Multiprogrammering en de X8. (Vervolg van EWD54)

3.3. De functie van de luisterbits.

Een ingreep (zie 2.3., EWD51 - 7) vindt slechts plaats als
a) de machine horend is
b) het collatieresultaat van ingreepwoord(en) en (overeenkomstige) luisterwoord(en) $\neq 0$ is.

Door een luisterbit = 0 te maken, kan men dus bereiken, dat de bijbehorende seinpaal, als de laatste allang positief is en de bijbehorende ingreepflipflop allang = 1 gezet is, tot nader order toch geen ingreep teweegbrengt.

Men is kennelijk niet geinteresseerd in het positief worden van een seinpaal, als er niets op staat te wachten. Speciaal bij een hardware seinpaal, waarvan het positief zijn in principe een ingreep teweegbrengt, is het begrotelijk, als deze ingreep plaats vindt en de coordinator niets anders doen kan dan gedesinteresseerd van dit feit nota nemen, gedesinteresseerd, omdat het aanbhet hele mogelijkheidspatroon niets verandert. Een ingreep kost nl. wel tijd (en we zullen aanhoudend alert moeten blijven om deze tijd niet te veel te laten toenamen).

Het idee is nu, om de coordinator ervoor te laten zorgen, dat de luisterbit van een hardware seinpaal aangeeft, of er op deze seinpaal iets staat te wachten of niet: als de wachtketen, die aan de hardware seinpaal hangt (zie 3.2.,EWD54 - 8 e.v.), leeg is, zal de bijbehorende luisterbit = 0 zijn, anders = 1. Het is de taak van de tandenpoetserij om initieel hiervoor te zorgen, het is de taak van de coordinator om dit steeds bij te houden.

Het gebruik van de luisterbit is hiermee duidelijk. Hoe effectief het is, kan ik op geen stukken na bekijken. Algemeen geldt, dat het effectiever zal zijn
a) als abstracte machines zo mogelijk liever aan non-hardware seinpalen gehangen worden
b) als de cumulatie van ingrepen hoger kan zijn. Dit laatste houdt verband met de omvang van het startmagazijn.

We moeten hier in elk geval naar kijken en wel om twee redenen: ten eerste geeft dit "letten op" waarschijnlijk nauwelijks aanleiding tot kostenverhoging -in hardware, tijd of aantal opdrachten van de coordinator-; ten tweede moeten we ons bewust zijn, dat het uitsparen van onnodige ingrepen hier zou werken, als de computer niet op de transputapparaten zou hoeven te wachten, dwz. als de hele fabriek "processor limited" zou zijn, dwz. juist de omstandigheid, dat het alle zin heeft, om zuinig met computertijd om te springen!

Een laatste opmerking is, dat de tijdsspendering per ingreep een betrekkelijk vast bedrag is: dit gaat procentueel des te zwaarder tellen, naarmate je randapparatuur frequenter de V-operatie uitvoert. (We moeten niet vervallen in de fout, die ik bij de X1 gemaakt heb, waar het ingreepprogramma voor de bandlezer afgestemd was op de 150 char./sec van de Ferranti-bandlezer -onzaliger nagedachtenis!-, met het droeve resultaat, dat het de EL1000 vanwege zijn nu niet meer negligeabele reactietijd niet meer bij kon sloffen!)

4. Automatische transporten tussen langzaam en snel geheugen.

4.0. Inleiding.

Een ander niet onbelangrijk probleem, nl. dat van de toevoeging of afvoering van abstracte machines laat ik voorlopig met opzet liggen, en wel om de volgende redenen. Binnen afzienbare tijd hoop ik de documentatie onder ogen te krijgen van hoe andere systemen, waarin dit probleem in een zekere mate van algemeenheid is op-

gelost. Daarin is ingetwijfeld een oplossing gevonden voor de toekenning van randapparatuur aan programma's; behalve concrete objecten zullen wij ook abstracte objecten -nieuw in te voeren seinpalen, bv.- moeten toekennen. Door de grotere flexibiliteit, waar we op mikken, zal het voor ons wat minder eenvoudig liggen, maar het zou onverstandig zijn om te schromen, van andermans ervaring te profiteren. De tweede reden is, dat de administratieve plichten bij toevoeging van nieuwe abstracte machines wel eens medebepaald kon worden door de structuur van deze machines. Deze laatste kon wel eens drastisch beinvloed worden door de manier, waarop we de automatische transporten tussen langzaam en snel geheugen denken te regelen. Dit laatste is dus fundamenteler -en ook moeilijker; vandaar dat ik dat nu eerst te lijf wil. Ter inleiding een historisch overzichttvan drie relevante stappen.

