Numéro d'ordre: 489

ANNEE: 1990

USTL FLANDRES ARTOIS — LIFL — U.A. 369 du C.N.R.S. — CNRS

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

50376
1990
23

THÈSE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES
DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

Les performances d'une architecture d'ordinateur destinée au support de la communication par message

par

Jean-Marie Place

Thèse soutenue le 2 Février 1990, devant la commission d'examen

Membres du Jury:

Président	:	M. Dauchet,
Directeur	:	E. Delattre,
Rapporteurs	:	C. Carrez,
	:	V. Cordonnier,
Examinateur	:	J-M. Geib,

Université des Sciences et Techniques de Lille Flandres Artois
U.F.R. d'I.E.E.A. Bât. M3. 59655 VILLENEUVE D'ASCQ CEDEX
Tél 20 43 44 92

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilitéss
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Françis	Systèmes électroniques
M. PARSY Fernand	Mécanique

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

Remerciements

• Je remercie Max Dauchet, président, pour m'avoir fait l'honneur de présider ce jury ainsi que pour ses nombreux encouragements.

• Je remercie Christian Carrez, rapporteur, qui m'a intégré dans son équipe de recherche et qui a accepté de rapporter cette thèse.

• Je remercie Vincent Cordonnier, rapporteur, qui a bien voulu examiner ce travail.

• Je remercie Eric Delattre, pour ses conseils multiples et précieux. La patience et la rigueur qu'il a su montrer lors de la mise au point de cet ouvrage m'ont été irremplaçables.

• Je remercie Jean-Marc Geib et Jean-François Méhaut. mes coéquipiers, qui ont beaucoup apporté dans le travail de cette équipe. Leur présence permanente au sein du laboratoire est un renfort d'une grande valeur.

• Je remercie Henri Glanc qui a assuré la reproduction de cet ouvrage avec sa digilence, son sérieux et son obligeance coutumière.

• Enfin je voudrais exprimer ma reconnaissance envers chacun des membres du laboratoire, enseignants, chercheurs, techniciens pour la qualité de vie et de travail qu'il contribue à créer quotidiennement.

TABLE DES MATIERES

TABLE DES FIGURES

INTRODUCTION

Cet ouvrage présente des travaux réalisés sous le nom "OMPHALE". Ce nom concerne en fait deux projets menés au sein du Laboratoire d'Informatique Fondamentale de Lille (L.I.F.L.). Les membres permanents de l'équipe de recherche sont au nombre de cinq:

- C. Carrez, Professeur, a dirigé l'équipe jusque mi-87.
- E. Delattre, Professeur, l'a remplacé dans cette fonction depuis cette date.
- J.M. Geib, Maître de conférences.
- J.F. Méhaut, Maître de conférences.
- J.M. Place, assistant.

Le premier projet concernait l'étude d'un système réparti, structuré en domaines de protection et muni d'un adressage par capacités. Ces recherches dirigées par C. Carrez ont donné lieu à la thèse de docteur ingénieur d'E. Delattre[Delattre79] . Entre autres résultats, ce projet a donné lieu à la conception d'une architecture matérielle dédiée à ce système qui a débouché sur la construction d'un prototype nommé "site0".

En 1983, OMPHALE s'est transformé en un second projet visant à la conception et au développement d'un noyau de système réparti fondé sur les notions de serveur et de communication par message. La conception a été effectuée par J.M. Geib. La faisabilité de cette architecture a été démontrée par J.F. Méhaut qui a réalisé son implantation sur SPS7. Par ailleurs, l'architecture du site0 convenait également à la mise en œuvre de ce second projet. L'utilisation du matériel nu du site0 permettait une évaluation complète des performances de l'ensemble matériel et logiciel. Ceci est le sujet de cet ouvrage. Cette thèse s'inscrit donc dans un contexte composé de:

- La définition de l' architecture logicielle OMPHALE qui propose:
  - Un mécanisme de fonctionnement pour la totalité des logiciels du système (notions de module, liste de primitives, schéma d'exécution, mécanismes de communication).
  - Une structure en trois couches : un noyau appelé NERF, une couche système d'exploitation de nom CORTEX et une couche formée de modules d'application.

  Le chapitre I expose les principales caractéristiques de cette architecture.

- Une organisation matérielle bi-processeur composée d'un processeur PN destiné à l'exécution du système et d'un autre processeur PC disposant d'un adressage protégé et chargé de l'exécution des programmes utilisateurs. Cette organisation doit permettre d'assurer une efficacité acceptable même pour un overhead système important. Une description détaillée de cette organisation est fournie par le chapitre II.

Une simulation a permis d'observer les grandes lignes du comportement du site0 et d'en déduire quelques conclusions sur la configuration optimale d'un site. Cette simulation, réalisée par nos soins est décrite dans le chapitre III

L'évaluation de l'ensemble site0/OMPHALE demandait au préalable l'accomplissement de plusieurs travaux:

- La réalisation du site0, effectuée par E. Delattre.
- L'assemblage d'une chaîne de développement. Y intervinrent A. Ataménia et D. Fouret sous la houlette de C. Carrez.
- L'écriture du noyau exécutif temps réel baptisé EXO fut d'abord assurée par F. Lebail puis reprise par E. Delattre.
- La définition d'un langage de programmation pour l'écriture de logiciels bâtis sur l'architecture OMPHALE et l'écriture d'un pré-processeur que nous avons achevés. La présentation de ce langage est intégrée dans la description de l'architecture logicielle du premier chapitre.

En disposant de tous ces éléments, nous avons mesuré les performances de cet assemblage et mettre en évidence une bonne adéquation entre l'organisation matérielle et l'architecture logicielle. Ces mesures sont rapportées et analysées lors du chapitre IV.

I. L'ARCHITECTURE OMPHALE.

Le travail exposé dans cet ouvrage a été réalisé dans le cadre du projet OMPHALE de définition d'une architecture de système réparti.Le chapitre présent a pour objet de présenter cette architecture telle qu'elle a été définie dans sa version la plus récente. Cette présentation n'a pas vocation à être une justification, ni même une description détaillée et complète de l'architecture: tous ces éléments ont déjà étés exposés dans la thèse de JM. GEIB [Geib89]. Il s'agit simplement de rappeler les concepts dont l'architecture OMPHALE est imprégnée, et les applications qui en ont été faites. Nous nous baserons pour cela sur le langage que nous avons défini pour supporter la programmation OMPHALE . Nous illustrerons la manière dont celle-ci fut mise en œuvre par des exemples de programmes. D'autre part, cet exemple nous fournira un prototype de programme qui nous servira lors des mesures.

I.A. INTRODUCTION.

OMPHALE est une architecture de système réparti ayant pour support un ensemble de sites connectés par un réseau local fiable et à haut débit. Les sites sont des ordinateurs autonomes comprenant un ou plusieurs processeurs, de la mémoire et éventuellement des périphériques. Nous nous intéresserons uniquement à l'architecture vue depuis un site: Le caractère réparti ainsi que sa mise en œuvre a été développé dans la thèse de J.F. MEHAUT [Méhaut89].

I.B. LES CONCEPTS UTILISES.

La définition de cette architecture a permis d'intégrer de façon harmonieuse dans un même ensemble un certain nombre de notions plus ou moins récentes:

- Le modèle client–serveur comme outil de conception des systèmes d'exploitation.
- La programmation par objet comme méthode structurante.

- La communication par message utilisée pour la réalisation des communications entre objets.

- Le mécanisme de capacité pour accroître la sécurité des systèmes.

Nous reprenons ici les thèmes développés dans le mémoire d'habilitation de E. DELATTRE [Delattre89a].

L'ensemble de ces concepts est intégré en un ensemble cohérent qui offre l'avantage d'une expression naturelle (car déduite de la structuration par objets) du parallélisme.

I.B.1. LE MODELE CLIENT-SERVEUR.

Le modèle client-serveur est une démarche structurante appliquée lors de la conception de systèmes d'exploitation. L'idée de base de ce modèle est de sortir un maximum de fonctionnalités des couches profondes du système, pour les proposer par l'intermédiaire d'entités logicielles distinctes : les serveurs. Un serveur regroupe tous les services liés à un aspect du système (par exemple serveur de mémoire, serveur de fichiers). Minix [Tanenbaum87] est l'exemple d'un tel système mené à terme.

Les avantages obtenus sont nombreux:

- **Modularité** car les serveurs travaillent dans un environnement bien spécifié. d'où une augmentation de la fiabilité et de l'extensibilité.

- **Minimalisation** du noyau qui est, par cette technique, allégé d'une multitude d'éléments non fondamentaux. Le noyau, plus petit, est ainsi rendu plus fiable et moins coûteux.

- **Portabilité** accrue: L'interface entre le système et le matériel peut être en grande partie localisée à l'intérieur d'un petit noyau et de serveurs interchangeables.

- **Ouverture**. Dans un cadre expérimental, la possibilité donnée à l'expérimentateur de réécrire tout ou partie des modules facilite grandement l'étude de l'architecture.

I.B.2. LA PROGRAMMATION OBJET.

L'approche orientée objet définit l'objet comme élément de structuration des grands systèmes de traitement de l'information.

D'un point de vue conceptuel, un objet est visible comme une entité qui possède un comportement et sur lequel on peut agir au moyen d'opérations (l'interface est l'ensemble des opérations disponibles).

Pour l'implantation, un objet peut être considéré comme un ensemble de données encapsulées avec les procédures qui les manipulent. Le procédures sont lancées en application des demandes d'opérations.

Cette démarche est généralisée à tous les niveaux de la conception du système. Les techniques d'héritage permettent de construire des objets conceptuellement de plus en plus complexes.

EN GENIE LOGICIEL.

L'approche objet apporte de nombreux avantages à tous les stades de production d'un logiciel [Meyer88] .

- Lors de la conception, elle suscite une décomposition en éléments indépendants, petits et de complexité limitée. Par ailleurs, les interactions entre les objets sont peu nombreuses et bien spécifiées.

- La réutilisabilité des objets, favorisée par le haut niveau conceptuel des interfaces, permet d'augmenter la productivité des équipes qui n'ont pas à réinventer à chaque fois des éléments similaires.

- Les contraintes d'intégrité sont un outil supplémentaire pour contrôler le bon fonctionnement des composants logiciels ainsi conçus. Le principe d'encapsulation permet de restreindre les effets des pannes logicielles en évitant leur propagation.

COMME EXPRESSION DU PARALLELISME.

La programmation par objet est classiquement vu comme un modèle de programmation. On s'arrête alors dans la démarche à la description du

logiciel sous forme textuelle, sans s'inquiéter du résultat final. Il est possible d'aller plus loin dans la démarche et de concevoir un système complet (environnement de programmation, système d'exploitation et matériel) qui préserve cette notion jusqu'au binaire. L'expression d'un logiciel selon cette approche fourni finalement un ensemble d'éléments indépendants. Il est alors tentant d'utiliser le découpage naturel issu de la conception comme support du parallélisme. Ce raisonnement conduit alors à associer un processus à chaque objet (ce n'est pas la seule approche possible[Yonezawa87] [Geib89]. On parle alors d'"objet actif" Nous rejoignons ainsi, par une autre démarche que la préoccupation de construction des systèmes, le modèle serveur mentionné plus haut. A cause du principe d'encapsulation des données, les problèmes liés au parallélisme (communication, synchronisation) sont alors reportés au niveau externe des objets (réaction d'un objet actif à deux demandes d'opération simultanées par exemple), ou encore gérés localement à chaque objet.

I.B.3. LA COMMUNICATION PAR MESSAGES

Outre la problématique de conception des grands systèmes, la maîtrise des phénomènes temporels dans les logiciels système encourage le concepteur à décrire le système comme un ensemble de processus communicant [Cheriton82] . Un point de passage obligé de cette description est la définition des moyens de communication entre ces processus. L'une des techniques employée est l'envoi de message. Les avantages que l'on compte en retirer sont les suivant :

- L'approche orientée objet fait souvent référence, au niveau conceptuel du moins, à ce mode de communication entre les objets (voir le déclenchement de méthodes par envoi de messages de Smalltalk [Goldberg83] .

- La copie par valeur permet d'assurer l'isolation – donc la protection –des données encapsulées.

- Cette technique est peu liée au matériel (par opposition à la communication par variable partagée par exemple, qui n'est facilement implantée qu'au moyen d'une mémoire commune). Elle est donc applicable sur des réseaux hétérogènes.

I.B.4. LA PROTECTION PAR CAPACITES.

L'intérêt de la mise en œuvre de mécanismes de protection des objets est évident: ces mécanismes apportent une sûreté accrue ainsi qu'une aide non négligeable durant la mise au point des logiciels. Depuis l'apparition des microprocesseurs 16 bits, les matériels commercialisés intègrent d'ailleurs différents mécanismes de protection.

Le principe des capacités [Levy84] préconise l'association indéfectible du moyen d'accès à un objet et des droits qui lui sont associés. Il participe ainsi également aux mécanismes de nommage des objets de l'environnement d'un processus et à leur protection. Hélas, les coûts (en matériel et/ou en efficacité) de ce mécanisme proscrivent son utilisation systématique lors de l'accès au objets élémentaires [Organick83] . On est alors amené à regrouper ces objets en segments selon leur statut vis à vis de la protection.

I.C. DESCRIPTION DE L'ARCHITECTURE.

Pratiquement, l'architecture OMPHALE unifie les notions décrites plus haut d'objet, de serveur et de processus sous le nom de *module*. Un module est un élément clos qui englobe données et procédures. Ces modules peuvent émettre des messages ou en recevoir sur des quais. Les quais sont les seuls éléments d'un module visibles par l'utilisateur de ce module. La réalisation pratique des quais demande la création d'une file d'attente de messages par quai.

Un module est :

- Un objet: C'est une entité autonome dont l'état évolue au gré des opérations qui lui sont demandées et qui encapsule des données.
- Un serveur: Il offre aux autres modules un ensemble de services en alternant périodes d'activités et périodes d'attente.
- Un processus séquentiel: Un objet ne traitera jamais qu'une seule demande d'opération à la fois.

Enfin, avec la notion de lien, le mécanisme des capacités est utilisé en ce qui concerne le nommage intermodules (mais non à l'intérieur d'un module) en vue d'obtenir un bon compromis entre protection et efficacité.

I.C.1. STRUCTURATION EN COUCHES.

OMPHALE est structuré en quatre couches.

- La description du matériel n'est pas abordée dans ce chapitre car OMPHALE se veut une architecture logicielle indépendante de la constitution physique de la machine.

Le logiciel système se compose des deux niveaux immédiatement supérieurs:

- Un noyau baptisé NERF assure le routage des messages ainsi que la gestion de l'état des modules. Un exemplaire de ce noyau existe par site.

- Le CORTEX (collection d'outils répartis) est un ensemble de modules qui offre aux modules généraux les fonctionalités indispensables qui sont absentes du noyau (gestion mémoire, nommage symbolique et création des modules, etc...). Tout comme les modules généraux, ces modules sont sollicités par l'intermédiaire de messages. Les modules de CORTEX sont répartis sur l'ensemble des sites.

- L'application se compose d'un ensemble de modules qui communiquent par le biais de messages. Un exemple d'application est détaillé dans la suite de ce chapitre. L'application constitue la couche supérieure du logiciel.

I.C.2. PRESENTATION

La structure de cette section est un peu particulière, puisque nous discuterons simultanément de l'architecture OMPHALE et de l'exemple de module que nous avons choisi. nous terminerons la description de chaque concept par le source décrivant l'exemple choisi selon le langage OMPHALE. Ainsi, chaque concept sera décrit par trois sous-sections:

- **Omphale**
  La sous-section OMPHALE présente les termes et spécifications de l'architecture. Nous ne décrirons que les éléments essentiels – parties réelles des constituants OMPHALE. D'autres notions (par exemple noms symboliques ou formats de message) rajoutés par le langage pour simplifier l'utilisation de l'architecture seront vue dans la sous-section langage.

- **L'Exemple**
  L'exemple montre comment nous pouvons, selon l'architecture OMPHALE, réaliser une pile d'entiers. Cet exemple, repris et extrapolé de la thèse de J.F. Mehaut [Méhaut89] est intéressant dans la mesure où il a été présenté sous deux versions qui différent par la granularité des modules conçus (un élément de pile ou une pile par module). En fait, nous pouvons faire évoluer graduellement la granularité entre ces deux extrêmes. Nous obtiendrons ainsi un étalon commode pour les mesures. La version que nous vous proposons ici est donc une version 'générique' de ces deux exemples.

- **Langage**
  Nous décrirons dans cette partie le reflet des principes de l'architecture dans le langage que nous avons défini (baptisé langage OMPHALE). Ce langage est un sur-ensemble de Modula-2 [Wirth84]. Nous avons volontairement simplifié ce langage – initialement trop ambitieux – et corrélativement l'exemple pour faciliter leur compréhension. Nous avons ignoré tout ce permet ou rend plus sûre la compilation des logiciels par exemple les problèmes strictement techniques de typage ou de portée des variables: Le lecteur qui voudrait une description complète de ce langage peut consulter l'annexe A [JMP88] La liste de l'exemple complet peut, également y être trouvée (annexe B). L'application de l'exemple au langage OMPHALE suivra immédiatement la description du langage.

DESCRIPTION DE L'EXEMPLE.

Il s'agit de réaliser un serveur "pile d'entiers" sur plusieurs modules. Une pile est constituée d'un module `Pile` et de un ou plusieurs modules `Element`. L'un de ces modules est le module 'Element sommet'. Le module `Pile` recueille les requêtes d'empilage et de dépilage et les redirige sur le module Element sommet. C'est également le module `Pile` qui décide de la création et

de la destruction des modules `Element`. Pour cela, il tient à jour pour son propre compte le nombre d'entiers mémorisés par l'élément sommet. Les modules `Element` sont chaînés (chaque module conserve un lien vers son successeur).

Il est évident ici que la séparation de la pile en plusieurs sous-ensemble n'a aucun intérêt pratique. Notre objet est uniquement de produire une application simple et de granularité variable. L'ordre de grandeur des modules reste malgré tout plausible dans un cadre "programmation orientée objet".

Les figures suivantes précisent le fonctionnement de l'ensemble lors des diverses opérations:

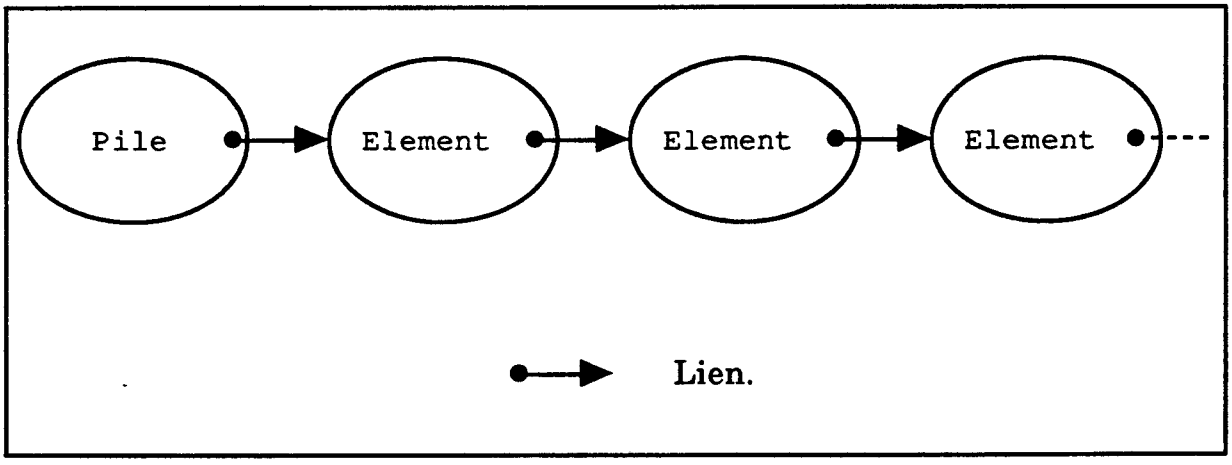


Fig. I-1. Chaînage des modules.

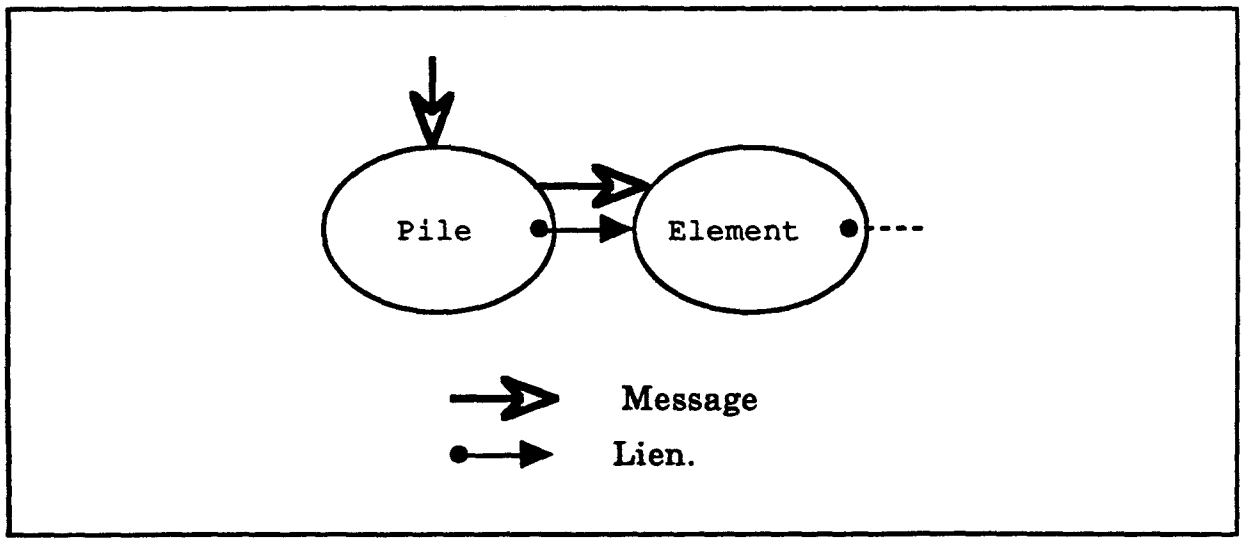


Fig. I-2. Empilage.Le module PILE reçoit une demande d'empilage. Il crée au besoin un nouveau module, qu'il situe en tête de la liste chaînée. Il envoie alors la valeur à empiler à l'élément sommet.

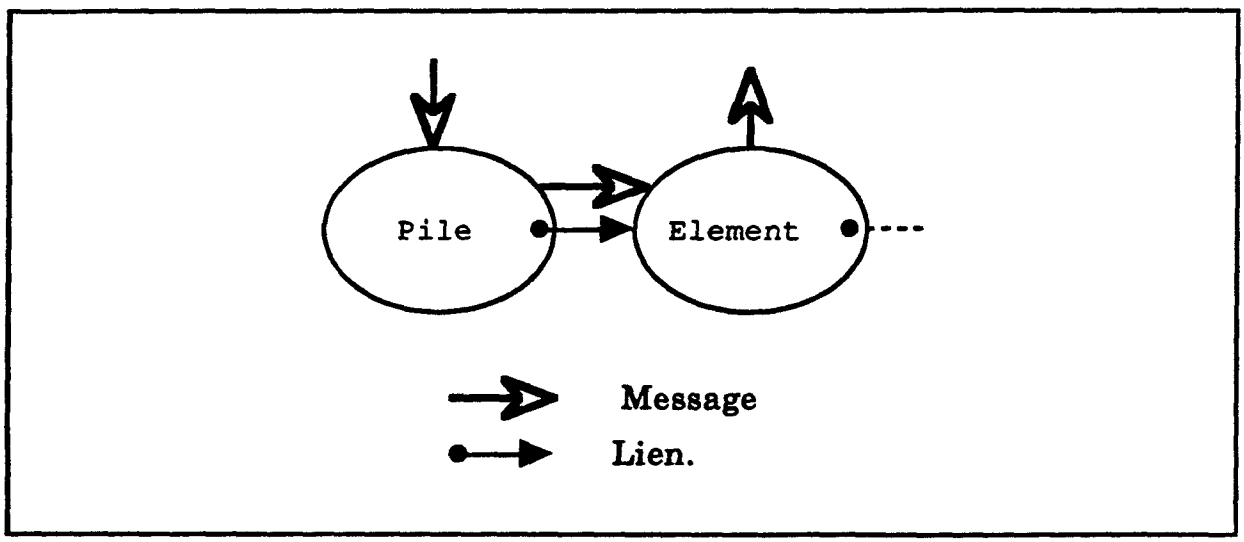


Fig. I-3. Dépilage. Le module PILE reçoit une demande de dépilage. Cette demande est transmise à l'élément sommet qui retourne directement la valeur (technique du long retour [Stroustrup81]) puis éventuellement, disparait. Nous n'avons pas mentionné, dans cette figure, les message destinés à maintenir la cohérence de la structure chaînée des modules.

I.C.3. LE MODULE

DESCRIPTION

OMPHALE ne connait qu'un seul type d'entité active: le module. Un module est autonome: il est constitué par un contexte d'exécution et un processus unique. Le processus peu réagir à la réception d'un message sur un plusieurs quais. L'ensemble des quais sensibles (i.e. sur lequel l'arrivée d'un message demande une réaction du module) est fixé par le module lui-même (attente sélective). La réaction du module peut provoquer la modification des variables locales (changement d'état de l'objet) et/ou l'envoi de messages.

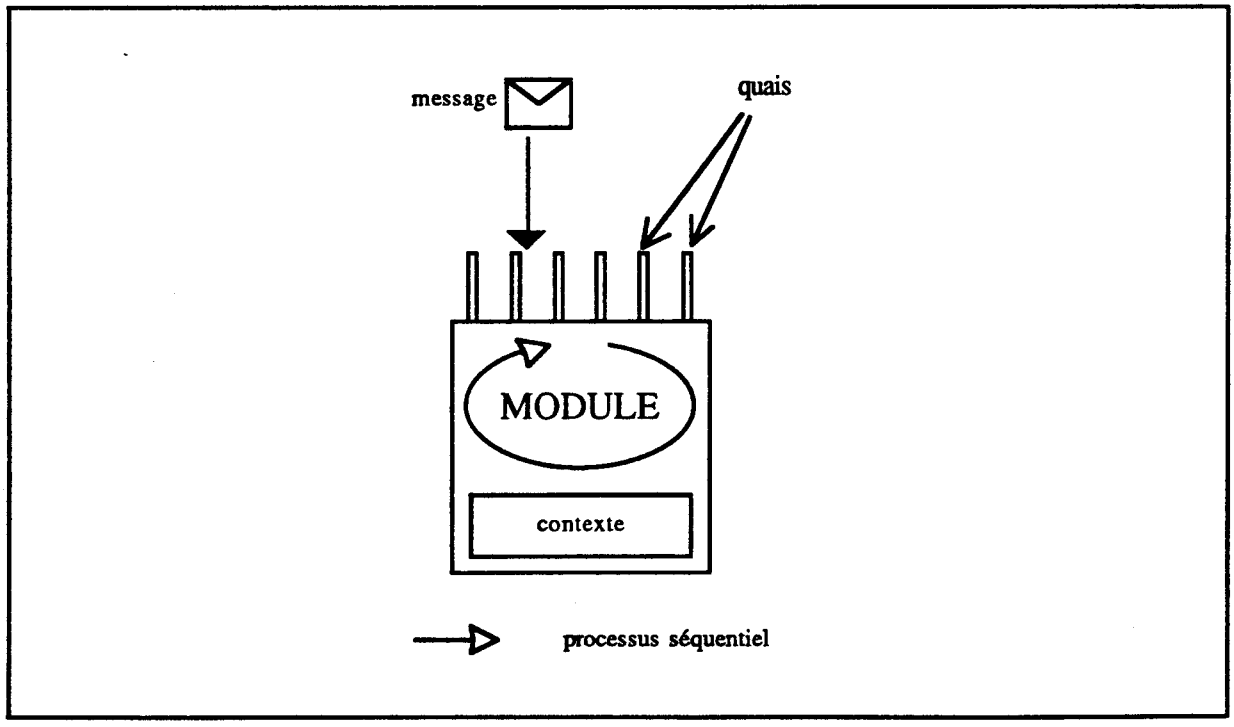


Fig. I-4. Constitution d'un module.

Un module OMPHALE peut ainsi être décrit par le texte source qui caractérise le comportement du module. A cette partie programmée, OMPHALE adjoindra quelques éléments indispensables au fonctionnement (outils de communication avec le noyau, code des primitives système).

LES ELEMENTS DEFINIS PAR LE PROGRAMMEUR.

- Le code du processus (unique) associé au module qui est, habituellement (mais non impérativement) décomposé en un contrôleur et plusieurs services.

  → Le contrôleur est chargé de piloter la politique de prise en compte des requêtes de service par ouverture et fermeture des quais et appels des services.

  → Les services implantent les opérations demandées. La même démarche systématique conduit à les écrire, au niveau du source, sous la forme de procédures sans paramètres.

- Les variables encapsulées dans le module forment le contexte d'exécution du processus associé.

- Les quais sur lesquels les messages des modules extérieurs sont reçus. **Il y a liaison univoque entre les quais et les services**: la réception d'un message sur un quai équivaut à la demande d'exécution d'une opération particulière. Ces quais (et, par extension, les services correspondants) sont désignés par des noms symboliques.

LES ELEMENTS FOURNIS PAR LE SYSTEME.

- Une zone mémoire particulière, baptisée environnement du module, regroupe les liens possédés par le modules. L'environnement recense donc l'ensemble des modules connus par le module courant. Les liens proprement-dits ne sont pas directement visibles de l'intérieur du module, mais sont référencés par l'intermédiaire de 'nom locaux de liens' (en fait des index dans l'environnement). A la création, le module dispose de quelques liens 'innés' tels que l'auto-lien[1]: lien d'un module vers lui-même.

---

[1] Il est indispensable que chaque module dispose d'un lien vers lui-même afin de pouvoir se désigner lors de sa communication avec autrui. L'adresse de l'expéditeur n'est pas automatiquement transmise avec le message.

- La zone de communication du module permet la communication entre le module et le noyau NERF. Le processus lié au module peut, grâce à cette zone, récupérer voire modifier des informations du système:

  → L'identificateur du quai sur lequel est arrivé le message qui a déclenché la période d'activité courante.

  → Une variable permet au module de consulter ou de modifier la liste des noms des quais sur lesquels l'arrivée d'un message pourra déclencher le réveil.

  → Une dernière variable contient le message reçu.

- La zone d'émission du module sert à stocker l'ensemble des messages émis durant la période d'activité courante. Son utilisation est cachée derrière toutes les primitives concernant les messages (cf. MESSAGES).

- A cela, il faut rajouter la primitive WAIT qui permet à un module de s'endormir jusqu'à la réception d'un message sur l'un des quais validés.

LE SCHEMA D'EXECUTION.

La vie d'un module est constituée de périodes d'activités entrecoupées de périodes d'attente.

Une période d'activité est déclenchée à la suite de la réception d'un message (requête de service). Lors de celle-ci, le module peut:

- Consulter le contenu du message et identifier le service demandé par l'intermédiaire de la zone de communication.

- Utiliser le contexte d'exécution du processus associé.

- Emettre à son tour des messages à destination d'autres modules.

- Valider ou invalider des services. Lorsqu'un service est invalide, les messages peuvent encore être reçus par le module, mais ne déclenchent plus de période d'activité, et ne peuvent donc plus être traités.

- Mettre fin à une période d'activité. Quand le service demandé est terminé, le module repasse en attente d'autres messages. Ceci est considéré, du point de vu du programmeur, comme un appel d'une primitive de nom WAIT. Cette primitive cache la totalité des opérations réalisées par le noyau entre la fin d'une période d'activité et le début de la période suivante. Il s'agit principalement du routage des messages et de la gestions des modules. WAIT est la seule primitive asynchrone: les autres primitives n'interrompent pas l'exécution du module.

LES ELEMENTS INTRODUITS PAR LE LANGAGE.

- Un nom symbolique de module permet, au niveau du langage d'établir les liaisons statiques et dynamiques entre les modules. Un module source peut donner lieu, soit à un module unique, soit à un module générique. Dans le dernier cas, un module source sera le prototype d'un nombre indéfini de modules réels dans le système: Il représentera une famille d'objet ayant le même comportement. Le nom symbolique du module sera alors fourni au dispositif créateur (le gérant de module).

Ainsi, un module offre les trois aspects déjà présentés:

- Processus: Un module est lié lors de sa création à un processus (et un seul). Ce processus gère de manière classique les variables locales à l'objet et dispose de primitives d'envoi asynchrone et de réception bloquante de messages. Globalement, on peut dire que le processus implante le comportement de l'objet.

- Serveur: Un module interprète tout message reçu comme une demande de service. De part sa faculté de réception sélective sur ses quais, un module est capable de gérer sa propre politique de service.

- Objet: Un module possède une interface (l'ensemble de ses quais) et encapsule des données. La réception d'un message sur un quai, interprétée comme une demande d'opération, peut éventuellement modifier l'état interne de l'objet.

