prosessorer å kjøre på ble vurdert ut ifra at problemet burde være av en viss størrelse samtidig som det helst ikke bør ta mer enn noen timer å kjøre den største av simuleringene. Grunnen til dette er at hver simulering skulle kjøres på fem arkitekturer på to platformer, slik at det totale tidsforbruket ville bli svært stort dersom en enkelt gjennomkjøring hadde tatt for lang tid. For hver av de fem arkitekturene valgte vi et oppsett der flest mulig av de

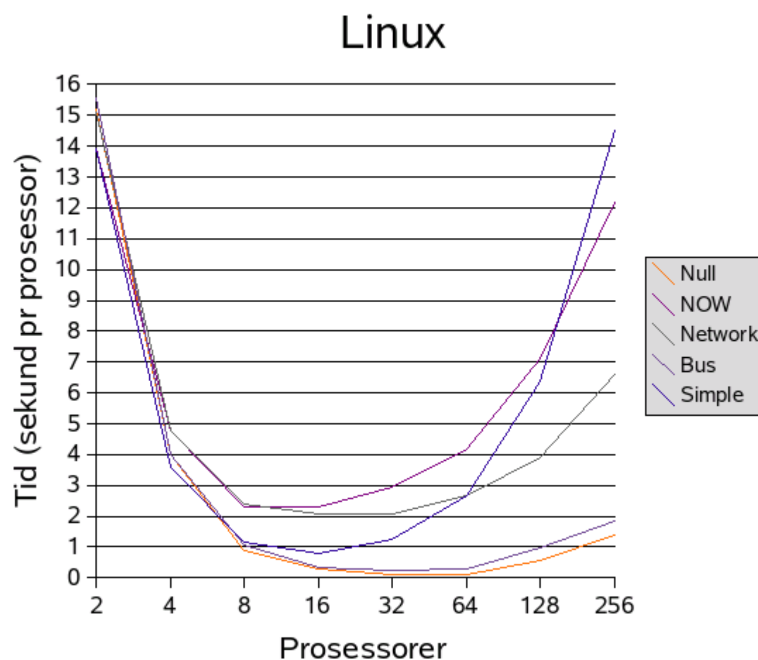
tilgjengelige parameterene for hver arkitektur ble benyttet. Parameterfilen kan ses i seksjon C.2. *MergeSort*-programmet som ble benyttet til alle simuleringene kan ses i seksjon C.1.

Gjennomkjøring av testen på Linux

Tabell 7.1 viser resultatene for simuleringene på Linux. Resultatene vi fikk hadde flere desimaler enn det som vises i tabellen, men vi valgte å bruke to desimaler for å gjøre tabellen mer oversiktlig. Vi støtte på få problemer i forbindelse med kjøringen av programmet. Det eneste å bemerke er at NOW-arkitekturen stoppen én gang med en *assertion* under kjøring med 64 prosessorer på Linux. Da vi forsøkte å kjøre simuleringen en gang til, fikk vi ikke denne feilen. Det er usikkert hva som forårsaket dette og vi vil ikke prioritere å undersøke det nærmere dersom problemet ikke oppstår igjen.

Prosessorer Arkitektur	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
Null	15.81	3.61	0.87	0.27	0.12	0.10	0.55	1.38
	15.66	4.19	0.92	0.29	0.13	0.09	0.55	1.40
	14.09	4.19	0.91	0.29	0.13	0.09	0.55	1.39
Snitt	15.18	4.00	0.90	0.28	0.13	0.09	0.55	1.39
NOW	14.16	4.64	2.34	2.27	2.88	4.24	7.03	12.16
	13.06	4.39	2.34	2.32	2.94	4.25	7.09	12.10
	14.54	5.28	2.26	2.30	2.98	4.10	7.23	12.27
Snitt	13.92	4.77	2.31	2.30	2.93	4.20	7.12	12.17
Network	16.15	4.70	2.23	2.08	2.06	2.63	3.87	6.14
	14.46	4.65	2.46	2.05	2.10	2.64	3.90	6.55
	14.72	4.92	2.44	2.07	2.03	2.68	3.92	7.05
Snitt	15.11	4.76	2.38	2.07	2.07	2.65	3.90	6.58
Bus	15.02	4.00	1.06	0.32	0.23	0.30	0.96	1.83
	16.59	3.84	1.12	0.35	0.24	0.31	0.98	1.83
	15.05	4.13	1.05	0.34	0.24	0.31	0.96	1.83
Snitt	15.55	3.99	1.08	0.34	0.24	0.31	0.97	1.83
Simple	13.38	3.50	1.15	0.77	1.25	2.67	6.40	14.49
	14.05	3.70	1.12	0.77	1.26	2.67	6.4	14.50
	14.32	3.53	1.22	0.81	1.25	2.67	6.4	14.49
Snitt	13.92	3.58	1.16	0.78	1.25	2.67	6.40	14.49

Tabell 7.1: Testresultater for Linux



Figur 7.3: Testresultater for Linux

Grafen i figur 7.3 viser en grafisk representasjon av resultatene. Vi ser at grafen stiger veldig bratt fra 64 prosessorer og utover. Vi antar at dette skyldes at operativsystemet er nødt til å gjøre en masse *swapping* for å møte minnebehovet til programmet på så mange prosessorer. Dette fører til at simuleringen går betydelig tregere enn dersom alt kan ligge i primærminnet under kjøring. Konsekvensen av dette er at dersom det skal kjøres store simuleringer på mange prosessorer med *MergeSort* er man avhengig av å ha mye primærminne for å få korrekt tidtakning. Dersom det kjøres et BSP-program som ikke bruker like mye minne per prosessor som det *MergeSort* gjør, kan man kjøre simuleringer med et høyere antall prosessorer uten å få problemer med *swapping*. Hva sammenlikningen med Windows angår antar vi at det fortsatt vil være mulig å sammenlikne disse tallene til tross for *swappingen* ettersom Windows også vil bli nødt til å laste deler av programmet ut av primærminnet under kjøring. En annen ting som også påvirker kjøretiden på mange prosessorer er at antallet tall å sortere blir lite i forhold til antallet prosessorer som benyttes. Dette fører til at mye tid går bort i *overhead* til parallelliseringen i forhold til hvor mange tall som sorteres på hver prosessor.

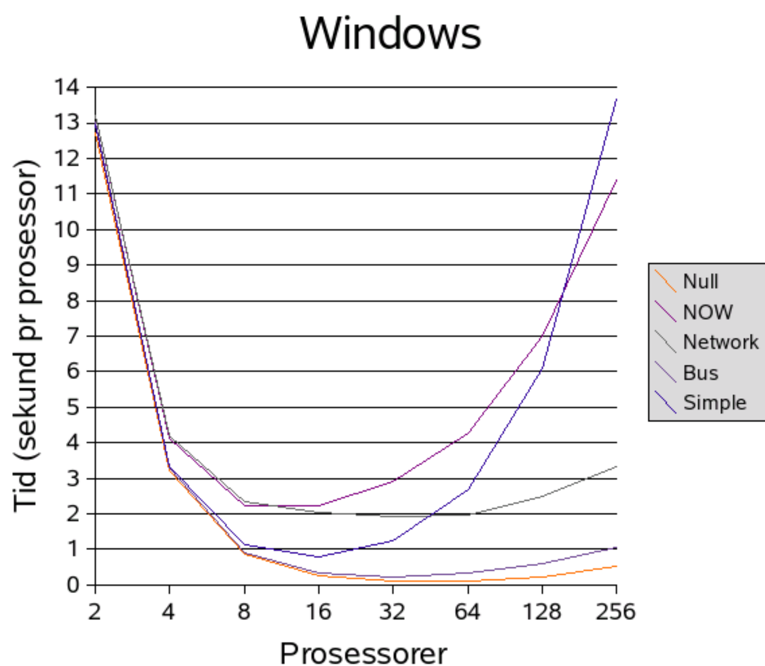
Gjennomkjøring av testen på Windows

Tabell 7.2 viser resultatene for simuleringene på Windows. Figur 7.4 viser en grafisk oversikt over tidene. Vi ser at grafene har samme form som på Linux. Også her antar vi at den raske

stigningen vi får på mange prosessorer i hovedsak skyldes at operativsystemet må ta i bruk sekundærminnet for å klare å gi BSPlab nok tilgjengelig minne. Dette fører til en mye tregere simulering enn dersom hele programmet hadde fått plass i primærminnet.

Prosessorer Arkitektur	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
Null	12.82	3.18	0.84	0.26	0.12	0.08	0.25	0.49
	12.74	3.21	0.87	0.26	0.12	0.10	0.23	0.55
	12.62	3.23	0.87	0.27	0.12	0.10	0.24	0.50
Snitt	12.73	3.21	0.86	0.26	0.12	0.09	0.24	0.51
NOW	13.22	4.13	2.23	2.26	2.90	4.29	6.97	11.27
	13.33	4.07	2.25	2.25	2.98	4.26	7.16	11.34
	13.06	4.15	2.25	2.20	2.88	4.26	6.87	11.55
Snitt	13.20	4.12	2.24	2.24	2.92	4.27	7.00	11.39
Network	13.25	4.24	2.33	2.04	1.97	1.97	2.48	3.31
	13.08	4.18	2.33	2.06	1.93	1.93	2.49	3.34
	13.27	4.23	2.32	1.98	1.92	1.99	2.46	3.32
Snitt	13.20	4.22	2.33	2.03	1.94	1.96	2.48	3.32
Bus	12.83	3.27	0.91	0.33	0.23	0.32	0.60	1.05
	12.71	3.32	0.90	0.32	0.24	0.31	0.61	1.08
	13.48	3.27	0.89	0.33	0.22	0.32	0.57	1.05
Snitt	13.00	3.29	0.90	0.33	0.23	0.32	0.59	1.06
Simple	13.09	3.33	1.09	0.78	1.25	2.68	6.09	13.66
	12.95	3.32	1.08	0.77	1.26	2.68	6.09	13.65
	12.76	3.29	1.18	0.78	1.25	2.70	6.10	13.66
Snitt	12.93	3.31	1.12	0.78	1.26	2.69	6.09	13.66

Tabell 7.2: Testresultater for Windows



Figur 7.4: Testresultater for Windows

Sammenlikning av resultatene

Etter å ha kjørt gjennom alle simuleringene både på Windows og Linux sitter vi med et tallmateriale vi kan benytte til å undersøke om BSPlab gir sammenliknbare resultater på begge platformene. Vi vil se på hvor stort avvik det er på hver av arkitekturene på simuleringer på samme antall prosessorer. Grunnen til at vi velger nettopp denne måten å se på resultatene er at vi er interessert i å finne ut om det er noen arkitekturer der målingene avviker veldig mye, noe som kunne tydet på en feil i BSPlab.

Prosessorer Arkitektur	2	4	8	16	32	64	128	256	Snitt
Null Linux	15.18	4.00	0.90	0.28	0.13	0.09	0.55	1.39	2.82
Null Windows	12.73	3.21	0.86	0.12	0.26	0.24	0.09	0.51	2.25
Differanse	2.46	0.79	0.04	0.02	0.01	0.00	0.32	0.88	0.56
Relativt avvik	16.18%	19.74%	4.37%	7.18%	4.07%	-3.06%	56.84%	63.21%	19.99%
NOW Linux	13.92	4.77	2.31	2.30	2.93	4.20	7.12	12.17	6.22
NOW Windows	13.20	4.12	2.24	2.24	2.92	4.27	7.00	11.39	5.92
Differanse	0.71	0.65	0.07	0.06	0.01	-0.07	0.12	0.79	0.29
Relativt avvik	5.13%	13.66%	3.04%	2.61%	0.46%	-1.70%	1.66%	6.46%	4.71%
Network Linux	15.11	4.76	2.38	2.07	2.07	2.65	3.90	6.58	4.94
Network Windows	13.20	4.22	2.33	2.03	1.94	1.96	2.48	3.32	3.93
Differanse	1.91	0.54	0.05	0.04	0.13	0.69	1.42	3.26	1.00
Relativt avvik	12.62%	11.35%	2.15%	1.90%	6.18%	25.96%	36.43%	49.48%	20.32%
Bus Linux	15.55	3.99	1.08	0.34	0.24	0.31	0.97	1.83	3.04
Bus Windows	13.00	3.29	0.90	0.33	0.23	0.32	0.59	1.06	2.46
Differanse	2.55	0.70	0.17	0.01	0.01	-0.01	0.37	0.77	0.57
Relativt avvik	16.38%	17.62%	16.17%	2.11%	2.69%	-3.05%	38.68%	42.22%	18.83%
Simple Linux	13.92	3.58	1.16	0.78	1.25	2.67	6.40	14.49	5.53
Simple Windows	12.93	3.31	1.12	0.78	1.26	2.69	6.09	13.66	5.23
Differanse	0.98	0.26	0.05	0.01	0.00	-0.02	0.30	0.84	0.30
Relativt avvik	7.07%	7.39%	4.15%	0.99%	-0.10%	-0.61%	4.72%	5.77%	5.48%
Differanse Snitt	1.72	0.59	0.08	0.03	0.03	0.12	0.51	1.31	
Relativt avvik	11.47%	13.95%	5.98%	2.96%	2.66%	3.51%	27.66%	33.43%	

Tabell 7.3: Differansen mellom kjøretiden på Linux og Windows

Tabell 7.3 viser differansen mellom hver av arkitekturene på forskjellige antall prosessorer. Vi ser at variasjonen er størst på få prosessorer og på mer enn 64 prosessorer. Grunnen til at det er stor variasjon på høyt antall prosessorer skyldes antakeligvis litt forskjell på hvor mye tid det koster å benytte sekundærminnet når primærminnet går fullt på de to plattformene. På *Network*-arkitekturen er denne differansen ekstra stor. Grunnen til dette er antakeligvis at det er en viss forskjell i kostnaden på å opprette og avslutte tråder. På *Network*-arkitekturen foregår det veldig mye oppretting og avslutting av tråder i forhold til de andre arkitekturene.

Generelt ser vi at kjøretiden på Windows er noe lavere enn på Linux, men på 8, 16, 32 og 64 prosessorer ligger den svært nærme på alle arkitekturer bortsett fra *Network*. Det er sannsynlig å tro at den på *Null*-, *NOW*- og *Bus*-arkitekturene vil ligge nærmere på at høyere antall prosessorer dersom maskina som kjører simuleringen har mer primærminne. *Network*-arkitekturen vil nok ikke skalere like bra på grunn av den ekstra kostnaden med å opprette tråder på Linux i forhold til Windows. Konklusjonen blir at så lenge ikke problemet er så stort at operativsystemet må ta i bruk sekundærminnet, får man sammenliknbare resultater mellom plattformene. Dersom sekundærminnet må benyttes får man uansett ikke gode resultater med automatisk tidtakning på BSPlab ettersom simuleringen går tregere enn den egentlig skal. Den arkitekturen som viser seg å i følge dette eksperimentet gi dårligst grunnlag for sammenlikning mellom plattformene er *Network*-arkitekturen, men så lenge ikke antallet prosessorer er for høyt gir også denne nesten like resultater på Windows og Linux.

7.2.3 Andre testprogrammer

Vi søkte på internett for å forsøke å finne andre programmer å teste BSPlab på. Blant annet Bisseling [Bisseling, 2004] har laget en del testprogrammer for BSPlib som skal kunne kjøres på BSPlab. Vi testet noen av programmene hans og andre vi fant på nettet. Disse programmene ble også testet på den originale BSPlab-versjonen under *Microsoft Visual Studio .Net* for å forsikre oss om at resultatene ikke ble påvirket av endringer vi har gjort i forbindelse med flyttingen.

Fast Fourier Transformasjoner

Dette programmet implementerer *Fast Fourier Transformasjoner* med BSPlib. Vi fikk kompilert programmet opp og fikk startet det. Programmet krasjet da vi forsøkte å bruke det på noen tilfeldig valgte problemstørrelser. Vi forsøkte å finne ut hva som gikk galt, men fant ikke ut hvor feilen var. Testingen vi gjorde med programmet fikk oss til å tro at det dreier seg om en minnefeil et eller annet sted, men om denne ligger i testprogrammet eller BSPlab selv, er usikkert.

Matrisemultiplikasjon

MatMult er et BSP-program for å multiplisere sammen to like store kvadratiske matriser. Det fulgte også med et lite program for å generere slike matriser med en gitt størrelse og tilfeldige data. Vi testet å multiplisere to 81x81 matriser med hverandre på 9 prosessorer på alle arkitekturene BSPlab støtter. Vi fikk feilmeldinger fra BSPlab om at den fikk problemer med å sende meldinger mellom prosessorer og måtte avbryte. Om dette skyldes feil i *MatMult* eller BSPlab vites ikke.

Programmer ikke støttet av BSPlab

Vi fant et par programmer som benyttet funksjoner som ikke er implementert i BSPlab fordi de er lagt til BSPlib-standaren etter at BSPlab ble laget. Disse var *bspmv* fra boka til Bisseling [Bisseling, 2004] som benytter funksjonen `bsp_qsize` og *bspprobe* som benytter funksjonen `bsp_fold`.

Kapittel 8

Java-bindinger

I dette kapittelet vil vi først beskrive hvordan JNI-teknologien fungerer. Deretter vil vi forklare hvordan vi gikk frem for å konstruere Java-bindinger for BSPlab. Tilslutt vil vi oppsummere hvordan den endelige løsningen artet seg.