4.1. De ARMAC

Ik herinner me nog heel goed, hoe we wild enthousiast waren, toen we het ontwerp van de ARMAC klaar hadden. In zijn laatste dagen, met de X1 aan de andere kant van de gang, hebben we wel eens medelijdend de schouders over het oude beestje opgehaald. In retrospectie was dit enthousiasme uit 1955 toch niet zo ongerechtvaardigd!

In 't kort kwam de organisatie van de ARMAC, die in wezen een trommelmachine was, op het volgende neer. De trommel was ingedeeld in sporen van 32 woorden per spoor. Opdrachtselectie direct van de trommel was echter onmogelijk: de machine was uitgerust met een buffer op kernen met de capaciteit van 1 spoor en opdrachtselectie bestond uit
1) verificatie, dat het goede spoor in de buffer gecopieerd stond
2) selectie van de overeenkomstige plaats in de buffer.

Als aan de eerst voorwaarde niet voldaan was, werd het rekenproces zonder pardon 1 trommelomwenteling opgehouden, waarin de buffer met het nieuwe spoor gevuld werd. Om dit alles mogelijk te maken, waren 7 additionele flipflops ingevoerd (de trommel bevatte 128 sporen, de adreslengte was 7 + 5 = 12 bits) waarin het nummer van het trommelspoor onthouden werd, dat in de buffer gecopieerd stond. Bij elke opdrachtselectie werd getest op gelijkheid van deze 7 bits en de 7 meest significante bits van de opdrachtteller.

Als additionele feature werd bij elke getalselectie ditzelfde 7-tal vergeleken met de 7 meest significante adresbits in het opdrachtregister: in geval van gelijkheid werd bij lezen niet gewacht, tot de aangewezen trommelplaats onder de koppen doordraaide, maar werd meteen de overeenkomstige plaats uit de buffer geselecteerd; bij schrijven werd dan wel op de trommel geschreven, maar tevens werd de kopie in de buffer bijgehouden.

Deze additionele hardware heeft zijn geld meer dan opgebracht. Let wel, dat de comparatie met de 7 bits, die aangaven, welk spoor in de buffer stond, in elke gewone opdracht twee keer werd uitgevoerd: eerst in de opdrachtcyclus, daarna in de getalcyclus. In een conventionele von Neumann machine kon je ook moeilijk anders.

Op de keper beschouwd waren het merendeel van deze comparaties overbodig: logisch was het toelaatbaar -en dat was toen in zekere zin geavanceerd- om de ARMAC als boven beschreven non-track-bewust te bedrijven. In de praktijk liet je dit wel uit je hoofd: je besteedde dagen om net zo lang aan een subroutine te fiegelen, totdat je hem in 32 woorden geperst had! En alle hoofdprogramma's hadden op den duur ook alle constanten in het spoor, waar ze gebruikt werden; je moest wel

Gaan we er nu van uit, dat de machine track-bewust geprogrammeerd wordt, dan komen we tot de verrassende ontdekking, dat alle comparaties met de 7 bits weggepraat kunnen worden en daarmee de 7 flipflops zelf.

Het enige wat je nodig hebt zijn speciale sprong- lees- en schrijfopdrachten van en naar plaats zoveel van de buffer. De gewone sprongopdrachten kan je dan altijd hernieuwde buffervulling laten impliceren. De enige vraag is dan nog, hoe je van het ene spoor naar het andere overloopt. Je had kunnen detecteren op overloop naar de 7de bit van de opdrachtteller bij deszelfs ophoging; een alternatieve oplossing is om expliciet aan het einde van een spoor naar het begin van het volgende spoor te springen -iets wat we in de praktijk haast altijd deden, omdat je toch over een paar constanten heen moesten springen.

Op deze manier bekeken is de automatische 7-bits comparatie dus eigenlijk, hoewel mooi, volslagen overbodig: de programmeur kon weten hoe de uitslag zou zijn (en hij deed er goed aan dit te weten).

4.2. ATLAS.

Een volledig beeld van de ATLAS heb ik niet; toen ik het wilde weten, kon ik de documentatie niet te pakken krijgen. Wat mij ervan bijgebleven is, is dat elk programma zijn eigen sporen op de trommel via een of andere techniek van relatieve adressering kan blijven nummeren van 0,1,2,3,.....; tijdens executie zijn daar physische sporen aantoegevoegd. Hoe precies, weet ik niet; het is -althans nu- voor ons ook niet belangrijk.