EXEMPLE

Dans l'exemple de la pile d'entier, nous avons décrit deux types de modules:

- Le module `Pile` reçoit toutes les demandes et gère la pile de manière centralisée. Ce module existe en exemplaire unique.

- Le module `Element` conserve et gère une section de `TailleElement` entiers contigus dans la pile. Un nouvel exemplaire de ce module est créé à la demande du module `Pile` lors d'un débordement. Inversement un module `Element` qui n'est plus utilisé sera détruit.

COMPORTEMENT DES MODULES.

Nous décrirons le comportement de chacun des types de modules par la listes de leurs quais et des opérations associées:

LE MODULE "PILE".

- `Empiler`
  Un message envoyé sur ce quai provoquera la mise en sommet de pile de la valeur contenue dans le message.

- `Depiler`
  Un message envoyé sur ce quai représentera une demande de dépilage. Ce message contiendra une adresse (nom de module + nom de quai). L'entier situé au sommet de la pile en sera donc retiré et renvoyé au demandeur.

- `Reponse`
  Ce dernier quai banalisé est utilisé pour récupérer la réponse à des demandes spéciales. Il ne sera jamais utilisé à l'initiative d'autres modules (il n'est d'ailleurs pas mentionné dans la partie 'spécification'):

  → Récupération d'un lien vers un élément récemment créé (demande faite au gérant de module).

  → Récupération d'un lien vers le nouvel élément sommet (demande faite à un module `Element` lors de sa destruction).

LE MODULE "ELEMENT".

- `Empiler`
  Un message arrivant sur ce quai contiendra un entier qui devra être être empilé localement (un tel message ne peut arriver que sur un élément sommet).

- `Depiler`
  C'est vers ce quai que sera redirigé – par le module `Pile` – les messages de requête de dépilage. Ils contiendront l'adresse du demandeur.

Les deux quais suivants servent au dialogue entre le module `Element` et le module `Pile`. En effet, le stockage des liens étant réparti, le module `Pile` a besoin de stocker et de reprendre ces liens lors des changements de configuration (Création ou destruction de module `Element`).

- `Chaîner`
  Dès la création d'un module `Element`, le module `Pile` lui envoie un lien vers son successeur. Ce quai a cet usage.

- `Suivre`
  Un message arrivant sur ce quai signale au module qu'il n'est plus utile et va donc être détruit. Juste avant sa mort, le module renvoie le nom de son successeur.

LANGAGE

La description d'un module dans le langage OMPHALE comporte plusieurs parties:

- La partie `SPECIFICATION` comporte la liste des éléments importés et exportés par le module. On y trouvera en particulier la liste des quais mis à disposition des autres modules.

- La partie `IMPLEMENTATION` explicite le fonctionnement interne du module. Cette partie est elle-même découpée en :

  – Déclaration des quais à usages locaux.

- Déclaration des variables globales au module qui constituent son environnement.
- Déclaration des procédures qui implantent chacun des services.
- Déclaration du contrôleur.

Les éléments fournis par le systèmes sont utilisables par le programmeur sous la forme d'identificateurs prédéclarés:

- RECEIVINGSERVICE. Nom du quai qui a accepté le message de réveil.
- OPENEDSERVICE. Liste des quais ouverts.
- RECEIVEDMESSAGE. Message qui a déclenché le réveil.
- WAIT. Primitive dont l'appel marque une fin de période d'activité.

On peut dès lors donner la structure de la description du module PILE:

```
MODULE Pile ;
SPECIFICATION
USES FORMAT FEntier, FLien, FQui ;
SERVICE Empiler(FEntier), Depiler(FQui) ;
IMPLEMENTATION
USES MODULE Element ;
SERVICE Reponse ;
      ..................Déclarations...
PROCEDURE ProcEmpiler() ;
BEGIN
      ...
END ProcEmpiler ;
PROCEDURE ProcDepiler() ;
BEGIN
      ...
END ProcDepiler ;

BEGIN
      ..................initialisations
      LOOP
            OPENEDSERVICES := {Empiler, Depiler} ;
            WAIT() ;
            CASE RECEIVINGSERVICE OF
                  Empiler : ProcEmpiler() |
                  Depiler : ProcDepiler()
            END ;
      END ;
END Pile .
```

I.C.4. LES LIENS

DESCRIPTION

Les liens sont les supports des liaisons dynamiques et protégées entre les modules. Un module ne peut référencer un autre module qu'au travers d'un lien.

IMPLANTATION

Du point de vue du noyau, un lien est représenté par un nom unique de module (le module pointé) associé à un certain nombre de droits qui sont:

- Droit de duplication, permission de copier ce lien. C'est un booléen.
- Droits sur les services, le nombre de services du module pointé est limité à 16. Il est possible d'associer un droit sur chacun de ses services. Ces droits sont contrôlés au moment de l'émission du message à destination du service. Les valeurs possibles des droits sur les services sont au nombre de trois:
  - → 0: Aucune utilisation du service n'est possible.
  - → 1: Une seule utilisation du service est possible. Le droit est ramené à 0 après tout envoi de message à ce service.
  - → ∞: Pas de limitation dans l'utilisation de ce service.

Un lien se situera, soit à l'intérieur d'un message (entre l'émission et la réception du lien) soit dans l'environnement d'un module. Le noyau fourni bien entendu à un module des primitives permettant de restreindre les droits associés aux liens que ce module possède.

A la création, un module possède un lien sur lui-même que nous appellerons auto-lien. Ce lien est duplicable et tous les droits sur les services du module sont à ∞. Ceci permet à un module de transmettre ses références aux modules en relation directe ou indirecte (en ayant auparavant restreint les droits au minimum). Un second moyen d'obtention d'un lien est donné par le

gérant de module: Ce dernier retourne un lien vers le module créé en réponse à toute demande de création.

EXEMPLE

Dans l'exemple, chaque module ELEMENT possède un lien (de nom ElementSuivant) vers son successeur. Le module PILE possède un lien vers le premier élément constitutif de la pile. Pour simplifier l'exemple, nous n'avons pas utilisé explicitement le mécanisme de protection associé aux liens.

LANGAGE

Du point de vue du programmeur, un lien est repéré par un nom local du lien (type MODULEID du langage), qui est effectivement un index dans l'environnement du module. Les primitives du noyau relatives aux liens ne permettent d'utiliser que les noms locaux de liens. Le nom local du lien d'un module vers lui-même (autolien) est: MYSELF.

Nous donnerons en exemple le code du service CHAINER du module ELEM:

```
MODULE Element ;
...
VAR
      ElementSuivant : MODULEID ;
...
PROCEDURE ProcChainer() ;
BEGIN
ElementSuivant := RECEIVEDMESSAGE.FLien.NomModule ;
END ProcChainer ;
...
END Element.
```

I.C.5. LES MESSAGES

DESCRIPTION

Les modules OMPHALE communiquent par messages. Quand un module désire envoyer un message à un correspondant, il doit fournir

- Le destinataire du message. Le module fournit le nom local du lien qui repère le destinataire.

- Le nom du service destinataire. NERF contrôle que les droits inclus dans le lien invoqué permettent l'utilisation de ce service.

- Le contenu du message. Il y a recopie des informations transmises dans le message. C'est donc un envoi par valeur (on transmet les informations proprement dites) par opposition à un envoi par adresse (on transmet l'endroit où se trouvent les informations). Le message est de longueur quelconque et peut contenir n'importe quel type de donnée – y compris un nom de service ou un lien –. Seuls les liens sont interprétés lors de l'émission du message: NERF remplace alors le nom local de lien (qui n'a aucune signification hors du module) par son contenu réel. Le lien émis disparait alors de l'environnement de l'émetteur. La manœuvre inverse (remplacement du lien par le nom local du lien qui est rajouté à l'environnement du module destinataire) est faite automatiquement par NERF à la réception du message.

Un message est anonyme. Si le module expéditeur désire se faire reconnaître, il devra explicitement placer à l'intérieur du message un lien qui le désigne (éventuellement aussi un nom de service). Ce lien peut être obtenu par duplication du lien auto-identifiant.

L'envoi d'un message est asynchrone: l'envoi d'un message ne bloque pas l'émetteur. En fait, le message est stocké temporairement dans la zone d'émission et l'envoi n'est donc effectif qu'après la période d'activité de l'émetteur. C'est ainsi que tous les messages envoyés durant une période d'activité :

- Peuvent être annulés pendant cette période.

- Seront émis simultanément.

A cet effet, le programmeur peut désigner tout message envoyé lors de la période courante d'activité par un identificateur. Cet identificateur est retourné par la primitive d'envoi de message.

LES PRIMITIVES

Les primitives concernant les messages sont au nombre de trois:

- SEND permet l'envoi d'un message en précisant le module et le service destinataire ainsi que le contenu du message. L'envoi d'un message n'est effectif qu'à la fin de la période d'activité du module.
- DELETEMSG permet d'annuler l'envoi d'un message spécifié par son identificateur.
- DELETEALLMSG permet d'annuler tous les envois de message ayant eu lieu durant la période d'activité en cours.

La réception d'un message est automatique: Avant chaque période d'activité, le message ayant déclenché cette période est traité. Les liens en sont retirés, ajoutés à l'environnement du module et remplacés par leur nom local. Le contenu résultant est placé dans la zone de communication (variable RECEIVEDMESSAGE).

EXEMPLE

Nous allons détailler les mécanismes utilisés lors de l'envoi d'un message qui contient uniquement la référence à un quai. Un tel message comprend un lien (type MODULEID) et un nom de service (type SERVICEID). Ce format de message est typiquement utilisé pour transmettre une adresse de retour dans un message de requête. Le service DEPILER du module Element, reçoit de cette manière le moyen de retourner la valeur dépilée.

LANGAGE

FORMAT DE MESSAGE

L'échange de message entre deux modules ne peut donner un résultat positif que si le contenu du message est interprété de la même façon par l'émetteur et par le récepteur. Nous appellerons 'Format' d'un message le mode d'interprétation de ce dernier. La déclaration d'un format apparait comme la déclaration d'un type structure. Un format pouvant être commun à plusieurs modules, le programmeur sera amené à décrire les formats des

messages de manière indépendante des modules. Le langage permet ces descriptions à l'intérieur d'unités particulières: les unités de format. L'avantage de cette séparation format/module est triple:

- La description n'est pas répétée pour chaque module qui utilise le format. Ce qui donne un source plus compact et facilite la maintenace du logiciel.
- Le contrôle de cohérence entre les formats des messages émis et reçus est partiellement effectué par le compilateur.
- Les primitives liées aux formats des messages (telles que l'envoi par exemple) peuvent être générées automatiquement et en un seul exemplaire par le préprocesseur.

Dans l'exemple considéré, le format `FQui` sera déclaré de la manière suivante:

```
FORMAT FQui;
       STRUCTURE
       NomModule : MODULEID ;
       NomQuai : SERVICEID ;
       END ;
END FQui .
```

LES PRIMITIVES

A une exception près, l'utilisation des primitives de gestion des messages émis est peu novatrice: il s'agit simplement de procédures qui sont fournies par l' environnement de développement (ici un module MODULA2 baptisé OMPHALE).

Les opérations demandées par l'utilisation de la primitive SEND sont plus complexes: elles s'effectuent sur deux niveaux:

- Statiquement. Le préprocesseur transforme les appels de la primitive générique SEND en fonction du format de la variable-message qui est passé en paramètre. Cette transformation abouti à l'appel de la procédure créée lors de la déclaration du format.

- A l'exécution. La procédure typée d'envoi de message appelle à son tour une primitive SEND atypique (i.e. qui ne contrôle pas le type de ses paramètres) fournie par le module OMPHALE .

EXEMPLE.

Nous détaillerons le service DEPILER du module ELEMENT après avoir adopté les simplifications suivantes:

- Nous ne considérons pas la valeur retournée par la primitive d'envoi de message.
- Nous éludons les problèmes de typage de la variable RECEIVEDMESSAGE dûs à MODULA-2.

```
PROCEDURE ProcDepiler() ;
MESSAGE
      Retour : FEntier ;
VAR
      Valeur : INTEGER ;
      ModuleRetour : MODULEID ;
      QuaiRetour : SERVICEID ;
BEGIN
      ModuleRetour := RECEIVEDMESSAGE.NomModule ;
      QuaiRetour := RECEIVEDMESSAGE.NomQuai ;

      DEC(SommetLocal) ;
      Valeur := PileLocale [SommetLocal] ;

      Retour.Entier := Valeur ;
      SEND (Retour, ModuleRetour, QuaiRetour) ;
END ProcDepiler ;
```

Les opérations déclenchées par un appel à ce service seront:

→ Récupération de l'adresse (nom local et service) de retour.

→ Dépilage local et constitution du message de réponse.

→ Envoi du message de réponse.

I.C.6. LES PRIMITIVES DE GESTION DE L'ENVIRONNEMENT.

L'architecture OMPHALE offre, bien sûr, au module, la possibilité d'agir sur son environnement. Nous ne décrirons les primitives correspondantes qu'au niveau de l'architecture. En effet, celles-ci ,n'ont pas été utilisée dans l'exemple et de plus, leur mise en œuvre est complètement classique (appel de procédure du module OMPHALE).

DESCRIPTION

DUPLICATION DE LIENS.

L'appel de la procédure de nom DUPLICATELINK permet à un module de demander la duplication d'un lien. Le module fournit en paramètre le nom local du lien qu'il veut dupliquer (qui doit bien sûr être duplicable) et un booléen qui indique si la copie du lien obtenu est elle-même duplicable (dans ce cas l'original doit être duplicable) ou non. En retour, le module récupère un nouveau nom local de lien. Les droits associés aux services sont traités de la manière suivante:

- Les services à utilisations non limitées restent à utilisations non limitées dans l'original et dans la copie.
- Les services à utilisation unique restent à utilisation unique dans l'original et deviennent non utilisables dans la copie.
- Les services non utilisables restent non utilisables dans l'original et dans la copie.

RESTRICTION DES DROITS D'UN LIEN.

Il est possible pour un module de diminuer volontairement les droits associés à un lien qu'il possède. Il s'agit, soit des droits sur les services (SETNOUSE (rendre inutilisable) ou SETONEUSE (mettre à utilisation unique), soit des droits de duplication (SETNODUP)). Il faut évidemment que les droits initiaux soient plus forts que les droits demandés, sinon la demande est infructueuse.

- SETNOUSE, SETONEUSE. Le module spécifie le lien (au moyen de son nom local) et la liste des services (par un ensemble) concernés.

- SETNODUP. Le module précise le lien qu'il veut rendre non duplicable.

Ces primitives sont particulièrement utilisées lorsqu'un module désire rendre acceesible un ou plusieurs de ses quais à d'autres modules. Elles précédent alors l'envoi du duplicata de lien à l'intérieur d'un message.

I.D. FONCTIONNEMENT GLOBAL DU NOYAU.

En résumé, nous pouvons décrire le fonctionnement global de l'architecture OMPHALE par les étapes suivantes:

- Un module, lors de sa phase d'activité, traite le message qui l'a réveillé. Il en résulte l'émission d'un certain nombre de messages, une mise à jour du contexte local au module ainsi que le positionnement des quais sur lesquels ce module acceptera d'être réveillé à nouveau. Cette phase d'activité se termine par l'exécution d'une primitive WAIT.

- Le noyau NERF récupére les messages émis. Si un message contient des liens, ceux-ci sont repris depuis l'environnement du module. Ensuite NERF effectue le routage de ces messages vers les modules destinataires. Si l'un de ces messages arrive sur le quai 'ouvert' d'un module en attente, ce module devient prêt.

- NERF choisi alors l'un des modules prêts et prépare son exécution en lui fournissant le contenu du message et quelques autres informations. Si le message reçu contient des liens, ces derniers sont rajoutés à l'environnement du module et remplacés par un nom local.

De l'ensemble de ce fonctionnement il résulte :

- Une isolation complète entre l'exécution des modules et la mise en œuvre du noyau.

- Les liens ne sont jamais directement manipulés par les modules. Ce qui est un bon outil de protection.

I.E. CONCLUSION.

Nous avons décrit les bases de l'architecture OMPHALE:

- Structuration du logiciel en modules depuis la conception jusqu'à l'exécution.
- Les modules communiquent entre eux au moyen de messages (envoi asynchrone, messages anonymes). La destination d'un message est précisée sous la forme d'un couple (identificateur de module, nom de quai).
- La protection est acquise par deux mécanismes:
  - L'encapsulation des données à l'intérieur des modules.
  - Les liens qui permettent le nommage réciproque des modules, ne peuvent être manipulés directement par un module (mécanisme des capacités).

Selon cette architecture, un module est, après réception d'un message, complétement autonome et peut être complètement exécuté indépendemment des autres éléments logiciels. Ceci permet en particulier:

- L'exécution parallèle de l'ensemble des modules prêts.
- La simultanéité entre les opérations du noyau (routage des messages, préparation des modules en vue de l'exécution) et l'exécution d'autres modules.

Nous avons développé un langage spécifique à cette architecture. Ce langage a été construit à partir de MODULA2. Nous pouvons en retenir les particularités suivantes:

- Utilisation maximale du typage proposé par MODULA2. Nous avons dû, à ce niveau entamer une réflexion sur les problème de typage des messages (notion de format).
- Automatisation maximale des tâches fastidieuses (description du format d'un message à l'envoi, appel de procédure à distance).

- Intégration de l'interface au noyau dans un langage de programmation de haut niveau.

II. L'ARCHITECTURE DU SITE ZERO.

Le présent chapitre a pour objet de décrire l'architecture matérielle du site 0. Cependant, pour une bonne compréhension du rôle de chacun de ses éléments, il est nécessaire de connaître les idées de base qui ont inspiré la conception du site. C'est pour cela que nous débuterons par un descriptif de l'objectif visé, que nous prolongerons par l'explication du fonctionnement du site. Après cette étape, nous pourrons revenir sur la description fine de chaque constituant.

II.A. DESCRIPTION GENERALE DU SITE 0.

L'architecture du site 0 est intermédiaire entre un multiprocesseur avec mémoire commune et un multi-ordinateur (multicomputer), chaque processeur disposant d'une mémoire locale.

II.A.1. LES OBJECTIFS DU SITE.

Le site 0 est un prototype destiné à l'évaluation d'une architecture matérielle adaptée pour l'exécution performante de certains programmes que l'on peut qualifier, en général, de fortement modulaires. Les caractéristiques souhaitées du logiciel sont les suivantes:

- L'exécution globale du logiciel peut être effectuée en exécutant une ou plusieurs fois un certain nombre de modules.
- Un module est autonome: il contient tout les éléments nécessaires à son exécution.
- La taille d'un module est faible.
- L'action du système lors de l'activation d'un module peut être décomposée en trois étapes (initiale, intermédiaire et finale). Les deux étapes initiale et finale regroupent les opérations système nécessaires avant et après l'exécution du module. L'étape intermédiaire est l'étape d'exécution proprement dite du module. C'est le seul moment on l'on peut exécuter du code non système (code utilisateur).

- A des fins de protection, un module sera structuré en segments qui peuvent être le support de dispositifs de protection. C'est le système qui est chargé de la mise en oeuvre de cette protection.

On devine déjà, au vu de ces contraintes, les idées fortes de l'architecture:

- La dichotomie supposée du logiciel permettra un découpage fonctionnel de l'architecture. Le site aura un coté système et un coté utilisateur.

- La protection sera assurée par un adressage segmenté et bien entendu gérée par le coté système de l'architecture.

- Une partie de la mémoire, structurée en blocs de faible taille, servira à stocker les modules. Cette mémoire sera accessible des deux cotés de l'architecture.

II.A.2. LES COMPOSANTS DU SITE.

Le site 0 est composé de deux processeurs, un processeur noyau (noté PN) et un processeur de calcul (noté PC), chacun d'entre-eux est doté des éléments lui permettant de fonctionner (mémoire propre ROM et RAM). Ces deux processeurs partagent le reste du site 0, à savoir des unités de gestion mémoire (UGM) et de la mémoire commune découpée en blocs. Cette architecture, qui impose un fort couplage entre PC et PN est justifiée par une grande simplicité de réalisation et le peu d'effet des conflits.Pour les besoins du développement et des entrées-sorties, nous avons également ajouté des circuits d'entrées-sorties sur les deux processeurs. Une particularité du matériel du site 0 est que les UGM ainsi que les blocs mémoire sont connectés dynamiquement à l'un ou l'autre des processeurs selon les besoins.

II.A.3. LES CHEMINS DE DONNEES DU SITE.

Les processeurs ont accès aux données (au sens large du terme) du site 0 par les chemins suivants (Fig. II-1):

• Le processeur PN peut accéder par l'intermédiaire de son bus local BNI:

→ Aux éléments qui lui sont propres (mémoire privée, entrées-sorties).

→ Directement (sans passer par les UGM) aux blocs mémoire.

→ Aux registres internes des unités de gestion mémoire UGM (à des fins de programmation).

• Le processeur PC peut accéder, grâce à son bus local BCI:

→ A sa mémoire privée.

→ A des éléments d'entrée/sortie.

→ A travers les unités de gestion mémoire UGM, au bus BC, donc aux blocs mémoire.

II.B. FONCTIONNEMENT GLOBAL DU SITE.

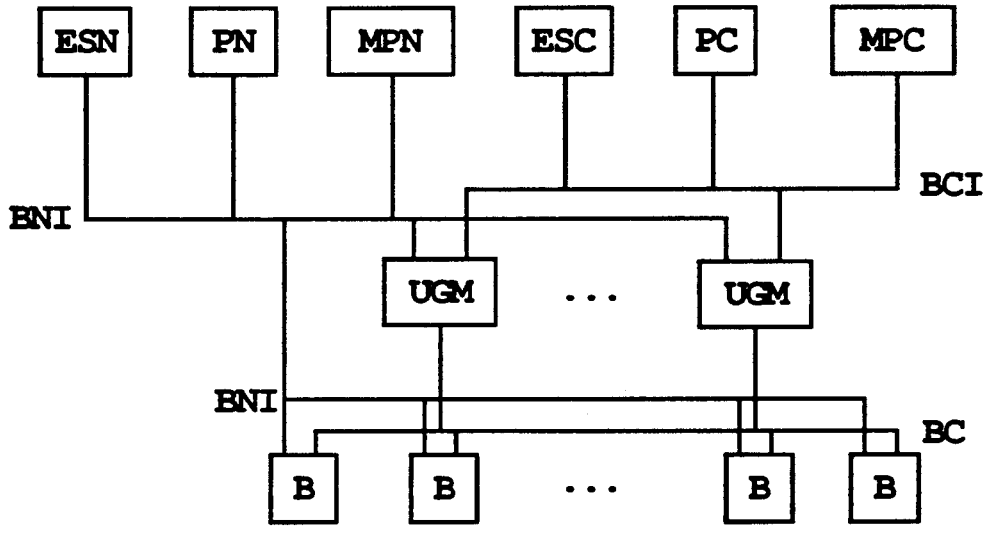
Nous allons maintenant décrire les différentes étapes de traitements qui composent la période d'activité d'un module. Il faut garder à l'esprit que, pour tirer parti le plus possible du parallélisme permis par la présence de deux processeurs, il y aura simultanéité entre ces étapes. L'étape utilisateur d'un module sera - on l'espère - souvent contemporaine à l'étape initiale ou finale d'un autre module. Nous ne mentionnons ici que les temps où le module est actif. Nous ignorerons donc à la fois les périodes d'attente (entre deux périodes d'activités) et les temps morts entre les phases internes à la période d'activité.

Nous aurons les hypothèses suivantes:

• Un module contient toutes les données nécessaires à son exécution. En particulier, ceci interdit à un module

- D'effectuer des entrées-sorties.
- Le partage de données avec un autre module. A fortiori, il ne pourra utiliser des variables globales.

- La partie usager (écrite par l'utilisateur) d'un module ne peut être initiée que si elle a été l'objet d'un traitement effectué par le système (fourniture du message reçu, etc... voir le chapitre "l'architecture OMPHALE"). Hormis les structures de données propres au noyau, ce traitement ne pourra demander ou modifier que des données locales au module.



ESN: Entrées/sorties PN
PN : Processeur Noyau
MPN: Mémoire privée PN
ESC: Entrées/sorties PC
PC : Processeur Calcul
MPC: Mémoire privée PC
UGM: Unité de gestion mémoire
B : Bloc de mémoire
BNI: Bus interne PN
BCI: Bus interne PC
BC : Bus C d'accès aux blocs mémoire

Fig. II-1. Schéma général du site 0.

A partir de ces hypothèses, on voit apparaître à l'intérieur de chaque module, une partition du travail : Les traitements et données relatifs à l'activité système d'une part, à la partie usager d'autre-part. On soulignera également l'indépendance complète entre les modules. On peut dès lors imaginer de spécialiser les processeurs PN et PC respectivement dans les tâches système et usager. C'est dans ce sens que le site 0 a été conçu.

L'étape utilisateur, qui consiste en une seule phase dite phase d'exécution est traitée sur PC, qui dispose de toutes les données qui lui sont nécessaires dans un des blocs mémoire. La protection voulue lors de la définition générale du site n'est opérante que lors de cette étape.

La partie système est effectuée sur PN et se décompose en deux étapes:

- L'étape initiale est elle même découpée en deux phases: la phase de préparation et la phase de chargement. La préparation regroupe les travaux système étrangers aux unités de gestion mémoire. En effet, pour un fonctionnement normal durant la phase d'exécution, les circuits de gestion mémoire demandent à être programmés durant une phase appelée phase de chargement. Chronologiquement ceci se passe entre la phase de préparation et d'exécution d'un module. Cette phase est attribuée à PN car il s'agit essentiellement d'une tâche du système (liée d'ailleurs fortement à la gestion de la mémoire).

- L'étape finale consiste en une seule phase de sauvegarde qui assure le déchargement des UGM (sauvegarde de la description des segments) également assurée par PN.

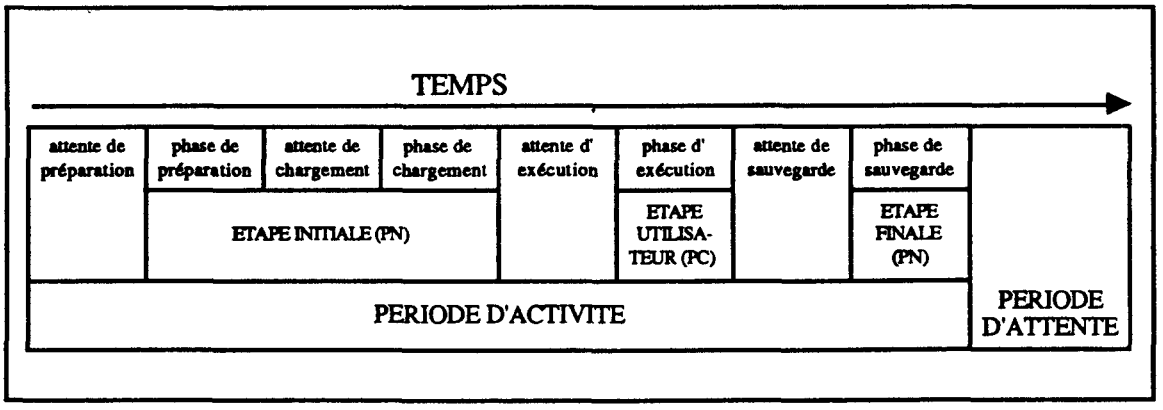


Fig. II-2. Cycle d'exécution d'un module.

En résumé, un module sera dans l'ordre:

- Préparé par PN.
- Les registres de l'UGM correspondante seront chargés par PN.
- Le module sera exécuté par PC.
- L'état de l'UGM pourra être relu par PN afin de contrôler le bon déroulement de l'exécution.

Chacune de ces phases sera détaillée par la suite.

II.B.1. SCHEMA D'EXECUTION D'UN MODULE.

Nous avons résumé dans le schéma présenté ci-dessous la correspondance entre les composants matériels d'un site et les éléments constitutifs d'un module tout au long d'une période d'activité de ce module.

L'ensemble de gauche représente l'ensemble des modules du site. Le noyau peut, dans une phase de préparation, réveiller un module. Les circonstances de ce réveil ne sont pas considérées dans ce chapitre. A la suite de la préparation, nous avons symbolisé le chargement d'une UGM. L'ensemble de ces UGM est figuré par l'ensemble des cercles. La couleur du cercle est caractéristique de l'état de l'UGM (libre, chargée, exécutée). Ainsi le chargement prend une UGM libre et la rend chargée. Viennent ensuite l'exécution par PC (à droite du schéma) puis la sauvegarde de

l'UGM. A la suite de la sauvegarde, nous récupérerons donc le module qui entame une nouvelle période d'inactivité.

Nous obtenons ainsi une description chronologique et 'topologique' (répartition des opérations dans les sous-ensembles de la machine) des phases successives. La topologie sera exploitée lors de l'étude du fonctionnement des UGM. Nous voyons que toutes les interactions entre les deux processeurs passent par l'intermédiaire des unités de gestion mémoire.

Nous avons ignoré les blocs mémoire qui, nous le verrons plus tard, induisent peu de conflits entre PC et PN.

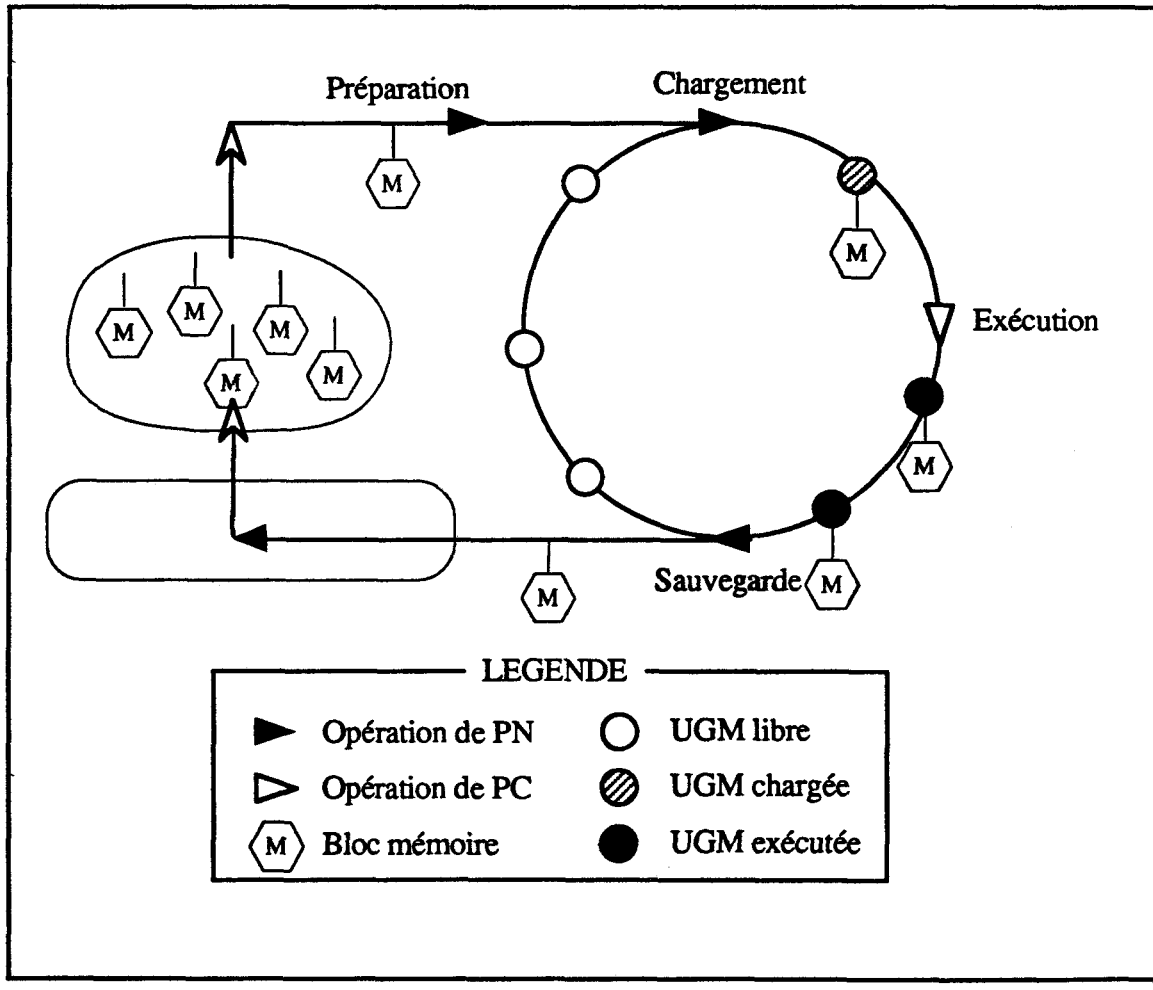


Fig. II-3. Schéma d'exécution, composants matériels et logiciels.

LA PHASE DE PREPARATION.

Le processeur PN réalise la préparation des modules. Pour ce faire, il dispose de toutes les informations nécessaires pour chaque module: localisation du module, structuration du module en segments, données systèmes liées, etc... Quand le système a décidé de lancer une période d'activité pour un module, il doit mettre à jour ses tables, éventuellement, en cas de création d'un module, trouver un bloc de mémoire libre et l'organiser, fournir les paramètres au module. Il prépare également la liste des segments destinée à être chargée dans l'UGM. Lors de cette phase, PN utilise le BUS BNI, soit pour accéder à ses éléments périphériques, soit pour manipuler directement (i.e. sans protection mémoire) les données situées dans le bloc mémoire.