8.1 Java Native Interface

JNI [JNI, 2005, Liang, 1999] er et omfattende grensesnitt for programmeringsspråk-bindinger. JNI gir programmereren mulighet til både å kalle C/C++-funksjoner fra Java, og til å kalle Java-metoder fra C/C++-programmer. Grensesnittet inkluderer også funksjoner for enkelt å konvertere mellom Java og C datatyper. I de påfølgende avsnittene vil det bli gitt forklaring og eksempel-kode for hvordan JNI fungerer. Vær oppmerksom på at det er utelatt en del feilhåndtering i eksempel-koden for å gjøre den lettere å forstå.

8.1.1 Kall til C++-funksjoner fra Java

For å kalle C++-funksjoner fra Java går man frem på følgende møte:

1. Deklarer de Java-metodene som skal bruke C++-funksjoner som `native`.
2. Fortell Java hva C++-biblioteket som skal lastes heter.
3. Kompiler opp Java-programmet.
4. Kjør programmet *javah* på de klassefilene som har `native` funksjoner.
5. Implementer de nye C++ wrapper-funksjonene hvis signatur har blitt generert av *javah*.
6. Kompiler inn wrapper-funksjonene i C++-biblioteket som skal lastes av Java.

Listing 8.1: Deklarasjon av Java native metoder

```
public class JNIExample {
    public native void helloWorld();
    public static native String[] quickSort(String[] words);

    public static void main(String[] args) {
        new JNIExample().helloWorld();

        String[] words = {"Ole", "Dole", "Doffen"};
        String[] sortedWords = JNIExample.quickSort(words);
        for(int c=0;c<sortedWords.length;++c)
            System.out.println(sortedWords[c]);
    }

    static {
        System.loadLibrary("jniexample");
    }
}
```

I listing 8.1 ser vi et eksempel på en Java-klasse som inneholder `native` metoder. Som eksemplet viser er `native` metodene deklarert, men selvfølgelig ikke definert da den utførende koden skal ligge i et C++-bibliotek. Eksemplet viser også at `native` metodene kan være både `static` og ikke `static`. Det siste `static` utsagnet i eksemplet forteller Java hvilket bibliotek det skal forsøke å laste med en gang Java klassen blir tatt i bruk.

Når så dette programmet er kompilert, må man kjøre programmet *javah* på den resulterende klasse-filen. Vi får da generert C/C++-headerfiler med deklarasjoner for `native`-metodene. For vårt eksempel blir deklarasjonene som vist i listing 8.2.

Listing 8.2: C/C++ deklarasjon av eksempel Java native metoder

```
JNIEXPORT void JNICALL Java_JNIExample_helloWorld
(JNIEnv *, jobject);

JNIEXPORT jobjectArray JNICALL Java_JNIExample_quickSort
(JNIEnv *, jclass, jobjectArray);
```

Dette er de såkalte wrapper-funksjonene for Java `native`-metodene. Java kan ikke kalle de faktiske C++-funksjonene direkte ettersom man i de fleste tilfeller må konvertere input-parameterene før C++-funksjonen blir kalt, og output-parameterene etterpå. Denne konverteringen gjøres i wrapper-funksjonene. For ikke å forurense navnerommet har disse funksjonene en spesiell navnekonvensjon. Først kommer det et prefiks, som alltid er “Java”. Deretter følger metodens fulle kvalifiserte klassenavn, og til slutt metodenavnet.

Vi ser at `helloWorld` tar inn to parameter selv om Java-deklarasjonen ikke tar inn noen. Disse to parameterne må alle wrapper-funksjoner ta inn. Det første parameteret `JNIEnv` er en peker til JNI-miljøet. Det er dette objektet som må brukes for å aksessere Java typer og konvertere disse. Det andre parameteret er en peker til objektet selv, man kan si at dette er Javas `this` variabel. Grunnen til at den er definert som `jobject` i `helloWorld` og `jclass` i `quickSort` er ganske enkelt at `quickSort` er en statisk metode. Så i `helloWorld`-tilfellet refererer parametret til instansen av klassen, mens i `quickSort`-tilfellet refereres det til selve klassen.

Listing 8.3: Definisjon av helloWorld wrapper-funksjon

```
void helloWorld()
{
    std::cout << "Hello World" << std::endl;
}

JNIEXPORT void JNICALL Java_JNIExample_helloWorld
(JNIEnv *, jobject)
{
    helloWorld();
}
```

I listing 8.3 ser vi definisjonen av `helloWorld`-wrapper-funksjonen. Ettersom den faktiske `helloWorld`-funksjonen hverken tar inn eller returnerer noen variabler trenger vi bare å kalle den faktiske funksjonen.

I motsetning til `helloWorld` er `quickSort` en del mer komplisert. Metoden tar inn en array med Java-strenger, og returnerer en slik sortert array. For at vi skal kunne kjøre C++ sin sorteringsfunksjon på dataene blir vi nødt til å gjøre om arrayen med Java-strenger til en C++-vektor med strenger. Likeledes blir vi nødt til å konvertere den sorterte C++-vektoren tilbake til en Java-array med strenger før vi gir den sorterte dataen tilbake til Java-programmet. Listing 8.4 viser oss definisjonen av `quickSort`-wrapper-funksjonen. Her ser vi at den første parameteren *env* av typen `JNIEnv` brukes til å kalle metoder for å få konvertert den tredje parameteren (den første parameteren fra Java-deklarasjonen av metoden) til en datatype vi kan bruke mot C++ sin innebygde quicksort-funksjon, i dette tilfellet en vektor med strenger.

Listing 8.4: Definisjon av quickSort wrapper-funksjon

```
JNIEXPORT jobjectArray JNICALL Java_JNIExample_quickSort
(JNIEnv *env, jclass, jobjectArray jwords)
{
    std::vector<std::string> cwords;

    // Convert Java String array to C++ vector
    jsize num = env->GetArrayLength(jwords);
    for(int c=0;c<num;++c) {
```

```
        jobject wordobj = env->GetObjectArrayElement(
                                    jwords,
                                    c);
        const char *word = env->GetStringUTFChars(
                            static_cast<jstring>(wordobj),
                            0);
        cwords.push_back(word);
        env->ReleaseStringUTFChars(
                    static_cast<jstring>(wordobj),
                    word);
    }

    std::sort(cwords.begin(), cwords.end());

    // Convert sorted C++ vector to Java String array
    jobjectArray ret = env->NewObjectArray(
                        num,
                        env->FindClass("java/lang/String"),
                        env->NewStringUTF(""));
    for(int c=0;c<num;++c) {
        env->SetObjectArrayElement(
                ret,
                c,
                env->NewStringUTF(cwords[c].c_str()));
    }

    return ret;
}
```

8.1.2 Kall til Java-metoder fra C++

Java programmer blir som kjent kompilert til såkalt *bytecode* istedet for maskinkode. Denne *bytecoden* kan bare kjøres i en *Java Virtual Machine* (JVM). For å kunne kalle Java-metoder fra et C++-program bruker man ganske enkelt JNI APIen til å starte en JVM. Den instansierte JVMen gir så mulighet til å laste inn ønsket Java-metode, for så å kalle denne. Konvertering mellom datatyper gjøres med samme grensesnitt som når man kaller C++-funksjoner fra Java.

Listing 8.5 viser et enkelt Java-program som skriver ut alle input parametre.

Listing 8.5: Enkelt Java-program

```
public class JavaProg {
    public static void main(String[] args) {
        for(int c=0;c<args.length;++c)
            System.out.println(args[c]);
    }
}
```

For å få kjørt dette programmet fra et C++-program må man gjøre følgende:

1. Starte opp en JVM via JNI API-kall.
2. Finne riktig klasse, i dette tilfellet *JavaProg*.
3. Finne riktig metode i klassen ved hjelp av både navn og datatype-signatur (type inn og ut parametere).
4. Konvertere data som skal inn til metoden fra C++ datatype til Java datatype.

Listing 8.6: C++-program som kaller Java-metoder

```
int main(int argc, char *argv[])
{
    [...]

    JNIEnv *env = NULL;
    JavaVM *jvm = NULL;
    // Create the Java Virtual Machine.
    JNI_CreateJavaVM(
            &jvm,
            reinterpret_cast<void**>(&env), &vm_args);

    jclass cls = env->FindClass("JavaProg");

    jmethodID mid = env->GetStaticMethodID(
                        cls,
                        "main",
                        "([Ljava/lang/String;)V");

    jstring jstr = env->NewStringUTF(
                        "Argument from C++");

    jobjectArray args = env->NewObjectArray(
                    1,
                    env->FindClass("java/lang/String"),
                    jstr);
```

```
    env->CallStaticVoidMethod(cls, mid, args);
    jthrowable exc = env->ExceptionOccurred();
    if( exc ) {
        env->ExceptionDescribe();
        exit(1);
    }

    return 0;
}
```

I listing 8.6 ser vi et enkelt C++-program som kaller Java-metoden i listing 8.5. Det er viktig å merke seg at i `GetStaticMethodID`-kallet sendes datatype-signaturen til metoden inn i tillegg til navnet. Grunnen til at signaturen også må med er det faktum at Java kan ha flere metoder med samme navn som tar forskjellige inn-parametere i samme klasse. Signaturene beskrives på måten "(inn-data-type)ut-data-type". Datatypene har forkortelser, for eksempel vil en metode som tar inn en `int` og returnerer `void` (ingenting) få signaturen "(I)V". For mer informasjon om datatype-signaturer se [JNI, 2005].

Det er også viktig å sjekke for unntak etter at man har kalt en Java-metode. Selv om C++ har støtte for unntak, er ikke disse kompatible med Java sine unntak. Hvis man kaller en Java-metode, kan det hende at denne metoden kaster et unntak helt ut. Hvis `ExceptionOccurred`-metoden ikke returnerer en *NULL*-peker etter et metode-kall har det oppstått et unntak i Java-programmet. I så tilfelle blir feilmeldingen skrevet ut og programmet avsluttet.

8.2 Løsningsskisser

Vi vil nå beskrive noen løsningsskisser for Java-bindinger som tilfredsstiller de kravene som ble satt for rammeverket i seksjon 3.5.1. Vi gjentar listen over krav her for å gjøre dette mer oversiktlig.

8.2.1 Krav til rammeverk for Java-bindinger

1. Bør ikke kreve noen forandringer på selve BSPlab.
2. Grensesnittet mot BSP-funksjonene bør ligne mest mulig på C-grensesnittet.
3. Det bør være ukomplisert å få kjørt BSP-simulering av Java-programmer.
4. Må kunne kjøre både på Windows og Linux.

8.2.2 Løsningsskisse for Krav 2

For å tilfredsstille dette kravet ønsker vi å gi metodene samme navn og parametre så langt det lar seg gjøre. Som nevnt tidligere støtter ikke Java brukerdefinerte funksjoner i det

globale navnerommet, så eksakt samme "navn" vil ikke funksjonene kunne få. En løsning på problemet vil være å definere en klasse med metoder hvis datatype-signatur tilsvarer BSP C-funksjonene. Navnet gjøres likt med C-navnet bortsett fra at `bsp`-prefiksen fjernes, ettersom vi uansett vil få en `BSP.`-prefiks fra klassenavnet hvis vi kaller klassen `BSP`. Vi kan også gjøre metodene `static`, slik at man ikke trenger å instansiere klassen før man kaller BSP-metodene. Med klassenavnet `BSP`, vil for eksempel BSP C-kallet i listing 8.7 bli seende ut som i listing 8.8 i Java.

Listing 8.7: BSP initialiseringskall i C

```
bsp_begin(bsp_nprocs());
```

Listing 8.8: BSP initialiseringskall i Java

```
BSP.begin(BSP.nprocs());
```

8.2.3 Løsningsskisse 1 for Krav 1 og 3

Før vi kan begynne å beskrive løsningsskissen er det viktig å skjønne hvordan programflyten i et BSPlab program er. Kort fortalt ligger `main`-koden i BSPlab-biblioteket, slik at denne koden alltid blir startet før koden til programmet som skal simuleres. BSPlab leter deretter opp en funksjon med navn `bsp_main`, og kjører så denne i simulatoren. Når `bsp_begin(antall_prosesser)` blir kalt startes `bsp_main` også opp i $antall_prosesser - 1$ ekstra tråder. [Uthus and Dybdahl, 1997]

I denne programflyten skal følgelig Java-metoden med BSP-simuleringskode kjøres istedet for `bsp_main`. Dette kan la seg gjennomføre ved å utnytte `bsp_main`-funksjonen til å starte en JVM og så kjøre Java-programmet i denne JVMen. Dette vil gi oss en kjørbar fil *jbspsim* med følgende program-flyt:

1. `main`-koden i BSPlab starter opp og initialiserer simulatoren.
2. Vår `bsp_main` metode blir kalt av simulatoren.
3. `bsp_main` starter opp en JVM, og kjører `main`-koden i en bestemt Java-fil.
4. Java-metoden kaller `BSP.bsp_begin(antall_prosesser)`; en `native`-metode som mapper til `bsp_begin` i BSPlab-biblioteket som er linket inn i *jbspsim*.
5. BSPlab starter `bsp_main` i $antall_prosesser - 1$ antall tråder.
6. Hver tråd med `bsp_main` starter en JVM og kjører `main`-koden i Java-filen.

Mulige problemer og begrensninger

Løsningsskissen beskriver at det skal startes en JVM per prosess som simuleres. Èn JVM krever en god del ressurser, noe som kan medføre dårlig skalerbarhet for denne løsningen. Det vil si at i forhold til C/C++-varianten av BSPlab vil denne løsningen for Java-bindinger kunne kjøre et langt mindre antall simulerte samtidige prosesser før ressursene i operativsystemet går tom. Hvis løsningen viser seg å fungere i praksis kan det hende at det finnes noen måter å omgå dette problemet på.

Et annet problem med denne løsningsskissen er at Java-programmet blir nødt til å kalle funksjoner (BSP-funksjonene) som ligger i den kjørbare filen som i utgangspunktet startet Java-programmet. Dette fenomenet kalles mer generelt for *back-linking*. Slik JNI-grensesnittet beskrives i [JNI, 2005] støtter Java i utgangspunktet bare å kalle funksjoner lastet inn fra et dynamisk lastbart bibliotek, og det er derfor ikke sikkert at det er mulig å kalle funksjonene direkte i den kjørbare filen. Det taler heller ikke til løsningens fordel at selv om *back-linking* er støttet av Linux, kreves det en del triks for å få *back-linking* til å fungere på Windows (j.fr. 8.3.1).

Selv om løsningsskissens problemer kan virke overveldene ved første øyekast, velger vi likevel å utforske løsningen videre i den tro at vi kan finne måter å omgå problemene på.

Ettersom vi er mest komfortable med Linux som utviklingsmiljø valgte vi å prøve og komme frem til en fungerende løsning på dette operativsystemet, for så å gjøre denne løsningen kjørbar også på Windows. For å gjøre denne "flyttingen" enklest mulig passer vi på å bruke konsepter vi vet det er mulig å få til å fungere også på Windows.

8.2.4 Utprøvning av konseptene til Løsningsskisse 1

Før vi bega oss ut på å implementere *Løsningsskisse 1* mot BSPlab, valgte vi å prøve ut konseptene bak denne løsningen. Som beskrevet i kapittel 8.2.3 er den største usikkerheten knyttet til denne løsningen fenomenet *back-linking*. For å prøve ut konseptet laget vi et C++-program, *backlink*, som starter opp JVM og kaller `main`-metoden i et Java-program. C++-programmet inneholder også en funksjon som er deklarert `native` i Java-programmet, og forsøkes kalt derfra. Listing 8.9 viser hele Java-programmet.

Listing 8.9: Java testprogram for Løsningsskisse 1

```
public class JavaProg {
    public static native void cppTestFunction();

    public static void main(String[] args) {
        cppTestFunction();
    }
}
```

C++-programmet som skal starte Java-programmet er tilsvarende figur 8.6 pluss definisjonen av funksjonen `cppTestFunction` vist i figur 8.10.

Listing 8.10: Definisjon av cppTestFunction

```
JNIEXPORT void JNICALL Java_JavaProg_cppTestFunction
(JNIEnv *, jclass)
{
    std::cout << "cppTestFunction" << std::endl;
}
```

Programflyten for disse programmene blir som følger:

1. C++-programmets `main`-funksjon startes opp.
2. `main`-funksjonen starter opp en JVM.
3. `JavaProg`-klassen letes opp.
4. `main`-metoden i `JavaProg` blir kalt.
5. `JavaProg` kaller `native`-metoden `cpptestFunction`.
6. `cppTestFunction` i C++-programmet blir kalt?

Forskjellen på dette Java-programmet og de beskrevet i kapittel 8.1 er at vi i dette tilfellet ikke kaller Java-metoden `System.loadLibrary` for å laste den dynamisk lastbare biblioteket hvor `native`-metoden er definert. Istedet er `native`-metoden definert i C++-programmet som starter Java-programmet i sin egen JVM.

Resultat

Resultatet av *back-linking*-testen ble som antatt udefinert. Det vil si at JVM krasjet i de fleste tilfeller når `native`-metoden `cppTestFunction` ble forsøkt kalt fra *JavaProg*. Dette resultatet medfører at *Løsningsskisse 1* ikke vil fungere slik den er beskrevet i seksjon 8.2.3. Vi gikk derfor videre med å utforske en ny løsningsskisse.