Belangrijk is, dat de ATLAS in mijn geheugen beklijfd is als een ARMAC met een aantal buffers. (Ik geloof 32 of 64 als je rijk was.) Voor de programmeur van ieder individueel programma is niet toegankelijk, welke sporen zich op elk moment in de buffers bevinden. Hij kan vrijelijk als in een echte von Neumann machine naar elk woord in zijn programma (opdrachten + werkruimtes) refereren. Hardware detecteert of het geselecteerde spoor in een van de buffers staat; zoja, dan wordt het overeenkomstige woord van die buffer genomen, zo nee, dan wordt een van de buffers vrijgemaakt en wordt daarin het gevraagde spoor gecopieerd, waarna deze berekening door kan gaan. Dit alles in hardware uitgevoerd.

Dit project is in meer dan 1 opzicht angstig. In de eerste plaats moeten de comparaties met het ene gevraagde en de 32 (of 64) aanwezige sporen simultaan uitgevoerd worden en de hiervoor benodigde hardware is niet kinderachtig. Het moet allemaal rap gebeuren, want anders compareer je het grootste gedeelte van de tijd. In de tweede plaats -hoewel dit geloof ik niet inhaerent aan het systeem is- is de strategie volgens welke een buffer vrijgemaakt wordt, ook in hardware vastgelegd. Bij de ARMAC hadden we dit strategieprobleem niet: er was maar 1 buffer en een keuze van 1 uit 1 geeft weinig vrijheid.

4.3. GIER ALGOL.

De GIER is een machine met een kerngeheugen van zeg 100 woorden en een 12 keer zo grote trommel met sporen van ongeveer 40 woorden (Precies weet ik het niet, het is in elk geval een raar getal.)

De implementatie van ALGOL op deze machine is voor wat de kerngeheugentoewijzing voor tussenresultaten geent op de X1-implementatie, voor opdrachten op de ATLAS, met dat verschil, dat het objectprogramma (uiteindelijk) wel trackbewust geformuleerd wordt.

Karakteristieke eigenschappen van de GIER zijn in dit verband dat het kerngeheugen echt wel klein is, dat de machine snel is en tijdens tracktransporten zelf wel door kan werken. De machine heeft geen interrupt features, multiprogrammering komt niet ter sprake.

De oplossing van Peter Naur, Jørn Jensen c.s. is in wezen als volgt.

Een zo klein mogelijk hoekje van het kerngeheugen is gereserveerd voor het "run time system", dwz. de administratie (programma + gegevens) welke programmasporen in het kerngeheugen staan en waar. Er is alles aan gedaan om dit gedeelte zo compact mogelijk te houden, iets wat je aan een begenadigd codeur als Jensen gerust kunt overlaten.

De rest van het kerngeheugen is nu beschikbaar voor variabelen en programma. De programmeur heeft de plicht om te zorgen, dat het aantal variabelen de 500 tot 700 niet overschrijdt. Als hij meer variabelen bespelen wil, kan hij dit, mits hij expliciet -via uit de bibliotheek beschikbare standaardprocedures- variabelen expliciet van en naar de trommel transporteert. De variabelen in het kerngeheugen worden gestapeld, de rest is -buiten controle van de programmeur maar onder controle van het run time system- ter beschikking voor copieen van programmasporen. Gedurende die periodes, dat het aantal variabelen klein is, is er meer ruimte in het kerngeheugen voor programma en wordt dus minder tijd aan tracktransporten verspeeld. (Een gedeelte van die tijd maakt hij ten nutte door een gecompliceerde administratie bij te werken.)

De werkwijze in het GIER ALGOL system lost voor ons niet alles op: het houdt geen rekening met multiprogrammering en legt de transportlast voor variabelen nog op de schouders van de programmeurs.

Voorts moeten we een vrij wezenlijk verschil niet uit het oog verliezen: bij de GIER moest men er van uitgaan, dat als regel het kerngeheugen te klein zou zijn om opdrachten en variabelen van een programma alle te bevatten; bij de X8 mogen we er dunkt me van uitgaan, dat het kerngeheugen in de regel daarvoor wel grootgenoeg zal zijn.

4.4. Paginaindeling.

Het idee is dus om de informatie op de trommel in "pagina's" in te delen en paginaasgewijs al of niet in het kerngeheugen te laten zitten. Wil dit aanleiding geven tot een redelijk vlotte afwerking, dan moet aan minstens drie voorwaarden voldaan zijn.

a) Zolang de benodigde informatie wel een plaats in het kerngeheugen heeft, moet er zo min mogelijk tijd verloren gaan door verificatie hiervan.
b) De informatie moet volgens een dynamisch zinvol criterium over de pagina's verdeeld zijn.
c) De strategie volgens welke "oude pagina's" uit het kerngeheugen verwijderd worden, moet door een of andere (dynamische) bespiegeling waarschijnlijk gemaakt kunnen worden.