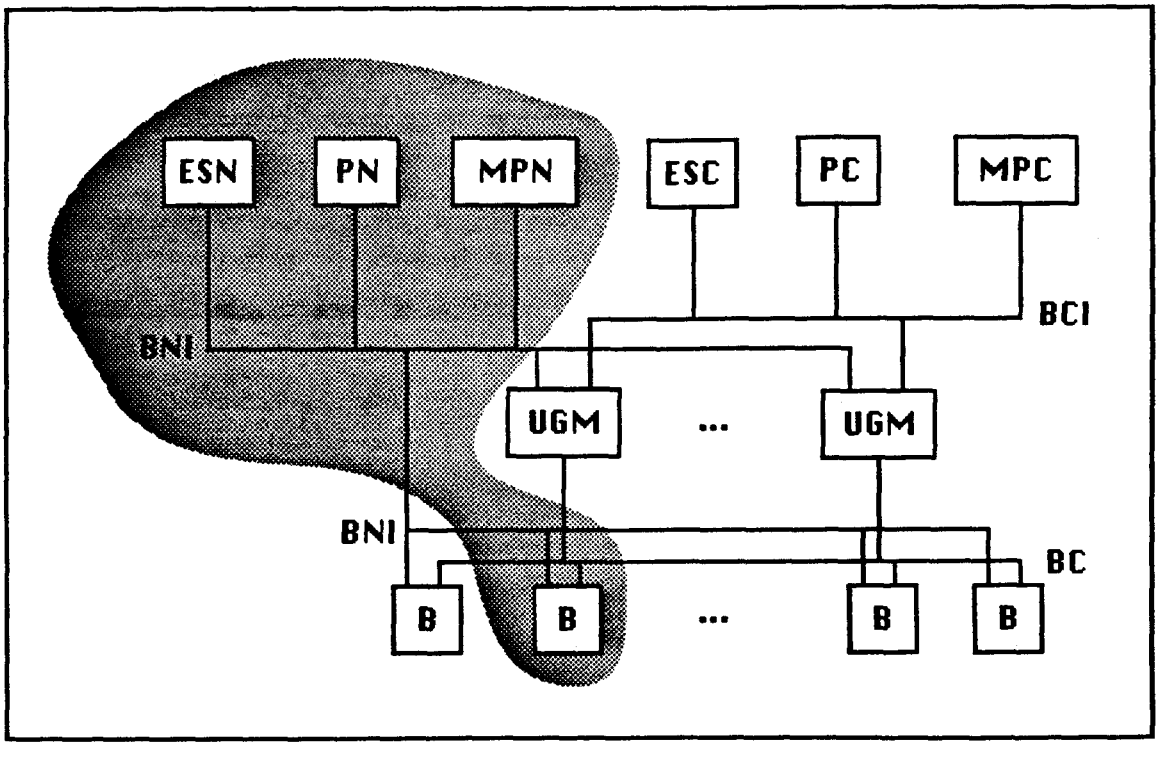


Fig. II-4. Les éléments actifs durant la préparation.

LA PHASE DE CHARGEMENT.

A la suite de la préparation, PN peut charger les registres d'une UGM libre avec la liste préparée précédemment. Implicitement, il attribue l'UGM au module jusque sa phase de sauvegarde. Cette étape est distincte

de la précédente car elle demande l'acquisition d'une UGM. Cette acquisition peut être refusée si aucune UGM n'est libre (PN devrait alors enchaîner soit sur une autre préparation, soit sur une sauvegarde). PN n'utilise alors que le bus BNI que pour lire la liste dans sa mémoire propre MPN et écrire dans les registres de l'UGM (opération d'entrée-sortie).

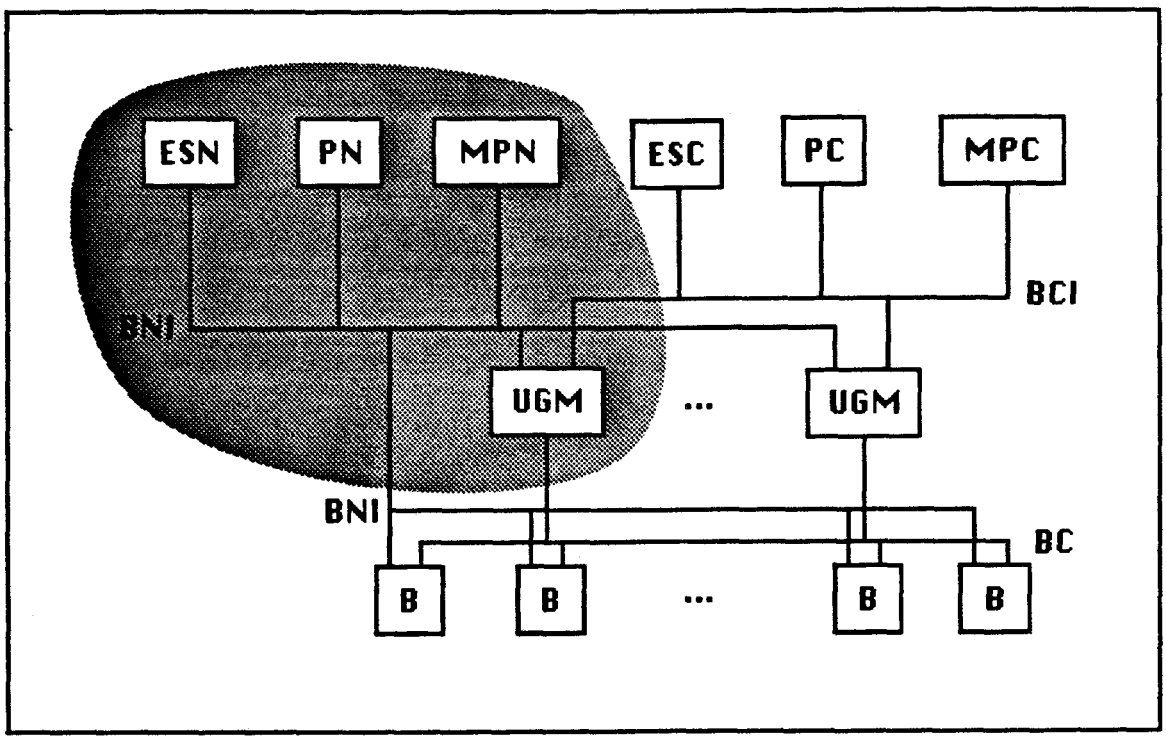


Fig. II-5. Les éléments actifs durant le chargement ou la sauvegarde.

LA PHASE D'EXECUTION.

La phase d'exécution est effectuée par PC. En fin de phase de traitement (i.e. lors de l'exécution de la primitive de terminaison du programme utilisateur), PC abandonne l'UGM qu'il possédait, puis essaie d'obtenir une nouvelle UGM chargée. En cas de succès, PC dispose donc d'une UGM qui repère un module prêt pour une nouvelle phase de traitement. Nous verrons que le module accédé et le bloc utilisé dépendent de la programmation des registres de l'UGM.

Les chemins de données utilisés lors de cette phase sont les bus BCI (accès à la mémoire privée) et BC (accès à la mémoire de bloc).

Notons que la communication entre le programme utilisateur proprement dit et le système ne peut se faire que de manière différée et indirecte. Différée parce que les opérations demandées par l'utilisateur ne seront prises en compte que lors de l'étape finale, donc après la terminaison de la période d'activité en cours. Indirecte parque que cette communication se fait uniquement par écriture sous un format prédéfini dans le bloc mémoire, à charge pour le système de le relire au cours de l'étape finale.

Il est possible de permettre une communication immédiate et directe en complétant l'architecture avec un dispositif de communication entre les deux processeurs (l'installation d'une FIFO avait été envisagée mais s'est avérée inutile pour le système auquel nous avons destiné le prototype).

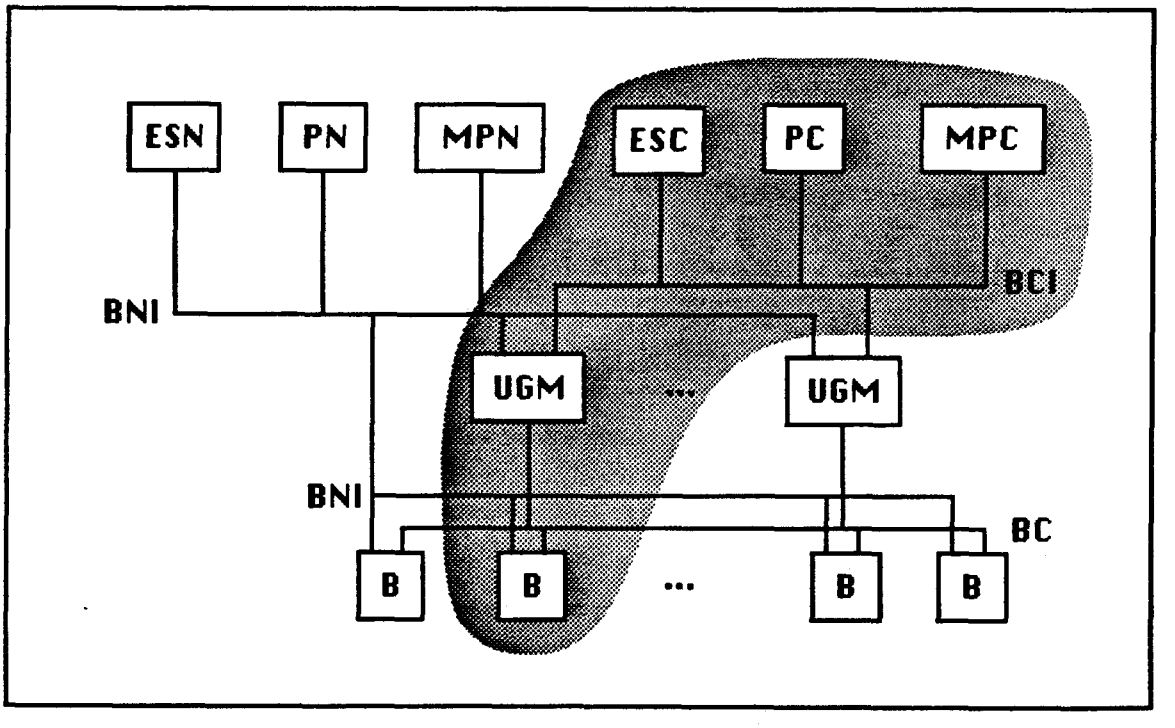


Fig. II-6. Les éléments actifs durant l'exécution.

LA SAUVEGARDE.

PN essaie d'acquérir une UGM 'exécutée' (donc indirectement le bloc mémoire et le module associé). En cas de succès, il peut alors traiter les requêtes systèmes émises lors de la phase d'exécution du module,

éventuellement sauvegarder l'état de cette UGM, ranger le module dans une mémoire plus stable et/ou moins couteuse, etc.... L'UGM peut alors être libérée. Les éléments mis en jeu lors de cette phase sont les même que pendant le chargement et la préparation (utilisation uniquement du bus BNI).

II.C. DESCRIPTION DES PRINCIPAUX ELEMENTS

II.C.1. LES PROCESSEURS.

Les processeurs PC et PN sont des Zilog Z8001[Zilog81] , processeurs 16 bits disposant d'un adressage segmenté. Pour PC, il fallait en effet un processeur (ou un ensemble processeur plus unité de gestion mémoire) qui offre un nombre de segments conséquent et un mode de fonctionnement protégé qu'on ne trouvait (à l'époque du choix) ni sur le 8086 d'Intel, ni sur le 68000 de Motorola. La protection imposait également de pouvoir annuler complètement les conséquences d'un accès illégal. Il fallait donc un processeur qui supporte les déroutements. Le choix de PN est uniquement un choix d'uniformité, les contraintes citées plus haut n'étant pas applicables sur PN qui est censé n'exécuter que du code certifié. L'uniformité simplifie autant la chaîne de développement de programme que les connexions matérielles des circuits annexes (périphériques, UGM).

Ces processeurs disposent chacun de mémoire locale (RAM et ROM) qui leur permet un fonctionnement indépendant de la mémoire de modules. Néanmoins l'accès à cette mémoire est différent pour chaque processeur. PN voit un espace mémoire uniforme qui comprend à la fois la mémoire locale et la mémoire de modules. Par contre, PC ,qui utilise un adressage segmenté, peut adresser sa mémoire locale dans le segment de numéro 0 et uniquement lorsqu'il est en mode superviseur. Ainsi, un programme utilisateur ne peut pas attenter aux données qui s'y trouvent. Un numéro de segment non nul est interprété par l'UGM et caractérise un accès à l'un des blocs de mémoire.

II.C.2. LES UNITES DE GESTION MEMOIRE.

OBJECTIFS

Les unités de gestion mémoire ont deux rôles: Elles assurent que le programme usager ne va pas intervenir sur des données qui lui sont étrangères; en ce sens, elles ont un rôle de protection. D'autre-part, quand une adresse correcte est fournie par PC, elle traduisent cette adresse "logique" (couple segment, déplacement) en adresse physique; elles ont alors un rôle de traduction. L'ensemble de ces deux fonctions est réalisé par un seul type de circuit V.L.S.I. : le Zilog Z8010 MMU [Zilog82] . Une UGM est constituée par deux circuits MMU.

LA SEGMENTATION.

Un programme qui s'exécute sur un processeur utilisant l'adressage segmenté ne voit plus un espace d'adressage homogène, mais une collection de segments. Chaque segment représente un espace mémoire contigu (tous n'ont pas forcément la même taille) auquel on a attribué certains droits. A chaque accès mémoire, l'adresse fournie est un couple (numéro de segment - déplacement). Le matériel compare alors les droits du segment désigné et l'état actuel du processeur ainsi que la taille du segment et le déplacement désiré. Ces deux comparaisons permettent de déterminer si l'accès mémoire est légal ou non. Dans l'affirmative, Il faut alors opérer la traduction de l'adresse à partir de la première adresse (ou base) du segment. Bien entendu, l'application de ce mécanisme implique que l'on ai "décrit" d'une manière ou d'une autre l'ensemble des segments accessibles par le programme. Les avantages apportés par ce schéma d'adressage sont nombreux:

Contrôle des accès: La protection matérielle mise en oeuvre permet, par une description assez fine des segments, la protection en lecture, écriture ou exécution d'un nombre appréciable de zones mémoires. Cette protection a principalement été utilisée pour interdire à un module (programme usager) l'altération de données appartenant au noyau ou à d'autres modules. Nous verrons qu'elle évite également l'accès simultané à un même bloc mémoire par les deux processeurs PC et PN.

Relocation: Le mécanisme de traduction apporte l'indépendance entre les adresses logiques (i.e. vues par le programmeur) et la localisation physique effective des objets. Il devient possible de translater un ou plusieurs segments en ne devant corriger que leur description. Cette possibilité facilite en particulier la gestion mémoire des blocs, qu'hélas nous n'avons pas approfondie.

Partage: Plusieurs processus peuvent partager les mêmes objets en mémoire avec des droits différents. Le principe OMPHALE interdisant le partage d'objet entre plusieurs modules, cette caractéristique ne sera mise à profit que pour le partage de données entre le noyau et le programme utilisateur d'un module.

LE CIRCUIT DE GESTION MEMOIRE.

Les unités de gestion mémoire sont des circuits VLSI distribués par ZILOG sous la référence Z8010. Elles assurent à la fois le contrôle de validité des adresses fournies par le processeur sur lequel elles sont connectées, ainsi que leur traduction en adresse physique. Ceci suppose:

1) Que l'UGM connaisse le type d'accès demandé.

2) Que l'UGM connaisse les droits et la taille de chaque segment.

3) Pour la traduction d'adresse, l'UGM n'a besoin que de l'adresse de base de chaque segment.

Le point 1 est réalisé par l'observation des signaux de commande émis par le processeur (c.f. Câblage des UGM).

Les points 2 et 3 demandent la programmation du circuit de gestion mémoire. Pratiquement, il suffit de charger les registres de description de segment avec les valeurs adéquates (chaque MMU dispose de 64 de ces registres; il suffit donc de deux MMU pour supporter les 128 segments adressables par le processeur). Le codage adopté pour ces registres est donné figure II-7.

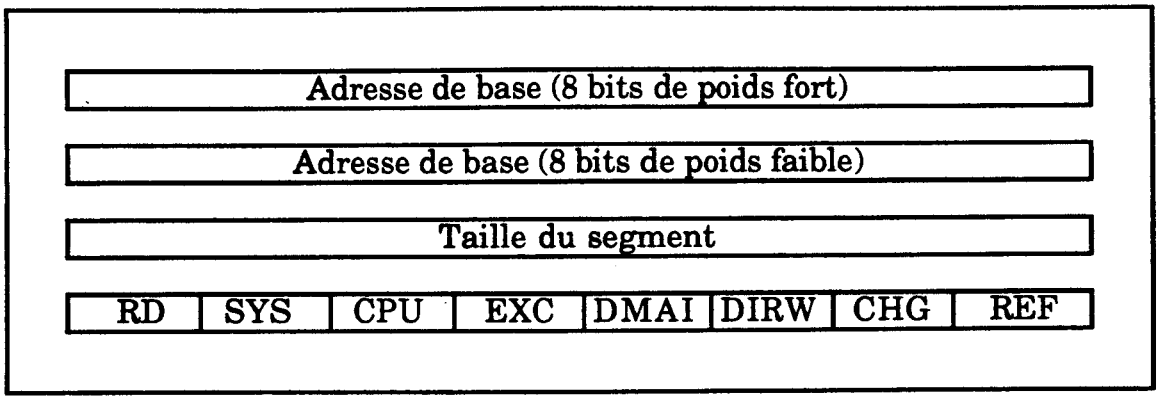


Fig. II-7. Registres de description d'un segment.

Nous avons utilisés les droits:

RD: Accès en lecture seulement.

SYS: Accès uniquement en mode système.

EXC: Accès uniquement en exécution.

Le chargement des registres est effectué par l'intermédiaire d'instructions spéciales d'entrées-sorties. Le problème qui se pose alors est de permettre la programmation des UGM par PN ainsi que leur utilisation (non simultanée) par PC. La solution apportée est une solution cablée. Pour l'expliciter, nous commencerons par décrire le câblage "standard" du circuit (programmation et utilisation par un processeur unique) avant de montrer les variations que nous y avons apportées.

CABLAGE STANDARD DES UGM.

Le câblage standard du Z8010 est donné par le schéma suivant:

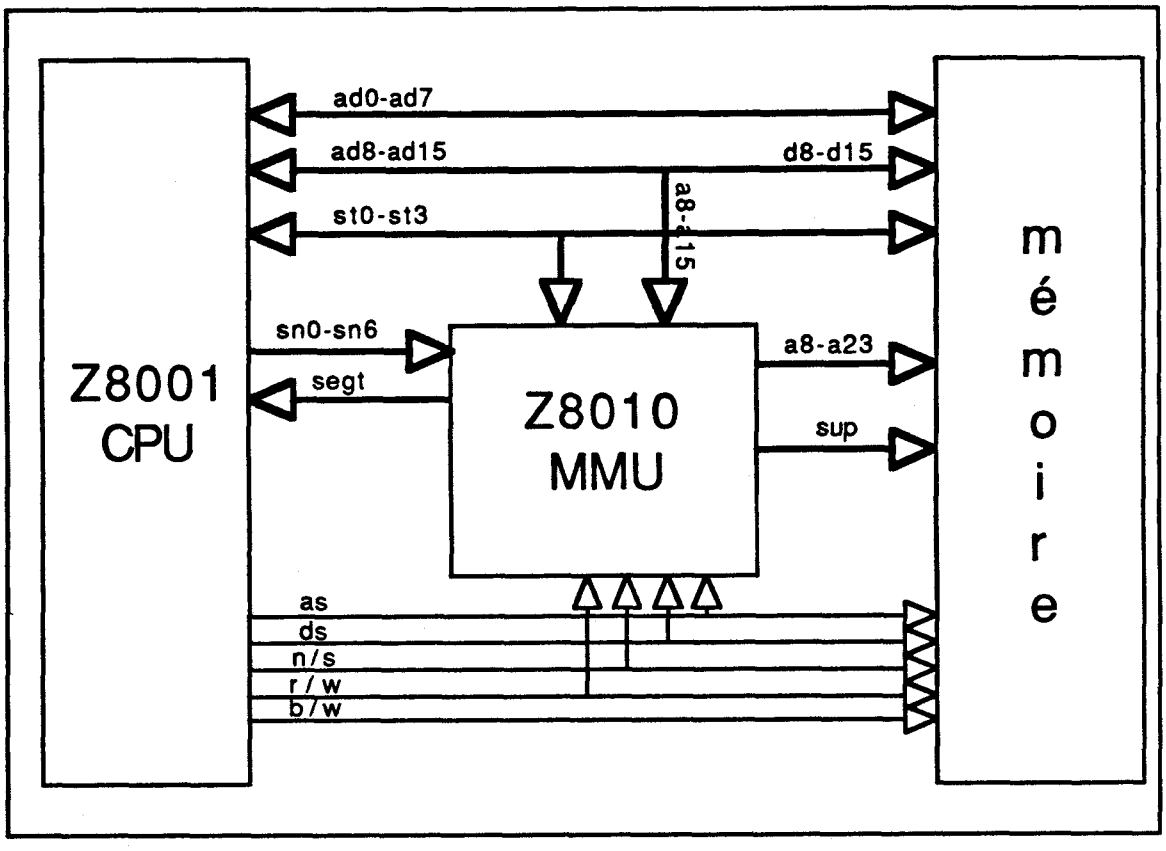


Fig. II-8. Câblage standard des MMU Zilog Z8010.

Nous isolerons principalement:

- Les lignes de commandes qui assurent la synchronisation des autres signaux. Deux lignes R/W et N/S permettent de connaître partiellement l'état courant du processeur (lecture ou écriture, mode normal ou système).

- Les signaux d'état du processeur (ST0 à ST3) exhibent le type d'accès demandé par celui-ci (Data memory request, Stack memory request, Instruction fetch...).

- Le signal SEGT permet à l'UGM de signaler une violation d'accès en provoquant un déroutement (TRAP) du processeur.

- Le numéro de segment (SN_0 à SN_6), partie de l'adresse logique.
- Le bus d'adresses/données est un bus multiplexé qui peut, comme son nom l'indique transmettre soit des adresses, soit des données. Nous remarquons que seul les 8 bits supérieurs d'adresses-données sont utilisés par le Z8010.

D'un autre coté, l'UGM doit également traduire les adresses logiques en adresses physiques. Elle pilote pour cela un bus d'adresses d'une largeur de 16 bits. Un signal supplémentaire SUP permet d'interdire une opération mémoire (d'écriture en particulier) illégale.

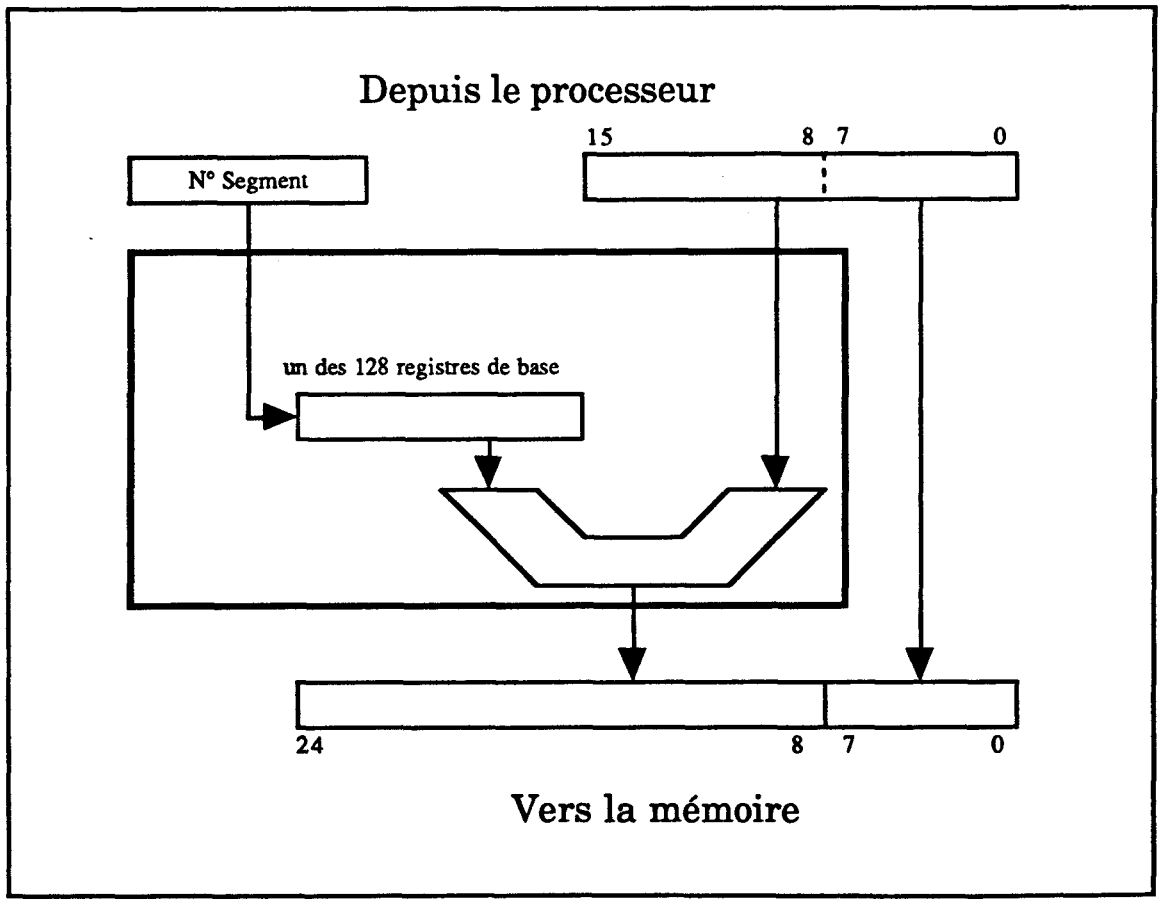


Fig. II-9. Mécanisme de traduction d'adresse des MMU.

Notons par ailleurs que les 8 bits de poids faible du bus multiplexé ne sont pas connectés à l'UGM (Fig II.9) et ne peuvent être, par conséquent, ni lus, ni traduits par cette dernière (ils sont par contre envoyés directement à la mémoire). Ainsi, l'UGM ne peut travailler que sur des

quanta de 2^8 = 256 octets: Les tailles et les bases des segments seront multiples de 256.

L'INSTALLATION EFFECTUEE.

Nous avons modifié le schéma standard d'installation pour répondre à deux problèmes: La taille du quantum et le partage entre les deux processeurs.

La conception du prototype pour un système différent à structure de domaine et adressage par capacités avait abouti à désirer une taille de quantum de 8 octets (un système comme Hydra [Levy84 :the Hydra system] confirme la petite taille des objets).

A posteriori, l'expérimentation de l'implantation d'OMPHALE allait nous montrer que les segments ne devaient pas être de grande taille. En effet, le principe d'encapsulation d'un objet dans un module restreint la taille de sa représentation (au moins pour les exemples que nous avons traités) car il oblige généralement à fractionner en éléments plus petits et plus faciles à spécifier. En outre, la contrainte de taille limite pour un module était assez contraignante (Du fait du coût de la mémoire utilisée, donc du peu de mémoire disponible, et du nombre de modules simultanés à considérer pour une expérimentation fructueuse). De fait, les structures de données les plus importantes sont les structures même du noyau celles-la même qui sont locales à PN. Cette constatation nous a incité à réduire la taille du quantum de 256 à 8 octets.

Pratiquement, il suffit de réduire le nombre de bits ignorés par l'UGM de 8 à 3. Cette manoeuvre a deux conséquences mineures: D'abord, l'espace adressable passe de 16 méga-octets à 512 kilo-octets (les adresses physiques sont alors représentées sur 19 bits). Ensuite, il y a une légère modification du dialogue entre ce circuit et le processeur du fait du décalage des lignes du bus. Cette modification a d'ailleurs demandé la réalisation d'un dispositif spécifique en composants discrets.

Le deuxième problème est nettement plus délicat. Il s'agit en effet de rendre l'UGM accessible par les deux processeurs (PN en programmation et PC en utilisation) alors que les deux modes utilisent les mêmes chemins de données. Nous avons été amenés à installer un commutateur

sur ceux-ci (Fig. II-10.). Ce dernier étant piloté par la circuiterie de partage d'UGM qui connait l'état (connecté à PN, connecté à PC ou non connecté) de chaque unité. Notons que ce dispositif n'a dû être installé qu'entre les UGM et les processeurs et non entre les UGM et la mémoire de modules: à un moment donné, une seule UGM est sensée fournir des adresses physiques sur le bus BC: La technique classique "trois états" permet de résoudre ce problème.

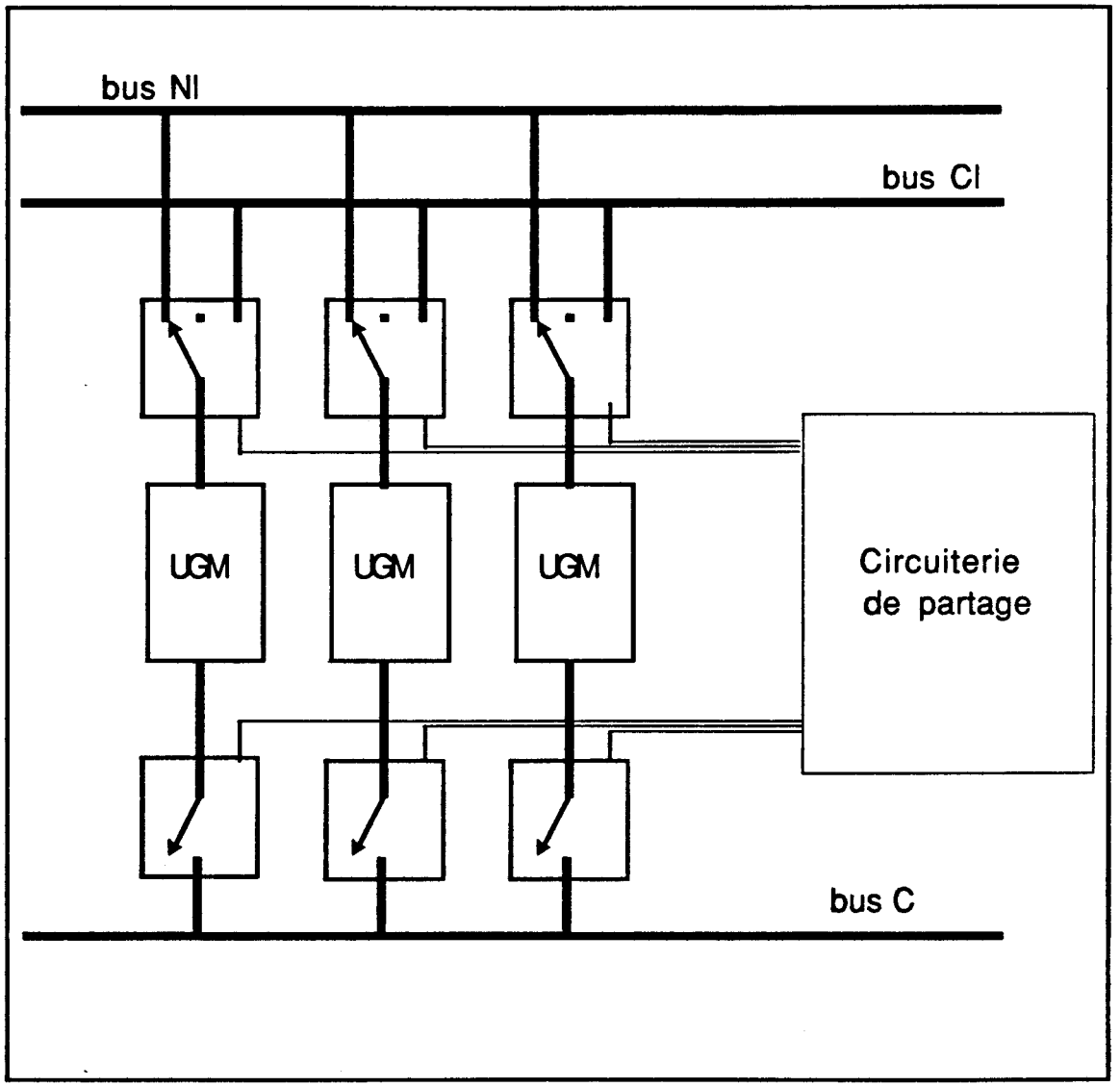


Fig. II-10. Commutation des UGM.

Nous avons ainsi pu connecter des unités de gestion mémoire de manière modulaire - donc en nombre quelconque - sur le site 0.

II.C.3. LA MEMOIRE DE MODULES.

OBJECTIFS.