8.2.5 Løsningsskisse 2 for Krav 1 og 3

I denne løsningen snur vi på flisa i forhold til *Løsningsskisse 1*. Her startes simuleringen ved å starte Java-programmet *JBSPSim*, som først laster inn Java BSP-programmet, og så laster det dynamisk lastbare biblioteket *jbspsim*. Biblioteket *jbspsim* er statisk linket mot biblioteket til BSPlab, og inneholder ellers metoden `bsp_main` som er skrevet spesielt for å kunne brukes i Java-bindinger sammenhengen. Deretter starter *JBSPSim* `main`-koden

i BSPlab, hvorpå resten av flyten nesten blir tilsvarende *Løsningsskisse 1*. Også i denne varianten benytter vi oss av funksjonen `bsp_main` i BSP-programmet, men denne gangen slipper vi å starte en JVM og laste Java-klassen, ettersom dette er gjort før simulatoren har blitt startet.

Program-flyten blir som følger:

1. Java-programmet *JBSPSim* laster Java BSP-programmet *BSPProg* og det dynamisk lastbare biblioteket *jbspsim*.
2. *JBSPSim* kaller så `main`-metoden i BSPlab.
3. `main`-koden i BSPlab starter opp og initialiserer simulatoren.
4. Vår `bsp_main` metode blir kalt av simulatoren.
5. `bsp_main` kaller en bestemt metode i Java-programmet, `javamain`, som innholder BSP-koden som skal simuleres.
6. Java-metoden kaller `BSP.bsp_begin(antall_prosesser)`; en `native`-metode som mapper til `bsp_begin` i BSPlab-biblioteket som er linket inn i *jbspsim*.
7. BSPlab starter `bsp_main` i $antall_prosesser - 1$ antall tråder.
8. Hver tråd med `bsp_main` starter `javamain`-metoden i Java-programmet.

Mulige problemer og begrensninger

Det største problemet som kan oppstå ved denne løsningsskissen er dersom JVMen ikke lar seg bli aksessert fra andre tråder en hovedtråden. Det er stor sannsynlighet for at dette ikke går og at man ender opp med en *access violation*. Dersom dette viser seg å være tilfelle er det for eksempel mulig at de andre trådene enn hovedtråden starter en egen JVM, men dette kan igjen gi andre problemer som beskrevet i *Løsningsskisse 1*.

8.2.6 Utprøvning av Løsningsskisse 2

Konseptene bak *Løsningsskisse 2* (j.fr. seksjon 8.2.5) er litt innviklet å teste ut for seg selv. Det vil si, det kreves at man setter opp et tråd-miljø i C++-programmet. Det er derfor raskere gjort å prøve ut løsningsskissen direkte mot BSPlab enn å sette opp et eget testmiljø uten BSPlab.

Listing 8.11: JBSPSim klassen for løsningsskisse 2

```
public class JBSPSim {
    public static native void cppMain();

    static {
        System.loadLibrary("jbspsim");
    }

    public void callback() {
        System.out.println("java callback");
    }

    public static void main(String[] args) {
        // Call the main-function of the BSPlab library
        cppMain();
    }
}
```

Listing 8.11 viser hoved-Java-programmet som skal starte opp `main`-funksjonen i BSPlab. `native`-metoden `cppMain` er en wrapper-funksjon i biblioteket *jbspsim* som igjen kaller den faktiske `main`-funksjonen til BSPlab som er statisk linket inn i *jbspsim*. Metoden `callback` er bare en test-metode som skal kalles fra alle trådene til BSPlab via `bsp_main`-funksjonen i *jbspsim*-biblioteket.

La oss nå gå over til den litt mer kompliserte implementasjonen av funksjonene til biblioteket *jbspsim* som vises i listing 8.12. Linje 1 til 3 viser den deklarasjonen av noen variable som vi trenger for å holde orden på Java-miljøet vårt. Variablene deklareres utenfor noen funksjon slik at de er tilgjengelige for alle funksjoner i filen, men også som `static` slik at de ikke er tilgjengelige fra funksjoner i andre kode-filer. Variablene *g_env* og *g_obj* representerer de variablene vi alltid får inn til en wrapper-funksjon kalt gjennom en `native` Java-metode. Disse trenger vi for å gjøre enhver operasjon mot Java fra C++. I linje 5 deklarerer vi `main`-funksjonen som ligger i BSPlib som `extern`. Dette forteller kompilatoren at metoden ikke ligger i biblioteket vårt, men i et bibliotek eller objektfil som skal linkes inn senere. Deklarasjonen gjør også `main`-funksjonen "synlig" for funksjonene i denne filen, slik at den kan bli kalt herfra. Linje 7 til 17 viser definisjonen av wrapper-funksjonen `cppMain`. Det første denne funksjonen gjør er å lagre Java-miljøet vårt til de globale variablene *g_env* og *g_obj* slik at disse kan aksessers fra andre funksjoner i den samme filen. Deretter letes `callback`-metoden opp, og en referanse til denne lagres også globalt i *g_mid*. Til slutt i linje 16 kalles `main`-funksjonen i BSPlab med inn-parameterene satt til 0 slik at BSPlab tror at den ikke har fått noen parameter fra kommandolinjen.

Når `main`-funksjonen har blitt kalt tar BSPlab over styringen og initialiserer simuleringsmiljøet. Hovedtråden som gikk inn i `main` vil til slutt lage en ny tråd som kaller `bsp_main` hvorpå hovedtråden venter til simuleringen er ferdig før den gjør noe mer. Linje 19 til 22 viser definisjonen av `bsp_main`-funksjonen. Det eneste denne funksjonen gjør er å bruke

referansene vi allerede har lagret til Java-miljøet og callback-metoden til å kalle metoden `callback` i Java.

Listing 8.12: Implementasjon av biblioteket "jbspsim"

```
 1 static JNIEnv *g_env = NULL;
 2 static jobject g_obj;
 3 static jmethodID g_mid;
 4
 5 extern int main (int argc, char **argv);
 6
 7 JNIEXPORT void JNICALL Java_JBSPSim_cppMain
 8 (JNIEnv *env, jobject obj)
 9 {
10     g_env = env;
11     g_obj = obj;
12
13     jclass cls = g_env->GetObjectClass(g_obj);
14     g_mid = g_env->GetMethodID(cls, "callback", "()V");
15
16     main(0,NULL);
17 }
18
19 void bsp_main(int argc, char **argv)
20 {
21    g_env->CallVoidMethod(g_obj, g_mid, 0);
22 }
```

Programflyten blir som følger:

1. Java-programmet *JBSPSim* blir kjørt.
2. *JBSPSim* laster biblioteket *jbspsim*.
3. *JBSPSim* kaller `native`-metoden `cppMain`.
4. `cppMain`-wrapperfunksjonen i *jbspsim* kaller `main`-funksjonen i BSPlab.
5. BSPlab kaller funksjonen `bsp_main` i *jbspsim*.
6. `bsp_main` kaller `callback`-metoden i *JBSPSim*.

Resultat

Også når denne løsningsskissen prøves ut i praksis viser det seg at noen av de antatte problemene faktisk oppstår. Problemene oppstår når hovedtråden deler seg og den nye tråden fortsetter inn i `bsp_main`-funksjonen. Når denne nye tråden så gjør kall mot en

JVM som er startet fra hovedtråden oppstår det *access violations*. Dette medfører at også denne løsningsskissen ikke er brukbar i sin nåværende form.

8.2.7 Raffinering av løsningsskissene

Ettersom ingen av løsningsskissene slik de ble beskrevet i kapittel 8.2 overvant de antatte problemene, ble vi nødt til å prøve å finne en forbedret løsning. *Løsningsskisse 1* er den mest elegante løsningen av de to, så i første omgang konsentrerte vi oss om å finne en løsningen for problemene til denne skissen.

Hovedproblemet til *Løsningsskisse 1* er at JNI ikke støtter *back-linking*. Java-programmet som blir startet fra C++-programmet *jbspsim* får dermed ingen mulighet til å kalle BSP-funksjonene som er statisk linket mot *jbspsim*. *Back-linking* er derimot støttet av C/C++-linkerene i Linux, så hvis man kunne forskjøvet problemet over til C++ ville det kanskje være løst. Dersom vi putter wrapper-funksjonene til BSP-metodene som skal deklareres som `native` i Java i et eget dynamisk lastbart bibliotek, *javabsp*, får Java i det minste tilgang til wrapper-funksjonene (noe Java ikke fikk da disse lå i den samme kjørbare filen som startet Java-programmet). Problemet nå er at wrapper-funksjonene ikke har tilgang til BSP-funksjonene i BSPlab. Dette kan selvsagt løses ved å linke biblioteket mot BSPlab, men selv om man på denne måten får tilgang til BSP-funksjonene, er det ikke de riktige BSP-funksjonene. Ettersom C++-programmet *jbspsim* også er statisk linket mot BSPlab vil det bli lastet to sett BSPlab hvorav den ene BSPlab aldri blir initialisert, men hvis funksjoner vil bli kalt fra wrapper-funksjonene.

Måten vi kan gi biblioteket *javabsp* tilgang til BSP-funksjonene i *jbspsim* på er ved hjelp av *back-linking*. Ved å linke *jbspsim* dynamisk mot *javabsp*, vil sistnevnte kunne kalle funksjoner i *jbspsim*. Det vil si at når vi starter *jbspsim* vil biblioteket *javabsp* også bli lastet inn i minnet og gitt tilgang til funksjoner i *jbspsim*. Det store spørsmålet er selvsagt om vi ikke også nå vil få to versjoner av det samme biblioteket lastet opp i minnet, i dette tilfellet *javabsp*. Først starter *jbspsim* og laster *javabsp* inn i minnet. Deretter starter *javabsp* opp et Java-program som også forsøker å laste *javabsp* inn i minnet. Vil vi ikke nå bli sittende med to versjoner av det samme biblioteket i minnet? Svaret i dette tilfellet er ganske enkelt nei. Dette skyldes måten den dynamiske linkeren fungerer på [dlopen, 2005]:

> "Only a single copy of an object file is brought into the address space, even if dlopen() is invoked multiple times in reference to the file, and even if different pathnames are used to reference the file."

Sitatet er tatt fra dokumentasjonen til den dynamiske linkeren for Unix/Linux¸ men den samme oppførselen gjelder også for Windows [LoadLibrary, 2005].

Dette gjelder bare for den dynamiske linkeren, og det var derfor vi fikk to versjoner av BSPlab når vi statisk linket det mot *jbspsim* og *javabsp*. Grunnen til at vi ikke brukte dynamisk linking i det tilfelle er at BSPlab og C++Sim må skrives om betraktelig for å kunne lastes dynamisk på Windows, og dette vil bryte med krav *1* for Java-bindingene.

Selv om løsningen bare ville kunne kjøre på Linux bryter dette igjen med krav *4*.

For å teste ut denne raffinerte løsningen laget vi et Java-program som vist i listing 8.13. Dette er Java-implementasjonen av BSP *Hello World*. Koden i `bsp_main` ble tilnærmet lik koden i listing 8.6, men i tillegg la vi til BSP-wrapper-funksjonene vist i listing 8.14. Disse metodene skal kalles fra *javabsp*-biblioteket ettersom dette biblioteket ikke kan kalle BSP-funksjonene som er linket inn i *jbspsim* direkte. I listing 8.15 ser vi koden for *javabsp*-biblioteket. Først `extern`-deklareres BSP-wrapper-funksjonene som ligger i *jbspsim*, så defineres Java sine `native`-metoder til å kalle disse wrapper-funksjonene.

Listing 8.13: Java testprogram for raffinert løsning

```java
class BSP {
    public static native void begin(int maxprocs);
    public static native void end();
    public static native void sync();
    public static native int pid();
    public static native int nprocs();

    static {
        System.loadLibrary("javabsp");
    }
}

public class JavaProg {
    public static void main(String[] args) {
        BSP.begin(BSP.nprocs());
        for(int i=0;i<BSP.nprocs();i++) {
            if( i == BSP.pid() ) {
                System.out.println("Hello Java World from process "
                                    +i+" of "
                                    +BSP.nprocs());
            }
            BSP.sync();
        }
        BSP.end();
    }
}
```

Listing 8.14: C++ testprogram for raffinert løsning

```
1  void __bsp_begin(int maxprocs)
2  {
3      bsp_begin(maxprocs);
4  }
5  void __bsp_end()
6  {
7      bsp_end();
8  }
9  void __bsp_sync()
10 {
11     bsp_sync();
12 }
13 void __bsp_pid()
14 {
15     bsp_pid();
16 }
17 int __bsp_nprocs()
18 {
19     bsp_nprocs();
20 }
21
22 void bsp_main(int argc, char **argv)
23 {
24     [...]
25 }
```

Listing 8.15: Kode for "javabsp"-biblioteket for den raffinerte løsningen

```
1  extern void __bsp_begin(int maxprocs);
2  extern void __bsp_end();
3  extern void __bsp_sync();
4  extern void __bsp_pid();
5  extern int __bsp_nprocs();
6
7  JNIEXPORT void JNICALL Java_BSP_begin
8  (JNIEnv *env, jclass obj, jint maxprocs)
9  {
10     __bsp_begin(maxprocs);
11 }
12
13 JNIEXPORT void JNICALL Java_BSP_end
14 (JNIEnv *env, jclass obj)
15 {
16     __bsp_end();
17 }
18
19 JNIEXPORT void JNICALL Java_BSP_sync
```

```
20 (JNIEnv *env, jclass obj)
21 {
22     __bsp_sync();
23 }
24
25 JNIEXPORT jint JNICALL Java_BSP_pid
26 (JNIEnv *env, jclass obj)
27 {
28     __bsp_pid();
29 }
30
31 JNIEXPORT jint JNICALL Java_BSP_nprocs
32 (JNIEnv *env, jclass obj)
33 {
34     __bsp_nprocs();
35 }
```

Programflyten til den raffinerte løsningen blir som følger:

1. *jbspsim* startes.
2. Den dynamiske linkeren laster biblioteket *javabsp* inn i minnet.
3. `main`-koden i BSPlab starter opp og initialiserer simulatoren.
4. Vår `bsp_main` metode blir kalt av simulatoren.
5. `bsp_main` starter opp en JVM.
6. `JavaProg`-klassen letes opp og lastes.
7. Når `JavaProg`-klassen lastes, lastes også `BSP`-klassen som inneholder alle `native` BSP-metodene.
8. `BSP`-klassen ber den dynamiske linkeren laste inn biblioteket *javabsp*. Ettersom biblioteket allerede er lastet inn i minnet, returnerer linkeren en referanse til det allerede lastede biblioteket.
9. `main`-metoden i `JavaProg` blir kalt.
10. Java-metoden kaller `BSP.begin(antall_prosesser)`; en `native`-metode hvis wrapper-funksjon ligger i biblioteket *javabsp*.
11. *javabsp* sin implementasjon av `native`-metoden kaller enda en tilsvarende BSP-wrapper funksjon `__bsp_begin` som ligger implementert i *jbspsim*.
12. *jbspsim* sin `__bsp_begin` kaller BSPlabs `bsp_begin`-funksjon.
13. BSPlab starter `bsp_main` i $antall_prosesser - 1$ antall tråder.
14. Hver tråd med `bsp_main` starter en JVM og kjører `main`-koden i Java-filen.

Resultat

Denne løsningen fungerte utmerket helt til andre enn den første tråden kom til kallet som starter en ny JVM i `bsp_main`. Det viste seg at selv om referansedokumentasjonen til Java sier at det skal være mulig å starte flere JVMer fra samme program, så støtter ikke SUN sin Java-implementasjon dette enda. Det viste seg at løsningen vår likevel var mulig å gjennomføre. JNI har en funksjon som gjør det mulig å koble en JVM som har blitt startet opp i èn tråd til andre tråder. Ved å bruke denne funksjonaliteten omgår vi ikke bare problemet med at man bare får startet en JVM per program, skaleringsproblemet som ble beskrevet i seksjon 8.2.3 blir også løst.

Listing 8.16 viser en forenklet utgave av `bsp_main` hvor vi har tatt i bruk den nye metoden `AttachCurrentThread`. I linje 1 og 5 bruker vi `static`-variabler. Variabelen `run` brukes til å avgjøre om det er den første tråden som starter `bsp_main`. I så fall må vi starte en JVM som vist i linje 8. `jvm`-variabelen brukes til å koble alle andre tråder enn den første opp mot JVMen. Ettersom variablen er `static` vil den beholde verdien den fikk i kallet til `JNI_CreateJavaVM` i alle etterkommende kall til `bsp_main`. Vi ser at i linje 14 utnyttes dette til å koble JVMen opp mot de andre trådene. Kallet til metoden `AttachCurrentThread` i `jvm` gir oss et JNI-miljø vi kan bruke videre i de andre trådene. Denne siste løsningen medførte av Java-versjonen av BSP *Hello World* (listing 8.13) kjørte utmerket på Linux.