4.4.1. De verificatie. (ad 4.4. sub a).

Bij de ARMAC geschiedde de verificatie door weinig, bij de ATLAS door veel hardware, die simultaan met het rekenproces in actie is. Dit in tegenstelling tot GIER ALGOL, waar gebruikt is, dat veel van deze verificaties weggepraat kunnen worden. Het meest voor de hanliggende feit om te gebruiken is wel, dat als een bepaalde opdracht zich in het kerngeheugen bevindt, dan dan de hele bijbehorende pagina zich in het kerngeheugen bevindt.

Het streven om het werkende programma zo min mogelijk te vertragen, zolang de benodigde informatie op de kernen staat is echter meer dan enkel uitbuiten van evidenties om het aantal verificaties te drukken: we willen er tevens voor zorgen, dat die verificaties, die nog wel uitgevoerd worden, zo min mogelijk tijd vergen. Dit is een strategisch uitgangspunt, slechts te rechtvaardigen door de veronderstelling dat een kerngeheugen van 32 K zo groot is, dat het voor vele problemen -dan wel voor lange tijd- toereikend zal zijn. Een kerngeheugen van 32 K lijkt mij zo groot, dat

het zinvol is om te eisen, dat alle programma's, die daaraan genoeg hebben, zo min mogelijk gehinderd worden door hun permissie die grens te overschrijden. (Hier scheiden wegen zich heel duidelijk van die van GIER ALGOL, dat in een kerngeheugen van 1000 40-bitswoorden gespeeld moest worden.)

In ons geval betekent dit, dat we, zodra de kerngeheugentoewijzing verandert, bereid moeten zijn om de consequenties hiervan uitgebreid na te gaan en op allerlei plaatsen passende notities te maken: we moeten de dynamische verificaties zoveel mogelijk prepareren.

Een enekel voorbeeld moge dit toelichten. Stel, dat we een minimumadministratie bijhouden, bestaande uit een lijstje van welke pagina's zich waar in het kerngeheugen bevinden. De vraag of een willekeurige pagina zich in het kerngeheugen bevindt, zou dan wel eens een tamelijk tijdrovende affaire kunnen zijn. Naarmate de opdrachtpagina's kleiner zijn, zal het dynamisch vaker voorkomen, dat een sprong naar de volgende -of daaropvolgende- pagina uitgevoerd moet worden. De hieruit voortvloeiende verificatieplicht kan je drastisch vereenvoudigen door bij elke pagina in kerngeheugen te vermelden of, en zo ja waar, zich de volgende pagina in het kerngeheugen bevindt. Aan de coordinator is dan natuurlijk wel de zoete plicht om bij wijzigingen deze notities steeds bij te werken.

Het idee van "de volgende opdracht" kan gebruikt worden om vele verificaties overbodig te maken, het idee van "de volgende pagina" om een aantal verificaties, die nog wel uitgevoerd moeten worden, te versnellen. Het versnellen van verificaties door passende preparatie zullen wij, als ik me niet vergis, nog wel vaker tegenkomen; er is nl. een tweede reden, waarom de techniek veelbelovend is. Als een verificatie een negatief resultaat oplevert, moet er getransporteerd worden: het proces in kwestie kan dus niet verder. Door multiprogrammering bestaat er wel de mogelijkheid, dat er iets anders kan gaan lopen, maar de zekerheid, dat dat andere niet de Dummy Abstracte Machine (zie 3.2.,EWD54 - 7) wordt. Een zinvoller zoethoudertje voor de computer is altijd beter: de coordinator is dan de meest voor de hand liggende cadidaat.

4.4.2. Dynamisch zinvolle indeling.

De drie beschreven voorbeelden werken met pagina's van uniforme lengte; bij de X8, waar de lengte van trommeltransporten binnen zeer ruime grenzen vrij kiesbaar is, zijn we dus niet aan vaste paginalengte gebonden. Het is dus redelijk, om zich af te vragen, of we de lengte van elke individuele pagina zich kunnen laten voegen naar de structuur van de informatie, die hij herbergt. Voordat ik nu de mogelijkheden van uniforme versus non-uniforme paginalengtes wil vergelijken, eerst een opmerking. Als we uniforme paginalengte kiezen, dan is het duidelijk, dat we voor deze uniforme paginalengte een keuze moeten doen -hoe is een andere zaak. De opmerking, die ik maken wilde is, dat we in geval van non-uniforme paginalengte ook een dergelijke quantitatieve keuze moeten doen. Doen we dat niet en kiezen we onze pagina's zo, dat alles wat bij elkaar hoort op 1 enkele pagina komt, dan wordt natuurlijk de uitkomst, dat we het hele programma op een enkele voldoend lange pagina onder gaan brengen. Dat was nu niet de bedoeling. De enige manier, waarop we de non-uniforme pagina-indeling kunnen aanbrengen, lijkt de volgende. Men neme een zekere ideale paginalengte in het oog. Is het programma daarvoor te lang, dan verdeelt men het programma op de meest natuurlijke manier in stukken; zijn die stukken nog te lang, dan kijkt men naar de interne structuur van deze stukken etc.etc. totdat de hele boel in passende mootjes is opgehakt.