La mémoire de modules est un autre élément important du site 0. Elle a pour rôle de mémoriser l'ensemble des modules avec les informations nécessaires à leur exécution. Nous pouvons dégager ses caractéristiques principales:

• Découpage en bloc. Cette mémoire est constituée d'un certain nombre de blocs banalisés et de taille identiques. Un module devra totalement être inclus dans un bloc.

• Double accès. Un bloc de mémoire sera manipulé à la fois par PN (préparation des modules, gestion mémoire) et par PC (exécution). Il faudra donc prévoir un chemin de données jusqu'à chacun de ces deux processeurs. Il ne s'agit toutefois pas de construire une mémoire permettant un accès simultané des deux cotés: le fonctionnement du site 0 est tel que l'on peut assurer l'absence de conflit. Une logique d'arbitrage lourde et complexe est alors superflue. Qui plus est, l'attribution d'un bloc à l'un ou l'autre des processeurs peut être faite pour la durée d'une opération (préparation, exécution) - donc de manière macroscopique - et uniquement à l'initiative de PN. Cette remarque n'a d'ailleurs pas été exploitée car nous avons pu construire une mémoire partageable sur la base (microscopique) d'un accès mémoire (au détriment de la sûreté de fonctionnement: on ne détecte pas les conflits d'accès).

D'autre-part, le caractère expérimental du site 0 nous a amené à désirer un maximum de souplesse dans la constitution de cette mémoire. En particulier, il était souhaitable de pouvoir faire varier le nombre ainsi que la taille des blocs. Nous avons heureusement pu bâtir cette mémoire de telle sorte que ce paramètrage puisse être totalement pris en charge par logiciel.

REALISATION.

Là encore, nous avons réalisé des éléments de base (éléments mémoire notés EM) qui permettent de constituer - par juxtaposition - des blocs variables en nombre et en taille. Ceci est possible car il n'est pas

nécessaire de concevoir des liaisons transverses entre les EM. Un EM est constitué de 8 Kilo-octets de mémoire statique et d'un commutateur qui connecte l'EM sur le bus BNl, sur le bus BC ou sur rien. La décision de commutation est prise simplement par le décodage des adresses provenant de chacun des bus.

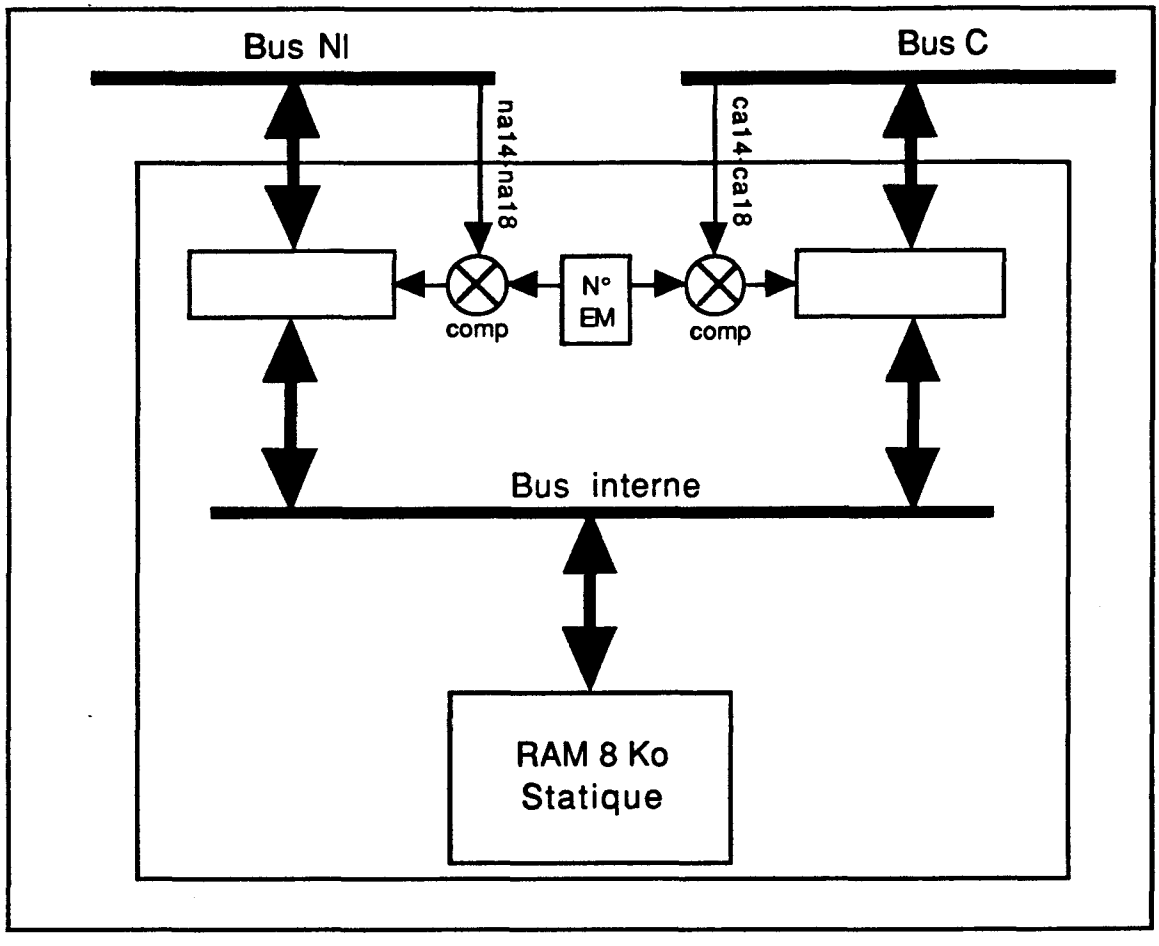


Fig. II-11. Constitution interne d'un élément de mémoire.

On utilise ici le fait que les adresses fournies par les processeurs ne peuvent être erronées. L'attribution des blocs aux processeurs est en fait indirectement réalisé par la programmation (par PN) et l'allocation des UGM (grâce à la circuiterie de partage).

Nous pouvons ainsi disposer d'au maximum 32 EM de 8 Kilo-octets. Ces EM peuvent être groupés pour former des blocs de taille multiple de 8 Kilo-octets. Il n'y a aucune opération explicite d'allocation.

La faible taille de la mémoire est due à la technologie utilisée lors de la conception du prototype. Cette limite n'hypothèque ni le principe de conception du site 0, ni les expérimentations qui ont été effectuées.

II.C.4. LA CIRCUITERIE DE PARTAGE DES UGM.

OBJECTIFS.

La circuiterie de partage des UGM est l'élément matériel qui permet:

1) De prendre en compte les requêtes et les libérations d'UGM par chacun des deux processeurs.

2) D'en déduire l'état de chacune des UGM et conséquemment, de piloter le commutateur qui réalise la connexion de celles-ci à l'un ou à l'autre des processeurs.

Nous remarquons que ce dispositif est le seul moyen de synchronisation entre PC et PN. Par exemple, c'est lorsque PN se verra refuser l'allocation d'une UGM parce qu'elles sont toutes en attente de sauvegarde qu'il s'apercevra qu'il vaut mieux commencer à effectuer des sauvegardes.

CYCLE D'UTILISATION DES UGM.

En observant le fonctionnement des UGM, on remarque que chacune d'entre elles suit un cycle, principalement confondu avec le cycle d'exécution d'un module. On peut ainsi distinguer les phases de chargement, d'exécution et de sauvegarde avec les phases intermédiaires (chargé, exécuté et libre). Il y a en effet une association stricte entre module et UGM juste avant le chargement et dissociation immédiatement après la sauvegarde. Une représentation simple de ce cycle peut être donné sous la forme d'un l'automate (Fig. II-12).

D'autre-part, comme nous considérons que l'attribution des UGM s'effectue selon une séquence stricte, nous pouvons établir la même partition sur l'ensemble des UGM installées sur le site. En d'autres termes, la description des états faite précédemment pour **une** UGM sur un intervalle de temps reste valable pour **l'ensemble** des UGM à un

instant donné. Nous pourrions ainsi, par exemple avoir l'évolution suivante (Fig. II-13 et II-14)

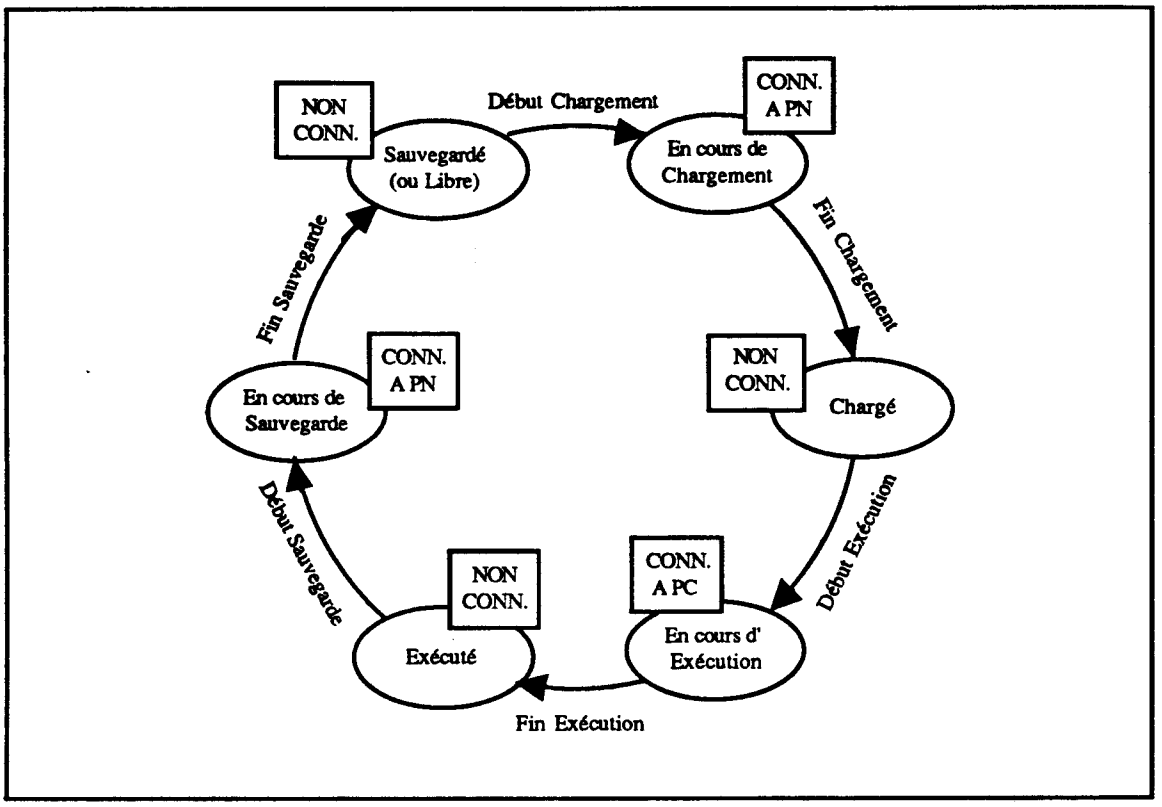


Fig. II-12. Cycle d'utilisation d'une UGM.

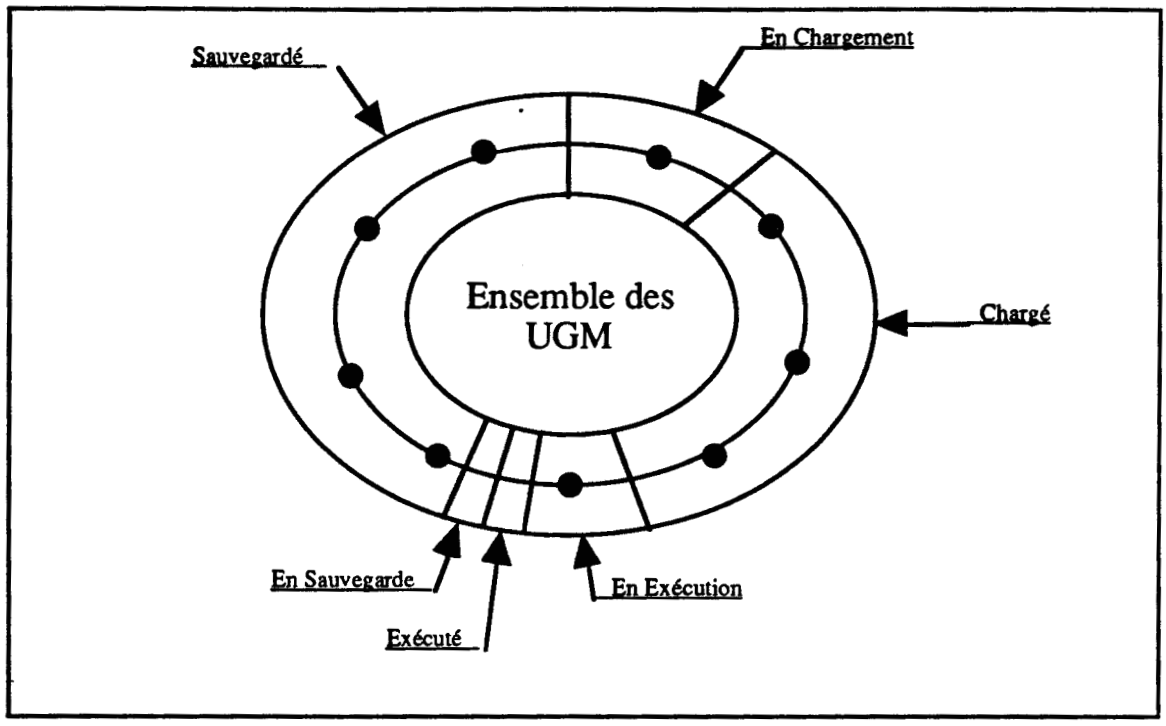


Fig. II-13. Partition de l'anneau à l'instant T. L'unicité du processeur PC implique qu'il n'y a pas plus d'une UGM en exécution à tout instant. De même pour PN; Le nombre d'UGM en cours de chargement ou de sauvegarde ne peut dépasser 1.

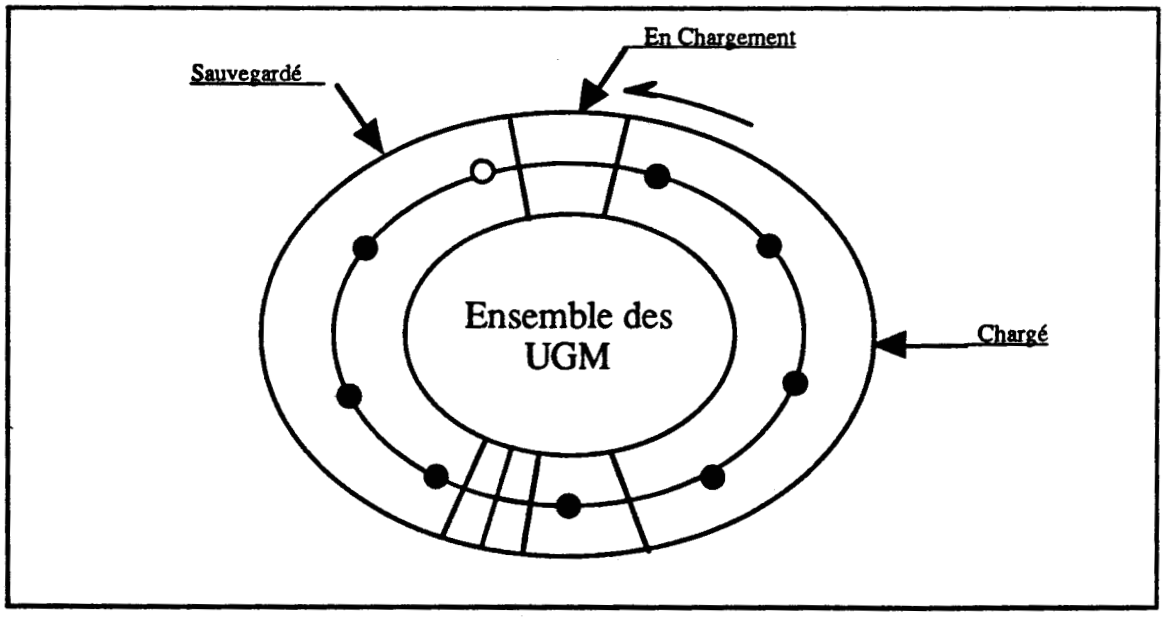


Fig. II-14. Partition après une "Fin de chargement" opérée par PN. La zone des UGM chargées s'est agrandie d'une unité. La prochaine UGM à être chargée sera l'unité blanche. La flèche périphérique indique le sens de progression de chaque zone.

COMMANDES DES PROCESSEURS.

Pour un fonctionnement correct de l'ensemble, il est donc impératif que les processeurs puissent émettre les commandes qui déclenchent les transitions de l'automate ainsi construit. Nous noterons que ces commandes sont particulières soit à PN, soit à PC (il n'y a pas de commande commune).

La répartition est la suivante:

- PN: Début Chargement, Fin Chargement, Début Sauvegarde, Fin Sauvegarde.

- PC: Début Exécution, Fin d'Exécution.

Les commandes "Début" sont des requêtes qui peuvent être satisfaites ou non. Il y a lieu de prévoir un protocole, de préférence non bloquant pour le processeur, qui lui permette de savoir si sa requête a été fructueuse. Sur le site 0, une requête refusée déclenchera un déroutement. Une commande "Fin" sera toujours satisfaite.

REALISATION.

La réalisation de cette circuiterie répartie est basée sur la simulation matérielle du mécanisme précédemment décrit à l'aide de jetons circulants entre différentes UGM. Une description détaillée en est donnée en annexe. Nous insisterons une nouvelle fois sur le caractère modulaire de celle-ci, qui apporte simplicité et extensibilité en nombre d'UGM. En outre, une requête fructueuse est exécutée en une seule instruction SOUT (instruction d'entrée-sortie spéciale).

II.D. DESCRIPTION DYNAMIQUE.

II.D.1. CURSUS D'EXECUTION D'UN MODULE SUR LE SITE.

L'historique d'une période d'activité d'un module est donnée par la description du schéma d'exécution d'un module. En effet, pour chaque période d'activité, on distingue l'étape initiale où le système met à jour l'environnement du programme utilisateur, l'étape utilisateur (phase

d'exécution du code utilisateur) et l'étape finale de recueil des messages envoyés.

Cette description peut être encore affinée par la prise en compte de l'organisation matérielle du site. En effet, à partir de l'hypothèse de base que les deux processeurs du site exécutent respectivement les parties système et utilisateur du module, nous avons attribué les étapes initiales et finales au processeur PN et l'étape utilisateur au processeur PC.

Qui plus est, on peut découper les étapes initiales et finales en considérant dans chacune de celles-ci d'une part, une phase 'manipulation des données du système' qui concerne uniquement PN, la mémoire locale au module et la mémoire système et d'autre-part une phase 'maintenance des UGM' qui permet à PN de paramètrer correctement une UGM pour l'exécution ultérieure du module. Ce découpage est intéressant pour l'évaluation des performances car il permet de définir quels éléments matériels (processeur, UGM ou mémoire) sont concernés par chaque phase et ainsi, de mettre en évidence les blocages dûs à ces unités. Il semble évident que ceux-ci vont grandement déterminer les performances du site.

Enfin, nous remarquons que les phases de "manipulations de données système" initiales et finales peuvent être confondues. Ceci pour deux raisons: Premièrement, l'étape finale sert uniquement à la récupération des résultats produits par la précédente phase d'exécution et qui seront sûrement réutilisées par un module. Elle peut alors être considérée comme une première partie de la phase initiale du module qui réutilisera ces données. Ces deux phases peuvent donc être consécutives. Deuxièmement, les deux phases demandent les mêmes ressources matérielles, à savoir le processeur N, la mémoire système et la mémoire du module.

En conclusion, nous avons établi quatre phases dans chaque période d'activité d'un module qui sont respectivement:

- Phase de préparation (manipulation des données systèmes relatives au module).
- Phase de chargement (Ecriture des registres d'une UGM).

- Phase d'exécution (Exécution du code utilisateur).
- Phase de sauvegarde (Récupération de l'état de l'UGM).

II.D.2. COMPORTEMENT DYNAMIQUE DES COMPOSANTS MATERIELS

L'ACTIVITE DES PROCESSEURS.

La figure II-15 montre la spécialisation des processeurs:

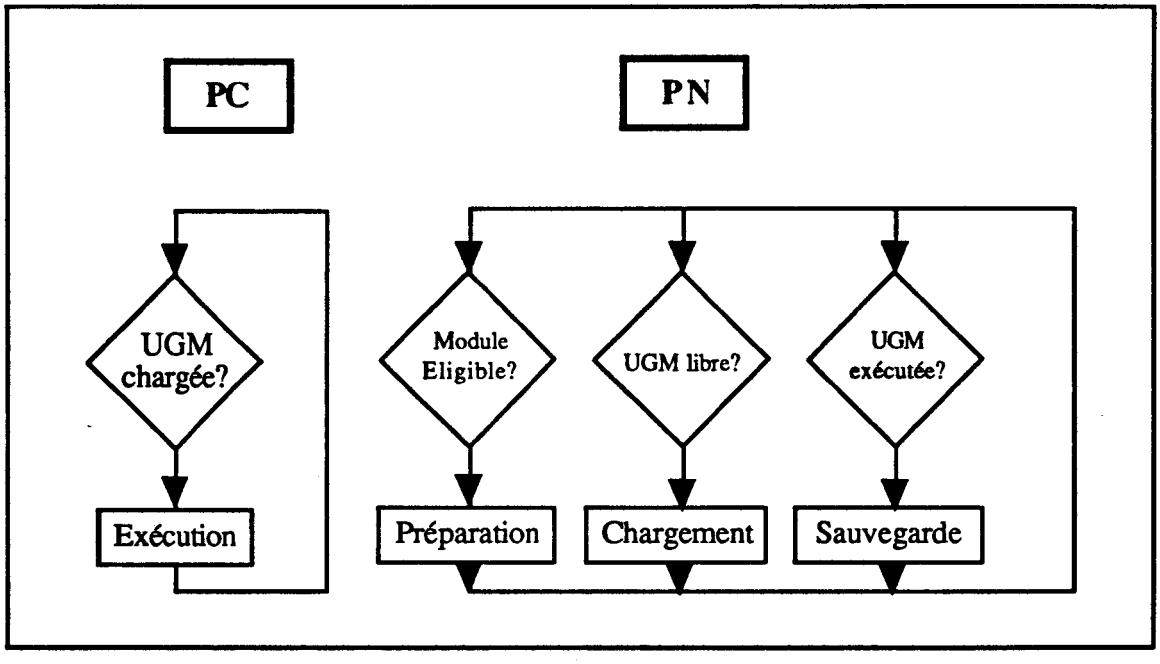


Fig. II-15. Occupation des processeurs.

PC ne réalise que la phase d'exécution, qu'il peut entamer si et seulement si une UGM chargée est disponible. Dans le cas contraire PC n'a simplement rien à faire.

PN peut préparer un module, charger ou sauvegarder une UGM. Pour choisir son activité, PN doit tester l'état des UGM, la disponibilité des blocs mémoires (si nécessaire) et déterminer quels sont les modules à réveiller.

Les deux processeurs se synchronisent indirectement par l'intermédiaire de la circuiterie de partage qui leur donne ou non la disponibilité d'une UGM pour l'opération projetée. Par exemple, PC ne

peut démarrer une phase d'exécution que s'il a demandé et obtenu une UGM de la circuiterie de partage en vue d'exécution, ce qui n'est possible que si PN a déjà préparé et chargé un module.

La communication entre PN et PC est faite uniquement par l'intermédiaire de la mémoire de bloc. En effet, tous les éléments écrits dans cette mémoire par PN lors de la phase de préparation pourront être lus (sous réserve que la programmation de l'UGM le permette) par PC lors de l'exécution. Inversement, PC ne pourra communiquer avec PN (donc grosso-modo avec le système) qu'en écrivant dans cette mémoire.

Les registres des UGM ne peuvent pas être utilisés dans ce sens à cause de leur spécificité.

ALLOCATION DES BLOCS DE MEMOIRE.

La description du fonctionnement du site 0 a mis en évidence une partition de l'ensemble des blocs mémoire: l'un d'entre-eux n'est accessible que par PC tandis que les blocs restants peuvent être librement manipulés par PN. Si ce découpage parait bien symétrique, les mécanismes mis en jeu pour le réaliser sont eux complètement différents.

PC ne peut ni lire, ni écrire dans un bloc mémoire qui ne lui est pas attribué. Toute tentative de violation est détectée puis inhibée par l'UGM couramment utilisée. Qui plus est, un déroutement en résulte et provoque l'abandon du programme courant (le programme utilisateur). De manière plus pratique, on peut connaître le bloc mémoire attribué au processeur PC en considérant l'état de l'UGM. L'allocation est ici une allocation physique (i.e. elle a un pendant matériel: le contenu des registres de l'UGM).

A l'opposé, rien de matériel n'empêche PN de lire ou d'écrire dans un bloc mémoire qui lui ne lui est pas attribué. On suppose simplement que le programme que va exécuter PN tiendra à jour de lui-même la liste des blocs mémoire disponibles et n'y accédera pas. Le code exécuté par PN est en effet uniquement du code système qui est supposé correct. La tenue de cette liste est possible car seul le programme système exécuté par PN gère l'allocation des blocs à PC ou aux UGM.

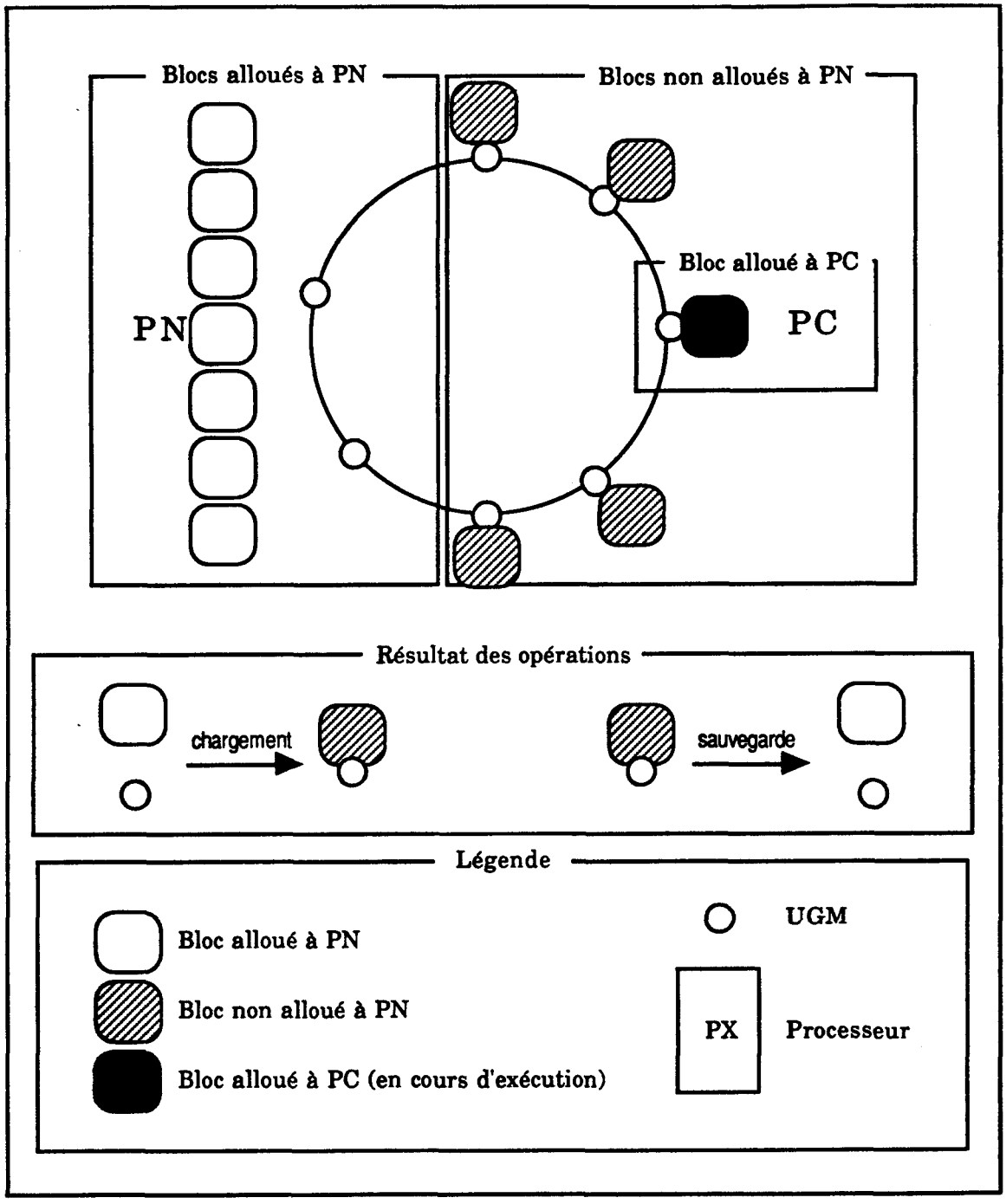


Fig. II-16. Allocation des blocs mémoire aux processeurs.

De manière plus fine, nous pouvons exposer la gestion des blocs mémoire au moyen les propositions suivantes:

- Initialement, PN dispose de tous les blocs alors que PC n'en a aucun.

- Lors du chargement de chaque module, PN marque le bloc mémoire où se trouve le module comme lui étant indisponible (autrement dit alloué à PC).

- Avant d'entamer la préparation d'un module, PN contrôle que ce module ne réside pas dans un bloc actuellement alloué à PC.

- Lors de la sauvegarde, PN enlève la marque du bloc comprenant le module sauvegardé. Ce bloc ne peut en effet plus être manipulé par PC car ce dernier ne dispose plus de l'UGM qui désigne le bloc.

Nous obtenons ainsi une gestion des blocs à deux niveaux:

- PN voit les blocs mémoire comme étants alloués à lui-même ou à l'ensemble des UGM. Dans ce dernier cas, un bloc est lié à une UGM.

- La circuiterie de partage attribue selon les demandes les UGM, à PN, à PC, ou encore à aucun processeur.

II.E. CONCLUSION

Dans ce chapitre, nous avons décrit une organisation de machine dont le principe de fonctionnement est essentiellement basé sur le partage de blocs de mémoire entre deux processeurs. Des unités de gestion mémoire (UGM) servent non seulement à la protection des accès mémoire vis à vis d'un des deux processeurs mais aussi à la commutation des blocs. Les requêtes émises par les processeurs sont prises en compte par une circuiterie spécifique qui commande les UGM.

Un prototype a été développé et réalisé à partir de circuits commercialement disponibles. Les éléments spécifiques dont dépendent la validation de cette organisation (circuiterie de partage, blocs mémoire) ont étés réalisés et testés.

Le modèle proposé apporte pour l'un des deux processeurs (PC) des caractéristiques qu'on retrouve actuellement sur des micro-processeurs récents (éventuellement dotés de circuits périphériques), à savoir protection mémoire et changement de contexte en une seule instruction. Mais de plus, l'organisation bi-processeur permet d'envisager, sous certaines conditions d'architecture logicielle, le doublement de la

puissance de calcul - à technologie équivalente -. La réalisation effective du prototype a également montré que ce modèle pouvait être appliqué à partir des microprocesseurs largement diffusés.

Il reste alors à déterminer quelles sont les conditions sur le logiciel mentionnées plus haut. De prime abord, cela revient à considérer sur un processeur courant les activités exécutées en mode protégé et non protégé, exécutées parallèlement sur deux processeurs. Le partage – à grande échelle de temps – de la mémoire commune, associé à la nécessité de minimiser les interactions entre les processeurs pose cependant de nouvelles questions:

- Le logiciel fonctionnant sur PC doit être autonome et entièrement contenu dans un bloc de mémoire.

- La gestion proprement dite des blocs mémoire peut poser des problèmes d'efficacité. En particulier, la copie de données d'un bloc à l'autre demande la disponibilité simultanée des deux blocs au même processeur.

- Quelles caractéristiques du logiciel doivent être évaluées pour déterminer son adéquation à l'architecture du site 0? Quelles sont les meilleures valeurs.

Une première approche consiste à simuler le fonctionnement du site zéro par rapport au modèle logiciel exposé dans ce chapitre. Cette étude a étée faite et est décrite lors du chapitre suivant.

III. LA SIMULATION DU SITE ZERO.

Au cours de l'élaboration du site zéro, nous avons décidé de réaliser une simulation de l'organisation matérielle du site. Celle-ci devait répondre à un certain nombre de questions qui se posaient alors. Il fallait observer le comportement du site zéro face à un ensemble de modules qui se présentent à l'exécution et mettre en évidence (et si possible quantifier) l'influence d'un certain nombre de paramètres matériels et logiciels sur les performances du site.

III.A. HYPOTHESES ET OBJECTIFS.

Une simulation générale qui prendrait en compte toutes les caractéristiques et du site zéro et des modules présentés à l' exécution n'est pas imaginable. Nous énumérerons donc un certain nombre de conditions restrictives (hypothèses) qui limiteront le champ d'investigations à un domaine raisonnable. Parallèlement, nous pourrons ainsi établir la liste exhaustive des paramètres de fonctionnement du site zéro.

III.A.1. LES HYPOTHESES.