Listing 8.16: Bruk av AttachCurrentThread i "jbspsim"

```
1  void bsp_main(int argc, char **argv)
2  {
3      static int run = 0;
4      ++run;
5
6      JNIEnv *env = NULL;
7      static JavaVM *jvm = NULL;
8
9      if( run == 1 ) {
10         jint jni_error = JNI_CreateJavaVM(&jvm,
11                          reinterpret_cast<void**>(&env),
12                          &vm_args);
13         jclass cls = env->FindClass("JavaProg");
14         [...]
15     } else if( run > 1 ) {
16         jni_error = jvm->AttachCurrentThread(
17                          reinterpret_cast<void**>(&env),
18                          NULL);
19         jclass cls = env->FindClass("JavaProg");
20         [...]
21     }
22 }
```

8.3 Flytting av Linux-varianten til Windows

Selv om vi nå hadde en BSP simulator som kunne kjøre enkle Java BSP-program på Linux, gjensto det en del forandringer før Java-versjonen også ville kompilere på Windows. Som nevnt tidligere brukte vi *back-linking* i den endelige løsningen på Linux, noe som ikke vil fungere uten videre på Windows. Problemet oppstår når vi skal kompilere og lage det dynamiske biblioteket *javabsp* vist i listing 8.15. På Windows vil vi da få *undefined reference* til de eksterne `__bsp_XXXXX` wrapper-funksjonene. *Undefined reference* i et dynamisk bibliotek på Linux skaper ingen problemer så lenge funksjonene finnes i programmet biblioteket til slutt blir linket mot (eller noen andre av bibliotekene programmet linker mot). På Windows er det derimot ikke mulig å lage et dynamisk bibliotek hvor ikke alle funksjonene er implementert.

8.3.1 Bruk av interface-klasse for å løse *back-linking* problemet

Vi kan komme oss unna *back-linking*-problemet på Windows ved help av en interface-klasse, også kalt en 100% abstrakt klasse. Istedet for å kalle `__bsp_XXXX` funksjonene fra *javabsp*-biblioteket. Kaller vi metodene i interface-klassen `BSPwrapper` vist i listing 8.17. I listing 8.18 ser vi den nye implementasjonen til *javabsp*-biblioteket. I linje 5 defineres en statisk peker til et objekt som arver fra `BSPwrapper`. Funksjonen i linje 6 må brukes av *jbspsim*-programmet for å gi *javabsp* en peker til en instansiert variant av `BSPwrapper`. Denne funksjonen er deklarert med makroen `DLLEXPORT` ettersom Windows krever at funksjoner i bibliotek som skal brukes direkte av andre programmer må være beskrevet med nøkkelordet `__declspec(export)` [declspec, 2005]. Derfor er makroen `DLLEXPORT` satt som ingenting på Linux, og som `__declspec(export)` på Windows. Hvis vi ser på alle BSP-kallene i listing 8.18 nå, ser vi at i steden for å kalle `__bsp_XXXX`-funksjonene, kalles `bspwrap->bsp_XXXX`-metodene. Ettersom klassen `BSPwrapper` er hundre prosent abstrakt, finnes det ingen implementasjon til den, og vi kan dermed ikke få *undefined reference* når vi kompilerer *javabsp*-biblioteket.

Listing 8.17: Den helt abstrakte klassen BSPwrapper

```
class BSPwrapper {
public:
    virtual ~BSPwrapper() {};

    virtual void bsp_begin(int maxprocs) = 0;
    virtual void bsp_end() = 0;
    virtual void bsp_sync() = 0;
    virtual int bsp_pid() = 0;
    virtual int bsp_nprocs() = 0;
};
```

Listing 8.18: Koden for siste versjon av "javabsp"-biblioteket

```
1  #include "BSPJava.h"
2  #include "jbspwrap.h"
3  #include "libjavabsp.h"
4
5  // Global variables
6  static BSPwrapper *bspwrap;
7
8  DLLEXPORT void set_bspwrapper(BSPwrapper *bsp)
9  {
10     bspwrap = bsp;
11 }
12
13 JNIEXPORT void JNICALL Java_BSP_begin
14 (JNIEnv *env, jclass cls, jint maxprocs)
15 {
16     bspwrap->bsp_begin(maxprocs);
17 }
18
19 [...]
```

For at *javabsp*-biblioteket skal kunne bruke `BSPwrapper`-pekeren sin, er den nødt til å få en peker til et faktisk objekt. Ettersom `BSPwrapper` er helt abstrakt kan den ikke instansieres. Vi lagde derfor en klasse `BSPwrapper_impl` som arver `BSPwrapper` og implementerer alle de abstrakte metodene. Et utsnitt av denne koden vises i listing 8.19. Ettersom linkeren i Windows ikke tillater at `BSPwrapper_impl::bsp_begin` direkte kaller `bsp_begin`-funksjonen i BSPlab-biblioteket, må BSP-metodene til `BSPwrapper_impl` kalle enda en BSP-wrapper-funksjon (`__bsp_XXXX`) som så kaller den faktiske BSP-funksjonen i BSPlab.

Listing 8.19: Implementasjonen av BSPwrapper

```
class BSPwrapper_impl : public BSPwrapper {
public:
    void bsp_begin(int maxprocs);
    void bsp_end();
    void bsp_sync();
    int bsp_pid();
    int bsp_nprocs();
};

void BSPwrapper_impl::bsp_begin(int maxprocs)
{
    __bsp_begin(maxprocs);
}

void __bsp_begin(int maxprocs)
{
    bsp_begin(maxprocs);
}
```

Etter at vi hadde laget implementasjonsklassen til `BSPwrapper` kunne vi la *jbspsim* instansiere et objekt av klassen `BSPwrapper_impl` første gang `bsp_main` blir kalt. I linje 6 i listing 8.20 ser vi at *jbspsim* lager et objekt av typen `BSPwrapper_impl`. Pekeren til dette objektet blir så sendt inn til *javabsp*-biblioteket ved hjelp av kallet til funksjonen `set_bspwrapper`.

Listing 8.20: Bruk av BSPwrapper i "jbspsim"

```
1  void bsp_main(int argc, char **argv)
2  {
3      [...]
4      if( run == 1 ) {
5          BSPwrapper_impl *bspwrap = new BSPwrapper_impl;
6          set_bspwrapper(bspwrap);
7
8          jint jni_error = JNI_CreateJavaVM(&jvm,
9                           reinterpret_cast<void**>(&env),
10                          &vm_args);
11         jclass cls = env->FindClass("JavaProg");
12         [...]
13     } else if( run > 1 ) {
14         [...]
15     }
16 }
```

Da vi endelig hadde omgått *back-linking*-problemet på Windows ved hjelp av interface-

klassen `BSPwrapper` kompilerte og kjørte *jbspsim* like bra på Windows som på Linux.

8.3.2 Finpuss av endelig løsning

Selv om løsningen vår nå kompilerte og kjørte både på Linux og Windows trengte den en ekstra finpuss for å være så enkel som mulig å bruke.

Forandre navn på Java BSP-program-klassen

I steden for `JavaProg` valgte vi å kalle Java-klassen *jbspsim* forventer å finne for `BSPProg`. Den eneste grunnen til dette er at navnet er mer selvforklarende for brukeren.

Legge BSP-klassen i egen fil

I den siste løsningen vår lå både Java BSP-programmet og deklarasjonen av BSP `native`-metodene i samme fil. Dette er kløneте ettersom det krever at brukeren kopierer BSP-deklarasjonene inn i hvert eneste Java BSP-program. Vi valgte derfor å putte disse deklarasjonene i en egen Java-fil ved navn *BSP.java*.

Sette riktig CLASSPATH

La oss si at Java BSP-simuleringsprogrammet vårt *jbspsim* er installert på brukerens maskin på området *C:/Program Files/BSPlab/jbspsim*. Brukeren har også lagt dette område i miljøvariabelen `PATH` slik at han skal kunne kjøre programmet med kommandoen *jbspsim* fra hvor som helst på maskinen. På området *jbspsim* ligger følgende filer (på Linux kan filnavn og *extension* variere litt fra Windows) :

jbspsim.exe Simuleringsprogrammet som brukes for å starte Java BSP-programmer. Er statisk linket mot BSPlab-biblioteket.

BSP.class Deklarasjonen av BSP-funksjonene som Java `native`-metoder.

javabsp.dll Det dynamisk lastbare biblioteket som inneholder wrapper-funksjonene til Java `native`-metodene som er deklarert i *BSP.class*.

Hvis brukeren nå står på et område hvor han har kompilert Java BSP-programmet sitt *BSPProg* og kjører *jbspsim* for å simulere Java-programmet, vil han få beskjed om at JVM ikke finner `BSP`-klassen. Grunnen til dette er at JVMen som blir startet fra *jbspsim* bare leter i det nåværende område for Java-klasser. Vi valgte å løse dette problemet ved å få JVMen til å lete etter Java-klasser ut fra miljøvariabelen *CLASSPATH* i steden. Dette ble gjennomført ved å forandre koden vist i listing 8.21 til den i listing 8.22.

Listing 8.21: Original kode for CLASSPATH

```
void bsp_main(int argc, char **argv)
{
    [...]
    if( run == 1 ) {
        [...]
        JavaVMOption options[1];
        // Setup the options list.
        options[0].optionString = "-Djava.class.path=.";
        options[0].extraInfo = NULL;
        [...]
    } else if( run > 1 ) {
        [...]
    }
}
```

Listing 8.22: Ny kode for CLASSPATH

```
1  void bsp_main(int argc, char **argv)
2  {
3      [...]
4      if( run == 1 ) {
5          [...]
6          JavaVMOption options[1];
7          // Setup the options list.
8          const char *cp = getenv("CLASSPATH");
9          string classpath = ".";
10         if( cp && cp[0] ) classpath = cp;
11         string optionString =
12             string("-Djava.class.path=") + classpath;
13         options[0].optionString =
14             const_cast<char*>(optionString.c_str());
15         options[0].extraInfo = NULL;
16         [...]
17     } else if( run > 1 ) {
18         [...]
19     }
```

8.4 Endelig resultat

Det endelige resultatet oppfyller alle krav satt i 8.2.1. Selv om vi bare har laget et *proof-of-concept*, det vil si vi har bare implementert de funksjonene som trengs for å lage Java-versjonen av BSP-programmet *Hello World*, mener vi at det skal være relativt enkelt å utvide programmet til å støtte alle BSP-funksjonene BSPlab støtter. Programmet vårt

kompilerer uten forandringer både på Windows og Linux. Det bruker også en uforandret utgave av BSPlab for å fungere, noe som gjør det enkelt å eksportere ny/forandret funksjonalitet i BSPlab til Java-bindingene.

8.4.1 Bruk av den endelige løsningen

Siden Java-bindingene bare en et *proof-of-concept* og ikke støtter alle BSP-funksjonene til BSPlab, følger det ikke med noen forklaring på hvordan du bruker bindingene i *tutorial*en. Vi velger derfor å ha med en kort forklaring på bruken her i stedet.

Windows

Når BSPlab er ferdig installert er Java-bindingene å finne under *C:/Program Files/BSPlab/jbspsim*. I underområdet *bin* finnes blant annet files *jbspsim-vars.bat*. Denne filen brukes til å sette miljøvariabler som må være satt for at Java-bindingene skal kunne brukes. Siden verdien til disse variablene kan variere fra Java-versjon til Java-versjon, er det viktig at man oppdaterer innstillingene i *jbspsim-vars.bat* før man kjører den.

Etter at *jbspsim-vars.bat* er oppdatert, kan man gjøre følgende for å kjøre BSP Java-programmet *Hello World*:

1. Åpne kommandolinjen i Windows.
2. Kjøre `jbspsim-vars.bat`.
3. Gå til *example*-området under *jbspsim*-området.
4. Kjøre kommandoen `jbspsim.exe`.

Linux

For å få testet Java-bindingene på Linux, må man kompilere opp alt selv. Det er viktig at stiene til *Java Runtime Enviroment* blir satt opp riktig i *CMakeLists.txt*-filene før CMake blir kjørt for å generere prosjektfiler. Etter at prosjektet er kompilert opp vil man få den kjørbare filen *jbspsim* i *src*-området under *jbspsim*. Ved å kjøre denne filen vil BSP Java-programmet *Hello World* som ligger i *BSPProg.java* på samme område bli simulert i BSPlab.

Kapittel 9

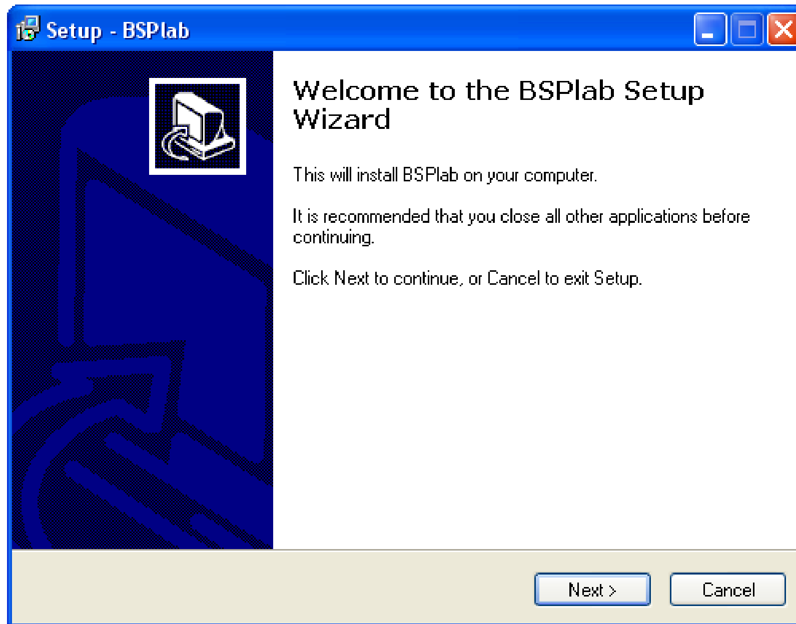
Installation Howto

This chapter will describe how to go about installing BSPlab on your computer. For the Windows installation guide refer to section 9.1, and for the Linux installation guide please look up section 9.2

9.1 Windows installation

The Windows installation of BSPlab uses standard Windows installation wizards, so there is no need to be a Windows expert to perform the BSPlab installation. For non-experienced users we recommend keeping the default settings and just clicking through the install wizard as described in 9.1.1. If you are a so-called expert user, you might want to perform a custom installation as described in 9.1.2. Common uses for the custom installation is if you already got *Dev-C++* and/or the *MinGW* compiler installed on your system. Other reasons to go for the custom installation might be if you want to use *Visual Studio .NET* as your develoment tool instead of *Dev-C++/MinGW*.

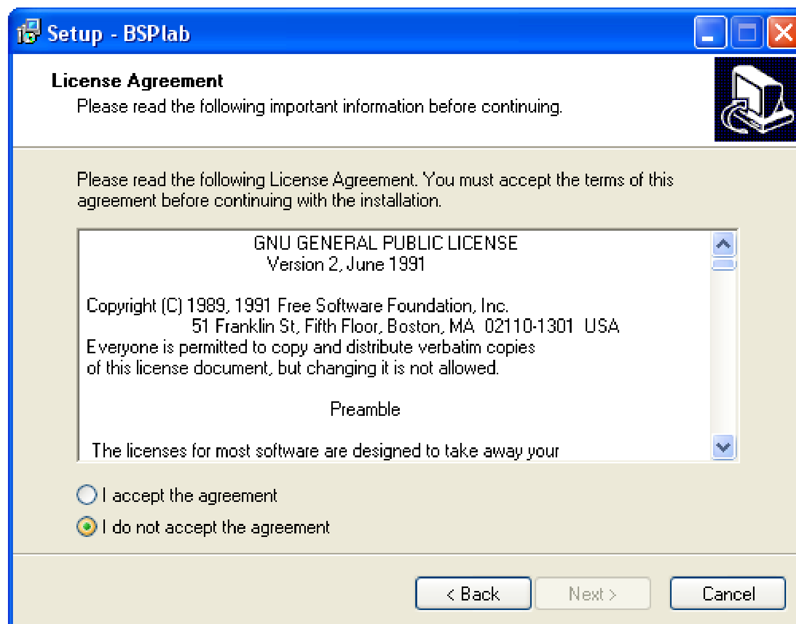
9.1.1 Default BSPlab installation with Dev-C++ and MinGW



Figur 9.1: Installation wizard startup screen

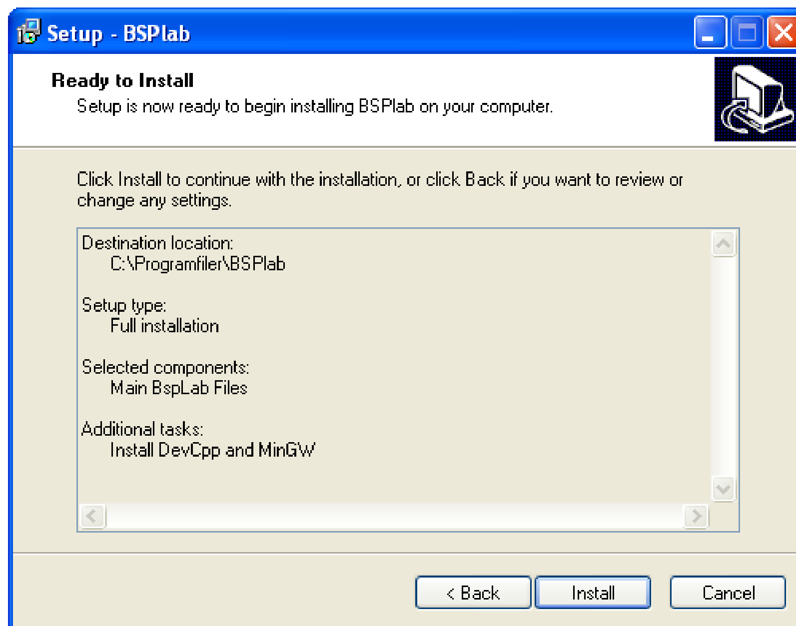
1. Before starting the BSPlab installation program we recommend that you switch to an user with administrator privileges. This is due to the fact that the development tools need to be installed systemwide to function properly.