In het geval van uniforme pagianlengte ga je, zolang je niet slim probeert te wezen, anders te werk. Het allergrofste is, dat je de tekst van het programma gewoon in vaste stukken van om de zoveel opdrachten ophakt. Een voor de hand liggende verfijning is, om de constanten, die in een pagina gebruikt worden over de pagina op te zamelen en achteraan in dezelfde pagina op te nemen. (Bij dit groeperen

moet je even voorzichtig zijn: als je nog ruimte voor een enkele opdracht hebt en die opdracht heeft net een nieuwe constante nodig, dan lukt het niet meer!) Tot zover is GIER ALGOL gegaan en verder niet; probeer je veel verder te gaan, dan krijg je steeds meer overwegingen, die bij de non-uniforme paginaindeling van meet af aan een rol moeten spelen.

En dit soort overwegingen is bedenkelijk, omdat op louter lexicographische gronden niet alle gegevens ter beschikking zijn, die uitmaken, of iets dynamisch zinvol is of niet. Het angstige is, dat het gevolg van vermeende slimheid zelfs averechts kan werken. Een enkel voorbeeld moge dit toelichten: stel dat we in een programma eerst een cyclus A hebben, gevolgd door een cyclus B met een binnencyclus C. Nu beschouwen we twee mogelijke indelingen: cyclus A zit schrijlings over een paginagrens en cyclus B kan nog in zijn geheel in de tweede pagina; het andere arrangement is, dat we de programmatekst wat doorschuiven, zodat cyclus A in zijn geheel aan het begin van een pagina komt te liggen en nu cyclus B schrijlings over een paginagrens ligt, maar zo, dat de binnencyclus C nog in dezelfde pagina als A past. Op grond van het bekende regeltje "optimalisering is vooraal van belang bij binnenste cycli" lijkt het tweede beter, want zowel A als C zitten nu niet meer schrijlings. Maar als A vele malen minder doorlopen zal worden dan B, dan is de eerste oplossing dynamisch toch echt te prefereren! Een en ander ter illustratie van het feit, dat we met minimaliseringen van pagina-overgangen erg voorzichtig moeten zijn. De ervaring heeft bovendien geleerd, dat dergelijke groepeerderijen, die al gauw op uitgebreide analyses van de samenhang, de "cross-references" in de aangeboden tekst neerkomen, vertaaltijden makkelijk tot een veelvoud doen exploderen. Het dubieuze effect en mijn voorliefde voor "load and go" maken, dat ik me niet erg aangetrokken kan voelen tot dergelijke optimaliseringen. Het ontaardt gauw in een poging tot raffinement op het verkeerde ogenblik.

Wat kan je wel doen? Wel, het onderbrengen van constantes in de pagina, waarin ze gebruikt worden, is een duidelijk voorbeeld. Het kost haast niets en het succes is verzekerd!

Immers: als we opeenvolgende opdrachten samen in een pagina onderbrengen, dan weten we, dat we niet zo gek zitten, omdat na een opdracht interesse in de volgende opdracht een heel waarschijnlijk gebeuren is. Als we nu in onze tekst "...* 2.5" tegenkomen, dan is de uit te voeren handeling "met 2.5 vermenigvuldigen" en als we dat in onze machinecode in twee losse stukken splitsen, nl. een vermenigvuldigopdracht en ergens anders de factor, dan is deze splitsing iets artificieels. Constanten in dezelfde pagina onderbrengen is dus niet: de aangeboden tekst "handig groeperen" maar "niet nodeloos ophakken". De vertaler heeft niet de informatie vernietigd, dat het hier om een constante operatie ging. En wat we in elk geval kunnen doen is proberen om het effect van de informatievernietiging door de vertaler te beperken, te mitigeren.