Les hypothèses sont des contraintes qui résultent soit des caractéristiques invariables du matériel utilisé (l'organisation matérielle d'un site, le temps d'accès à la mémoire des processeurs Z8001, etc...), soit de choix qui sont immédiats (la loi de probabilité des durées d'exécution des modules par exemple). Ces hypothèses peuvent quelquefois demander une justification. Dans ce cas, nous essayerons de les étayer, le plus souvent en montrant qu'un autre choix serait inefficace ou remettrait en cause la conception globale du site.

Les hypothèses énumérées seront de deux sortes: Les hypothèses sur le matériel et les hypothèses sur les modules.

TEMPS DE CHARGEMENT ET DE SAUVEGARDE.

Nous verrons par la suite que ces temps sont des constantes déduites des caractéristiques du matériel.

INTERVALLE D'ARRIVEE.

Le débit d'arrivée des modules est caractérisé par une variable aléatoire de loi exponentielle (hypothèse classique) dont la moyenne sera un des paramètres de la simulation.

TEMPS D'EXECUTION ET DE PREPARATION.

L'évaluation des temps d'exécution et de préparation est nettement plus ardue. Le manque d'information précise sur le rôle respectif de chacune de ces phases nous a conduit à reprendre la même hypothèse, à savoir pour ces deux durées deux variables aléatoires de loi exponentielle. Les valeurs des moyennes (paramètres de la loi exponentielle) seront deux paramètres de la simulation.

ORGANISATION FONCTIONNELLE DU SITE.

Nous avons considéré, dans cette étude, que les deux seuls agents actifs du modèle sont les logiciels qui s'exécutent sur PN et PC (que nous appellerons processus). Par opposition, le site zéro est décomposé en un certain nombre de composants matériels qui sont autant de ressources. Chaque phase demande pour son engagement la disponibilité de certaines ressources, qu'elle libère lorsqu'elle se termine. Pour être rigoureux, nous décrirons initialement chacun des deux processeurs comme une ressource. Ce n'est qu'au moment de l'analyse de leurs caractéristiques que nous montrerons que l'on peut confondre processeur et processus.

LES RESSOURCES.

LE PROCESSEUR PN.

Le processeur PN est celui qui exécute les opérations de préparation, chargement et sauvegarde des modules. Il peut accéder de manière permanente à une mémoire privée. Il travaille également avec une partie des mémoires de modules et avec une des UGM.

LE PROCESSEUR PC.

Le processeur C a un rôle beaucoup plus simple que le processeur PN . Il doit en effet se contenter d'exécuter des modules complètement

préparés. Pour cela, il aura à utiliser un jeu de UGM et un des blocs de la mémoire de modules. Pour être complet, on doit également mentionner une mémoire privée et des éléments périphériques.

LA MEMOIRE PRIVEE N.

Cette mémoire contient principalement les structures de données du système qui ne doivent être manipulées que durant la phase de préparation. Donc uniquement par le processeur PN . On y trouve également le code du noyau du système.

LA MEMOIRE PRIVEE C.

Cette mémoire sert à stocker le code système de PC. En effet, ce processeur exécute du code système pour organiser son travail (demander une UGM, se brancher sur le code utilisateur). On peut également trouver pratique de mettre dans cette mémoire les outils offerts aux utilisateurs sous forme de primitives.

LES UGM.

On dispose par site de plusieurs UGM. Les processeurs demandent l'allocation d'une UGM au moyen de la circuiterie de partage. Le processeur PN pour les opérations de chargement, sauvegarde et le processeur PC en utilisation. Les registres d'une UGM mémorisent la description des 128 segments du module, c'est à dire l'ensemble de son contexte mémoire à l'exécution.

LES BLOCS DE MEMOIRE DE MODULE.

On y installe les images mémoire des modules à exécuter. Une image comprend le code et les données utilisateur mais aussi les structures de données systèmes propres à chaque module quand elles doivent être utilisées durant la préparation et durant l'exécution. Ces structures peuvent être cachées à l'utilisateur.

Etant attribués alternativement à l'un puis à l'autre processeur, ce sont les outils privilégiés de communication entre les deux processeurs. Ceci explique qu'on y place les éléments d'un module qui servent à la fois durant l'exécution et durant la préparation.

On prend comme hypothèse le fait que les différents blocs sont de même taille.

LA CIRCUITERIE DE PARTAGE.

La circuiterie de partage introduira sûrement peu de dégradation dans les performances pour deux raisons. D'abord parce que l'utilisation de ce dispositif est instantanée (en ce sens que les processeurs ne doivent pas obtenir l'accès à cette ressource pour une certaine durée mais uniquement à l'occasion d'une seule instruction d'entrée-sortie. Mais surtout, de par sa conception matérielle, cette circuiterie est accessible simultanément par les deux processeurs.

TYPE DE CHAQUE RESSOURCE

Après avoir énuméré l'ensemble des ressources d'un site, nous distinguerons:

- Les ressources non allouables. Soit parce qu'elles sont constamment disponibles pour chacun des processus (la circuiterie de partage, par exemple) ou parce qu'elles sont indissociablement liées à une autre ressource (les mémoires privées de PC ou de PN sont liées à leur processeur respectif). Dans ce dernier cas, les conflits qui pourraient éventuellement se produire sur ces ressources se produisent d'abord sur les ressources "maîtres". Une ressource non allouable n'influe pas sur les performances du site.

- Les ressources allouables peuvent être banalisées ou non. Une ressources est dite banalisée quand un processus ne peut demander que "une ressource parmi un ensemble". Inversement, une ressource non banalisée devra être demandée explicitement; par exemple, il n'est pas question d'allouer PC lorque PN est demandé.

Les paragraphes suivants détaillent chacune des ressources vis à vis de ce critère.

PN.

En exemplaire unique, cet ensemble comprend le processeur N ainsi que ses éléments périphériques. C'est une ressources allouable.

PC.

En exemplaire unique, cet ensemble comprend le processeur C ainsi que ses éléments périphériques. C'est une ressources allouable.

Nous serons amené à simuler les activités des deux processeurs. Or celles-ci sont, bien entendu indéfectiblement liés aux processeurs mêmes, à leur mémoire privée ainsi qu'à leurs dispositifs d'entrée-sortie. Il ne peuvent donc être l'objet d'aucun conflit d'accès de la part des activités. Cette remarque fait perdre le statut de ressource à PC, PN et à leurs éléments associés.

Dorénavant, nous confondrons le nom du processeur sur lequel tourne un processus et ce processus lui-même.

UGM.

Le nombre d'UGM est variable. C'est la circuiterie de partage qui décide de l'allocation des UGM aux processeurs. Ce sont des ressources banalisées.

BLOC DE MEMOIRE.

Le nombre de blocs de mémoire est variable. D'autre-part, rien n'a encore été supposé sur la répartition des modules sur les blocs de mémoires. Les blocs sont à priori des ressources banalisées. Cependant, lors des phases de chargement, exécution et sauvegarde, un raisonnement similaire à celui que nous avons tenu pour les éléments associés aux processeurs nous indique que ceux-ci sont liés aux autres ressources que sont les UGM. En effet, sous réserve d'une bon comportement de PN , nous pouvons considérer que l'attribution d'une UGM chargée ou exécutée sous-entend l'attribution du bloc mémoire désigné par ses registres.

Par ailleurs, soulignons ici le fait que l'on ne considère que l'allocation d'un bloc à l'un ou à l'autre des processeurs, indépendamment du chemin d'accès utilisé (direct ou à travers l'UGM).

Le tableau suivant présente un résumé des considérations précédentes.

Ressources		
Allouables		Non Allouables
Banalisées	Non Banalisées	
UGM (Blocs)	(PN) (PC)	MPN, ESN MPC, ESC Circuiterie de partage

Fig. III-1. Inventaire des ressources.

ALLOCATIONS NECESSAIRES A CHAQUE PHASE.

PREPARATION.

Le processeur PN effectue la préparation des modules. Pour cela, PN a besoin d'un bloc de mémoire libre i.e. non alloué à PC (cf. chapitre 2) au hasard (ressource banalisée).

CHARGEMENT.

Le processeur charge les descripteurs de segment dans l'UGM. Le chargement se fait à partir de tables qui se trouvent en mémoire privée. PN n'a donc, à ce moment là, pas besoin d'accéder au bloc alloué au module en cours de traitement.

EXECUTION.

Le processeur PC exécute le code utilisateur de chaque module. Il accède alors au bloc de mémoire alloué au module au travers d'une UGM (traduction d'adresse).

SAUVEGARDE.

Le processeur PN récupère les données qui se trouvent dans le bloc de mémoire concerné et éventuellement dans les registres de l'UGM.

Nous pouvons alors établir un tableau des besoins propres à chaque phase.

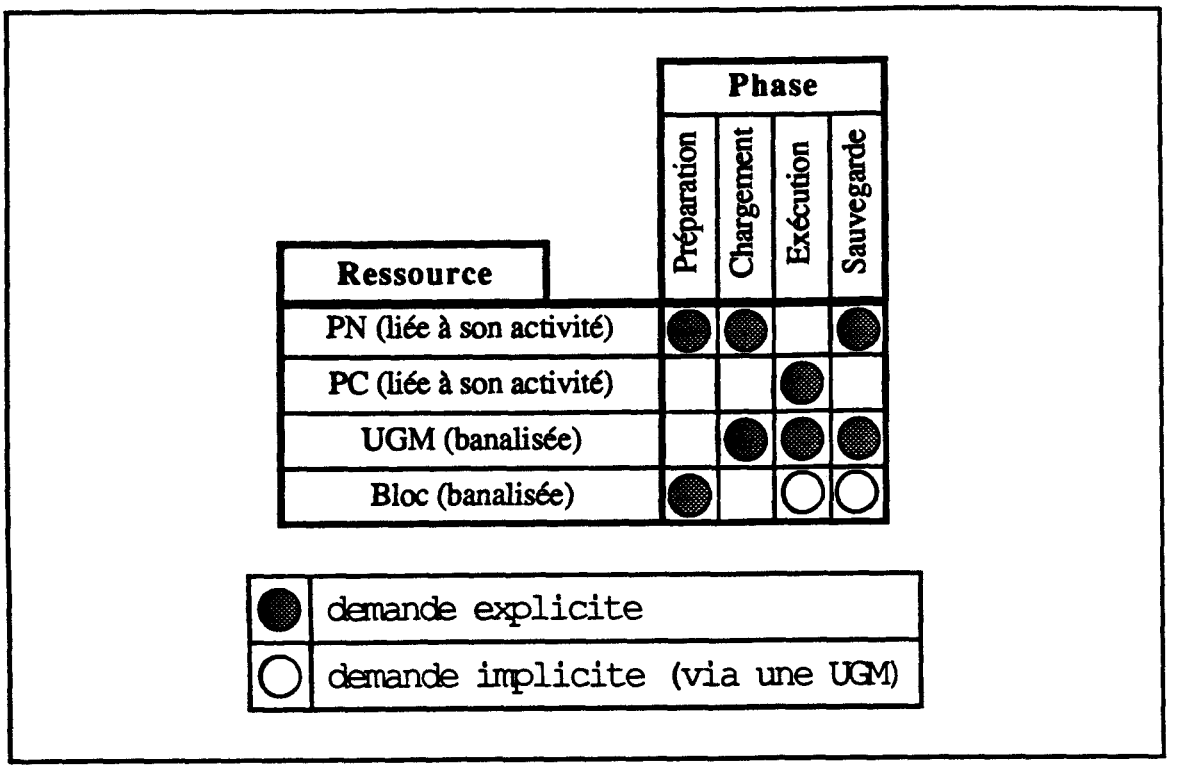


Fig. III-2. Inventaire des besoins par phase.

GESTION DES UGM.

Chaque UGM suit un cycle de 6 états qui est imposé par la circuiterie de partage. Les états sont respectivement "libre, chargement, chargé, exécution, exécuté, sauvegarde". Le choix de l'UGM attribuée au processeur qui en fait la demande est ainsi complètement fixé par le matériel.

CARACTERISTIQUES DE LA MEMOIRE DE MODULE.

Nous n'avions, au départ, qu'une idée assez vague de la façon dont seraient répartis les modules entre les différents blocs. D'une part parce que les caractéristiques même de la mémoire (taille d'un bloc, nombre de blocs) étaient une inconnue que nous devions résoudre, d'autre-part parce que les choix d'implantation (carte mémoire d'un module: nombre et position des segments) n'étaient pas encore établis.

Ces imprécisions ont été levées de deux façons: D'abord nous savions (en tant qu'hypothèse de base) qu'un module au moins pouvait tenir complètement dans un bloc de mémoire. Ensuite, nous avons considéré

les valeurs qui restaient inconnues (nombre de blocs, nombre de modules maximum par bloc) comme des paramètres de la simulation.

CREATION DES MODULES

La simulation d'un comportement réel de l'architecture, en tenant compte d'un enchaînement déterminé de modules connus à l'avance était irréaliste: D'une part, le choix de l'enchaînement est trop sujet à caution pour aboutir à des résultats intéressants; d'autre-part, la stratégie d'allocation d'un bloc mémoire à chaque module n'est pas triviale.

Nous avons préféré supposer qu'un module est créé de toutes pièces juste au début de sa préparation. C'est à ce moment précis que le noyau doit élire un bloc mémoire disponible pour recevoir le nouveau module.

GESTION DE LA MEMOIRE DE MODULE.

Une fois retenue l'hypothèse qu'un bloc pouvait contenir un certain nombre de modules, il restait à préciser la manière dont seraient associés blocs et modules. L'objectif de la stratégie établie est de minimiser les temps d'attente dûs à des conflits d'accès à la mémoire entre PC et PN , typiquement quand les deux processeurs, simultanément, préparent et exécutent des modules différents mais situés dans le même bloc de mémoire.

La gestion simulée consiste principalement en une allocation circulaire des blocs aux modules (gestion en anneau).

III.A.2. LES PARAMETRES CHOISIS.

Nous établissons ici la liste des paramètres de la simulation. Ce sont soit des valeurs numériques qui répondent aux questions posées sur le fonctionnement du site zéro, soit des éléments qui conditionnent ces même réponses. Ainsi, on peut imaginer que le nombre de jeux d'UGM optimal dépend de la durée moyenne d'exécution d'un module.

CONFIGURATION MATERIELLE.

NOMBRE D'UGM.

On conçoit facilement que le nombre d'UGM installées sur le site influe considérablement sur les performances de la machine.

Un cas extrême est un site qui ne comporte qu'une seule UGM. Alors, les deux processeurs attendent tour à tour la seule UGM existante. Les performances globales approchent alors franchement d'un mono-processeur.

D'un autre coté, un nombre important d'UGM ne devrait pas se justifier. En effet, dans ce cas, on observe un nombre important d'UGM non exploitées et donc complètement inutiles.

Entre ces deux extrêmes, il faut calculer un nombre d'UGM qui leur donne un taux d'utilisation raisonnable sans *trop* ralentir les processeurs (compromis coût/performance).

NOMBRE ET TAILLE DES BLOCS DE MEMOIRE.

Pour une taille globale donnée G de mémoire de modules, on peut choisir des découpages différents en B blocs de taille T (B x T = G). La complexité matérielle de l'ensemble est fonction, primo de la taille globale G, secundo du nombre de blocs B. En effet, chaque bloc doit se voir attribuer un accès aux deux bus BNl et BC. De cette réflexion, on peut déduire que pour une taille globale donnée, on a intérêt à minimiser le nombre de blocs, ce qui, corrélativement, augmente la taille de chacun des blocs.

Le nombre de blocs de mémoire influe sur les performances du site. En effet, un manque de bloc mémoire, quand PN n'a pas de bloc disponible pour préparer un nouveau module, peut ralentir la machine.

La taille des blocs importe pour l'évaluation des performances du site. Un bloc de bonne taille permettra de stocker plusieurs modules dans le même bloc. Il se peut alors que les deux processeurs doivent travailler sur deux modules qui se situent dans le même bloc. La fréquence de ce dernier cas de figure doit en principe être réduite au minimum par

l'utilisation d'un algorithme judicieux d'allocation des blocs de mémoires aux modules.

En résumé, à volume égal de mémoire, il faudra trouver le compromis entre le nombre et la taille des blocs, alors que les deux points de vues de coût et de performance militent dans des directions opposées.

CONFIGURATION LOGICIELLE.

STRATEGIE DE PN.

Il convenait également de bien préciser le mode de fonctionnement du système. Si la répartition des phases entre les deux processeurs était bien établie, le fonctionnement interne de PN n'était pas du tout éclairci. A la fin d'une phase, PN peut se trouver devant plusieurs possibilités parmi chargement, préparation ou sauvegarde de différents modules. Les régles de choix n'étaient pas énoncées. Cette simulation a été un moyen d'évaluation des différentes stratégies de choix qui vont être présentées.

Faisons la liste des critères à partir desquels le choix va être fait:

Le processeur PN peut tester la disponibilité d'un module en préparation puisque cette information est contenue dans les structures de données du système.

La demande d'UGM n'étant pas bloquante, le processeur PN peut connaître la disponibilité d'UGM en sauvegarde ou en chargement.

De fait, cette connaissance ne peut se faire qu'en faisant la demande effective d'UGM. Mais alors, si une réponse est affirmative, l'UGM correspondante est allouée (PN ne peut refuser la ressource). Par exemple, PN veut savoir si il existe une UGM libre (pour effectuer un chargement). Il fait donc la demande ad hoc à la circuiterie de partage. Si la demande est satisfaite, l'UGM est effectivement allouée. Toutefois, PN peut accepter l'UGM et la mettre de coté, pour ensuite entamer une autre opération. Cette procédure n'est pas pénalisante car PN est le seul à pouvoir demander, et à fortiori obtenir, une UGM libre. Le même raisonnement vaut pour PC car il est seul sur le site. Ceci est heureux car PN ne pourrait maintenir une table de disponibilité des UGM: deux des

opérations qui agissent sur l'état des UGM (demande d'exécution et fin d'exécution) ne sont pas de son initiative.

Compte-tenu de ces éléments, nous avons pu imaginer sept stratégies pour PN. Six d'entre-elles sont des stratégies que nous appellerons à priorités fixes. Pour celles-ci, PN évaluera dans un ordre fixe, la faisabilité des trois opérations qui sont à sa charge (chargement, exécution, sauvegarde) et exécutera la première opération possible. Les six stratégies résultent des six permutations possibles pour ces trois opérations. On notera ces stratégies par les trois initiales des opérations (C, P, S) dans l'ordre des priorités respectives. Par exemple, pour appliquer la stratégie **CSP**, le processeur N cherchera d'abord si un chargement est possible (il faut une UGM libre et un module prêt) puis, si le chargement est impossible, tentera de faire une sauvegarde (il a besoin pour cela d'une UGM exécutée) et en dernier recours regardera si il peut faire une préparation (avec un module non prêt). De même, nous testerons l'efficacité des stratégies **CPS, PCS, PSC, SCP** et **SPC**.

La dernière stratégie, baptisée **INT**, (évaluée à titre gratuit car elle demanderait une adjonction au matériel du site) suppose que le processeur PC peut, en fin d'exécution, signaler au processeur N qu'il vient de restituer au système une UGM qui est donc dans l'état "*exécutée*". Ceci permet d'élaborer la stratégie suivante: Initialement, le processeur PN prépare puis charge un module autant de fois qu'il y a d'UGM installées sur le site. Une fois cet amorçage réalisé, PN ne cherchera plus à faire que des préparations. Les sauvegardes et chargements sont toujours faits par PN mais uniquement sur la requête de PC (la requête est transmise par le mécanisme matériel supposé plus haut): En fin d'exécution, PC restitue son UGM et signale à PN qu'il a terminé sa phase. En réponse, PN interrompt la préparation en cours (préemption du processeur PN par le module qui vient d'être exécuté), sauvegarde le module qui vient de se terminer et recharge immédiatement l'UGM rendue libre par la sauvegarde.

Le but de cette stratégie est de minimiser les temps d'inoccupation des UGM. On peut ainsi espérer une charge maximale pour PC qui n'a besoin, pour travailler, que d'une UGM chargée (si PC dispose d'une UGM, il dis-

pose aussi forcément du module préparé correspondant en mémoire de module).

CARACTERISTIQUES DES MODULES.

Le but de ces paramètres est de caractériser le comportement des modules. Les options prises sont les suivantes:

- Tous les modules ont la même taille mémoire, cette valeur est prise comme unité. Ceci implique que l'on donne la valeur de la taille d'un bloc en nombre de modules stockables par bloc (c'est une valeur entière).

- On prend aussi comme paramètres les moyennes des lois de distribution des durées des phases de préparation et d'exécution. nous avons vu que les durées de chargement et de sauvegarde pouvaient être prises constantes. Ces deux paramètres TP et TX peuvent être remplacés par deux autres valeurs TT et RH (temps total de calcul du module et pourcentage d'overhead). Les relations entre toutes ces valeurs seront détaillées dans le paragraphe suivant.

- Intervalle d'arrivée. C'est la moyenne de la loi de distribution de la variable aléatoire qui caractéristique l'intervalle de temps entre l'arrivée de deux modules consécutifs.

III.A.3. EVALUATION DE L'ARCHITECTURE

Pour jauger précisément l'architecture, il faut disposer de valeurs chiffrées représentatives du comportement du site. Le problème qui se pose alors est celui du choix de ces valeurs. Lors de ce choix, deux préoccupations s'imposent au concepteur:

- Caractériser proprement les performances du site. Si l'on peut exhiber, en guise de données, un petit nombre de paramètres "purs" bien spécifiés, les valeurs calculées sont beaucoup plus nombreuses mais surtout interdépendantes ; par exemple, on peut trouver une relation évidente entre la durée totale de la simulation d'un jeu de données et la charge moyenne de chaque processeur. Or, cette relation dépend

.*uniquement* du jeu de données. Il est donc faux d'évaluer l'architecture avec les valeurs moyennes de charge des processeurs.

- Conjointement, la modélisation nécessaire à la simulation déforme la perception qu'a l'observateur du comportement de la machine. Nous verrons que le paramètre 'intervalle d'arrivée' qui quantifie, en principe, la charge de l'ensemble (débit des travaux présentés à l'entrée) n'a qu'un rôle mineur (en fait uniquement dû aux phases transitoires de montée en charge et de perte de charge du système).

Le but de ce chapitre est d'établir les relations qui peuvent exister entre les valeurs manipulées (paramètres ou résultats de mesure). On peut espérer ainsi mettre en évidence les meilleurs critères d'évaluation d'un site.

LISTE DES VALEURS.

Dans un premier temps, nous allons faire la liste exhaustive des valeurs numériques de la simulation. Nous noterons Σxx la valeur cumulée de chaque valeur xx relative à un module sur l'ensemble des modules.

PARAMETRES DES MODULES.

Quand les caractéristiques des modules sont des variables aléatoires, on peut retenir, le ou les paramètres de la loi utilisée (qui est toujours, ici, une loi exponentielle dont on prendra la valeur moyenne).

NM. Nombre de modules du jeu d'essai.

IA. Intervalle d'arrivée moyen des modules.

TP. Temps de préparation moyen des modules.

TX. Temps d'exécution moyen des modules.

TT. Temps total moyen de traitement d'un module.

RH. Ratio d'overhead d'un module. (C'est le rapport entre les temps de traitement système et utilisateur d'un module). On considère le temps de traitement système se compose des temps de préparation,

chargement et sauvegarde, alors que le temps de traitement utilisateur est uniquement le temps d'exécution. Soit:

$$RH = \frac{TP + TC + TS}{TX}$$

PARAMETRES DU SITE.

TC Temps de chargement d'un module. Cette valeur est considérée comme constante car déduite des caractéristiques matérielles du site 0.

TS Temps de sauvegarde d'un module. Même remarque que pour TC.

STRAT.
Stratégie employée. Ce n'est pas une valeur numérique.

UGM.
Nombre d'UGM.

NBBLOCS.
Nombre de blocs de mémoire.

SZBLOCS.
Taille de chaque bloc en nombre de modules contenus.

VALEURS MESUREES.

Ces valeurs sont déduites d'un déroulement de simulation, pour un jeu d'essais (composé d'un certain nombre de modules) ainsi que pour une configuration connue du site zéro.

DT. Durée totale de la simulation.

DP. Durée cumulée de préparation de l'ensemble des modules.

DC. Durée cumulée de chargement de l'ensemble des modules.

DX. Durée cumulée d'exécution de l'ensemble des modules.

DS. Durée cumulée de sauvegarde de l'ensemble des modules.

AP. Temps d'attente de préparation cumulé pour l'ensemble des modules.

AC. Temps d'attente de chargement cumulé pour l'ensemble des modules.

AX. Temps d'attente d'exécution cumulé pour l'ensemble des modules.

AS. Temps d'attente de sauvegarde cumulé pour l'ensemble des modules.

MM. Taux moyen d'occupation des UGM.

DM. Temps de présence dans une UGM cumulé pour l'ensemble des modules.

TR. temps de réponse (intervalle de temps entre l' entrée d'un module dans le système et sa sortie) cumulé de l'ensemble des modules.

MD. Nombre moyen de module traités par unité de temps.

CC. Taux de charge de PC.

CN. Taux de charge de PN.

LES FORMULES ETABLIES.

Les seules hypothèses prises lors de l'établisement de ces formules concernent le nombre des processeurs, ainsi que l'ordre et la répartition des phases entre les deux processeurs. En particulier, elles sont valables quelque soit la stratégie de PN.

$$TR = AP + DP + AC + DC + AX + DX + AS + DS \qquad (1)$$

Chaque module suit invariablement les étapes suivantes entre son entrée et sa sortie du système: attente de préparation, préparation, attente de chargement, chargement, attente d'exécution, exécution, attente de sauvegarde et sauvegarde. Donc son temps de réponse est égal à la somme des durées individuelles de chaque étape. Si on cumule cette propriété sur l'ensemble des modules, on obtient la formule *(2)*.

$$\Sigma TT = \Sigma TP + \Sigma TC + \Sigma TX + \Sigma TS \quad (2)$$

La durée totale de traitement est égale à la somme des durées des quatres phases du traitement.

$$MD = \frac{NM}{DT} \quad (3)$$

Le nombre moyen de modules traités par unité de temps est égal au rapport du nombre de modules sur la durée totale de la simulation.

$$DM = DC + AX + DX + AS + DS \quad (4)$$

Une UGM est utilisée durant les phases de chargement, d'exécution et de sauvegarde des modules, mais est également occupée quand ces modules sont en attente d'exécution ou de sauvegarde.

$$MM = \frac{DM}{DT} \quad (5)$$

$$CC = \frac{DX}{DT} \quad (6)$$

$$CN = \frac{DP+DC+DS}{DT} \quad (7)$$

Les taux de charges des UGM, de PC et de PN sont égaux au rapport entre leur durée respective d'activité et la durée totale de la simulation.

De l'ensemble de ces relations, on peut constater que, pour un jeu d'essais donné (donc avec **DP**, **DC**, **DX** et **DS** fixés), les valeurs obtenues pour **CC**, **CN**, **MM**, peuvent être déduites de **DT**, durée totale de la simulation. Cette dernière donnée est donc suffisante pour comparer deux simulations utilisant le même jeu de donnée mais des configurations matérielles différentes. Cette durée totale peut être comparée au temps total de traitement **ΣTT** qui est, incidemment, le temps que mettrait un mono-processeur pour traiter l'ensemble des modules. On peut également s'en servir pour comparer des résultats de jeux d'essais différents sous réserve que le temps total **ΣTT** de ces jeux d'essai soients identiques.

En revanche, les autres valeurs mesurées (temps d'attente des modules dans les différentes files) ne semblent pas devoir se prêter à une interprétation directe sinon à déterminer les causes de performances insuffisantes.

Par ailleurs, il faut souligner l'effet pervers du paramètre intervalle d'arrivée. Quand il est trop grand, il retarde l'arrivée des modules candidats en forçant les processeurs à des périodes d'inactivité: du point de vue de la durée totale de simulation, il est inutile de traiter très rapidement 5000 modules en une micro-seconde si le 5001ème module arrive 1 heure plus tard... Dans ce cas extrême, la durée totale de la simulation est déduite de la date d'arrivée du dernier module et du temps de traitement de ce dernier module, ce qui n'est évidemment pas souhaité.

A l'opposé, ce paramètre ne sert plus à rien dès que l'on peut assurer qu'il reste toujours au moins un module candidat à l'entrée dans le système. Nous verrons plus tard que même les valeurs 'actives' minimales et maximales de ce paramètre ne peuvent pas servir à évaluer les performance du site.

III.B. LES MOYENS UTILISES.

III.B.1. MODELISATION DU PROBLEME.

On peut assimiler l'ensemble du site à un réseau de files d'attente dont la description est:

- Quatres files d'attente, une par phase de traitement.
- Les clients sont des modules.
- Deux stations qui sont les deux processeurs. La première station (PN) réalise les services de la première, deuxième et quatrième file (préparation, chargement et sauvegarde), alors que l'autre station réalise le troisième service (exécution).
- L'ensemble de ces files d'attente sont disposées à la suite l'une de l'autre.
- Le fonctionnement de l'ensemble est régi par certaines contraintes:

→ Contraintes sur le nombre d'UGM. Le nombre d'UGM étant fixé, on ne peut avoir plus de modules dans l'ensemble des files d' exécution et de sauvegarde que le nombre d'UGM.

→ Contraintes sur le nombre de blocs: Le choix de l'algorithme de gestion des blocs de mémoire implique que dès qu'un module a été chargé, le bloc dans lequel il se trouve ne peut être manipulé par PN, ceci jusqu'au début de la phase de sauvegarde. Cela signifie qu'il ne peut y avoir en attente d'exécution ou même en exécution plus de modules que le nombre de blocs mémoire.

→ Contraintes sur la taille de la mémoire de module. PN a besoin d'un bloc de mémoire pour préparer un module. Il se peut qu'au cours de la simulation, PN trouve un module à préparer, mais aucun bloc disponible pour l'y placer. Dans ce cas PN se verra contraint de choisir une autre opération ou pire, de ne rien faire. Appliqué au réseaux de files d'attentes, cela signifie que l'on doit borner le nombre de modules dans l'ensemble du réseau (exception faite de la file d'attente de préparation FAP).

L'ensemble des contraintes est résumé dans le schéma suivant:

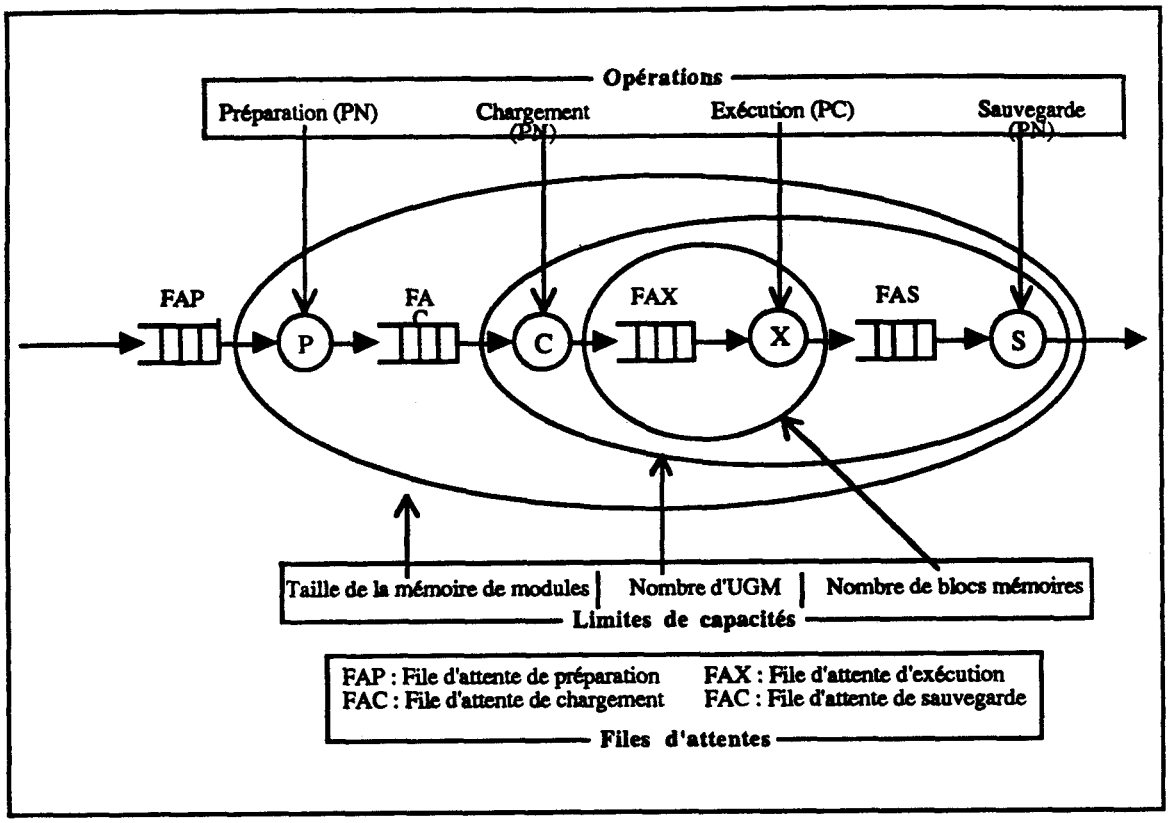


Fig. III-3. Réseau de files d'attente.