To install BSPlab on your computer, insert the BSPlab installation CD into the CD/DVD-ROM. The installation program should start immediately, if not, please access the CD/DVD-ROM drive and run the file *bsplab-install.exe*. When the installation program starts you should be presented with the screen shown in figure 9.1.



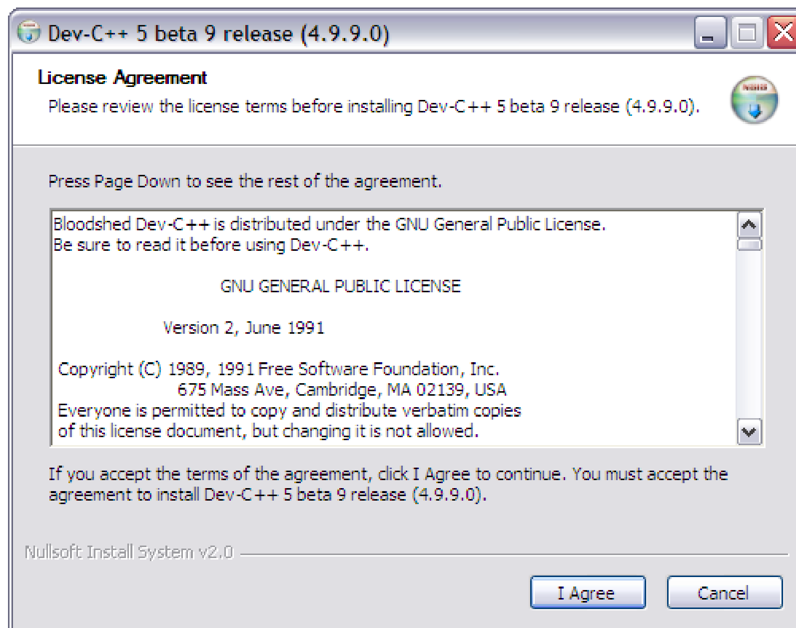
Figur 9.2: Installation wizard license agreement screen

2. On the wizard startup screen, please press *Next* to be presented with the *License Agreement* screen shown in figure 9.2. To continue you have to accept the license agreement by checking *I accept the agreement*, then press *Next*. If you do not agree with the license agreement, you can not continue with the installation.



Figur 9.3: Installation wizard Ready to install screen

3. If you accept the license agreement, the *Destination Location* screen will appear. We do not recommend to change the default location unless you are a expert user and/or have a good reason to do so. Just press *Next* three more times, and the *Ready to install* screen shown in figure 9.3 should appear.



Figur 9.4: Dev-C++ installation wizard startup screen

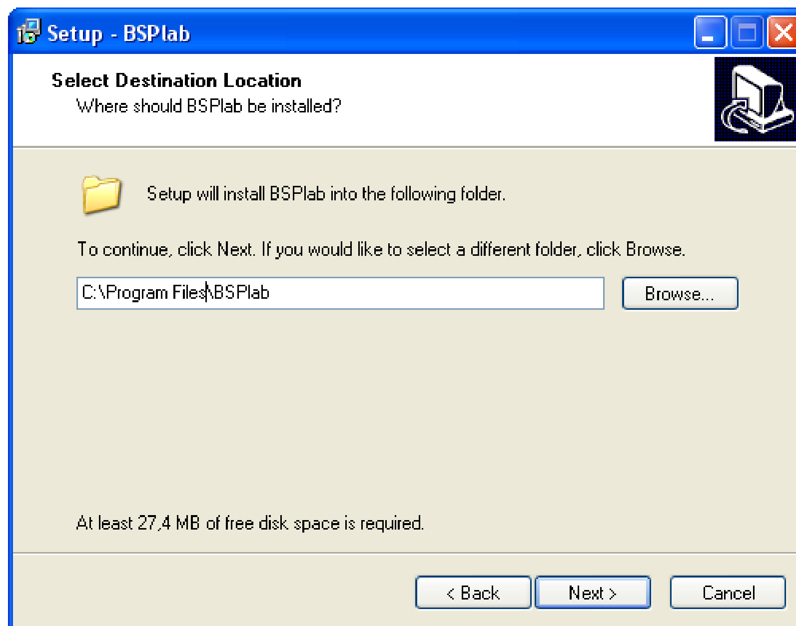
4. Now press the *Install* button and watch BSPlab being installed. When the installation wizard has finished installing BSPlab, it will launch the Dev-C++ development tools installation wizard presented in figure 9.4.

5. To continue installing the development tools, you must read and accept the *End User License Agreement* presented here. If you do not agree with the license agreement, you cannot continue with the installation. If you do agree, please press the *I agree* button to continue. Now just click through the installation wizard to let Dev-C++ be installed.

6. Congratulations, you have now finished installing BSPlab and its development enviroment. Please refer to section 10.1 to learn how to do simulations with BSPlab.

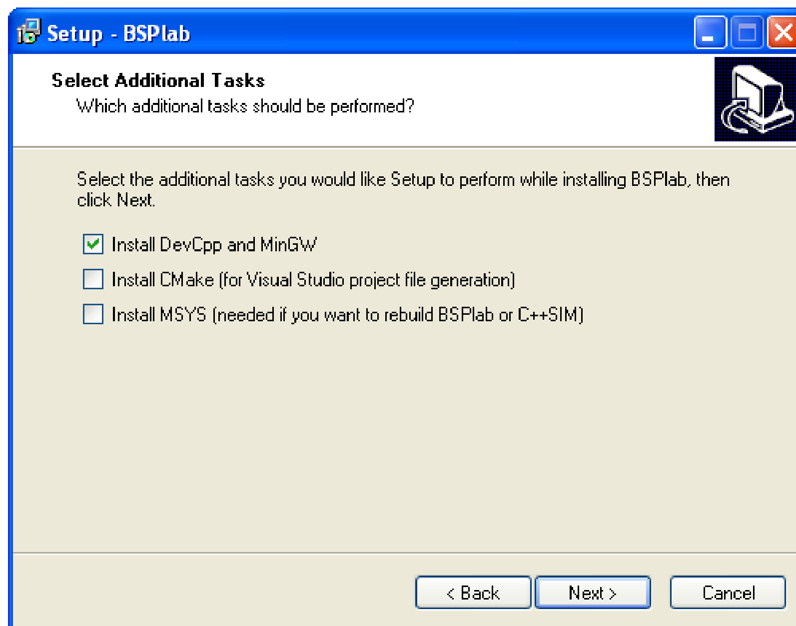
9.1.2 Custom BSPlab installation (for user with i.e. Visual Studio .NET)

1. To perform a custom BSPlab installation please follow the first two instructions in 9.1.1 to be presented with the *Destination Location* screen shown in figure 9.5. If you want to change the default location, be aware that all further install instructions and tutorials will use the default location as reference. Press *Next* when you are satisfied with the destination location.



Figur 9.5: Installation wizard destination location screen

2. Press *Next* once more to be presented with the *Selected Additional Tasks* screen shown in figure 9.6. Uncheck the *Install DevCpp and MinGW* option (unless you also want to experiment with these development tools), and check *Install CMake*. Press *Next*, then *Install* and watch BSPlab being installed. When the installation wizard has finished installing BSPlab it will launch the CMake installation wizard (and other installation wizards if you checked for more than CMake).



Figur 9.6: Installation wizard additional tasks screen

3. Click through the CMake installation wizard. Remember, the same apply to the CMake installation location as with BSPLab's, all further documentation will assume you installed CMake to the default location.

4. After you have finished installing CMake, it is time to create the BSPlab and C++Sim libraries. First you have to create the project files for your compiler/development enviroment.

1. Start the Windows Command line prompt. This may be done by pressing the *Start* button, then *Run command*, type in `cmd` and hit `[ENTER]`.
2. Change to the *cppsim* directory in the BSPlab direcory (e.g. `cd C:/Program Files/BSPlab/cppsim`).
3. Run CMake to generate project files by typing `C:/Program Files/CMake20/bin/cmake .` (change path if you installed CMake elsewhere). If you have more than one compiler installed on your system you might have to specify for which compiler you want to generate project files. Run CMake with the parameter `--help` to get a list of available generator, and then add the parameter `-G generator-name` to the above CMake command.
4. Change directory to the *bsplab* project directory (e.g. `cd C:/Program Files/BSPlab/bsplab`) and repeat item 3.

5. Change directory to the *bspprog* project directory (e.g. `cd C:/Program Files/BSPlab/bspprog`) and repeat item 3.

5. You have now finished generating project files. The next thing we have to do is compile the libraries. Do this by first opening the C++Sim project file in your developer enviroment (the project file should be found in the directory *C:/Program Files/BSPlab/cppsim*). When you have opened the project file, go on to compile the project. If you do not encounter any errors repeat the process with the BSPlab project file probably located in *C:/Program Files/BSPlab/bsplab*.

Congratulations, you have now finished installing BSPlab for a custom development enviroment. Please refer to section 10.1 to learn how to do simulations with BSPlab. Remember you have to use the project file you generated for *bspprog* instead of the one used in the examples.

9.2 Linux installation

On Linux you can either install BSPlab systemwide, or you can install it userwide. If you install it systemwide, all users on the system may use BSPlab without extra installation effort. If you instead install it for one user (userwide), only this user may use BSPlab. You must have root (administrator) access to the system to do a systemwide installation. Consult 9.2.1 to perform a systemwide installation. If you do not have root access, or want to do an userwide installation, consult 9.2.2.

Requirements

- GNU C++ Compiler version 3 or higher
- GNU Make utilities

The installer will detect if your system fullfills these requirements. If your system does not have these development tools installed, consult the *Troubleshooting* paragraph on page 117.

9.2.1 Systemwide BSPlab installation

To perform a systemwide installation, do the following:

1. Log in as user *root*.
2. Change path to the root of the installation CD and execute the command *./bsplab_install*.

3. Choose the installation path of BSPlab. Just pressing [ENTER] will install BSPlab in */usr/local* (libraries in */usr/local/lib* and header files in */usr/local/include*).
4. If the *CMake* automake utility is not already installed on the system, you will be asked where you want it installed. Unless you specify otherwise, *CMake* will also be installed in */usr/local*.
5. If the installation was a success, the message *Installation completed successfully* will be displayed on the screen. If this is not the case, you should be presented with an error message. Please refer to *Troubleshooting* on page 117 to solve the problem.

9.2.2 User BSPlab installation

To perform an user installation, do the following:

1. Change path to the root of the installation CD and execute the command *./bsplab_install*.
2. Choose the installation path of BSPlab. Just pressing [ENTER] will install BSPlab in *bsp* under your home directory.
3. If the installation was a success, the message *Installation completed successfully* will be displayed on the screen. If this is not the case, you should be presented with an error message. Please refer to *Troubleshooting* on page 117 to solve the problem.

Troubleshooting

g++ not found Please consult your linux distribution documentation on how to install the development tools.

make not found See above.

permission denied You are probably be trying to install the software in a location where you do not have write access.

Kapittel 10

BSPlab Tutorial

This chapter will try to describe how to use and run BSPlab. The chapter is split in two main sections, the first section, section 10.1, will try to explain how to run BSP simulations in BSPlab. The other section, section 10.2, will describe how an user may customize the BSPlab library and make new distributions of BSPlab.

10.1 How to run programs with BSPlab

BSPlab is an implementation of the *BSP Worldwide Standard Library*, and is used to run simulations of BSP programs. This section will describe how to write BSP programs and simulate them, and also how to run your existing BSP programs in the BSPlab simulator.

10.1.1 The *Hello World* BSP test program

BSPlab comes with the *Hello World* BSP test program. The first thing you should do when you have installed BSPlab is to compile and run this test program.

Compiling the *Hello World* test program

1. Go to the *bspprog* directory where you installed BSPlab (usually in C:/Program Files/BSPlab).
2. Double click on the file *bspprog.dev* to open the project in the Dev-C++ IDE.
3. Press *CTRL+F9* to build the program. Make sure no errors appear in the compiler log.

Running the *Hello World* test program

1. Start the Windows Command line prompt. This may be done by pressing the *Start* button, then *Run command*, type in *cmd* and hit [ENTER].
2. Change to the *bspprog* directory (i.e. *cd C:/Program Files/BSPlab/bspprog*).
3. Run the test program by typing *bspprog* followed by [ENTER].

When the test program has run, you should get a lot of output on your screen. The first you see is the simulator setup parameters and initialization, then the BSP program's output. In this *Hello World* test case, all the simulated processors should greet you with the phrase *Hello World*.

10.1.2 Running your own BSP programs in the simulator

The simplest way to running your own BSP programs in the simulator, is to replace the *bspmain.cpp* with your own. Then you have to change the name of your `main` function to `bsp_main`, and change the return value from `int` to `void`. The main function will probably look something like this:

```
int main(int argc, char *argv[])
{
    // bsp program main code
}
```

Change it to:

```
void bsp_main(int argc, char *argv[])
{
    // bsp program main code
}
```

You will also have to comment out the `return`-statements from the `bsp_main` function. After you have made the changes, compile the program, and then execute the produced executable to run the program in the simulator.

Running BSP programs with multiple files in the simulator

If you have a BSP program with more than one file, the previously explained strategy will not work. Instead you can do the following:

1. Create a new folder in the BSP directory with an appropriate name for your project. It is important that the new folder is placed in the BSP directory so the path to the include files and libraries will be correct. If you do not want this restriction, please read section 10.1.3 to learn how to make your own Dev-C++ projects run in the simulator.

2. Copy the file *bspprog.dev* from the *bspprog/templates* folder into your new folder.
3. Open the *bspprog.dev* file in your folder.
4. Go to menu option *Projects/Add to Project*, and add your source files.
5. Change the signature of the *main* method as described above.
6. Compile and run.

10.1.3 Making your own Dev-C++ projects run in the simulator

If you already have got a Dev-C++ BSP program project, or you want to create a BSP Dev-C++ project from scratch, this section describes how to set up the project to include the correct header files and libraries.

Setting up the correct include paths

When you have opened up your BSP program project in the Dev-C++ editor, choose the *Project/Project options* menu option or press *ALT+P*. You should be presented with a smaller window with some tabs at the top. Click the *Directories*-tab, then the *Include directories*-tab that shows up below. Select the input field almost at the bottom of the window and type in *C:/Program Files/BSPlab/bsplab/src* and press the *Add* button. Repeat the process with the following path: *C:/Program Files/BSPlab/cppsim/Include*. If you have installed BSPlab in another directory, please change the paths accordingly.

Linking against the necessary libraries

In the *Project options* window, go to the *Parameters* tab. Then press the *Add Library or Object* button, and select the BSPlab library *C:/Program Files/BSPlab/bsplab/lib/mingw/libbsplab.a*. Repeat the process with the following libraries:

- *C:/Program Files/BSPlab/cppsim/lib/libC++SIM.a*
- *C:/Program Files/BSPlab/cppsim/lib/libCommon.a*
- *C:/Program Files/BSPlab/cppsim/lib/libEvent.a*

Please make sure the paths correspond with where you chose to install BSPlab.

If your BSP program already is linking against a BSP library, make sure you remove this library from the library list. Else you might end up getting "multiple definitions" errors from your linker as the BSP library and the BSPlab library both implement the same functions.

Compiling the project

When you have updated your include paths and library linkage, you should make sure you have followed the instructions in section 10.1.2 before you try to compile the project.

10.1.4 The *Parameters.dat* file

The *Parameters.dat* file placed in the same directory as the BSP program to be simulated controls a lot of the variables to be used in the simulation. Most of the options in the file are described in comments inside the file. For further information about the parameters please refer to [Uthus and Dybdahl, 1997].

Later versions of BSPlab has added two new parameters to the *Parameters.dat* file:

StackSize This option specifies in bytes how much stack should be allocated to the BSP program to be simulated.

RealTime This option specifies whether the operating system should schedule the simulation in real time or not.

StackSize

If you are using lots of processors in your simulation you might run into a problem where the operating system refuses to create new threads because it is out of memory. This is most likely attributed to the fact that the operating system allocates 1-2MB of memory to each thread created, and that the OS has a virtual memory limit of 2GB. If your BSP program does not need a stack this large, you may use the *StackSize* parameter in the *Parameters.dat* file to lower the allocated stack size.

RealTime

If you want to make sure that other program running on the OS does not interfere with your simulations, you should quit as many programs as you can, and then run the simulation in real time scheduling. To enable real time scheduling set the *RealTime* parameter in the *Parameters.dat* file to 1. Also make sure that you are running the program as administrator/root as real time scheduling requires administrator rights.

10.2 Customizing BSPlab

This section will describe how to go about to change parts of the BSPlab simulation core, and then making an usable library for your self and/or others. The section is split in two parts. Subsection 10.2.1 will describe what you should do to make sure the CMake

configuration system is correct after you have added and/or removed files from the project, and how to distribute the new version. Subsection 10.2.4 will describe how to customize BSPlab from Dev-C++.