De opmerking, dat je verificatieplichten kunt verkorten door bij elke pagina in kerngeheugen te noteren (zie 4.4.1.) waar je de volgende kunt vinden, is op hetzelfde soort overwegingen geent. Je zult dit eerst raadplegen van de volgende nl. vooral doen bij die sprongopdrachten, die niet in de tekst voorkomen, maar die de vertaler er bij gemaakt heeft. En dat zijn nu juist bijna allemaal voorwaartse sprongen; denk aan het overlopen naar de volgende pagina of de vertaling van "if..then..else". Er is geen enkele reden om de if-clause aanleiding te laten geven tot hetzelfde stuk objectprogramma, dat je gekregen zou hebben als de sourcetekst het met expliciete sprongen -die "overal heen" kunne zijn- geformuleerd had.

Ik hoop hier meer voorbeelden van te kunnen vinden.

Ik ga er nu maar van uit, dat we "opdrachten + comstanten" samen zullen indelen in pagina's van vaste lengte. Blijft natuurlijk de vraag welke lengte.

Als je aanneemt, dat uiteindelijk het kerngeheugen in het merendeel van de gevallen groot genoeg is, dan is er een praa voor lange pagina's. Immers, het aantal malen, dat verificatie uitgevoerd moet worden is dan minder, omdat je vaker in dezelfde pagina blijft en verificatie dus achterwege kan blijven. Bovendien: hoe minder pagina's er in het spel zijn, hoe goedkoper de verificatie. Zodra je ook ernstig rekening moet houden met het geval, dat je kerngeheugen te klein blijkt,dan moet je trommeltransporten vermijden en een van de manieren om dat te doen is zo zuinig mogelijk met je kerngeheugen omspringen. Je verspilt natuurlijk als je maar in een stukje van een pagina geinteresseerd bent, de ruimte, die de rest van de pagina onherroepelijk inneemt. Ik denk op het ogenblik -12 juni 1963- over pagina's van 256 opdrachten. (Mijn idee over de ideale paginalengte is tot 256 gegroeid; als het zo door gaat, zou ik nog wel eens bij 512 terecht kunnen komen.)

De volgende overwegingen zijn voor mij wel verhelderend geweest om in te zien, welke factoren in het spel waren. Als het kerngeheugen voor het totale programma te klein is, dan kunnen we slechts lange tijd trommeltransporten uitsparen, als de besturing in een gedeelte van de programmatekst rondcyclet. Als prototype van de situatie, waarin winst is te boeken, bekijken we dus een cyclus. Dynamisch beschouwd bestaat een cyclus uit een opeenvolging van referenties naar een aantal door het programma verspreid liggende stukjes consecutief programma (de uitgeschreven cyclus, daarin gebruikte subroutines, formele parameters etc.);de vraag, die we kunnen stellen is "Als al deze stukjes M opdrachten lang zijn, wat is dan de verwachtingswaarde van de geheugenbezetting bij een paginalengte N?" Je zoudt N zo willen kiezen, dat die verwachtingswaarde minimaal is. Dat is hij natuurlijk als we N = 1 kiezen, maar dat is kennelijk niet de bedoeling: sterker, we gaan er van uit, dat $N << M$ onpractisch is, omdat het begrip "pagina" dan onvoldoende is om zo'n stuk van M opdrachten bij elkaar te laten horen. De kans, dat het stuk van M opdrachten binnen 1 pagina komt te liggen is nu $p1 = (N - M)/N$, de kans, dat het stuk schrijlings over een paginagrens komt is $p2 = M/N$. De verwachtingswaarde van de geheugenbezetting is nu $N*p1 + 2*N*p2 = N + M$. Onder de bijvoorwaarde $M \ll N$ vind je dus dat je ideale paginalengte $N = M$ zou moeten zijn en een cyclus stouw je gemiddeld in twee keer zo veel geheugen als je eigenlijk nodig zou hebben. (De kans, dat je twee stukken M in dezelfde pagina zou aantreffen is verwaarloosd.)

Op deze manier bezien is de paginalengte van circa 40 woorden in GIER ALGOL heel goed verdedigbaar: M = 40 lijkt redelijk afgestemd op allerlei standaard subroutines, de stukjes tekst, die actuele parameters vastleggen etc. Het is ook duidelijk, dat wij een grotere paginalengte moeten invoeren.

4.4.3. Voorstel tot versnelling van de verificatie. (19 juni 1963)

Het volgende handelt alleen over de pagina's van uniforme lengte, die ik denk te gebruiken voor het bergen van opdrachten + constanten (dwz. het statische gedeelte van de procesbeschrijving). Om spraakverwarring te voorkomen spreek ik van kernpagina's en trommelpagina's; een kernpagina is een stuk kerngeheugen, bedoeld om een copie van een trommelpagina te herbergen.