Il faut souligner que ces contraintes **quantitatives** sont insuffisantes pour décrire complètement le fonctionnement du site. Il faudrait ajouter à celles-ci les algorithmes d'allocation de la mémoire de bloc et la stratégie de choix d'activité de PN qui n'y sont pas entièrement prises en compte.

III.B.2. LES OUTILS DISPONIBLES.

Une évaluation analytique théorique de ce réseau par la théorie des files d'attente n'est pas possible (il suffit, pour s'en convaincre de regarder l'article de [Pujolle79] sur les réseaux de files d'attentes à capacités limitées). Qui plus est, ces méthodes ne permettent pas de tenir compte des algorithmes d'allocation mémoire et de PN. Il ne restait plus qu'à simuler le fonctionnement du site. Pour cela, nous pouvions choisir d'utiliser un langage de spécification de réseaux de files d'attente associé à son simulateur (QNAP2 était le seul outil disponible). Nous avions

également le loisir de programmer entièrement le simulateur à l'aide d'un langage de programmation classique. L'étude de QNAP2 nous montrera que le réseau ne pouvait être complètement décrit au moyen du langage QNAP2. Il nous fallait donc programmer la simulation.

QNAP2.

Nous avions l'opportunité d'utiliser un logiciel de simulation de réseau de files d'attente baptisé QNAP2. Nous n'en avons rien fait car ce que nous pouvions simuler à l'aide de ce logiciel était, soit trop compliqué, soit inexact. L'objet de ce chapitre est de décrire globalement QNAP2, puis, à partir de cette description d'expliciter les problèmes que nous avons rencontrés dans l'application envisagée. Il n'est nullement question de faire une analyse exhaustive du logiciel mais plutôt de montrer les limitations qui nous ont empêché de l'utiliser.

DESCRIPTION.

ROLE ET OBJECTIFS DE QNAP2.

QNAP2 est un logiciel qui permet de d'étudier le comportement de réseaux de files d'attente. Pour cela, il permet la description d'un modèle qu'il peut ensuite résoudre (exhiber la valeurs de certaines caractéristiques), soit par simulation, soit analytiquement.

Nous allons principalement nous intéresser aux moyens de description du réseau car c'est à ce niveau que nous avons éprouvé les principaux problèmes d'utilisation (Peu importent les résultats fournis par l'outil si le problème n'est pas décrit correctement). Il faut remarquer que cette démarche dévalue forcément le produit qui est par ailleurs intéressant pour la technique de description de réseau qu'il apporte et pour l'automatisation des nombreuses méthodes de résolution qu'il réalise.

Le langage de description de réseau de files d'attente utilisé par QNAP2 permet de décrire le réseau complet comme un ensemble d'objets. Ces objets peuvent être statiques (créés en même temps que le réseau et inamovibles) ou dynamiques (créés et détruits à la demande, au cours de la simulation). Notons que le principal avantage de QNAP2 réside dans le

fait que, contrairement aux langages de programmation classiques, il met à disposition de l'utilisateur outre les types standards des langages de programmation, des types d'objet fréquemment utilisés lors des simulations et d'un niveau d'abstraction élevé (client, serveur, station, file d'attente...). Le comportement des stations peut même être décrit par un algorithme dont la formulation est très voisine d'un langage de programmation classique (PASCAL).

LES OBJETS DE QNAP2.

Nous détaillons ici les objets originaux du langage. On y trouve en effet des éléments classiques des langages de programmation courants et de peu d'intérêt ici. Par contre, nous allons expliciter les objets construits fournis par QNAP2 en mettant en exergue les caractéristiques utilisées dans la démonstration qui suit.

FILE D'ATTENTE.

QNAP2 permet de créer des files d'attente. Ce sont des structures de données qui permettent de stocker les clients qui sont en attente devant une station. Ces files peuvent implanter un certain nombre de disciplines (LIFO, FIFO), avec ou sans priorité.

STATION.

Une station est le lieu où un client obtient un service. Ce peut être une station à serveur unique (on rend au plus un service à la fois) ou à serveurs multiples. Dans ce dernier cas, la station peut satisfaire plusieurs clients simultanément, mais ces clients viennent de la même file d'attente et demandent le même service. De ce fait on peut associer chaque station au service qu'elle rend.

Il existe un type spécial de station baptisée **source** qui engendre des clients à des intervalles de temps fixés par le temps de service (voir paragraphe suivant). Une telle station ne peut recevoir de client.

SERVICE.

Les services ne sont pas à proprement parler des objets du langage; Il n'ont pas d'existence indépendante des stations, mais ils possèdent cependant des propriétés utiles.

Un service est décrit sous la forme d'un algorithme qui indique les opérations à effectuer lorsque client et serveur sont disponibles. Ces algorithmes sont constitués de demandes de travaux et de manipulation d'objets.

Les demandes de travaux représentent des opérations qui ne changent pas l'état des objets de la simulation mais qui prennent du temps. Ce temps peut avoir une valeur contante ou déterminée par une loi de probabilité.

Les manipulations d'objet sont permises par les primitives offertes par QNAP2, entre autres, il est possible de:

- Forcer le transfert d'un client vers une file d'attente.
- Demander ou relâcher une ressource, un sémaphore.
- Tester, ou positionner un drapeau.

etc...

CLIENT.

Les clients sont les objets circulants de la simulation. C'est leur passage au travers les différentes stations qui cadence le comportement du réseau et déclenche les services. La principale primitive qui les concerne est donc une primitive de routage (TRANSIT) qui permet de terminer un service en dirigeant le client courant vers un autre station.

Il est possible de modéliser des comportements complexes de clients grâce aux priorités et aux classes. Ces deux propriétés des objets clients permettent d'affiner les stratégies des files d'attente.

La priorité d'un client est un attribut qui permet d'ordonner plusieurs clients dans la même file. Cette priorité peut changer tout au long de la durée de vie du client.

La classe d'un client permet de caractériser les services qu'il requiert à chaque station. Tout comme la priorité, la classe d'un client peut évoluer durant la simulation.

SEMAPHORES, RESSOURCES ET DRAPEAUX.

QNAP2 fournit également un certain nombre d'outils qui permettent de synchroniser les différents serveurs: sémaphores, ressources et drapeaux. Tout ces objets possèdent une file d'attente où se trouvent les clients (ou du moins leur représentant) en attente d'un état particulier de l'objet.

VUE GLOBALE D'UN RESEAU.

Ainsi, on peut décrire un réseau de files d'attente comme la juxtaposition de stations. Chaque station se voit attribuer une file d'attente et un service. La file d'attente implante une discipline. Le service indique le comportement de l'ensemble client—serveur lorsque un service est rendu. En particulier, on y précise le temps que met ce service à être rendu, le routage du client à la sortie de la station et éventuellement la mise à jour de données locales. L'ensemble peut être paramètré par les caractéristiques du client (priorité, classe).

EXEMPLE D'UTILISATION DE QNAP2.

Afin d'illustrer les concepts décrits ci-dessus, il nous est apparu souhaitable d'exposer un exemple de description de réseau selon QNAP2. Nous prendrons comme exemple une version simplifiée du site 0 .

DESCRIPTION SIMPLIFIEE DU SITE ZERO.

Nous considérerons dans un premier temps le site 0 comme un réseau linéaire de quatres stations associées respectivement aux quatres opérations de traitement: préparation, chargement, exécution, sauvegarde. Nous y inclueront la contrainte suivante:

- A tout moment, il ne peut y avoir globalement plus de M clients en chargement, exécution et sauvegarde. Autrement dit, un chargement ne peut commencer que si une UGM est disponible.

Il s'agit ici d'une version volontairement simplifiée où l'on ne mentionne absolument pas l'impossibilité d'une préparation et d'un chargement simultané (du fait qu'il n'y a qu'un seul processeur PN).

LE SOURCE QNAP2.

On distinguera principalement dans cette description:

- la station de nom INP (ligne 4 à 10) qui a pour rôle d'engendrer les différents exemplaires de modules (ici au nombre de 5). L'intervalle de temps entre deux arrivées de modules est de 10. Il n'aurait pas été difficile de la remplacer par une loi exponentielle.

- Les quatres stations de préparation (lignes 12-14), chargement (16-21), exécution (23-25) et sauvegarde (27-32).

- La station de nom UGM est en fait une ressource (ici en deux exemplaires) conformément à la contrainte que nous avons donnée plus haut. La réservation d'une UGM (P(UGM)) s'effectue lors du chargement (ligne 19) alors que la libération prend place après la sauvegarde (V(UGM) - ligne 31).

- Les appels CST(10) et EXP(1000) permettent de déterminer la durée d'un service, durée constante égale à 10 unités ou aléatoire selon une loi exponentielle de moyenne 1000.

```
  SIMULOG   ***  QNAP2  ***  (OCTOBRE 1984) V03
(C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1982

 1   /DECLARE/ QUEUE PREP,CHAR,EXEC,SAUV;
 2   /DECLARE/ QUEUE INP, UGM;
 3
 4   /STATION/ NAME=INP;
 5             INIT=5;
 6             TRANSIT=PREP;
 7             SERVICE=
 8             BEGIN
 9             CST(10);
10             END;
11
12   /STATION/ NAME=PREP;
13             SERVICE = EXP(500);
14             TRANSIT= CHAR;
15
16   /STATION/ NAME=CHAR;
17             TRANSIT=EXEC;
18             SERVICE= BEGIN
19                      P(UGM);
20                      CST(10) ;
21                      END;
22
23   /STATION/ NAME=EXEC;
24             SERVICE=EXP(1000);
25             TRANSIT=SAUV;
26
27   /STATION/ NAME=SAUV;
28             TRANSIT=OUT;
29             SERVICE= BEGIN
30                      CST(1);
31                      V(UGM);
32                      END;
33
34   /STATION/ NAME=UGM;
35             TYPE=RESOURCE,MULTIPLE (2);
36
37   /CONTROL/  TMAX=10000;
38              TRACE=0., 100000.;
39
40   /EXEC/     BEGIN
41              SIMUL;
42              END;
43
44   /END/
STOP
```

Fig. III-4. Qnap2: Description simplifiée du site0.

QNAP2 APPLIQUE A LA SIMULATION DU SITE ZERO.

Nous allons voir comment la description du site zéro, qui parait simple grâce aux outils fournis par QNAP2, se révèle vite, sinon impossible, du moins très complexe. Les diverses solutions expérimentées ressemblent plus à des adaptations plus ou moins 'bricolées' d'un langage limité qu'à l'utilisation limpide d'un puissant modèle. Au mieux, on aboutira à la description artificielle d'un phénomène pourtant intuitivement simple.

PROBLEMES NON RESOLUS.

Le premier jet cité plus haut simule effectivement bien le comportement d'un site 0 qui disposerait de quatre processeurs. Avant d'aller plus loin, il fallait introduire l'hypothèse de l'unicité de PN: les opérations de préparation, chargement et sauvegarde devant être mutuellement exclusives.

Nous avons trouvé deux réponses à la question posée:

Une première idée est de laisser la description des trois serveurs (préparation, chargement et sauvegarde), mais en aménageant l'algorithme de telle manière qu'un seul d'entre-eux puisse être actif à la fois.

La deuxième solution que nous avons imaginée regroupe les trois opérations sur un seul serveur baptisé PN. A charge pour ce dernier de sélectionner - à l'aide du mécanisme des priorités dans les files d'attentes - l'opération à exécuter selon la stratégie testée.

HYPOTHESE 1 (TROIS SERVEURS).

La méthode utilisée pour interdire le fonctionnement simultané de plusieurs serveurs parmi les trois cités (PREP, CHAR et SAUV) et d'associer un sémaphore à chacun d'entre-eux, appelés respectivement PSEM, CSEM et SSEM. Dès son arrivée dans le service, le client en cours demande l'acquisition du sémaphore qui le concerne (phase 1). Si un autre service est déjà en cours, cette action sera refusée et le client sera mis en attente. Cette attente s'effectuant à l'intérieur du service, les autres clients de ce serveur resteront dans la file associée. Inversement, dès qu'un client a

terminé son service, il détermine quelle est l'opération suivante que devrait effectuer PN et libère le sémaphore correspondant (phase 2). Ainsi, l'opération décidée peut s'exécuter si un client est disponible pour cela. D'un autre coté, l'opération qui vient de se terminer, si elle n'est pas élue, ne peut pas reprendre car le client partant n'a pas relâché son sémaphore. Son successeur se trouvera alors bloqué dès son entrée dans le service (lors de la phase 1). Initialement, seul PSEM doit avoir une valeur de 1 afin de rendre possible la première préparation. Les autres sémaphores (CSEM et SSEM) sont positionnés à 0.

Le choix de l'opération suivante (procédure choix) peut se faire au vu des cardinalités (nombre de clients présents) des files d'attentes. Notons qu'un client en cours de service est encore considéré comme appartenant à la file associée, même si il est bloqué sur un sémaphore.

Le principal problème qui se pose alors est de détecter les cas où aucun des serveurs n'a de client en attente. Il n'est alors pas possible de déterminer avec précision quel est la prochaine opération à exécuter car c'est l'opération demandée par le prochain client qui n'est pas encore connue. Cet état correspond à l'état oisif du processeur N. Une parade est de reprendre la même procédure de choix en sortie des services qui fournissent des clients à PN (soit INP et EXEC) procédure qui serait appliquée quand les trois files (PREP, CHAR et SAUV) sont vides.

```
SIMULOG   ***  QNAP2  ***  (OCTOBRE 1984) V03
(C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1982


 1      /DECLARE/ QUEUE PREP,CHAR,EXEC,SAUV;
 2              QUEUE INP ;
 3              QUEUE PSEM,CSEM,SSEM;
 4              INTEGER NBMMU;
 5
 6              PROCEDURE CHOIX (IP,IC,IS);
 7              INTEGER IP,IS,IC;
 8              BEGIN
 9              PRINT ("NBMMU =  ",NBMMU) ;
10              PRINT ("PREP  =  ",IP) ;
11              PRINT ("CHAR  =  ",IC) ;
12              PRINT ("SAUV  =  ",IS) ;
13
14              IF (IS>0) THEN
15                 BEGIN
16                 PRINT ("CHOIX = SAUV ") ;
17                 V(SSEM) ;
18                 END
19              ELSE IF (NBMMU > 0) AND (IC>0) THEN
20                 BEGIN
21                 PRINT ("CHOIX = CHAR ") ;
22                 V(CSEM);
23                 END
24              ELSE
25                 BEGIN
26                 PRINT ("CHOIX = PREP ") ;
27                 V(PSEM) ;
28                 END ;
29              END ;
30
31   /STATION/ NAME=PSEM;
32              TYPE=SEMAPHORE;
33   /STATION/ NAME=CSEM;
34              TYPE=SEMAPHORE,MULTIPLE(0);
35   /STATION/ NAME=SSEM;
36              TYPE=SEMAPHORE,MULTIPLE(0);
37
38   /STATION/ NAME=INP;
39              INIT=5;
40              TRANSIT=PREP;
41              SERVICE=
42              BEGIN
43              CST(10);
44              END;
45
46   /STATION/ NAME=PREP;
47              TRANSIT= CHAR;
48              SERVICE =
49                 BEGIN
50                 P(PSEM);
51                 EXP(500);
52                 CHOIX (PREP.NB-1, CHAR.NB+1, SAUV.NB) ;
53                 END ;
54
55   /STATION/ NAME=CHAR;
56              TRANSIT=EXEC;
57              SERVICE= BEGIN
```

```
58                      P(CSEM);
59                      NBMMU := NBMMU-1;
60                      CST(10) ;
61                  CHOIX (PREP.NB, CHAR.NB-1, SAUV.NB) ;
62                      END;
63
64    /STATION/ NAME=EXEC;
65              SERVICE=EXP(1000);
66              TRANSIT=SAUV;
67
68    /STATION/ NAME=SAUV;
69              TRANSIT=OUT;
70              SERVICE= BEGIN
71                      P(SSEM);
72                      CST(1);
73                      NBMMU := NBMMU+1;
74                      CHOIX (PREP.NB, CHAR.NB, SAUV.NB-1) ;
75                      END;
76
77    /CONTROL/  TMAX=10000;
78               TRACE=0., 100000.;
79
80    /EXEC/     BEGIN
81               NBMMU := 2 ;
82               SIMUL;
83               END;
84
85    /END/

STOP
```

Fig. III-5. Qnap2: Description du site0 selon trois serveurs.

Les critiques que l'on peut faire à cette solution sont, d'une part la complexité de programmation des algorithmes de choix (en ne parlant que des stratégies à priorités fixes) mais surtout l'imprécision des résultats locaux (mesures de charge des processeurs par exemple). En effet, les temps de blocage des clients sur les sémaphores sont comptabilisés comme temps de service - c'est normal puisque les client sont bloqués **à l'intérieur** des services -.

HYPOTHESE 2 (UN SEUL SERVEUR)

L'alternative à la solution précédente est de ne considérer qu'un seul serveur (ce qui est d'ailleurs plus proche du modèle physique). Nous avons alors testé le programme suivant:

```
SIMULOG    ***  QNAP2  ***  (OCTOBRE 1984) V03
(C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1982

 1   /DECLARE/ QUEUE PN, PC;
 2   /DECLARE/ QUEUE INP, UGM;
 3   /DECLARE/ CLASS CLSP,CLSC,CLSS;
 4
 5   /STATION/ NAME=INP;
 6             INIT(CLSP)= 5;
 7             SERVICE=
 8                BEGIN
 9                CST(10);
10                TRANSIT (PN,CLSP,0) ;
11                END ;
12
13   /STATION/ NAME= PN;
14             TRANSIT(CLSP) = PN,CLSC,1;
15             TRANSIT(CLSC) = PC;
16             TRANSIT(CLSS) = OUT;
17             SERVICE =
18                BEGIN
19                IF (CUSTOMER.CCLASS = CLSP) THEN
20                   BEGIN
21                   EXP(500) ;
22                   TRANSIT (PN, CLSC, 1) ;
23                   END
24                ELSE IF (CUSTOMER.CCLASS = CLSC) THEN
25                   BEGIN
26                   CST(10) ;
27                   TRANSIT (PC) ;
28                   END
29                ELSE IF (CUSTOMER.CCLASS = CLSS) THEN
30                   BEGIN
31                   CST(1) ;
32                   TRANSIT (OUT) ;
33                   END ;
34                END;
35
36   /STATION/ NAME= PC;
37             SERVICE=
38                BEGIN
39                EXP(1000);
40                TRANSIT (PN,CLSS,2) ;
41                END ;
42
43   /CONTROL/  TMAX=10000;
44              TRACE=0., 100000.;
45
46   /EXEC/     BEGIN
47              SIMUL;
48              END;
49
50   /END/
STOP
```

Fig. III-6. Qnap2: Description du site0 selon un seul serveur.

Nous voyons qu'ici, la complexité du choix de l'opération suivante est localisée dans le choix de la classe et de la priorité des clients qui arrivent dans la file d'attente de PN. Cette solution répond bien aux critiques précédentes: les états oisifs de PN sont bien traités, de manière naturelle et sans perte d'efficacité. Qui plus est, les valeurs issues de la simulation sont des valeurs exactes directement exploitables.

Néanmoins, cette méthode perd rapidement de son intérêt au fur et à mesure que les contraintes arrivent. En effet, les priorités assignées aux clients lors de leur trajet sont fixes: il est impossible de modifier les priorités d'un client lors qu'il est en attente ou, du moins, la documentation n'indique pas les effets d'une telle manipulation.

Or, le choix de l'opération suivante n'est pas seulement déduit de la stratégie et de la liste des opérations possibles. En particulier, lorsque deux clients se présentent devant PN, l'un pour la préparation, l'autre pour le chargement, la réponse à ces sollicitations pourra être différente selon la disponibilité des UGM: si aucune UGM n'est libre, la préparation sera choisie de préférence à un chargement même prioritaire.

Le pendant de cette remarque en programmation QNAP2 est la possibilité de bloquer une classe (ici se serait la classe de chargement CLSC) pour servir les clients en attente de préparation (classe CLSP). Ceci n'est pas possible car nous avons vu que les blocages - qu'ils soient devant un sémaphore, une ressource ou un drapeau - ne peuvent avoir lieu qu'à l'intérieur d'un service.

PROPOSITIONS.

Ainsi, malgré un abord séduisant, la programmation en QNAP2 du site 0 parait bizarrement artificielle. Après réflexion, nous dirons qu'il nous a manqué un seul outil: La possibilité de décrire sous forme algorithmique la procédure de choix d'un client (stratégie) dans une file. C'est d'autant plus dommage que les algorithmes peuvent connaître les données relatives à ces files. Une telle procédure aurait pu prendre en compte des éléments aussi complexes que la disponibilité des UGM et même - sujet que nous n'avons pas abordé ici - la gestion de la mémoire de bloc -. Ceci est une nouvelle illustration de la puissance des descriptions par algorithmes par rapport aux descriptions par données.

Pour être juste, nous devons mentionner le gain de productivité impressionnant que permet ce logiciel. Nous estimons que l'utilisation de ce dernier nous aurait fait gagner environ 80% du temps de développement.

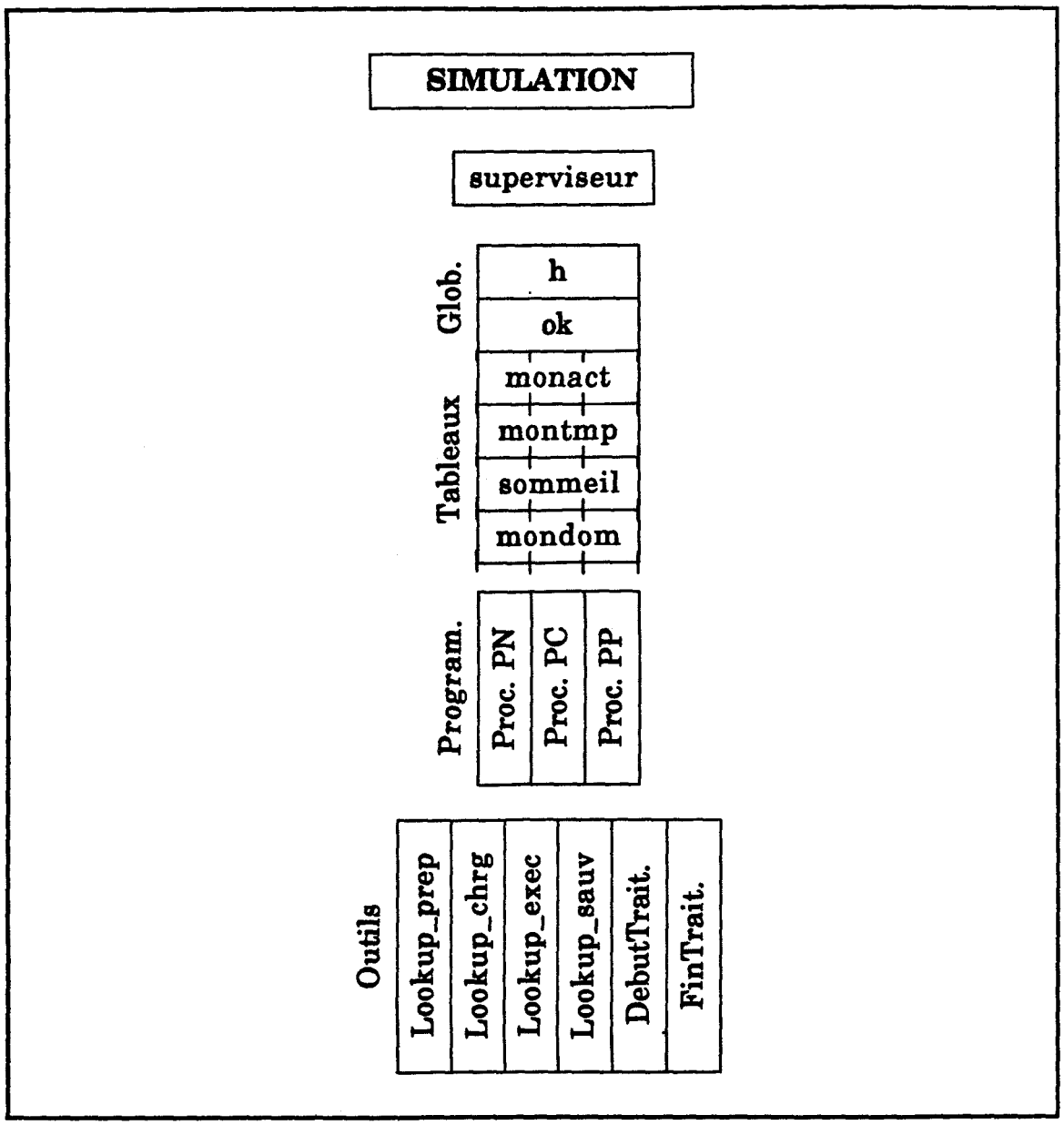
III.B.3. LE PROGRAMME PASCAL.

Après avoir constaté les insuffisances des outils que nous avions à notre disposition, nous nous sommes tournés vers une solution plus rudimentaire mais éprouvée: écrire de toutes piéces un programme de simulation en PASCAL. Ce programme a été ralisé sur système MULTICS.

L'ensemble de l'application est décomposé en trois parties: la création des données, la simulation et l'analyse des résultats. Chacune de ces parties comprend plusieurs programmes qui ont été écrits au fur et à mesure des besoins. Cette section doit vous donner les caractéristiques du programme de simulation proprement dit.

LE PRINCIPE DE LA SIMULATION.

Pour écrire la simulation, nous avons considéré que les entités actives du site zéro sont les activités propres à chaque processeur (par extension, les processeurs eux-même) plutôt que les modules. Ceci permet une programmation plus naturelle et une réduction du nombre des objets actifs. A partir de cette base, nous programmons indépendamment chaque processus selon une spécification commune: les processus présenteront donc une interface uniforme. Un programme superviseur est alors chargé de considérer les différentes interfaces pour déterminer la date du prochain événement dans le système, puis d'activer chacun des processus pour la date calculée à la manière des systèmes temps-réel: Il donne la main au(x) processus dont la date de terminaison d'opération est nulle. Ce processus met à jour les différentes variables puis rend la main au superviseur. La fin de la simulation se produit quand il n'y a plus de processus actif ni de module en file d'attente. On fournit bien sûr aux processus des outils pour les tâches que ceux-ci doivent exécuter. Nous décrirons également ces outils.



SPECIFICATIONS.

Les processus sont écrits sous la forme d'autant de procédures à un paramètre. Ce paramètre indique le numéro du processeur (le programme est prévu pour simuler plusieurs processeurs PC, mais cette possibilité n'a pas été exploitée). De ce fait, les spécifications des processus consistent en un ensemble de variables de communication entre le programme superviseur d'une-part et chacun des processus programmés d'autre-part. Ces variables sont de deux types:

Les variables dites 'simples' sont des variables qui sont mises à jour par le programme superviseur et qui peuvent être consultées par tous les processus. l'heure `h` est un exemple de variable simple.

Les variables 'multiples' sont des tableaux indicés par le nom des processus. Chaque élément de ces tableaux est lu ou écrit par le superviseur lui-même ou par le processus qui lui sert d'indice. Ainsi `montmp(PN)` désire la variable multiple de nom `montmp` qui est attribuée au processus PN.

LES VARIABLES SIMPLES.

- `h`. heure courante.
  Cette variable entière affiche l'heure actuelle de la simulation. Cette variable est entièrement gérée par le programme superviseur.

- `ok`.
  Cette variable booléenne permet la détection d'une anomalie par le superviseur. Sa mise à vrai déclenche l'arrêt immédiat de la simulation.

LES VARIABLES MULTIPLES.

MONACT().

Permet à un processus de mémoriser le type de l'activité qui est entreprise actuellement par le processus. Cette activité est une valeur dans l'ensemble {`rien`, `prep`, `chrg`, `exec`, `sauv`}.

MONTMP().

Cette variable permet à un processus d'afficher le temps restant avant son réveil. Quand le processus dort, cette variable contient 0. Le superviseur va décrémenter les compteurs positifs à chaque fois qu'il fait avancer l'état du système.

SOMMEIL().

Cette variable booléenne permet au superviseur de savoir si un processus est en sommeil ou pas. La définition du sommeil est la suivante: Un processus est en sommeil lorsqu'il a besoin d'un événement

extérieur pour changer d'état. Ainsi, un processus qui traite un module n'est pas en sommeil car il changera d'état au plus tard quand le traitement du module sera terminé (ce qui n'est pas un événement extérieur). A partir de cette définition, on estime que lorsque toutes les files d'attente sont vides et que tous les processus sont endormis, il n'y aura plus de changement d'état et que la simulation est terminée. Vu la programmation du processus fournisseur décrite plus bas, ceci ne se produira que quand tous les modules auront été exécutés.

Cette variable est positionnée par les appels aux procédures FINTRAITEMENT et DEBUTTRAITEMENT.

MONDOM().

Permet à un processus de mémoriser le numéro du module qui est en train d'être traité.

EXEMPLE.

Nous vous proposons un exemple restreint (nous ignorons le processus producteur).

Supposons qu'à un moment donné, on ait la configuration suivante:

```
h = 100 ;
monact(PN)= chrg; montmp(PN)= 10; mondom(PN)= 4; sommeil(PN)= faux ;
monact(PC)= exec; montmp(PC)=  5; mondom(PC)= 3; sommeil(PC)= faux ;
```

Le superviseur peut en déduire qu' à l'instant t = 100 :

- PN est en cours de chargement du module 4, opération qu'il terminera dans 10 unités de temps (soit en h + 10 = 110).
- PC est en cours d'exécution du module 3. Cette opération sera terminée à h + 5 (soit en 105).

Et donc, que le prochain événement aura lieu en 105. Il avance alors h, diminue les montmp de la valeur du pas (5), puis active le processus PC.

Ce dernier verra l'environnement suivant:

```
h = 105 ;
monact(PC)= exec; montmp(PC)=  0; mondom(PC)= 3; sommeil(PC)= faux ;
```

Qui signifie qu'il vient de terminer l'exécution du module 3. Si l'on suppose que la file d'attente FAX des modules chargés est vide, PC se mettra alors en sommeil:

```
monact(PC)= rien; montmp(PC)=  0; mondom(PC)= 0; sommeil(PC)= vrai ;
```

Il se trouve que dans ce cas particulier une seule activation de processus est nécessaire (PN continue l'opération en cours). Ce n'est pas toujours vrai: En effet au temps t = 110, l'état des variables sera le suivant:

```
h = 110 ;
monact(PN)= chrg; montmp(PN)=  0; mondom(PN)= 4; sommeil(PN)= faux ;
monact(PC)= rien; montmp(PC)=  0; mondom(PC)= 0; sommeil(PC)= vrai ;
```

Au vu desquelles le superviseur activera le processus PN qui décrétera alors la fin de chargement et choisira son opération suivante. Or, la fin de chargement remplira le file FAX. Il faudra donc réveiller PC car ce dernier a du travail. L'ordre d'activation a une importance; il aurait été inutile ici de réveiller PC avant que PN ait mis le module 4 dans la FAX. Inversement lorsqu'une fin d'exécution entraine le réveil de PN (en vue d'une sauvegarde), il faut activer PC avant PN. Cependant, l'activation d'un processus qui n'est concerné ni directement, ni indirectement par l'événement en cours ne gène pas la simulation. La solution adoptée est donc la suivante: dés qu'un événement se produit, on active tous les processus (dans un ordre arbitraire). On recommence ces activations tant que l'environnement se modifie. Une variable est mise à vrai au début de chaque cycle. Un processus la met à faux si il modifie l'environnement. De cette manière, PN peut réagir à une action de PC, elle-même déclenchée par un événement survenu sur PN...

DETAIL.

PROGRAMMATION DU SUPERVISEUR.

Le programme superviseur a pour but de gérer l'activation des processus et l'horloge de la simulation .

Un processus doit être activé quand:

- Son compteur `montmp` atteint 0. Ceci signifie que le processus en question vient de terminer une opération et qu'il doit donc décider de la suite des événements (lancement d'une nouvelle opération ou sommeil).

- L'état du système a changé. Or, l'état du système ne peut changer que par l'action des processus eux-même (les modules sont introduits par un processus producteur).

Le superviseur devra alors activer tous les processus jusqu'à l'obtention d'un état stable (toutes les réactions des processus ont été prises en compte). Pour les algorithmes utilisés, l'ordre d'activation des processus n'a pas d'importance. Cette remarque ne peut pas être généralisée pour tous les algorithmes.

L'horloge sera ensuite avancée pour atteindre le prochain événement. Il y a recherche du minimum des valeurs de `montmp` strictement positives; Les valeurs nulles de `montmp` correspondent à des processus endormis. En effet, un processus endormi ne produira pas d'événement de sa propre initiative tandis qu'un processus **p** réveillé ne produira pas d'événement avant `montmp(p)`.

Quand cet intervalle est déterminé, il suffit d'avancer l'horloge et de décrémenter les compteurs des processus actifs de la valeur de l'intervalle.

PROGRAMMATION DES PROCESSUS.