Before you start making changes to the BSPlab source code we recommend that you read about its design and architecture in [Uthus and Dybdahl, 1997].

10.2.1 Making a new distribution of BSPlab

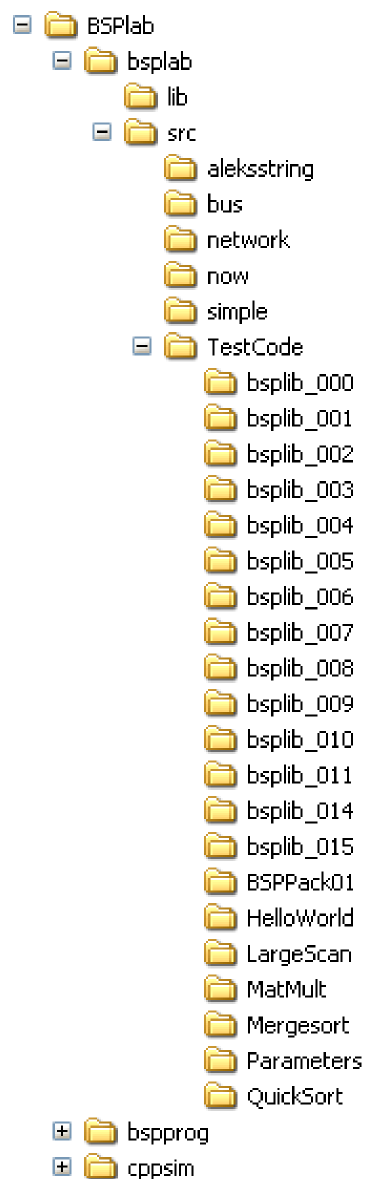
In order to make a new distribution of BSPlab you will need to have installed the *CMake* and *MSYS* programs. Please follow instruction 1 to 3 of section 9.1.2 if you have not installed these programs already. Be sure to also check the option for *MSYS* in instruction 2. The instructions only apply to the Windows platform, if you are using Linux all the necessary tools to make a new distribution of BSPlab should already be installed. Please be aware that MSYS renames the file *C:/Program Files/Dev-Cpp/bin/make.exe* to *C:/Program Files/Dev-Cpp/bin/mingw32-make.exe*. Please copy the file back to it's original name for Dev-C++ to continue to work.

10.2.2 BSPlab structure and CMake setup

In order for you to compile BSPlab and its support library C++Sim you will have to know how to use the CMake automake utility. This section will try to explain the most crucial elements of the CMake setup, for further information please refer to [CMake, 2005].

BSPlab package structure

When you have installed BSPlab the structure of the BSPlab directory (usually *C:/Program Files/BSPlab* should look like Figure 10.1.



Figur 10.1: The BSPlab directory structure

Here is a short explaination of each directory and sub-directories.

bsplab The BSPlab source code and library, including some BSP test programs.

cppsim BSPlab's support library, provides the process simulation core. This library does not include any BSP simulation code.

bspprog A BSP test program, may be used as a template to create new BSP programs to

be simulated in BSPlab.

BSPlab

The *bsplab* directory has the following sub-directories:

lib This is where the library file gets compiled and stored.

src All the BSPlab source code is stored in or below this directory.

The *src* directory contains most of the BSP simulator core. Code that is specific to each architecture that BSPlab can simulate is stored in the *bus*, *network*, *now* and *simple* sub-directories. The *src* directory also includes a directory called *TestCode*. This directory contains many of the BSP programs used to test BSPlab during development.

The CMake build files

BSPlab

The CMake build files for BSPlab, CMakeLists.txt, are placed in the *bsplab* root directory, the *src* directory and the *TestCode* directory below *src*. The *CMakeLists.txt* file in the root directory is used to set up some global options, like where the finished library file should be stored. The file also defines which sub-directories should be traversed for more *CMakeLists.txt* files, like the *src* directory.

The *CMakeLists.txt* file in the *src* directory contains most of the settings for compiling the BSPlab library. The `ADD_LIBRARY` command in this file is used to define which files are part of the BSPlab library. If you want to remove or add files from/to BSPlab, you will have to change the `ADD_LIBRARY` command to reflect these changes.

C++Sim

C++Sim is built up of 5 libraries. BSPlab uses 3 of these libraries, called *C++Sim*, *Common* and *Event*. The *CMakeLists.txt*-files are placed in the following directories:

- The C++Sim root directory.
- *ClassLib*.
- *ClassLib/src*. The *CMakeLists.txt* file in this directory produces the library called *C++Sim*.
- *Common*.
- *Common/src*. The *CMakeLists.txt* file in this directory produces the library called *Common*.
- *Event*.

- *Event/src*. The *CMakeLists.txt* file in this directory produces the library called *Event*.

We will not go into more detail about the C++Sim setup, as it is only a support library for BSPlab and it should not be necessary to change it in order to modify BSPlab.

10.2.3 Making the BSPlab distribution library

Be aware you will not need to compile the BSPlab library to distribute on Linux, as we do not distribute binary files of BSPlab on the Linux platform. We also strongly recommend that you run through the usual BSP test programs before you distribute a new version av BSPlab. Please read section 10.2.5 to learn how to perform the testing.

To compile the BSPlab library on Windows you will have to start *MSYS* and follow these instructions:

1. Change your working directory to the *bsplab* directory (i.e. `cd /c/Program Files/BSPlab/bsplab`). Notice that the *C* drive is accessed by the path */c* in the MSYS enviroment.
2. Remove the CMake cache file *CMakeCache.txt* by running the command `rm CMakeCache.txt`. This is only necessary to do if you have been using CMake to generate Visual Studio project files, or other project files not of the "Unix Makefiles" type.
3. Generate makefiles by running the command `cmake -G"Unix Makefiles" .`
4. Type *make* to compile the BSPlab library.

The new BSPlab library should now be located in the *bsplab/lib* directory and named *libbsplab.a*.

Cleaning up the directory before distribution

From the BSPlab root directory (i.e. *C:/Program Files/BSPlab/bsplab*) run the command `./distclean` to remove all unnecessary files from the BSPlab directory and make it ready for distribution (on Windows you must remember to be in the *MSYS* environment). If you have been using build tools other than Visual Studio or MinGW, some non-essential build files may still remain and you must identify and remove them manually.

10.2.4 Customizing BSPlab for dummies

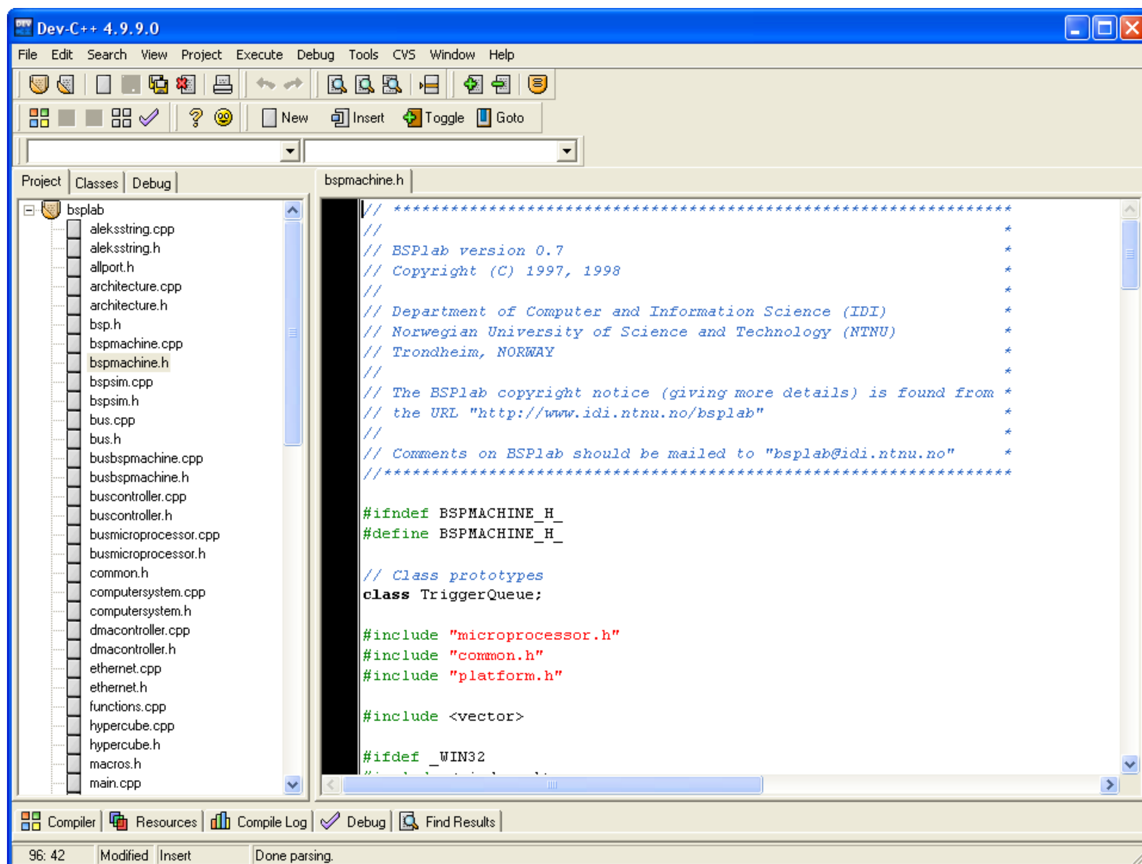
If you want to customize BSPlab just for the fun of it, this is the tutorial for you. However, if you want to publish/distribute your customized BSPlab we recommend that you also read section 10.2.1.

Opening the BSPlab project in the Dev-C++ IDE

The first thing you need to do before you can start making changes to BSPlab is to open the project in the Dev-C++ IDE. The file should be located in *C:/Program Files/BSPlab/bsplab*, and named *bsplab.dev*.

Making changes to BSPlab

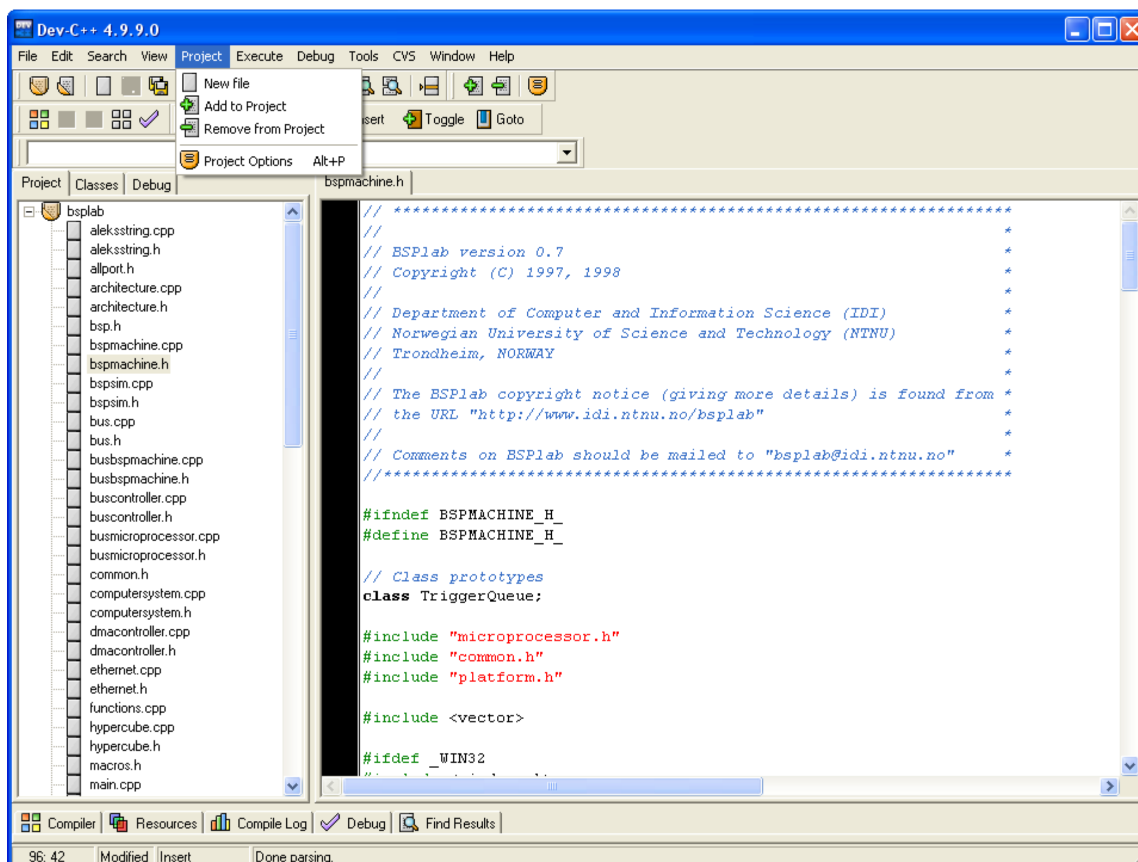
After you have opened the BSPlab project file, Dev-C++ should present you with a list of all the files in the project on the left hand side of your screen. If you want to make changes to some of the existing files, just click on the file you want to edit. The contents of the file should now appear on your screen. An overview of Dev-C++ with the BSPlab project loaded editing the *bspmachine.h* file is shown in Figure 10.2.



Figur 10.2: Dev-C++ with the BSPlab project loaded

If you want to add some of your existing code files to the BSPlab project, make sure to

place the files in the correct directory before you proceed. The BSPlab directory structure is explained in section 10.2.1 with the visual representation in Figure 10.1. After you have placed the source files in the correct directories, use the *Project/Add to Project* menu option to include the files in the BSPlab project. Figure 10.3 shows Dev-C++ with the *Project* menu option open.



Figur 10.3: Dev-C++ with the BSPlab and the Project menu open

If you are going to add a new source file (not yet written) to the BSPlab project, please use the *Project/New file* menu option shown in Figure 10.3. The editor will now present you with a blank page, press *CTRL+S* to save and give the file a name. Also remember to save the file in the correct place inside the BSPlab directory structure.

Compiling the customized BSPlab

When you have finished making your changes to BSPlab (or just want to check if what you are doing compiles), you will have to compile your changes into the BSPlab library, to

do this press *CTRL+F9*. If you have made any changes to header-files you might want to make sure these changes have propagated to all the source files as well. The best way to make sure this has happened is to recompile the whole project by pressing *CTRL+F11*.

Running your programs with the customized BSPlab

After you have made a new version of BSPlab, you might want to run some of your BSP programs in the new simulator. It is important to remember that all the BSP programs are linked statically against the BSPlab library, so you will have to relink the BSP programs to make them use the new BSPlab library. The easiest way to relink your BSP programs is by pressing *CTRL+F11* to rebuild the project when you have loaded your BSP project in Dev-C++. If your BSP project takes a lot of time to compile, a better solution would be to delete the executable and press *CTRL+F9* to just relink the file instead.

10.2.5 Running tests on the customized BSPlab simulator

Before you distribute a new version of the BSPlab simulator we recommend that you run through the standard BSPlab test programs. These programs reside in the *src/TestCode* directory inside the *bsplab* directory.

Running all the test programs

To run the test programs, follow these instructions:

1. If you are on Windows, start up the MSYS enviroment.
2. Change current path to the *TestCode* directory (i.e. *C:/Program Files/BSPlab/bsplab/src/TestCode*).
3. Execute the command `./TestAll`.

The `TestAll` script will run 14 BSP test programs through 25 different parameter files. These parameter files are set up to test the different architectures and architecture options of the BSPlab simulator. Running all these tests may take several hours.

Examining the results

When the tests have finished running, you should take some time to examine the results. In the *diffs* directory you will find files with the names *<bsp-program>.<parameter-file-id>.diff*. For example the difference file for the BSP program *MergeSort* run with parameters file number 20 will be named *MergeSort.20.diff*. The diff-files describe the difference between the output of the program and the expected output.

Sometimes the output from the program depends on the parameter options, and you will get diff-files even though the program ran perfectly. Nontheless, you should look through the diff-files for any unexcepted print outs or results.

If some of the programs crash during the testing you should get a notification on your screen. Please try to track down any new problems running the test programs before you distribute your version of BSPlab.

Kapittel 11

Konklusjon

Etter at prosjektet er ferdig sitter vi igjen med endel erfaringer og tanker om hvordan utførelsen av oppgaven vi ble gitt har gått. I dette kapittelet vil vi dele noen av våre erfaringer, tanker om videre arbeid som kan gjøres på BSPlab og vår konklusjon på hele prosjektet.

11.1 Erfaringer

Gjennom hele prosjektet har vi hatt svært god nytte av våre tidligere erfaringer med kryssplatformutvikling i C/C++. Til tross for at dette ikke var et nytt felt for noen av oss ble vi gjennom prosjektet kjent med flere nye verktøy vi vil ha nytte av senere. Noe av det viktigste for oss når det gjelder verktøy har vært å ha satt oss inn i *CMake* og *Inno Setup Compiler*. Disse programmene vil vi benytte oss av ved senere anledninger. I tillegg kom vi borti et fagfelt som var helt nytt for oss begge; parallelle modeller, spesifikt *BSP*. Ettersom mye tyder på at parallelle systemer kommer til å bli viktigere og viktigere i fremtiden, er dette erfaringer som er vel verdt å ta med seg. Både kjenskap til *BSP* og andre modeller kan bli nødvendig ved utvikling på fremtidens platformer.