Ik ga in hoofdzaak uit van een paginalengte van 256 woorden -de consequenties der onderscheiden technieken zal ik af en toe terloops ook voor pagina's van 512 woorden vermelden-, een kerngeheugen van 32 K en 1 trommel van 512 K.

Het kerngeheugen bergt dan maximaal 128 pagina's, de trommel maximaal 2048. Ik neem aan, dat de coordinator er hoge prijs op stelt om per kernpagina enige historie bij te houden, welke trommelpagina er in gecopieerd staat, wanneer er voor het laatst naar gerefereerd is etc. Laten we aannemen, dat voor deze administratie een woord per kernpagina genoeg is. We hebben dan 128 woorden "kernpaginatabel".

Deze tabel mag dan ideaal zijn om bij een gegeven kernpagina vast te stellen, welke trommelpagina er in gecopieerd staat, hoewel voldoende is deze administratie nauwelijks adequaat om de omgekeerde vraag rap te beantwoorden "Staat trommelpagina zo en zo soms ergens op de kernen gecopieerd en zo ja, waar?".

Als we gewoon het lijstje afgaan om te kijken of hij er bij is, dan kost dit bv. 4 geheugencycli per test; aangenomen, dat de helft van het geheugen met uniforme pagina's gevuld is, dan moeten we minimaal 1, maximaal 64 tests aanleggen en komen we op een gemiddelde van 130 geheugencycli per verificatie. (Als het kerngeheugen wat te klein is en we vaak nul op het rekest zullen krijgen, ligt dit gemiddelde hoger, omdat je om afwezigheid te kunnen constateren, de hele boel doormoet.) Dit zou, als we er niets aan doen wel eens 25 procent van de rekentijd kunnen gaan vergen. (Dit is geschat op een verificatie om de 40 opdrachten van gemiddeld 10 geheugencontacten of 50 van gemiddeld 8.) In elk geval te veel. Ik wil dit nu op twee manieren bestrijden:

1) onderzoeken hoe door extra geheugenruimte en eventueel extra werkzaamheden bij verandering der kerngeheugentoewijzing deze algemene test versneld kan worden
2) onderzoeken hoe we, zolang de kerngeheugentoewijzing niet verandert het aantal van deze algemene verificaties kunnen drukken.

Analoog aan de kernpaginatabel kunnen we een trommelpaginatabel invoeren. Die zou dan 2048 woorden beslaan en in elk woord zouden we kunnen noteren of en zo ja, waar de trommelpagina in de kernen gecopieerd staat. Dit maakt de omgekeerde vraag inderdaad rap beantwoordbaar, maar tegen niet verwaarloosbare kosten. Bij wijziging in de kerngeheugentoewijzing moet de coordinator nu behalve de kernpaginatabel ook de trommelpaginatabel bijwerken, deze werkzaamheden zijn verwaarloosbaar, de geheugenbezetting van 2048 woorden maximaal is echter iets, waartegen ik wel een beetje hik. Bedenkend, dat je slechts vraagt naar een 7-bits antwoord, kan je met drie in een woord de omvang van deze tabel tot 683 maximaal drukken, maar dit wordt "tradind speed for space". (Bij pagina's van 512 woorden heb je er maximaal 1024 op de trommel en maximaal 64 in het kerngeheugen: 1024 6-bits antwoorden kunnen op 256 woorden worden afgebeeld. De versnelling door de trommelpaginatabel is nu minder indrukwekkend, omdat scannen van de kernpaginatabel dan ook gemiddeld twee keer zo snel is afgelopen. Hier zien we overigens mooi, hoe grotere pagina's de administratie drukken.)

Een alternatief om de comprimatie -van 11 tot 7 of van 10 tot 6 bits- te verrichten is om in te voeren een lijstje van de in het kerngeheugen gecopieerde trommelpagina's, naar opklimmend trommelpaginanummer gerangschikt. Dit vergt bij verwijdering en toevoeging enige extra administratie -wat geschuif- maar de vraag van het "waar" kan nu met een logarithmische zoekerij, die in gemiddeld 6 (of 5) slagen beeindigd is De tijdsduur, die voor de logarithmische zoekerij nodig is is iets groter -volgens voorlopige probeersels- dan die nodig voor het raadplegen van de gecomprimeerde trommelpaginatabel (30 tot 40 geheugencontacten) kortom: op de 400 nog te veel om te licht over te denken.