Nous nous intéressons ici à la manière de programmer les processus. Nous avons décomposé le fonctionnement du site zéro selon trois processus. Un processus pour simuler le fonctionnement de chaque processeur et un processus producteur qui sert à cadencer l'arrivée des nouveaux modules. Nous pouvons, ainsi étendre le programme pour simuler un site qui disposerait de plusieurs processeurs de calcul identiques.

Un processus devra réagir à chaque activation. Pour cela il pourra consulter les variables qui font partie des spécifications énoncées plus haut. Il lui faut d'abord regarder si il est directement concerné par

l'événement actuel (test de nullité de montmp). Si c'est le cas, le processus peut consulter monact pour savoir si lui-même a une opération en cours. Si la réponse est positive, il lui faut terminer cette opération. Enfin, il doit choisir sa nouvelle activité à partir de l'état de la machine et de sa stratégie (nous avons écrit un programme N par stratégie testée et un programme C). Pour cela, le programmeur dispose d'outils généraux fournis sous la forme de procédures.

OUTILS GENERAUX.

Nous avons fourni deux sortes de procédures: Les tests de faisabilité (Lookup_xxxx) et les primitives de début et de fin d'opération.

Les tests de faisabilité calculent si une opération particulière est possible:

- Lookup_prep:
  Calcule si une préparation est possible. Pour cela, il faut au moins un module en attente de préparation. Il faudra ensuite au moins un bloc mémoire disponible (voir la gestion mémoire).

- Lookup_chrg:
  Pour qu'un chargement soit possible, il faut qu'il y ait au moins un module en attente de chargement et au moins une UGM libre. Ce test peut ne pas être effectué au moyen de la circuiterie de partage car PN peut compter lui-même le nombre d'UGM libres (sous réserve qu'il connaisse le nombre d'UGM installées sur le site) car seul ce processeur effectue les libérations (fins de sauvegarde) et chargements des UGM.

- Lookup_exec:
  Pour qu'une exécution soit possible, il suffit qu'il y ait une UGM dans l'état attente d'exécution. Une telle UGM désigne forcément un module en attente d'exécution.

- Lookup_sauv:
  Une opération de sauvegarde peut être engagée si on trouve une UGM en attente de sauvegarde. La aussi, un module dans le bon état lui est associé.

Les primitives de début et de fin d'opération DEBUTTRAITEMENT et FINTRAITEMENT permettent de systématiser les manipulations mise en jeu lors des changements d'opérations. Elle effectuent l'écriture de l'événement sur le fichier 'compte-rendu' qui sera analysé après coup. Elles assurent également la mise à jour des variables de la simulation.

- DEBUTTRAITEMENT force les valeurs des variables locales mondom, montmp, sommeil et monact et enregistre l'événement réalisé.
- FINTRAITEMENT marque la fin de l'activité en cours.

LE PROCESSUS PRODUCTEUR P.

Ce processus producteur est un peu particulier dans le sens où il ne correspond pas à un processeur réel et, de ce fait, nous n'avons pas besoin, pour l'évaluation des performances, de connaître ses phases d'activité. Son seul rôle et d'injecter dans la machine les modules aux moments opportuns calculés à l'aide des intervalles d'arrivée des modules. Si elles ne sont pas constantes, les caractéristiques des modules sont lues à partir du fichier de nom **JDxx** qui contient le jeu de données. La création de ces fichiers est assurée par un utilitaire annexe **gene**. Ce processus s'endort uniquement quand il a épuisé le fichier de données.

LES PROCESSUS DE CALCUL C.

Ce processus simule le comportement de PC. Ce processeur n'a qu'une alternative quant-au choix de son activité. Il peut travailler quand il a un module disponible (ce qui peut être détecté par la réponse affirmative à une demande d'UGM), et dans ce cas, il entre dans une phase d'exécution. Dans l'hypothèse contraire, ce processeur ne peut rien faire et il entre dans une phase de sommeil.

On peut également signaler le fait que ce processeur est seul maître de son destin. J'entends par là que si il a engagé une opération, il ne changera pas de phase d'activité avant sa terminaison totale. Ceci est à opposer au comportement de PN selon la stratégie INT – décrite plus loin –.

L'algorithme de programmation du processus C est donc:

Processus C (noproc)

```
si montmp [noproc] = 0 alors
|     FINTRAITEMENT (noproc) ;
|     si lookup_exec alors
|  |        DEBUTTRAITEMENT (exec, init, noproc)
|     sinon
|  |        DEBUTTRAITEMENT (rien, init, noproc)
|     fsi
fsi
```

Fig. III-7. Algorithme du processus C.

Ce processus peut éventuellement déclencher une interruption en fin de traitement à destination de PN.

LE PROCESSUS NOYAU N.

L'algorithme de description du comportement de PN reflétera la stratégie adoptée. Pour illustrer la méthode générale de programmation de PN pour les stratégies à priorités, nous détaillerons la stratégie CSP.

STRATEGIES A PRIORITES (CSP, CPS, PCS, PSC, SCP, SPC)

Pour ces stratégies, il suffit de tester successivement la faisabilité des opérations dans l'ordre de priorité choisi et d'engager la première opération possible. Si aucune opération n'est réalisable, le processus s'endort. Ceci donne le style d'algorithme suivant :

```
Processus N (noproc) { priorité CSP }
si montmp [noproc] = 0 alors
|     FINTRAITEMENT (noproc)
|     si lookup_chrg alors
|  |        DEBUTTRAITEMENT (chrg, init, noproc)
|     sinon si lookup_sauv alors
|  |        DEBUTTRAITEMENT (sauv, init, noproc)
|     sinon si lookup_prep alors
|  |        DEBUTTRAITEMENT (prep, init, noproc)
|     sinon
|  |        DEBUTTRAITEMENT (rien, init, noproc)
fsi
```

Fig. III-8. Algorithme du processeur N.

EXTENSIONS POUR LA STRATEGIE INT.

La simulation d'interruptions déclenchées par PC et traitées par PN demande l'introduction de nouveaux mécanismes. Ces extensions se présentent sous deux aspects :

- Il est possible que PN doive répondre à un événement alors même qu'il n'a pas terminé l'opération en cours. Par bonheur, les régles d'activation, comme elle ont été définies plus haut répondent à cette extension.

- Une opération peut être réalisée par PN en plusieurs fois: une préparation est interrompue par une sauvegarde puis par un chargement avant d'être complétée. Cette extension a été réglée par l'adjonction de nouveaux éléments: le tableau mon_pc et les procédures DEBUT_IT et RETOUR_IT.

ADJONCTIONS NECESSAIRES.

MON_PC().

La connaissance de l'activité en cours n'est pas toujours suffisante à un processus pour déterminer son comportement. En particulier lorsque qu'une activité se déroule en plusieurs fois. L'information supplémentaire est alors fournie par un pseudo-compteur ordinal qui doit être géré localement par le processus lui-même. Le style de programmation qui en résulte est détaillé dans la partie suivante. A l'initialisation, ces pseudo-compteurs ont la valeur conventionnelle "init". Les autres valeurs que peuvent prendre cette variable sont locales à chaque processus.

NOUVELLES PROCEDURES.

On disposera également, pour les cas où il y a préemption d'un processus par un module, des opérations DEBUT_IT et RETOUR_IT qui assurent les services et les retours d'interruption.

PROGRAMMATION DU PROCESSUS N.

C'est du coté PN que la compléxité de programmation de la stratégie INT apparait: Elle nécessite la faculté de sauvegarder et de restaurer l'état du processus. En effet, il est possible qu'un changement d'activité se

produise au milieu d'une phase quelconque. Nous verrons également qu'il faut tenir compte d'une phase initiale de chargement des UGM.

Le principe de cette stratégie est le suivant: Nous allons essayer de réduire la durée d'inoccupation des UGM (par des modules en attente ou par l'absence de module).

En regardant le cycle d'utilisation des UGM, nous pouvons nous apercevoir que les périodes passives (c'est-à-dire celles où une UGM n'est pas utilisée) sont les états d'occupation par les modules en attente d'exécution, en attente de sauvegarde ou encore l'état 'libre'. Nous voulons réduire le temps d'attente de sauvegarde par interruption; une UGM peut être sauvegardée quand elle contient un module dont l'exécution a été complétée. Si PC signale de manière asynchrone à PN qu'il vient de terminer une exécution, et il a donc alors restitué l'UGM concernée, PN peut alors immédiatement commencer la sauvegarde (il suffit qu'une UGM soit en attente de sauvegarde pour pouvoir engager la-dite opération). Nous pouvons ainsi assurer un temps d'attente de sauvegarde nul pour l'ensemble de la simulation.

Mais nous n'allons pas nous arrêter en si bon chemin! Nous pouvons encore réduire la durée de liberté de l'UGM sauvegardée. Il suffit simplement qu'un seul module soit en attente de chargement pour pouvoir recharger l'UGM dans la foulée. L'UGM arrive alors en attente d'exécution. A ce niveau la stratégie choisie par PN est sans effet sur ce temps d'attente. En revanche (et sauf cas d'espèce), PC disposera d'un maximum de modules prêts à être exécutés, ce qui diminue d'autant son temps d'oisiveté. La préparation des modules devient alors une tâche de fond de PN, moins prioritaire que les opérations de chargement et de sauvegarde et interruptible par elles.

Ce fonctionnement demande cependant quelques ajustements. Nous avons supposé que nous avions assez de module prêts pour enclencher le chargement immédiatement après la sauvegarde. Ceci n'est pas toujours vrai. Par ailleurs, au démarrage, l'ensemble des UGM est libre et il faut bien amorcer le comportement décrit en chargeant au moins un module (un maximum serait préférable). Ces deux problèmes se résolvent de la même manière; il suffit de déclencher un chargement chaque fois que

cela est possible, soit en fin de préparation, soit à l'initialisation. Le test de faisabilité du chargement restant conforme à la procédure générale.

Quand l' exécution d'un module dure moins longtemps que la sauvegarde du module précédent, une interruption arrive à PN alors qu'il est en cours de sauvegarde. Etant donné que cette opération est liée à une UGM particulière, PN ne peut traiter l'interruption avant d'avoir terminé la sauvegarde en cours. Le masquage des interruptions permet d'éviter d'imbriquer les opérations de sauvegarde. Notons que ce cas de figure est impossible dans la simulation actuelle (avec TS = 1 et TX ≥ 1). La remarque reste valable dans le cas général.

En résumé, la stratégie pourrait s'écrire de la manière suivante sur le processeur PN:

```
boucle
    si lookup_chrg
    |     alors chargement
    sinon si lookup_prep
    |     alors préparation
    { sinon rien }
    fsi
finboucle
```

```
service d'interruption
masquer_les_it
sauvegarde
si lookup_chrg alors chargement
demasquer_les_it
fin service
```

Fig. III-9. Stratégie INT service d'interruption.

Il reste à écrire cet algorithme de manière compatible avec les spécifications des processus. Ceci se fait de la façon suivante:

Quand PN prend la main, il regarde d'abord si une interruption non masquée est arrivée. Si c'est la cas, le masque est positionné et le service d'interruption est enclenché, il comporte:

1- Masquer les interruptions.

2- Sauver l'état courant du processus.

3- Lancer une sauvegarde.

4- Lancer un chargement.

5- Restaurer l'état du processus.

6- Démasquer les interruptions.

Ces 6 étapes peuvent se programmer classiquement sauf les étapes 3 et 4 qui prennent un certain temps "de simulation". Pour ces dernières, le processus doit rendre la main au superviseur ne serait-ce que parce que d'autres événements peuvent se produire dans l'intervalle.Le service se déroule donc en trois fois (étapes 1, 2, 3, puis 4, puis 5 et 6). Le pseudo-pc est utilisé pour retenir où on en est dans ce traitement. L'étiquette `suit_it` marque l'étape 4, l'étiquette `ret_it` marque l'étape 5. Notons que hormis pour l'interruption elle-même, les deux dernières activations du processus se font dans des circonstances classiques, c'est-à-dire lorsque le compteur `montmp` du processus atteint 0.

Si une interruption n'est pas en attente ou est masquée, on regarde la valeur de `montmp`.

Si `montmp` est non nul, il n'y a rien a faire.

Si la nullité de `montmp` est établie, ceci peut être dû à plusieurs causes établies grâce à `mon_pc`:

- Une interruption est en cours de service (`mon_pc` vaut `suit_it` ou `ret_it`). Il suffit de poursuivre le service.

- Une préparation vient de se terminer ou un chargement initial (un chargement qui n'est pas la conséquence d'une interruption) ou encore la simulation vient de commencer. Il convient alors de lancer un chargement si ceci est possible, sinon de lancer une préparation, sinon de dormir.

Le processus programmé est donc le suivant:

```
Processus N  { stratégie INT }
si it et non masque alors
|     masque = vrai
|     debut_it { sauver l'état courant }
|     DEBUTTRAIT (sauv, suit_it, N)
fsi
si montmp = 0 alors
|     FINTRAITEMENT
|     cas mon_pc:
|     init :
|     |     |     initialise { les variables rémanentes it et
|     |     |  masque}
|     idle:
|     |     si lookup_chrg alors
|     |     |     DEBUTTRAIT(chrg, idle, N)
|     |     sinon si lookup_prep alors
|     |     |     DEBUTTRAIT(prep, idle, N)
|     |     sinon
|     |     |     DEBUTTRAIT(rien, idle, N)
|     |     fsi
|     suit_it:
|     |     si lookup_chrg alors
|     |     |     DEBUTTRAIT (chrg, ret_it, N)
|     |     |     sinon {pas de chargement possible après
|     |     |  sauvegarde }
|     |     |     retour_it
|     |     |     masque = faux
|     |     |     REPRISETRAITEMENT
|     |     fsi
|     ret_it:
|     |     fin_it
|     |     masque = faux
|     |     REPRISETRAITEMENT
|     fin cas
fsi
```

Fig. III-10. Stratégie INT: le programme.

III.C. LES EXPERIMENTATIONS EFFECTUEES.

Ce chapitre a pour objet de décrire les manipulations réalisées ainsi que les résultats que l'on a pu en déduire. Nous verrons que, au fur et à mesure que le modèle s'est compliqué, nous avons introduit de plus en plus d'outils, soit pour réaliser la simulation elle-même, soit pour analyser plus facilement les résultats. A cause de cela, nous allons, pour chaque expérimentation, donner son principe, ses outils propres, ses résultats bruts et les analyses que nous en avons faites.

De plus, il parait opportun de décrire à ce niveau la démarche intellectuelle entreprise. Quand le modèle fut créé et le simulateur réalisé, il fallut établir un plan d'étude: le nombre des paramètres en jeu disqualifiant une évaluation 'au petit bonheur'. L'idée naturelle qui vient alors est de partir d'un modèle simple et de le compliquer peu à peu. La section suivante expose les méthodes utilisées pour simplifier le modèle.

III.C.1. REDUCTION DU NOMBRE DE PARAMETRES

Une première simplification peut être faite en ne considérant que des jeux d'essai constants (que nous avons appelé tests sans dispersion). L'idée sous-jacente est d'obtenir des situations reproductibles et facilement analysables.

Ensuite, il fallait réduire le plus possible le nombre de paramètres. Cette réduction peut être le résultat de trois méthodes :

- Evaluation des facteurs constants spécifiques de l'architecture matérielle. Ceux-ci sont au nombre de deux: le temps de chargement et le temps de sauvegarde. Cette méthode qui réduit la généralité des résultats est toutefois la moins constestable.

- On peut montrer qu'un paramètre n'intervient que de façon simple dans les résultats. Il faut pour cela effectuer des simulations les plus diverses, en exhibant alors une loi simple. C'est, par exemple, ce que nous avons fait pour le paramètre 'intervalle d'arrivée'.

- Une autre possibilité est de négliger complètement ce paramètre en première approche – quitte à idéaliser quelque peu le modèle – Il suffira en fin d'étude de montrer que l'extrapolation des régles mises en évidence au début reste valable pour le modèle complet.

Quand nous avons déterminé l'influence d'un paramètre, il est alors possible de continuer l'étude sur le reste des paramètres : le volume de jeu d'essais à effectuer en est réduit d'autant. D'où l'importance de l'ordre d'étude des paramètres. En étudiant les éléments les plus facilement simplifiables d'abord, on économise en volume de calcul et en complexité lors des étapes suivantes.

CALCUL DE TC ET TS.

Le chiffrage des temps de chargement et de sauvegarde des registres des MMU est simple: La puissance des instructions du Z8001 permet en effet de réaliser ces deux opérations en un minimum d'instructions.

L'opération de chargement se fait en environ une centaine de micro-secondes grâce à l'instruction OTIR qui effectue l' écriture dans l'espace d'entrée-sortie d'une zone mémoire complète (voir annexe). Un calcul précis donne une durée de $TC = 55 + 40 * n$ cycles si n est le nombre de descripteurs de segments qui doivent être chargés. Soit pour 8 descripteurs (voir l'implantation d'OMPHALE) une durée de 375 cycles ou encore 93,75 µs pour la fréquence d'horloge utilisée de 4 MHz). Nous arrondirons à 100 µs soit 10 unités de temps (nous avons pris l'unité de temps égale à 10 µs).

L'opération de sauvegarde a une durée nettement plus courte du fait que, en fonctionnement normal, il n'y a pas d'information à aller rechercher dans les registres des MMU. Il s'agit alors uniquement d'une commutation électronique. Cette opération dure 19 cycles ou encore 4,75 µs, valeur arrondie à 10 µs (1 unité de temps).

III.C.2. LES TESTS SANS DISPERSION

Une première série d'expérimentations a été effectuée en proposant comme jeu de données une série de modules de caractéristiques identiques. En éliminant le facteur aléatoire du jeu d'essai, on peut, espérer avoir une bonne approximation du comportement réel du site, facilement reproductible, et permettant de définir des lois simples qui donnent la durée d'exécution en fonction des paramètres.

D'un autre coté, l'uniformisation des caractéristiques entraîne une fréquence plus grande de cas particuliers dûs à la coïncidence de plusieurs événements. Par exemple, pour un jeu d'essai qui comprend uniquement des modules de caractéristiques «IA=1,TP=489,TX=500» (la notation utilisée est celle de la section "paramètres des modules"), il y aura toujours simultanéité entre la fin de l'exécution d'un module et la fin de préparation d'un autre module et donc le résultat de la simulation

indiquerait un temps d'attente de chargement nul pour tous les modules, résultat qu'on ne peut évidemment pas généraliser.

Il faut donc plutôt voir ces tests sans dispersion comme une approche permettant de deviner la forme générale des résultats et non comme une simulation absolument réaliste.

INFLUENCE DE L'INTERVALLE D'ARRIVEE.

Le paramètre 'intervalle d'arrivée' est de loin le plus discutable de tous les paramètres. En effet, si il influe de manière flagrante sur la durée d'une simulation, c'est de manière complètement indépendante de l'architecture. C'est celui-ci que nous étudierons en premier.

HYPOTHESE TESTEE.

L'ambition de cette étude est d'évaluer la puissance des différentes configurations. Le révélateur principal de cette puissance pourrait être la durée de la simulation. Si on adopte ce critère de mesure, on s'aperçoit que l' intervalle d'arrivée introduit un biais dans cette mesure; un module arrivé à l'instant t ne pourra pas sortir du site avant l'instant t . Inversement, nous constaterons que la longueur de la file d'attente de préparation (c'est à dire le nombre de modules qui ne sont pas encore dans le site) constamment positive n'influe pas sur la durée de la simulation: peu importe qu'un seul ou que cent modules attendent à l'entrée, la machine n'ira pas plus vite pour autant. Nous allons vérifier ces hypothèses.

LES RESULTATS.

Nous avons simulé de nombreuses configurations et jeux d'essai, en étudiant les implications d'une variation de la valeur de l'intervalle d'arrivée IA. Tous les résultats obtenus ont montré de grandes similarités. Pour éviter les redites, nous n'en présenterons qu'une seule série, pour illustrer les conclusions générales qui en ont été tirées.

Les courbes suivantes résument la série présentée. Elles représentent la durée totale de simulation en fonction de l'intervalle d'arrivée, à configuration constante, à raison d'une courbe par stratégie. En fait l'égalité des résultats pour des stratégies différentes réduit à trois le

nombre de courbes visibles. Les trois courbes sont notées GR A, GR B et GR C (Il y a là une anticipation des résultats qui seront confirmés lors de l'expérimentation suivante).

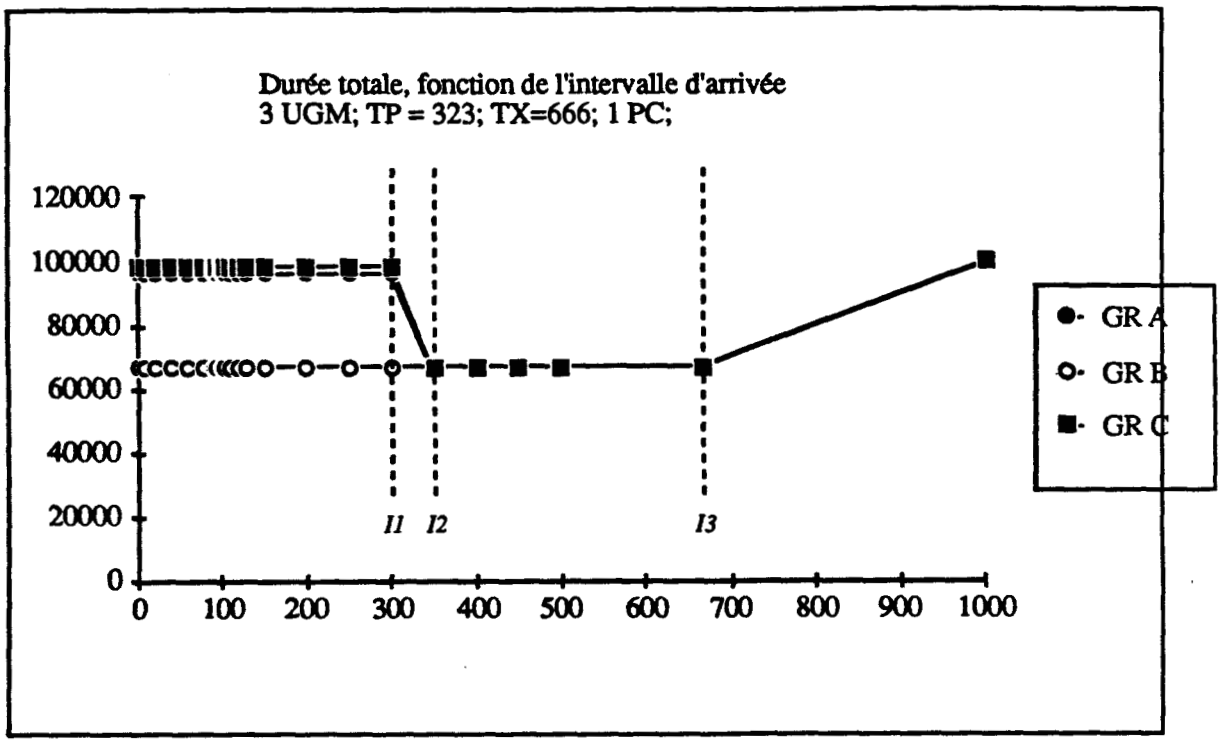


Fig. III-11. Durée totale en fonction de l'intervalle d'arrivée.

INTERPRETATION QUALITATIVE.

Les tests effectués (voir Fig. III-11) montrent que l'on peut définir quatre zones d'effet de ce paramètre Ces zones sont délimitées par trois valeurs I_1, I_2, et I_3.:

- En deça de I_1, la durée de la simulation est constante. Il y a toujours au moins un module en attente de préparation: le nombre de modules qui attendent à l'entrée du site n'a pas d'importance pourvu qu'il soit toujours strictement positif. Il y a saturation du site (dans le sens où des modules surnuméraires ne pourraient être traités qu'au dépend de la durée de la simulation).

- Entre I_1 et I_2, La loi est décroissante (strictement décroissante pour les groupes A et C, les stratégies qui privilégient la préparation): dans ce cas, l'absence de module en entrée peut provoquer un changement de

stratégie effective. C'est ainsi que, pour certaines valeurs de l'intervalle d'arrivée, une stratégie PCS peut prendre le comportement de la stratégie CPS par suite d'un manque de module à préparer (les choix des deux stratégies seront alors identiques car forcés). Paradoxalement, l'augmentation de l'intervalle d'arrivée permet alors un accroissement des performances.
La continuité de la loi (le fait qu'il n'y a pas de chute brutale de la durée de simulation entre les deux zones) résulte d'une action progressive du paramètre:
Au cours d'une simulation, on distingue une date critique t_x à partir de laquelle la file FAP est vide. C'est à partir de t_x que la stratégie peut se transformer. Pour IA=I_1, t_x=DT (fin de la simulation). Pour IA=I_3, t_x=0 (début de la simulation). Ainsi t_x va progressivement évoluer de la fin au début de la simulation à mesure que IA augmente, rendant de plus en plus prédominant, donc visible, le changement de stratégie.

- Entre I_2 et I_3, la durée de simulation reste à nouveau constante mais à un niveau minimal. Dans cette zone, les conditions sont telles que les choix de PN sont presque toujours obligés. Dans un tel contexte, les stratégies ne peuvent pas influer significativement sur les performances du site.

- Au delà d'une troisième valeur I_3, la durée de la simulation varie selon une loi linéaire croissante. Dans ce cas, la durée de la simulation est déduite de la date d'arrivée du dernier module. Ceci est caractéristique d'un sous-emploi de la machine.

Nous remarquons que, pour certaines stratégies, les durées de simulation (constantes) lues dans les première et troisième zone sont égales. On peut alors confondre les trois premières zones et décréter que l'on observe plus que deux comportements.

INTERPRETATION QUANTITATIVE.

Quand les caractéristiques des modules sont fixées (tests sans dispersions), le calcul des valeurs est simple :

- I_1 a représente l'intervalle d'arrivée à partir duquel la longueur de la file d'attente de préparation peut s'annuler. Il s'agit ici du temps de préparation. $\boxed{I_1 = TP.}$

- I_2 est la valeur minimale pour laquelle la file d'attente de préparation est toujours vide. Il s'agit de la valeur $\boxed{I_2 = TP+TC+TS}$.

- I_3 est la valeur d'intervalle d'arrivée pour laquelle les modules n'arrivent pas plus vite qu'ils ne sont traités. C'est donc le temps effectif de traitement d'un module soit $\boxed{I_3 = SUP(TP+TC+TN,TX)}$.

CONCLUSION.

Pour la suite de l'étude, nous considérerons un intervalle constant d'arrivée de 1. Au pire, nous obtiendrons des résultats que l'on pourrait améliorer, mais au prix d'une perversion de la stratégie choisie. Une autre manière de dire les choses est: "Quand les meilleurs choix ont été faits, on ne peut que diminuer l'efficacité quand on augmente l'intervalle d'arrivée".

INFLUENCE DE LA STRATEGIE.

Une caractéristique qui apparait évidente après l'examen des premiers résultats est la similitude des performances obtenues selon quelques stratégies. On peut ainsi classer celles-ci en trois groupes à l'intérieur desquels les stratégies ont des performances identiques pour la plupart des tests effectués. L'étude à postériori montre que les écarts ne se révèlent que dans des cas qui sortent de notre étude.

Le groupe B regroupe les stratégies qui offrent les meilleurs résultats qui peuvent même atteindre le niveau du bi-processeur idéal (celui qui exécuterait le jeu d'essai en la moitié du temps demandé par un mono-processeur). On y trouve la stratégie INT ainsi que les stratégies CSP et SCP qui défavorisent systématiquement la préparation par rapport aux autres opérations.

Les deux autres groupes ont des résultats différents bien que très proches. Le groupe A comprend les stratégies SPC, PSC et PCS. Il offre les pires résultats, comparables en ordre de grandeur à ceux d'un

monoprocesseur. Le groupe C qui comprend la seule stratégie CPS a un niveau intermédiaire entre les deux groupes précédents.

On peut retirer de cette observation les constatations suivantes:

- Il faut à tout prix pénaliser la préparation par rapport aux autres opérations. Ceci pour deux raisons:

  → Les opérations de chargement et à moindre titre de sauvegarde favorisent une utilisation maximale des UGM donc, indirectement, du deuxième processeur tandis que la préparation occupe le processeur N alors même que PC peut être oisif.

  → L'opération de préparation est la plus longue de toute. Dans les stratégies sans préemption, une préparation, quand elle est engagée monopolise complètement PN pour une durée assez longue. Ce dernier ne peut alors plus réagir aussi vite à une évolution de l'état des UGM, l'ensemble y perdant alors évidemment en souplesse.

- Lors des tests sans dispersion, la stratégie INT n'est pas plus rapide que les meilleures stratégies à priorité fixe. On peut expliquer cette remarque : la coïncidence des événements décrite au début de cette section joue un rôle unificateur entre les résultats des stratégies. En fait, nous verrons que les expérimentations ultérieures infirmeront cette remarque.

INFLUENCE DU RATIO D'OVERHEAD.

A ce stade, on peut se poser la question de savoir quelle est l'influence de ratio d'overhead sur les performances de l'architecture. Nous avons donc analysé les courbes obtenues – pour une configuration matérielle constante – de la durée totale DT en fonction du ratio d'overhead, pour toutes les stratégies. Là encore, les sept stratégies testées se sont réparties selon les trois groupes décrits plus haut.

RESULTATS.

Nous prendrons comme exemple l'évolution de DT, en fonction du ratio d'overhead soit avec 3, soit avec 1 seule UGM.

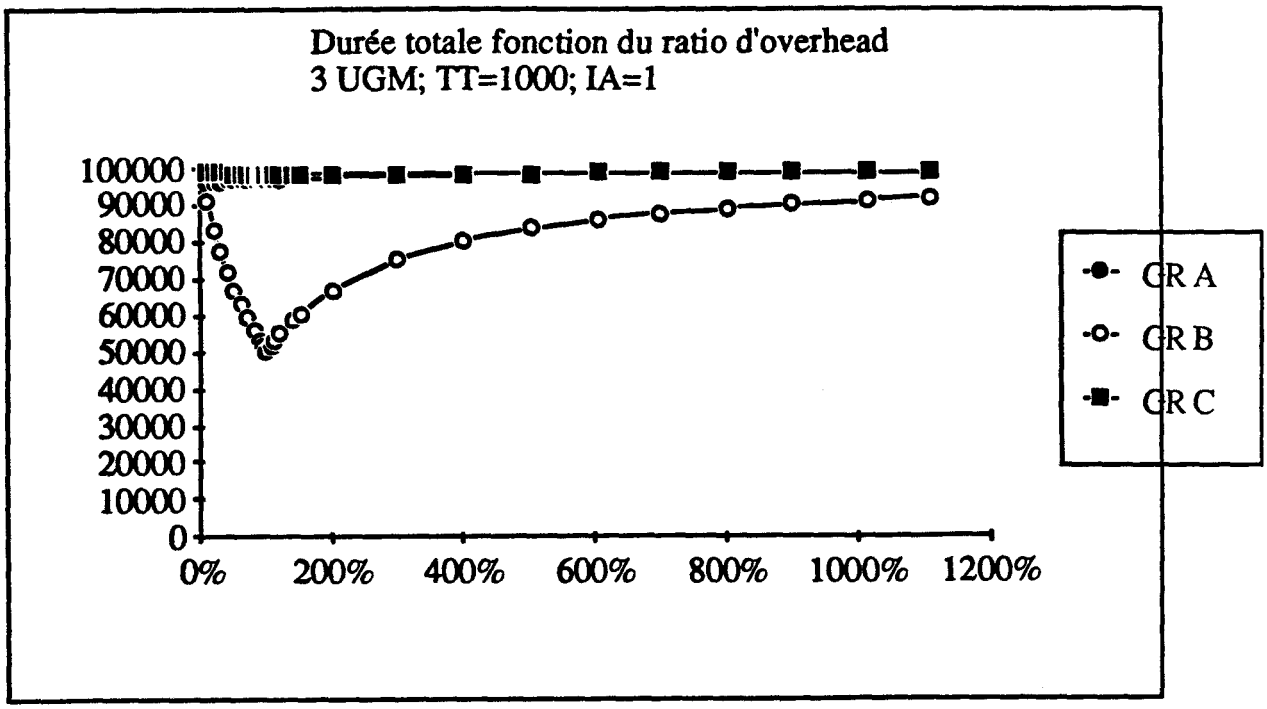


Fig. III-12. Durée totale, en fonction du ratio d'overhead (3 UGM).
TP et TX sont calculés à partir de TT et RH par les formules:

$$TX = \frac{TT}{RH+1} \quad \text{et} \quad TP = TT - TX - TC - TS.$$

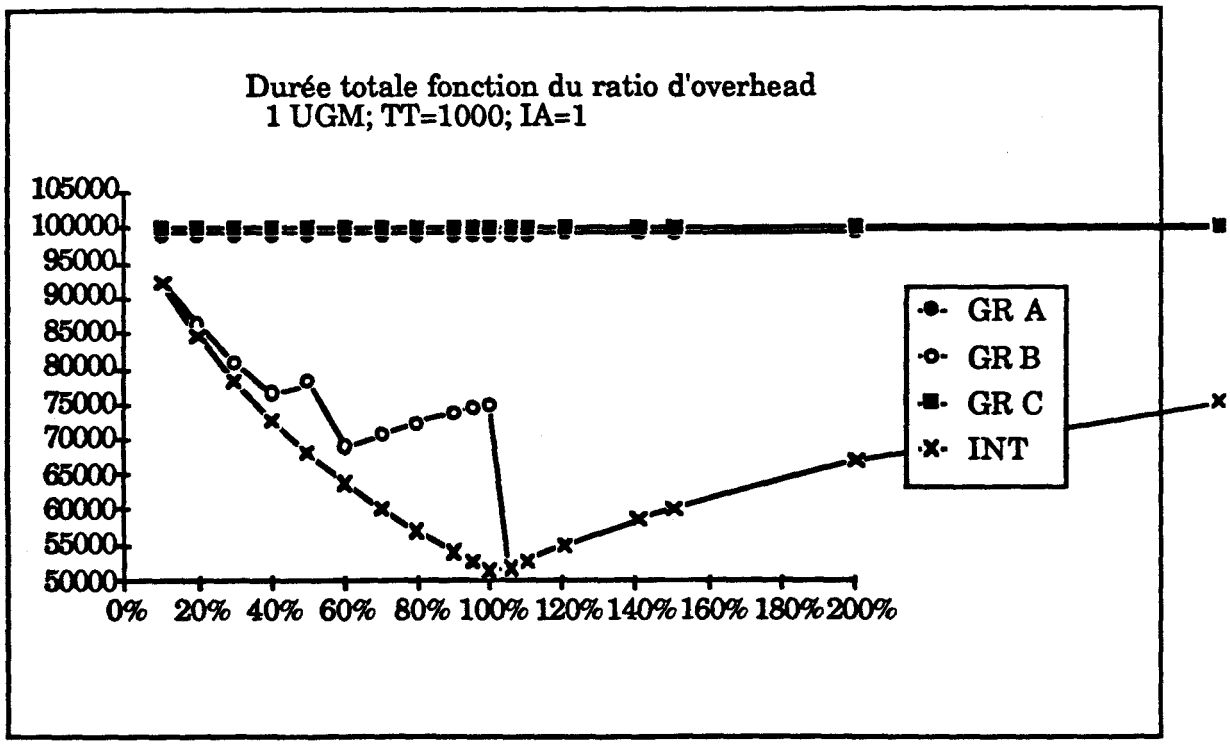


Fig. III-13. Durée totale, en fonction du ratio d'overhead (1 UGM).