11.1.1 Erfaringer med programvarepakker

Vi har benyttet oss av en rekke forskjellige verktøy for å løse oppgavene i prosjektet. Her er noen av våre erfaringer med noe av denne programvaren.

Latex

Hele rapporten er skrevet i Latex [Latex, 2005]. Vi hadde fra før god erfaring med å benytte Latex til å skrive store dokumenter og det viste seg også denne gangen å være et godt valg. Som editor benyttet vi *Kile* på Linux [Kile, 2005]. Dette er en helt utmerket Latex-editor og

møtte alle våre behov og mye mye mer. Latex er også godt egnet til versjonskontrollsystemer ettersom koden ikke inneholder noe binær data.

Subversion

Til versjonskontroll av rapporten og all BSPlab-koden benyttet vi *subversion* [Subversion, 2005]. *Subversion* er et moderne versjonskontrollsystem som møtte alle våre behov for trygg oppbevaring av dokumentet og koden.

MinGW og Dev-C++

GCC er standard kompilator på mange Unix plattformer og er godt utprøvd og testet. Det er veldig nyttig å vite at kode som er kompilert med GCC på Linux, vil kompilere uten problemer på Windows så lenge man ikke har benyttet seg av ikke-portable funksjoner. Selv om de fleste C++-kompilatorer skal være syntaktisk like, er ikke dette alltid sannheten. Vi har bare hatt positive opplevelser av MinGW i løpet av BSPlab prosjektet, bl.a. gikk flyttingen fra Linux til Windows MinGW helt smertefritt.

Dev-C++ tilbyr på langt nær så mye tilleggsfunksjonalitet som IDEen til Visual Studio, men i mange tilfeller gjelder filosofien “less is more”, og vi synes minimalistiske Dev-C++ tilbyr akkurat den funksjonaliteten vi trenger. For en person som ikke er vant med Visual Studio vil nok Dev-C++ fremstå som mye mer oversiktelig, men for veldig store programmeringsprosjekt kan nok denne IDEen komme for kort. For det mellomstore prosjektet BSPlab synes vi at Dev-C++ fungerte ypperlig til sitt bruk.

KDevelop

Mesteparten av koden ble ikke editert i *Dev-C++*, men i en editor på Linux som heter *KDevelop* [KDevelop, 2005]. Dette er det utviklingsmiljøet vi har mest erfaring med fra før, så vi valgte å skrive kode i denne editoren. *KDevelop* tilbyr det meste man trenger for å editere kode effektivt og for å kompilere og kjøre programmer og hadde alle fasiliteter vi trengte til dette prosjektet.

CMake

Vi hadde ikke brukt automake-verktøyet CMake før vi begynte med BSPlab-prosjektet. CMake fungerte utmerket som plattform-uavhengig bygge-verktøy. De konfigurajonsfilene vi opprinnelig satte opp for Linux trengte ytterst små forandringer for å fungere på Windows med både MinGW og Visual Studio. Som vi har oppsummert i prosjektet finnes det fra før få automake-verktøy som også har tatt hensyn til Windows-plattformen, dette har virkelig CMake gjort noe med. Vi kommer sikkert til å bruke CMake på passende prosjekter også i fremtiden.

Inno Setup Compiler

Vi hadde en svært positiv opplevelse av installasjonsprogram-byggeren *Inno Setup Compiler*. Dette produktet gjorde det mulig å ved hjelp av en svært enkel skript-fil bygge et installasjonsprogram for BSPlab med absolutt all den funksjonaliteten vi hadde bruk for.

11.2 Videre arbeid

Det gjenstår fortsatt mye arbeid som kan gjøres på BSPlab for å få et enda bedre produkt. Først og fremst bør BSPlab testes grundigere på større BSP-programmer og man må få luket ut de problemene som er observert her. BSPlab er skrevet for å støtte funksjonene til BSPlib 0.72 alpha. Nyeste versjon av BSPlib er 1.14 og denne har endel flere funksjoner enn 0.72 alpha. Dette medfører at for å støtte nyere BSP-programmer må BSPlab oppdateres til å støtte alle funksjonene beskrevet i standaren for BSPlib.

For å gjøre BSPlab mer brukervennlig kan det utvikles et grafisk brukergrensesnitt der man kan sette parametere og kjøre eksperimenter. BSPlab fungerer nå like bra både på Windows og Linux, men det kan tenkes at det er behov for BSPlab på andre plattformer enn dette. For eksempel på Mac. Det bør ikke være noe stort problem å få BSPlab til å fungere på *Mac OS X* og andre *Unix*-varianter ettersom det allerede fungerer på Linux.

Det er i denne oppgaven laget et lite eksempel som viser hvordan BSPlab kan benyttes fra Java. Denne muligheten kan utvides både for Java og andre programmeringsspråk. Å gjøre BSPlab tilgjengelig for andre enn C/C++-programmerere vil kunne hjelpe til med å gjøre BSPlab mer populært.

Mye kan nå gjøres for å få BSPlab tatt i bruk av brukere rundt om i verden. Nå som BSPlab ikke lenger er knyttet til en proprietær plattform bør det være enklere enn før å få brukere som har nytte av det til å laste ned BSPlab og benytte seg av det. På denne måten er det også mulig at videre utvikling av BSPlab kan skje ved at brukere som benytter seg av det videreutvikler selv og sender inn sine endringer til den som vedlikeholder BSPlab. Det finnes en rekke sider på internett som støtter å administrere prosjekter av denne typen. En av de mest brukte er *SourceForge* [SourceForge, 2005]. *SourceForge* blir benyttet av tusenvis av brukere over hele verden og kan hjelpe til med både å gjøre BSPlab tilgjengelig for interesserte brukere og til å la andre personer bidra til å vidreutvikle BSPlab.

11.3 Konklusjon

Flyttingen av BSPlab fra *Microsoft Visual Studio* over til GCC på Linux og MinGW på Windows har etter vår mening gått svært bra. Ikke bare har vi fått BSPlab til å kjøre stabilt på begge plattformene, men vi har også identifisert og fjernet feil som fantes i den originale versjonen. Nye BSPlab fungerer like godt som den originale versjonen på alle de testene som er kjørt.

Det ble kjørt gjennom en grundig test av *MergeSort* på begge plattformene for å undersøke om tidsforbruket er sammenliknbart mellom Windows og Linux. Konklusjonen ble at så lenge BSP-programmet ikke bruker for mye minne, slik at operativsystemet ikke må benytte seg av *swap*, er tidsforbruket svært likt.

Det er laget et *proof of concept* på hvordan det er mulig å integrere BSPlab med Java. Dette arbeidet kan være nyttig dersom noen senere vil integrere BSPlab fullstendig med Java. Dette eksempelet på integrasjon med et annet programmeringsspråk kan også være nyttig i integrasjon med andre språk.

Vi har laget installasjonsverktøy for både Windows og Linux som gjør det enkelt for en bruker å legge inn og ta i bruk BSPlab. På Windows er det benyttet et grafisk brukergrensenitt som er utformet mest mulig likt standard installasjonsprogrammer på Windows. På Linux er det benyttet et tekstbasert installasjonsprogram, slik at man ikke er avhengig av å ha grafiske komponenter inne for å installere BSPlab.

Det følger med en CD til rapporten som inneholder ferdigkompilerte biblioteker til bruk med MinGW og alle kodefilene til BSPlab og C++Sim. På CDen ligger installasjonsprogrammene til både Windows og Linux samt koden for Java-integrasjonen.

Tillegg A

Installasjons-CD

Med prosjektrapporten sender vi også med en CD som viser hvordan vi har tenkt at en installasjons-CD som følger med for eksempel en bok om BSPlab skal fungere. Innholdet på CDen er delt opp i fire hoveddeler:

1. Installasjonsprogram for Windows med tilhørende filer.
2. Installasjonsprogram for Linux med tilhørende filer.
3. HTML installasjonsdokumentasjon og tutorial for BSPlab.
4. Nyttige referanserfiler som ikke skal være en del av den endelige installasjons-CDen.

De påfølgende avsnittene vil beskrive hoveddelene i mer detalj.

A.1 Installasjonsprogram for Windows

I installasjonsprogrammet for Windows *bsplab-install.exe* ligger alle filer som trengs for å installere BSPlab. Det vil si at filen i tillegg til å inneholde BSPlab og C++Sim, også inneholder installasjonsprogrammene til Dev-C++, MSYS og CMake. Dette gjør at hele utviklingspakken til BSPlab kan gjøres lett nedlastbar fra nettet som en fil.

A.1.1 Automatisk oppstart av installasjonsprogrammet

CDen inneholder i tillegg til *bsplab-install.exe* filen *autostart.inf*. Denne filen gjør at installasjonsprogrammet til Windows automatisk starter når man putter installasjons-CDen i CD-ROMen. *autostart.inf* vises i listing A.1. Som vi kan se trenger man bare å spesifisere hvilken fil som skal startes bak `open=` i linje 2, og så tar operativsystemet seg av resten.

Listing A.1: autostart.inf

```
[autorun]
open=bsplab-install.exe
```

A.2 Installasjonsprogram for Linux

Installasjonsprogrammet for Linux *bsplab_install* innholder ikke selv alle filer som skal installeres. Filene *bsplab-1.0-x86-linux.tar.gz* og *cmake-2.0.5-x86-linux-files.tar.gz* i området *linux* på CDen inneholder henholdsvis BSPlab-pakken og CMake-verktøyet. For at disse programmene praktisk skal kunne legges ut på en hjemmeside bør de bli arkivert ned til en fil ved hjelp av *tar*. Det er mulig å bytte ut hvilken versjon som skal installeres av BSPlab eller CMake ved å bytte ut filene i *linux* området og oppdatere hodet til *bsplab_install* med de nye filnavnene.

A.3 HTML dokumentasjon

Vi har konvertert kapittelene *Installation Howto* og *Tutorial* i denne rapporten til HTML og lagt dem med på CDen. De er å finne i området *documentation/html* under henholdsvis *installhowto* og *tutorial*.

Dette er de mest relevante kapittelene for installasjon og bruk av BSPlab, noe som også er grunnen til at de er skrevet på engelsk. Vi har valgt å konvertere dem til HTML for at det skal være enkelt for brukeren å åpne og lese dem dersom han har mottatt CDen. HTML-versjonen vil også gjøre det lettere for de som skal administrere BSPlab-prosjektet å legge ut disse beskrivelsene på internett, og eventuelt endre de ettersom de ikke vil trenge tilgang til Latex-rapporten vår.

A.4 Referansefiler som ikke skal være en del av installasjons-CDen

Filene i området *not.to.be.included.on.install.cd* er, som navnet tilsier, ikke ment for å være med på den endelige installasjons-CDen.

A.4.1 Diplomrapporten

I underområdet *report* ligger den originale PDF-filen for rapporten vår. Vi har valgt å legge med denne filen av praktiske årsaker. PDF-filen innholder hyperreferanser som gjør det langt raskere å navigere i den elektroniske varianten enn papirvarianten. Det er også mye enklere å søke etter ord eller uttrykk i PDF-filen.

A.4.2 Skript-filen for installasjonsprogrammet

I underområdet *Inno.Setup.Compiler* ligger skript-filen som brukes for å kompilere opp installasjonsprogrammet til Windows. Ved hjelp av denne filen kan vedlikeholdere av BSPlab-prosjektet enkelt lage installasjonsprogram for nye versjoner av BSPlab.

A.4.3 Forskjellige versjoner av BSPlab

I underområdet *Older.BSPlab.versions* ligger kildekoden til det opprinnelige BSPlab, samt en del av overgangsversjonene våre. Når vi rensket opp i BSPlab-koden kjørte vi tester på koden for hver milepæl for å se om vi hadde ødelagt noe av funksjonaliteten. Vi ønsker å ta vare på disse milepælene slik at videre vedlikeholdere av BSPlab skal kunne gå tilbake å se om et nylig oppdaget problem i BSPlab skyldes flyttingen, og eventuelt i hvilken milepæl dette problemet har oppstått.

En kort forklaring til hver milepæl er lagt med i filen *README*, men følger for ordens skyld også med her:

```
Beskrivelse av zip-arkivene (arkivene vises i kronologisk rekkefølge):

originale_bsplab.zip:
Inneholder den umodifiserte koden fra det originale BSPlab-prosjektet.

bsplab_vs_.net.zip:
Inneholder den første versjonen vi fikk til å kompilere på VS.Net.

bsplab_fjernet_boolean.zip:
Inneholder et mellomstadie der vi hadde fjernet Boolean
og std::Bool typene.

bsplab_fikset_stl:
Inneholder et mellomstadie der vi har fjernet bruk av
3. parts STL-implementasjonen.

bsplab_endelig_versjon_.net.zip:
Inneholder den siste versjonen av BSPlab vi hadde som kun
var for VS.Net
```

Tillegg B

Kjente problemer med BSPlab

Etter at prosjektet er ferdig sitter vi igjen med endel problemer med BSPlab som er observert underveis, men som av forskjellige grunner ikke har blitt fikset. I dette kapittelet vil vi liste opp disse med noen kommentarer på hver. Dette gjøres for å gjøre det enklere for andre som vil videreutvikle BSPlab å finne problemer å ta tak i og som en referanseliste dersom det oppdages en feil i BSPlab og man lurer på om denne er dokumentert tidligere.

Minnelekasje i posix_thread.cc

På grunn av måten C++Sim er bygd opp er det ikke mulig å slette *Entity*-objekter med det samme tråden i objektet termineres. Det er i *pthread* heller ingen funksjon tilsvarende `WaitForSingleObject` (som benyttes i *nt_thread.cc*) for å finne ut når tråden virkelig har terminert. Dersom objektet slettes før tråden har terminert krasjer BSPlab. Løsningen ble å ikke slette *Entity*-objekter på Linux der *pthread* benyttes. Dette fører til en minnelekasje som muligens kan få konsekvenser ved kjøring av store simuleringer.

Enkelte BSPlib-funksjoner er ikke implementert

BSPlab er implementert til å støtte BSPlib versjon 0.72 alpha. Nyeste versjon er 1.14 og denne inneholder en del flere funksjoner enn 0.72 alpha. Dette fører til at BSPlab ikke støtter å simulere BSP-programmer som benytter seg av noen av disse nye funksjonene.

Ustabiliteter på NOW-arkitekturen

Det er observert at BSPlab enkelte ganger stopper under kjøring av *MergeSort*-programmet på NOW-arkitekturen med mer enn 64 prosessorer. BSPlab stopper med en *assertion* og det er ikke kjent hva dette skyldes. Problemet er kun observert på Linux.

Tillegg C

Testkode

I dette appendikset finnes koden til *MergeSort* som ble brukt i forbindelse med testingen av tisforbruket på Windows og Linux. I tillegg finnes parameterfilen som ble brukt under kjøringen av alle simulasjonene under denne testen.

C.1 MergeSort

Listing C.1 viser programmet vi benyttet til å sammenlikne tidsforbruket mellom Windows og Linux. Programmet sorterer et gitt antall tall ved å benytte et valgt antall prosessorer. Både antall tall som skal sorteres og antall prosessorer som skal benyttes tas inn når programmet starter. Vi fikk best resultat når vi lot antallet prosessorer være en potens av 2 og antallet tall som skal sorteres være delelig på antall prosessorer.