Je kunt ook de kernpagina's aan een ketting rijgen, aftastbaar in de volgorde van laatste referentie. Het programma, dat zo'n ketting afwandelt is een snoesje, 6 geheugencontacten per slag, als ik me niet heb vergist. Zolang je in een paar pagina's je rondjes draait, gaat de verificatie leuk en kan deze methode met de anderen concurreren, maar als je meer dan vijf, zes schakels af moet tasten, wordt het tijdrovend en moet je rekening gaan houden met maximale zoektijden van boven de milliseconde (zeg 400 geheugencontacten), nl. voor een zo vergeten pagina, dat hij al bijna op de nominatie staat om overschreven te worden. Het effect zou zijn, dat bij een plotselinge wisseling de machine "wat op zijn kop moet krabben" zich afvragend waar dat en dat ook al weer stond. In verband met multiprogrammering lijkt me dit niet aantrekkelijk. Een technisch nadeel (dat door scannen van de kernpagina tabel gedeeld wordt) is bovendien het volgende: als een trommel-

pagina niet op de kernen aanwezig is, ontdek je dat pas, nadat je de hele ketting, resp. tabel hebt afgewerkt en je trommeltransport kan pas daarna gestart worden. Liever een snellere detectie, dat het mis zit en administratief gehannes, nadat het transport vast in gang gezet is. Als generale administratie is het ketting rijgen dus niet bruikbaar; hoogstens kunnen we voor de vier jongste pagina's bv. de nummers in een klein cyclisch magazijntje bijhouden om kleine cycli wat te helpen, als dit nodig mocht zijn. Ik hoop van niet.

Vooralsnog zie ik het meeste heil in de naar opklimmend trommelpagina nummer gesorteerde aanwezigheidslijst, en wel voornamelijk omdat deze lijst evenredig lang is met de kernpaginatabel, zodat ik de twee door elkaar kan vlechten. Ik moet de alternatieven gewoon uitwerken. Hoe zwaar we ruimte tegen tijd laten wegen, hangt er mede van af, hoe frequent we proberen moeten om via deze administratie bij een gegeven trommelpaginanummer een kernpaginanummer te vinden. Vooralsnog vergt dit naar schatting 10 procent van de rekentijd, ook als alles in het kerngeheugen zit. En dit vind ik nog wat veel.

Nu de tweede mogelijkheid: de opmerking is nl. dat het niet nodig is om de vraag "waar staat trommelpagina nummer zoveel" zo rap te kunnen beantwoorden, als we een mechanisme kunnen bedenken, waardoor het aantal malen, dat deze vraag gesteld wordt, maar genoeg [illegible] gedrukt wordt.

Voor sprongen naar de volgende pagina is deze oplossing gevonden. Het feitenmateriaal, dat hiervoor door de centrale administratie bijgehouden moet worden is onafhankelijk van de omvang van het programma en onafhankelijk van de intensiteit van interpaginareferenties en tenslotte altijd up to date.

Om verdere verificatie's te onderdrukken heb ik allerlei dingen geprobeerd; de grote moeilijkheid was, dat je op grond van aanwezigheid dat gegevan wel overal rond kunt strooien, maar dat het heel moeilijk wordt om na te gaan, wat je dan allemaal herroepen moet, mocht zo'n pagina verdwijnen. Wat wij echter moeten exploiteren is het volgende: we moeten zorgen, dat de vraag "welke trommelpagina staat in die en die kernpagina?" rap beantwoordbaar is. Dat kan en uit het volgende zal blijken, dat het de moeite loont om daar alles en alles aan te doen.

Laten we als specifiek voorbeeld eens beschouwen het meegeven van de link aan een procedure (het meegeven van het beginadres van een impliciete subroutine als actuele parameter is van hetzelfde laken een pak). Het is duidelijk, dat we dit terugkeeradres invariant, dwz. in termen van trommelpaginanummer moeten meegeven, zeg om de gedachten te bepalen een adres in trommelpagina 287. Het is echter onverstandig om bij de Return de centrale administratie zonder meer te confronteren met de vraag "waar staat trommelpagina 2877". We kunnen er allicht op gokken, dat die nog op dezelfde kernpagina staat. Wat we dus moeten doen is als terugkeeradres twee dingen meegeven: een invariant trommeladres en een kernenadres, het laatste als suggestie! De localisatie van een trommelpagina kunnen we beschouwen als een tijdrovend sommetje, waarvan we het antwoord echter heel makkelijk kunnen controleren en waarvoor we een schatting hebben, die in 95 procent van de gevallen raak zal zijn, nl. als de oude pagina nog op zijn plaats staat. Als dat niet meer het geval is, dan is er in de tussentijd zoveel gebeurd, dat de verificatietest er nog wel bij kan.