Listing C.1: MergeSort-programmet benyttet ved testingen av BSPlab

```
/*=====================================================
PROGRAM        MERGE SORT
Written by     D. Kim     (dkim@home.korea.ac.kr)
Date            July 30, 1996
Description:

This is a merge sort program using BSPlib of Oxford
Parallel. The number of data to be sorted is given
by SIZE, and the number of processors used is
decalared by NPROC (currently #PROC=4, #data=32).
The data array is splited into the number of
processors with equal number of data. The partitioned
array each is sorted sequentially by each assigned
processor and then merged in log (#PROCS) steps.
Superstep synchronization is made at the merging process.
======================================================*/
```

```
#ifndef BSP_H_
#include <bsp.h>
#endif

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <iostream>

using namespace std;

unsigned int SIZE;
unsigned int NPROC;


void bsp_msort(int *my_array,
               int *your_array,
               int n,
               int *out_array)
{
  int i;
  int my_ptr, your_ptr;

  for (i=0, my_ptr=0, your_ptr = 0; i<2*n ; i++)
  {
    if ((my_ptr <n)&&(your_ptr<n))
      if (my_array[my_ptr] < your_array[your_ptr])
        out_array[i] = my_array[my_ptr++];
    else
      out_array[i] = your_array[your_ptr++];
    else if (my_ptr ==n)
      out_array[i] = your_array[your_ptr++];
    else if (your_ptr ==n)
      out_array[i] = my_array[my_ptr++];
    else printf("Something wrong in comparison\n ");
  }
}


void bsp_exchg_sort(int *array, int curr_size)
{
  int i, curr_bound, j, jj;
  int *result= (int*) calloc(curr_size,sizeof(int));
  int *your_data = (int*) calloc(curr_size,sizeof(int));
  int *mine = (int*) calloc(curr_size,sizeof(int));
  bsp_pushregister(your_data,curr_size*sizeof(int));
  bsp_sync();

  curr_bound = curr_size/NPROC;
```

```
  for (j=0; j< curr_size; j++)
    if (bsp_pid() == j /(curr_size/NPROC))
      mine[j%(curr_size/NPROC)] = array[j];

  for(i=1;i<NPROC ;i*=2)
  {
    for (j=0;j < NPROC; j+=2*i)
    {
      if ((bsp_pid()- i ) % (2*i) == 0)
      {
        bsp_put(bsp_pid() -i,
                mine,
                your_data,
                0,
                curr_bound*sizeof(int));
      }
      bsp_sync();

      for (j=0;j < NPROC; j+=2*i)
        if (bsp_pid() == j) {
          bsp_msort(mine, your_data, curr_bound,result);
        }
        bsp_sync();
    }
    for (j=0;j < NPROC; j+=2*i) {
      if (bsp_pid() == j) {
        for (jj=0; jj < 2* curr_bound; jj++) {
          mine[jj] = result[jj];
        }
      }
    }

    curr_bound *= 2;
    bsp_sync();

  }
  bsp_popregister(your_data);
  /* NOW COPY THE SORTED ARRAY TO MAIN ROUTINE */
    if (bsp_pid() == 0)
      for (jj=0; jj < curr_size; jj++)
        array[jj] = result[jj];
}


void print_array(int *array, int n)
{
  int i,j;
  for (i=0; i < NPROC; i++){
```

```
    if ( i== bsp_pid()){
      printf("\n INIT. DATA AT PID_%d = \n", bsp_pid());
      for(j=0;j<n;j++) printf("%d ",array[j]);
      printf("\n");
      fflush(stdout);
    }
  }
}


void init_sort(int *dat,int m, int pid)
{
  int i,j,amin, index,tmp;

  for (i=0; i< m; i++)
  {
    amin = dat[i];
    tmp =dat[i];
    index =i;
    for (j=i; j< m; j++)
      if (dat[j] < amin)
      {
        index=j;
        amin = dat[j];
      }
    dat[i] = amin;
    dat[index] = tmp;
  }
}


void copy_array(int *array, int *result, int n, int pid)
{
  int i;
  int start = pid * (SIZE/NPROC);
  for(i=0; i< n/NPROC; i++)
    array[start+i] = result[i];
}


void init(int *array, int n)
{
  int start;
  int i,j;
  int *result= (int*) calloc(n, sizeof(int));
  int *my_array= (int*) calloc(SIZE/NPROC, sizeof(int));
  for (i=0; i< NPROC; i++)
    if (bsp_pid()==i)
    {
```

```
      start = i * (SIZE/NPROC);
      for (j=0; j<SIZE/NPROC ; j++)
        my_array[j] = array[start +j];
      init_sort(my_array, SIZE/NPROC, i);
      copy_array(array, my_array,n,i);
   }
}

void do_sort()
{
  int i,j,n,*xs;

  double before, TimeUsed;

  bsp_begin(NPROC);

  before = bsp_time();

  n=SIZE;
  xs = (int*)calloc(n,sizeof(int));

  for (i=0;i<n;i++) {
    if ( i%2 ) {
      xs[i]= i % n;
    }
    else {
      xs[i] = (n-i) % n;
    }
  }
  init(xs,n); /* Sequentially sort and placement
                 to each proc before merging */

  bsp_end();
  if(bsp_pid() == 0) {
    TimeUsed = bsp_time() - before;
    cerr << "TIME USED: " << TimeUsed << endl;
  }
}


void bsp_main(int argc, char **argv)
{
  bsp_init(do_sort, argc, argv);

  printf("How many processors do you want to use?\n");
  fflush(stdout);
  scanf("%d",&NPROC);
  if (NPROC > bsp_nprocs()){
    printf("Sorry, not enough processors available.\n");
```

```
    fflush(stdout);
    exit(1);
  }

  printf("How many numbers would you like to sort?\n");
  fflush(stdout);
  scanf("%d",&SIZE);

  do_sort();
}
```

C.2 Parameterfilen

Listing C.2 viser parameterfilen som ble benyttet i seksjon 7.2.2 ved testing av BSPlab. Den eneste parameteren som ble endre i denne filen var *[BSPMachine]* for å endre arkitektur på simuleringene.

Listing C.2: Parameterfilen benyttet ved testingen av BSPlab

```
/////////////////////// General parameters /////////////////
        // Network, NOW, Simple, Null or Bus
[BSPMachine] = Null
[Warnings] = On

//Print a dot after each super step?
[ProgressIndicator] = Off

//The file where logging is directed
[Probe_OutputFilename] = ProbeResults.txt

//Maximal file size for log file
[Probe_OutputFileMaxSize] = 1000000

//Automatic timing of the code?
[AutomaticTiming] = On

//Manual timing of the BSP code?
[ManualTiming] = Off

//Automatically benchmark the CPU ??
[AutomaticCPUTiming] = Off

//The simulator machine CPU speed compared
//with a 100 MHz 486
//Not used when AutomaticCPUTiming is true
```

```
[ThisCPUSpeed] = 1.0

//The speed of the simulated CPU
//(1.0 is equal to ThisCPUSpeed
//or the benchmarked value)
[VirtualCPUSpeed] = 1.0

//The stack size of each prosessor (thread).
//0 means to use the default stack size of
//the operating system
[StackSize] = 307200

// Use real time priority? (1 = true, 0 = false)
[RealTime] = 1

// Used to set the overhead sizes
[BytesInGetMessage] = 8
[BSMPMessageMarkSize] = 2
[BytesInSyncMessage] = 8
[TimeToStartPacketizing] = 3.2e-6
[TimeToPacketizeOneByte] = 4.0e-8

////////////////// Logging parameters /////////////////

[LogSuperstepSentData] = False
[LogSuperstepReceivedData] = False
[LogSuperstepTimeSpent] = False
[LogBarrierSyncTimeSpent] = False
[LogSuperstepDRMASent] = False
[LogSuperstepDRMAReceived] = False
[LogSuperstepBSMPSent] = False
[LogSuperstepBSMPReceived] = False

[LogTotalSuperstepSentData] = False
[LogTotalSuperstepReceivedData] = False
[LogAvgSuperstepTimeSpent] = False
[LogAvgBarrierSyncTimeSpent] = False
[LogTotalSuperstepDRMASent] = False
[LogTotalSuperstepBSMPSent] = False
[LogTotalSuperstepBSMPReceived] = False

[LogTotalSentData] = False
[LogTotalReceivedData] = False
[LogTotalTimeSpent] = False
[LogTotalBarrierSyncTimeSpent] = False
[LogTotalDRMASent] = False
[LogTotalBSMPSent] = False
[LogTotalBSMPReceived] = False
```

```
// Parameters for both null, bus, simple and NOW machine //
[NumberOfProcessors] = 1000

////// Parameters for the Simple BSP machine /////////////
[Simple_TimeToSynchronizeOneProcessor] = 0.001
[Simple_TimeToSendOneByte] = 0.00000008

//////// Parameters for the network machine //////////////
//Network topology:
//TwoDTorus, TwoDMesh, TwoDMeshHP, ThreeDMesh, (HyperCube)
[Network_Topology] = TwoDMesh

//Network size

//Used for TwoDTorus, TwoDMesh, ThreeDMesh
[Network_Xsize] = 16
[Network_Ysize] = 16

//Used for ThreeDMesh
[Network_Zsize] = 8
//Used for HyperCube and TwoDMeshHP
[Network_Dim] = 6
//Communication port architecture: OnePort, AllPort
[Network_PortModel] = AllPort

//Message passing: Hazard, (Safe)
[Network_Stability] = Safe
//Transission time of a link in the network
[Network_TransTime] = 0.000001
//Start up latency when sending a message
[Network_StartUpLat] = 0.00002

//Outport priority for a microprocessor: (Fifo), Random
[Network_OutPortPriority] = Fifo

//Routing methods: TwoPhaseWorm, (Worm), StoreAndForward
[Network_RoutingMethod] = TwoPhaseWorm

//Delay messages to barrier synchronization: On, (Off)
[Delayed_messages] = Off

// Find an intermediate node (in two phase routing) from a
// "square" with the to and from nodes as corners, or from
// all nodes ?
[Intermediate_node_from_square] = False

// Maximal aize for messages sent over the network (both
// store and forward, and wormhole routing)
[NetworkMaxPacketSize] = 16384
```

```
// The grain of the simulation.  If set to one, the load
// of each link is calculated for each byte in a worm, if
// set to 2, the load is calculated for every two bytes,
// and so on
[Network_VirtualChannel_Simulation_GrainSize] = 1


/////////// Parameters for both NOW and the Bus /////////

// The fan in of the reduction tree used in barrier
// synchronizations
[Bus_BarrierTreeFanIn] = 4

// The fan out of the broadcast tree
[Bus_BarrierTreeFanOut] = 2

//// Parameters for the bus used in the Bus machine //////
// Width of bus in bits
[ProcessorBus_Width] = 32
// Frequency of bus in Hz
[ProcessorBus_Frequency] = 33000000.0
// Set to one cycle (ca 30 ns)
[ProcessorBus_ArbitrationGap] = 0.00000001

////////// Parameters for the NOW machine ////////////////
// Activate noise (external traffic) on the Ethernet
// cable ??
[NOW_NoiseActivated] = False

// Mean time between noise
[NOW_TimeBetweenNoise] = 1.0

/////////// Parameters for the Ethernet /////////////////
// The number of bytes of overhead for each Ethernet
// message
[Ethernet_PacketOverhead] = 26

// The minimal and maximal message length (without
// the overhead)
[Ethernet_MinDataSize] = 46
[Ethernet_MaxDataSize] = 1500

// Maximal number of retries for a message, before fail.
// When the Ethernet fails, a warning is written, and
// the message is "reset" (try to send it as it was new)
[Ethernet_MaxRetries] = 16

// The maximal backoff value used in the backoff algorithm.
```

```
[Ethernet_MaxBackoff] = 10

// The bandwidth in Bits / sec
[Ethernet_BandWidth] = 10.0e6

// The interfram gap of the Ethernet
[Ethernet_InterFrameGap] = 9.6e-6

// The propagation delay of the Ethernet cable
[Ethernet_PropagationDelay] = 22.5e-6

// The slot time used in the the backoff algorithm.
// Must be more than twice the prop. delay !!
[Ethernet_SlotTime] = 51.2e-6

// The length of the jam of the stations.
[Ethernet_JamDelay] = 3.2e-6
```

Bibliografi

[4DOS, 2005] 4DOS (2005). http://www.4dos.info/. Sist sett: 14/06-05.

[ActivePerl, 2005] ActivePerl (2005). http://www.activestate.com/Products/ActivePerl/?mp=1. Sist sett: 25/05-05.

[Apache HTTP Server, 2005] Apache HTTP Server (2005). http://httpd.apache.org/. Sist sett: 01/06-05.

[autopackage, 2005] autopackage (2005). http://autopackage.org/. Sist sett: 13/05-05.

[BASH, 2005] BASH (2005). http://cnswww.cns.cwru.edu/~chet/bash/bashtop.html. Sist sett: 01/04-05.

[Bisseling, 2004] Bisseling, R. H. (2004). *Parallel Scientific Computation*. Oxford University Press.

[Bitrock Installbuilder, 2005] Bitrock Installbuilder (2005). http://www.bitrock.com/products_installbuilder_overview.html. Sist sett: 20/05-05.

[BSP Worldwide, 2005] BSP Worldwide (2005). Bsp worldwide. http://www.bsp-worldwide.org. Sist sett: 18/05-05.

[BSPlib, 2005] BSPlib (2005). Bsplib. http://www.bsp-worldwide.org/implmnts/oxtool.htm. Sist sett: 14/06-05.

[CMake, 2005] CMake (2005). http://www.cmake.org. Sist sett: 01/04-05.

[CreateThread, 2005] CreateThread (2005). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createthread.asp. Sist sett: 06/06-05.

[C++SIM, 1997] C++SIM (1997). C++sim. http://cxxsim.ncl.ac.uk/. Sist sett: 17/03-05.

[Cygwin, 2005] Cygwin (2005). http://www.cygwin.com/. Sist sett: 17/03-05.

[declspec, 2005] declspec (2005). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccelng/htm/msmod_17.asp. Sist sett: 07/06-05.

[Dev-C++, 2005] Dev-C++ (2005). http://www.bloodshed.net/devcpp.html/. Sist sett: 17/03-05.

[dlopen, 2005] dlopen (2005). http://www.opengroup.org/onlinepubs/009695399/functions/dlopen.html. Sist sett: 25/05-05.

[Flanagan, 1996] Flanagan, D. (1996). *Java Examples in a Nutshell*, chapter 2. O'Reilly, 1st edition.

[Freshmeat, 2005] Freshmeat (2005). http://freshmeat.net. Sist sett: 25/05-05.

[Gentoo, 2005] Gentoo (2005). http://www.gentoo.org/. Sist sett: 11/06-05.

[GNOME, 2005] GNOME (2005). http://www.gnome.org/. Sist sett: 01/06-05.

[GNU Autoconf, 2005] GNU Autoconf (2005). http://www.gnu.org/software/autoconf/. Sist sett: 25/05-05.

[GNU Automake, 2005] GNU Automake (2005). http://www.gnu.org/software/automake/. Sist sett: 25/05-05.

[GNU Libtool, 2005] GNU Libtool (2005). http://www.gnu.org/software/libtool/. Sist sett: 25/05-05.

[Gorman, 2004] Gorman, M. (2004). *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, 1st edition.

[GPL, 2005] GPL (2005). GNU General Public License. http://www.gnu.org/copyleft/gpl.html. Sist sett: 06/06-05.

[Hundhammer, 2005] Hundhammer, S. (2005). http://lists.suse.com/archive/suse-programming-e/2003-Apr/0090.html. Sist sett: 30/05-05.

[Inno Setup Compiler, 2005] Inno Setup Compiler (2005). http://www.jrsoftware.org/isinfo.php. Sist sett: 26/05-05.

[JNI, 2005] JNI (2005). Java Native Interface. http://java.sun.com/j2se/1.4.2/docs/guide/jni/. Sist sett: 18/05-05.

[KDE, 2005] KDE (2005). K Desktop Enviroment. http://www.kde.org/. Sist sett: 01/06-05.

[KDevelop, 2005] KDevelop (2005). http://www.kdevelop.org. Sist sett: 25/05-05.

[Kile, 2005] Kile (2005). http://kile.sourceforge.net/. Sist sett: 07/06-05.

[Latex, 2005] Latex (2005). http://www.latex-project.org/. Sist sett: 07/06-05.

[Liang, 1999] Liang, S. (1999). *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1st edition.

[Lilleaas, 1998] Lilleaas, E. (1998). Evaluation of bsplab. Technical report, NTNU. Fordypningsrapport.

[LoadLibrary, 2005] LoadLibrary (2005). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50lrfloadlibrary.asp. Sist sett: 25/05-05.

[lokisetup, 2005] lokisetup (2005). http://www.icculus.org/loki_setup/. Sist sett: 15/05-05.

[McCall, 2003] McCall, A. (2003). Stop the autoconf insanity! why we need a new build system. http://freshmeat.net/articles/view/889/. Sist sett: 25/05-05.

[McColl, 1990] McColl, W. F. (1990). *Scalable Computing.* Oxford University Computing Laboratory.

[Meyers, 1997] Meyers, S. (1997). *Effective C++.* Addison-Wesley Professional, 2nd edition.

[MinGW, 2005] MinGW (2005). Minimalist GNU for Windows. http://www.mingw.org/. Sist sett: 17/03-05.

[MSDN, 2005] MSDN (2005). Microsoft software developer network. http://www.msdn.com. Sist sett: 18/05-05.

[MSYS, 2005] MSYS (2005). MinGW - Minimal SYStem. http://www.mingw.org/msys.shtml. Sist sett: 25/05-05.

[OpenSSL, 2005] OpenSSL (2005). http://www.openssl.org/. Sist sett: 12/06-05.

[Qt, 2005] Qt (2005). http://www.trolltech.com/products/qt/. Sist sett: 15/05-05.

[RPM, 2005] RPM (2005). Redhat Package Manager. http://www.rpm.org. Sist sett: 15/05-05.

[SourceForge, 2005] SourceForge (2005). http://sourceforge.net. Sist sett: 25/05-05.

[Subversion, 2005] Subversion (2005). http://subversion.tigris.org/. Sist sett: 07/06-05.

[Sund, 2004] Sund, E. Å. (2004). Bsplab to the people. Technical report, NTNU. Fordypningsrapport.

[Unixhelp, 2005] Unixhelp (2005). http://unixhelp.ed.ac.uk/CGI/man-cgi?sched_setscheduler+2. Sist sett: 11/06-05.

[Uthus and Dybdahl, 1997] Uthus, I. and Dybdahl, H. (1997). Simulation of the bsp model on different computer architectures. Master's thesis, NTNU.

[Valiant, 1990] Valiant, L. G. (1990). *A Bridging Model for Parallel Computation, Communications of the ACM, Vol. 33, No. 8.*

[Visual Studio Installer, 2005] Visual Studio Installer (2005). `http://msdn.microsoft.com/vstudio/downloads/tools/vsi11/default.aspx`. Sist sett: 26/05-05.

[Webopedia, 2005] Webopedia (2005). `http://www.webopedia.com/TERM/p/polymorphism.html`. Sist sett: 18/05-05.

[Win32 Thread Stack, 2005] Win32 Thread Stack (2005). `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/thread_stack_size.asp`. Sist sett: 06/06-05.