INTERPRETATION.

Le ratio d'overhead influe différemment selon la stratégie testée.

- Pour le groupe A, ce ratio n'a pratiquement aucune importance, les temps réalisés restent toujours constants sauf quand la configuration matérielle ne comporte qu'une seule UGM. Dans ce cas, la variation est positive, c'est-à-dire que plus ce ratio est élevé, plus la durée de simulation est grande et par conséquent, moins bonnes sont les performances.

- En ce qui concerne le groupe B (qui offre les meilleurs résultats), la courbe de durée de simulation en fonction du ration est d'abord décroissante, jusqu'à un minimum obtenu quand il y a égalité parfaite entre les charges de PN et de PC (le ratio d'overhead est alors de 100%). La courbe reprend une allure croissante sans dépasser le temps de calcul d'un monoprocesseur. Notons que le minimum atteind la durée du bi-processeur idéal (à un temps constant près: le temps de montée en charge du site).
  On peut faire globalement les mêmes remarques lorsqu'une seule UGM est installée. La différence entre le groupe B et la stratégie INT (les dents de scie) est due au phénomène de synchronisme signalé au début de cette section.

- Le groupe C a, tout comme le groupe A, des durées de simulation quasiment constantes mais de valeur légèrement plus faible.

CONCLUSION.

S'il était évident que les cas extrèmes (rh = 0% et rh = ∞), les performances seraient celles d'un monoprocesseur, on pouvait craindre, pour le meilleur cas de figure, les ralentissements dus aux croisements des modules avant et après l'exécution. Il n'en est rien. Cet optimisme est malgré tout hypothéqué par la remarque concernant l'absence de dispersion des caractéristiques des modules.

L'idée intuitive selon laquelle l'architecture est plus efficace quand le ratio est de 100%, c'est-à-dire quand les charges sont équilibrées sur les deux processeurs se vérifie au moins pour les meilleures stratégies.

INFLUENCE DU NOMBRE D'UGM.

Quand ont été établis (du moins en première approche) les avantages apportés par l'architecture, il reste à évaluer la configuration matérielle optimale permettant de réaliser ce gain. Il est intéressant d'évaluer l' influence du nombre d'UGM sur les performances. Il est flagrant qu'une configuration comportant une seule UGM est complètement inefficace. Pour les autres cas, la corrélation n'est pas aussi évidente.

RESULTATS.

Nous présentons les mesures obtenues pour un ratio d'overhead RH de 30 %. Les expériences menées avec d'autres valeurs n'ont pas donné des résultats différents sur la forme.

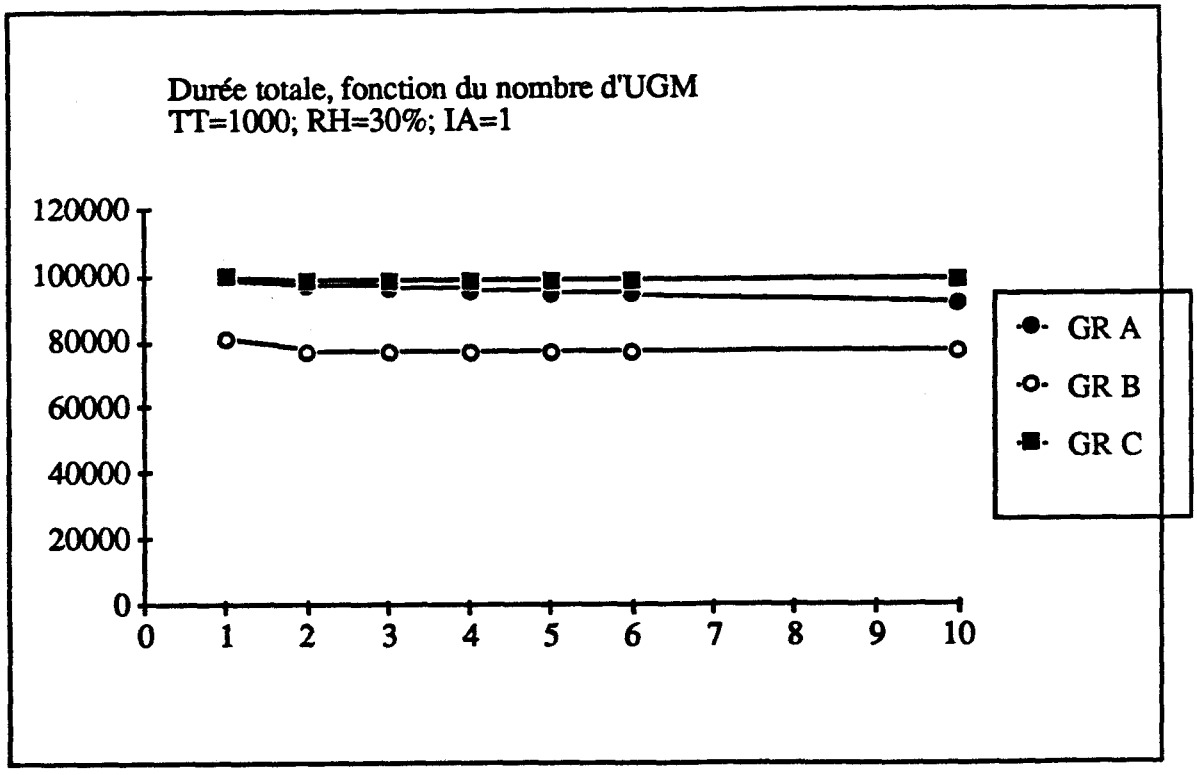


Fig. III-14. Durée totale, en fonction du nombre d'UGM.

INTERPRETATION.

Les simulations effectuées ont révélé les propriétés suivantes :

- Pour les groupes B et C, le meilleur gain est obtenu quand on rajoute la deuxième UGM. Les UGM supplémentaires offrent peu ou pas d'avantage.

- Les stratégies du groupe A bénéficient progressivement d'une augmentation du nombre d'UGM (sans toutefois surpasser celles du groupe B). Les UGM supplémentaires se révèlent bénéfiques jusqu'à 100 (valeur directement liée au nombre de modules du jeu d'essai). Cette dernière remarque n'est toutefois d'aucun intérêt pratique. L'expérimentation est en effet de moins en moins rigoureuse à mesure que le nombre d'UGM et le nombre de modules du jeu d'essai deviennent comparable: on n'observe plus alors que des régimes transitoires.

- Dans des conditions difficiles (1 UGM, RH < 100 %), la stratégie INT prend l'avantage sur les autres stratégies du le groupe B. La différence n'apparait pas dans cette courbe (car trop peu marquée) mais avait déjà été mise en évidence lors de l'expérimentation précédente (cf figure III-13).

CONCLUSION.

Pour des conditions réalistes, c'est à dire pour un faible nombre d'UGM, les stratégies du groupe B sont de loin les plus efficaces. Ce choix fait, il est inutile de considérer un nombre d'UGM plus grand que 2.

III.C.3. LES TESTS AVEC DISPERSION

Les tests sans dispersion permettent un dégrossissement de l'étude du site. Mais ces tests sont loin d'être parfaits; en effet, l'arrivée d'un flux constant de modules tous identiques conduit inévitablement à l'établissement d'un régime complètement stationnaire. C'est ainsi que si le nombre de modules en attente de préparation augmente sur les premiers instants, on peut être sûr que ce nombre sera croissant (et au même taux durant l'ensemble de la durée de la simulation). Ce genre de régime n'est pas réaliste et empêche d'évaluer l'utilité de certains éléments. C'est ainsi que nous n'avons pas pu observer l'influence de la configuration mémoire (nombre de bloc, taille des blocs) car une grande taille mémoire ne fait que faciliter le passage d'état transitoire (pointe de charge par exemple). L'introduction de dispersion dans les

caractéristiques des modules d'un jeu d'essai nous permettra d'avancer dans cette étude.

Pour approfondir l'étude du site 0, nous avons eu besoin de simuler des groupes de modules de caractéristiques différentes à l'entrée de la machine. Ce paragraphe a pour objet de décrire la manière dont ces groupes ont été créés ainsi que les méthodes que nous avons utilisées pour contrôler leur validité au sens statistique.

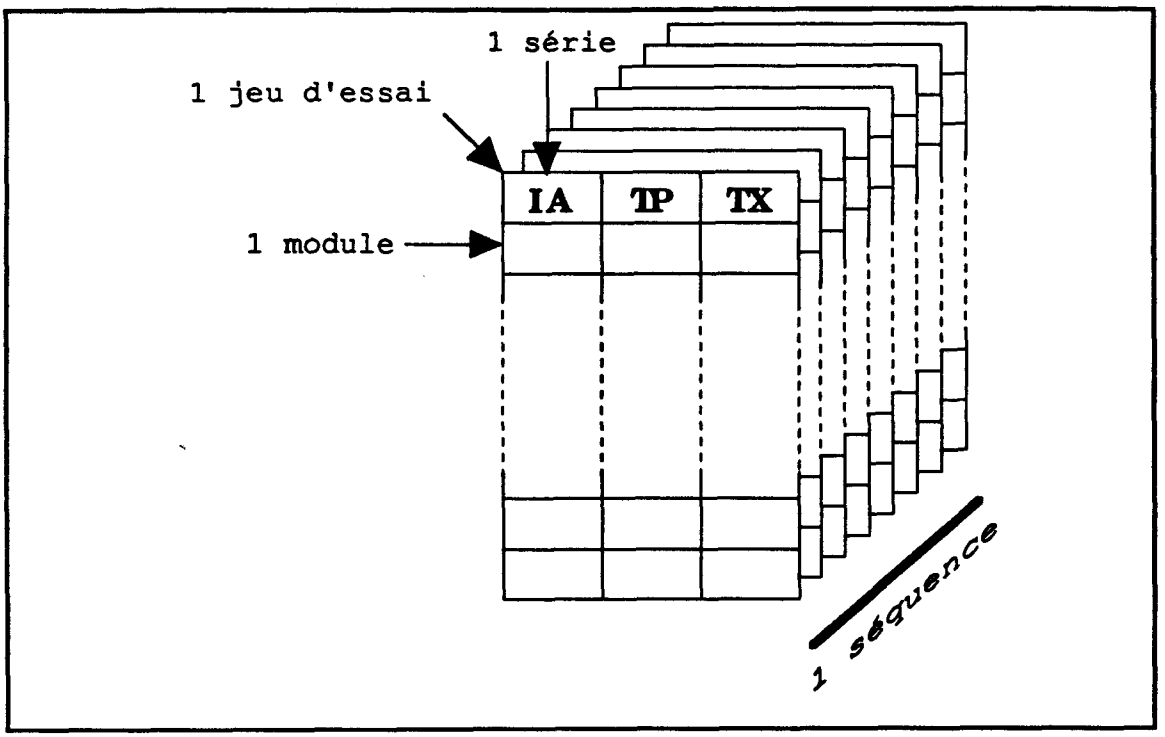
VOCABULAIRE.

MODULE: Un triplet qui regroupe les 3 caractéristiques d'un module, à savoir, IA: intervalle d'arrivée, TP: temps de préparation, TX: temps d'exécution.

JEU D'ESSAIS: Un ensemble de NM modules (généralement 100).

SERIE: L'ensemble de toutes les valeurs d'une caractéristique pour un jeu d'essai. Par exemple 100 intervalles d'arrivée. Un série est donc l'échantillon d'une variable aléatoire de cardinalité NM (nombre de modules du jeu d'essais). Ou encore un jeu d'essais peut être considéré comme le regroupement de trois séries.

SEQUENCE: Les tests sans dispersion sont commodes car ils permettent d'obtenir des résultats avec une seule simulation (soit encore un seul jeu d'essai composé de séries constantes). A contrario, les tests sans dispersion demandent de disposer d'un ensemble de jeux d'essais. Ce sont les valeurs moyennes de ces jeux d'essais qui sont caractéristiques de l'étude. Nous appelons séquence cet ensemble de jeux d'essais.



LA CREATION DES JEUX D'ESSAIS.

Chaque module est caractérisé par trois valeurs : IA, TP et TX. Il est précisé dans les hypothèses de la simulation que ces caractéristiques sont indépendantes et suivent une loi de probabilité exponentielle

$$f(x) = \frac{1}{A} e^{-\frac{x}{A}} \quad \text{si } x >= 0 \; ; \; f(x) = 0 \quad \text{si } x < 0$$

Or, la bibliothèque mathématique FORTRAN "NAG" existant sur MULTICS nous fournit un générateur de nombres pseudo-aléatoires (loi uniforme) de période suffisante (2^{57}) adaptable à une loi exponentielle par utilisation de la formule :

$x = -A \ln(c)$ où c est une variable aléatoire de loi uniforme sur $[0, 1[$ et x est la variable engendrée selon la loi exponentielle de paramètre A. ln est le logarithme néperien.

Ainsi, nous pouvions utiliser le même générateur pour les trois lois exponentielles bien qu'elles ne fussent pas de même moyenne. Tout ceci sous réserve que le nombre de module à engendrer est faible par rapport à

la période du générateur, ce qui est évident pour les jeux d'essais que nous avons utilisés (moins de 1000 modules). Nous obtenons ainsi trois séries de valeurs par jeu d'essai.

CONTROLE DES JEUX D'ESSAIS

Nous avons contrôlé la validité statistique des jeux d'essais utilisés par trois outils qui étaient tous disponibles sous NAG. Nous n'en donnerons pas une description théorique mais une idée des objectifs visés et les méthodes générales utilisées par ceux-ci.

CONTROLES SIMPLES

Il s'agit ici de calculer les évaluateurs courants (moyenne, écart-type) de chaque série pour les comparer avec les valeurs théoriques des lois de distribution. Ce contrôle, un peu brutal, ne donne pas de mesure de la validité d'un jeu d'essai. Il n'a, de toute manière, pas permis de détecter d'anomalies (qui auraient dues pour cela être flagrantes).

TEST DE KOLMOGOROV-SMIRNOV

Ce test est utilisé pour vérifier la distribution d'un échantillon selon une loi donnée par sa fonction de répartition F. Une formule est établie qui donne une "mesure" de l'écart entre la fonction de répartition théorique et la fonction de répartition déduite de l'échantillon. Il suffit alors de regarder la probabilité d'occurence de cette valeur pour obtenir une estimation de la vraisemblance de la loi F.

Nous avons utilisé, pour établir le tableau suivant, le seuil, habituel pour ces tests, de 5% (risque d'erreur de première espèce, c'est-à-dire de réfuter à tord une répartition théorique exacte). Ce tableau montre pour trois séquences de 90 jeux d'essais, le nombre de jeux d'essai refusés:

Jeu d'essai			IA	TP	TX
IA=1	TP=489	TX=500	0	3	2
IA=1	TP=289	TX=700	0	3	2
IA=500	TP=489	TX=700	1	3	1

Ces résultats sont tout a fait acceptables et ne mettent pas en évidence d'erreur de méthode. La seule réserve que l'on peut émettre est le trop bon

résultat obtenu par les intervalles d'arrivée en particulier quand IA=1 qui peut être expliqué par la discrétisation de la variable aléatoire.

TEST DE KENDALL

L'objet de ce test est de vérifier la non-corrélation entre les variables aléatoires. Un tel phénomène pourrait se produire par exemple si la période du générateur aléatoire et la taille de chaque jeu d'essai étaient identiques. Le test de Kendall évalue la concordance ou la non concordance des classements des triplets selon chaque série.

Là aussi les valeurs obtenues ne révèlent pas d'erreur systématique dans la procédure de génération des jeux d'essais.

VALIDATION DES TESTS SANS DISPERSION.

Nous avons introduit les trois séquences testées précédemment dans le simulateur avec trois configurations (3 UGM, pas de limitation mémoire et stratégies CPS, CPS et INT). Nous avons comparé ensuite μ (durée moyenne de DT issue de la simulation d'un jeu d'essais dans chaque séquence) avec M (durée de simulation d'un jeu d'essai de caractéristique constante)

Après avoir observé que μ est toujours inférieur à M, nous avons pu distinguer trois types de résultats selon l'écart $\boxed{\Delta M = \frac{(M - \mu)}{M}}$:

- La durée calculée sans dispersion est très proche de la valeur moyenne de la séquence ($\Delta M < 3\%$). Ceci se produit pour les stratégies assez peu efficaces (groupes A et C).

Nous avons fait figurer dans les schémas ci-dessous, les fréquences des simulations réparties en classes de même amplitudes (5000 unités de temps). Nous y avons superposé les valeurs de M et de μ.

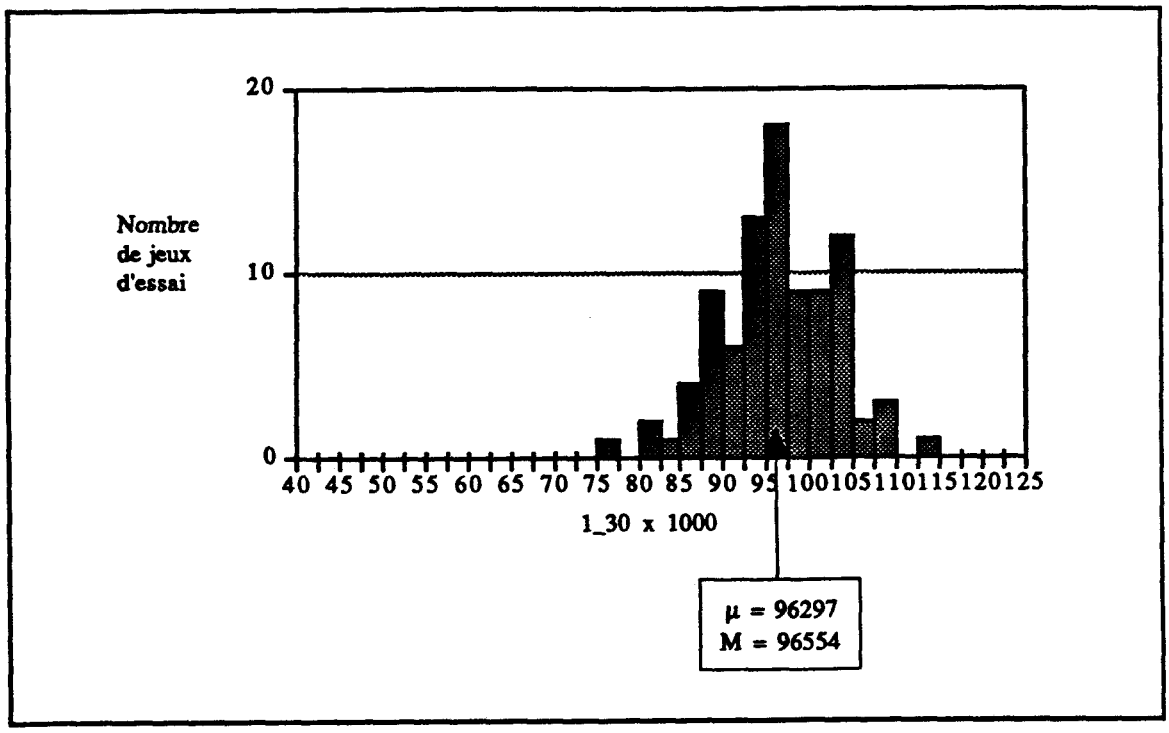


Fig. III-15. Tests avec dispersion (groupes A et C).

- La durée calculée est proche de la valeur moyenne de la séquence ($3 < \Delta M < 10\%$). Ce cas de figure est observé pour les meilleures stratégies (groupe B).

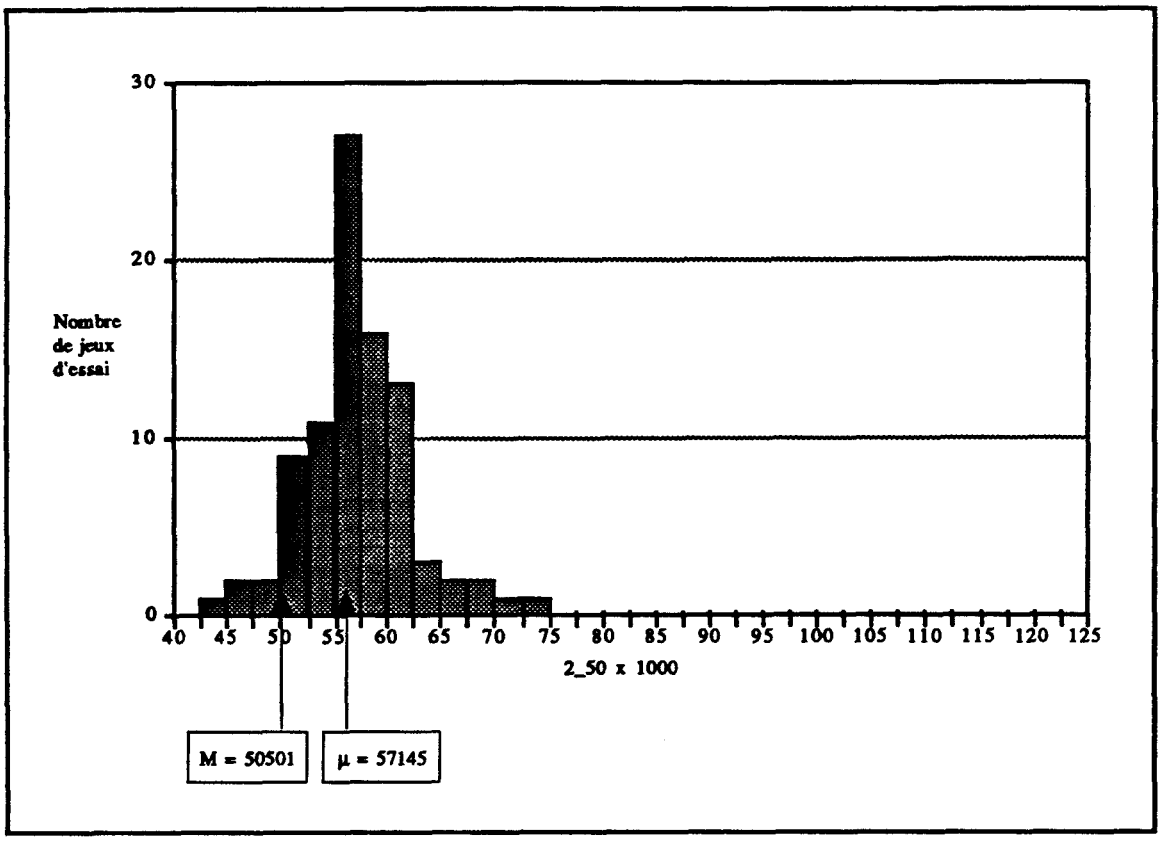


Fig. III-16. Tests avec dispersion (groupe B).

- La durée calculée est très éloignée de la durée moyenne (ΔM vaut environ 40%). Ce qui se produit dans un cas unique, pour la séquence IA = 500 et la pire des stratégies.

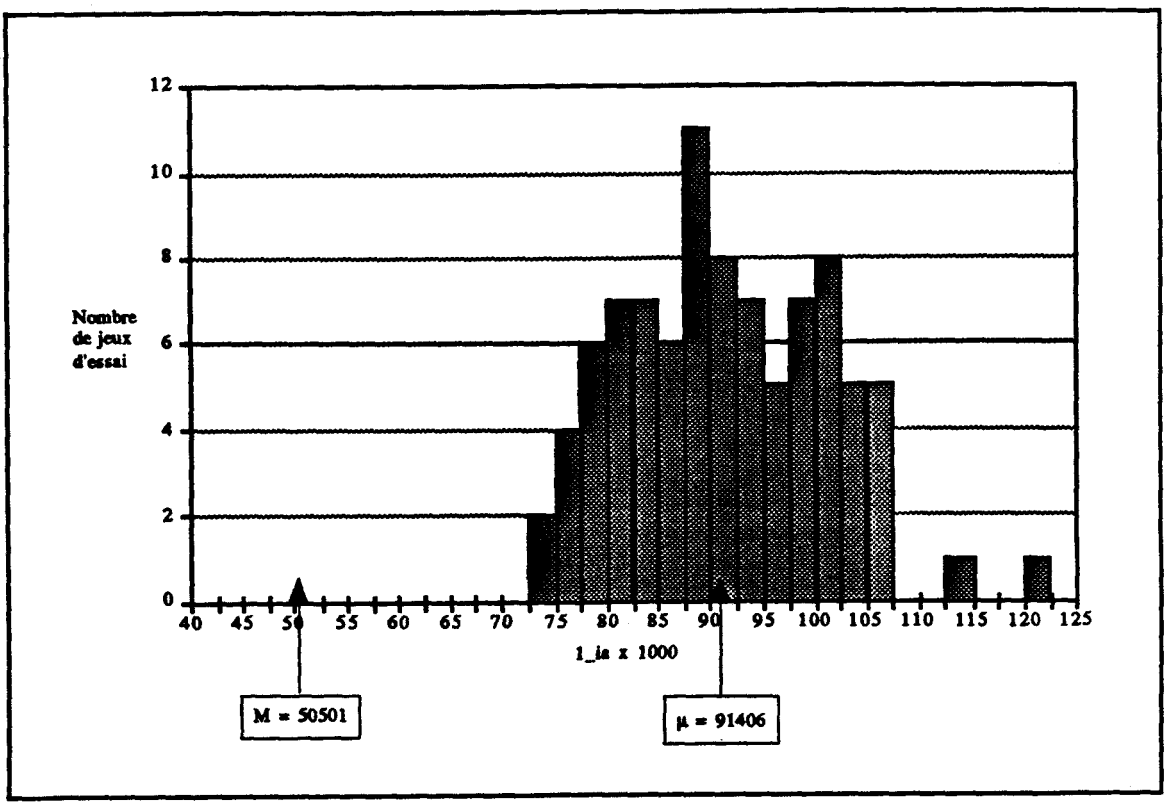


Fig. III-17. Tests avec dispersion (Cas extrême).

Les conclusions que nous avons tirées de ces expériences sont au nombre de deux:

- La remarque concernant la simultanéité des événements faite plus haut est tout à fait justifiée. Elle joue toujours dans un sens défavorable (i.e. la non simultanéité d'événements entraîne toujours des pertes et jamais des gains de temps). En ce sens, elle diminue l'écart de performance entre les différentes stratégies: une stratégie inefficace qui perd déjà du temps par ailleurs n'est pas trop gênée par les dispersions des caractéristiques alors qu'une stratégie parfaitement réglée y perdra forcément plus.

- L'intervalle d'arrivée provoque effectivement de grands écarts dans les durées de simulation, écarts qui ne peuvent pas être rattrapés par les mauvaises stratégies. Cela justifie à posteriori l'élimination de ce paramètre.

Globalement, nous retiendrons que les résultats obtenus sans dispersion sont optimistes et que de fait, les écarts observés entre les

différentes configurations seront plus petits dans les cas réels. Cependant, ceci ne remet pas en cause les remarques concernant les performances *relatives* (entre les stratégies par exemple). Par ailleurs, ces premières observations nous confortent dans la résolution de minimiser le facteur 'intervalle d'arrivée' lors des expériences ultérieures.

LES TEST AVEC SIMULATION DE LA MEMOIRE

Il n'est pas possible d'évaluer les conséquences d'un changement de la taille de la mémoire de module si on ne choisit pas un algorithme, sinon parfait, du moins acceptable de gestion de cette mémoire (nous parlons là de la gestion des blocs – attribution des blocs aux modules – qui est bien entendu entièrement distincte de la gestion mémoire interne à chaque bloc): Le premier algorithme testé était si mauvais que l'ajout de mémoire était purement et simplement inutile. Les blocages mutuels des modules lors de l'attribution de mémoire aux modules ralentissaient par trop la machine.

Nous allons détailler cette gestion afin de bien préciser les conditions expérimentales. Nous proposons un algorithme de gestion qui n'est peut-être pas parfait mais qui a l'avantage d'être simple et facilement programmable.

POLITIQUE DE GESTION DE LA MEMOIRE.

La méthode de gestion de la mémoire se décompose en deux problèmes:

- Assurer l'exclusion mutuelle d'accès à un bloc de la part des deux processeurs PN et PC. Nous avons déjà vu que ceci était réalisé par le matériel du coté PC et par le logiciel du coté PN: Il est possible pour le logiciel noyau de tenir à jour l'ensemble des blocs mis à disposition de PC et non encore récupérés.

- Définir la méthode d'attribution d'un bloc de mémoire à un module. Cette attribution a lieu juste avant la préparation. Le bloc reste attribué au module jusque la sauvegarde.
  Quand un module est de taille 1 (un bloc peut contenir un seul module), cette méthode n'a pas d'importance pour peu qu'elle respecte le

premier point (ne pas attribuer un bloc à un module proposé à PC et non récupéré). Par contre, si plusieurs modules peuvent se situer dans le même bloc, il faut être prudent dans le choix de l'attribution: il faut non seulement respecter le premier point, mais aussi éviter les blocages qui peuvent se produire (par exemple PN ne peut plus faire de préparation car le module prévu se trouve dans le même bloc qu'un module en cours d'exécution).

La méthode de gestion choisie est un mécanisme d'allocation tournante: Pour n blocs de mémoire, et quelque soit la taille de chaque bloc, le premier module arrivé est placé dans le bloc 1, le second dans le bloc 2, le $n^{ème}$ dans le bloc n, le $n+1^{ème}$ module est mis dans le bloc 1 etc... Quand un module ne peut être placé dans un bloc (car ce bloc n'est pas accessible par PN ou parce que ce bloc est déjà plein), on essaie le bloc suivant. Si aucun bloc ne peut être trouvé de cette manière, alors l'allocation n'est simplement pas possible.

L'objectif de cette méthode est d'équilibrer le nombre de modules sur chaque bloc. En effet, plus la répartition est équitable, plus la probabilité de blocage est faible. PN y gagne en nombre d'activités possibles, donc en souplesse. Nous verrons que les performances calculées de cette méthode de gestion (placée dans les meilleures configurations) sont proches des performances calculées sans tenir compte de la mémoire. Nous en déduisons que cette politique de gestion mémoire est acceptable.

ETUDE DE L'INFLUENCE DE LA MEMOIRE

Nous avons déjà vu que la configuration mémoire est caractérisée par deux paramètres (le nombre de blocs ainsi que la taille de chaque bloc). Nous verrons que leurs influences respectives sont très liées. De ce fait, nous avons dû étudier ces deux paramètres simultanément. La section présente fournira au lecteur, un aperçu des résultats, suivi d'une analyse sommaire destinée à structurer les chiffres recueuillis. Nous essayerons ensuite d'interpréter cette analyse.

LES RESULTATS.

CHAMP D'ESSAI

Nous présenterons ici simplement les extraits de simulation les plus significatifs en guise d'illustration aux quelques observations que nous avons déduites de nos essais. Ces essais étant, par soucis d'économie, restreint par deux régles:

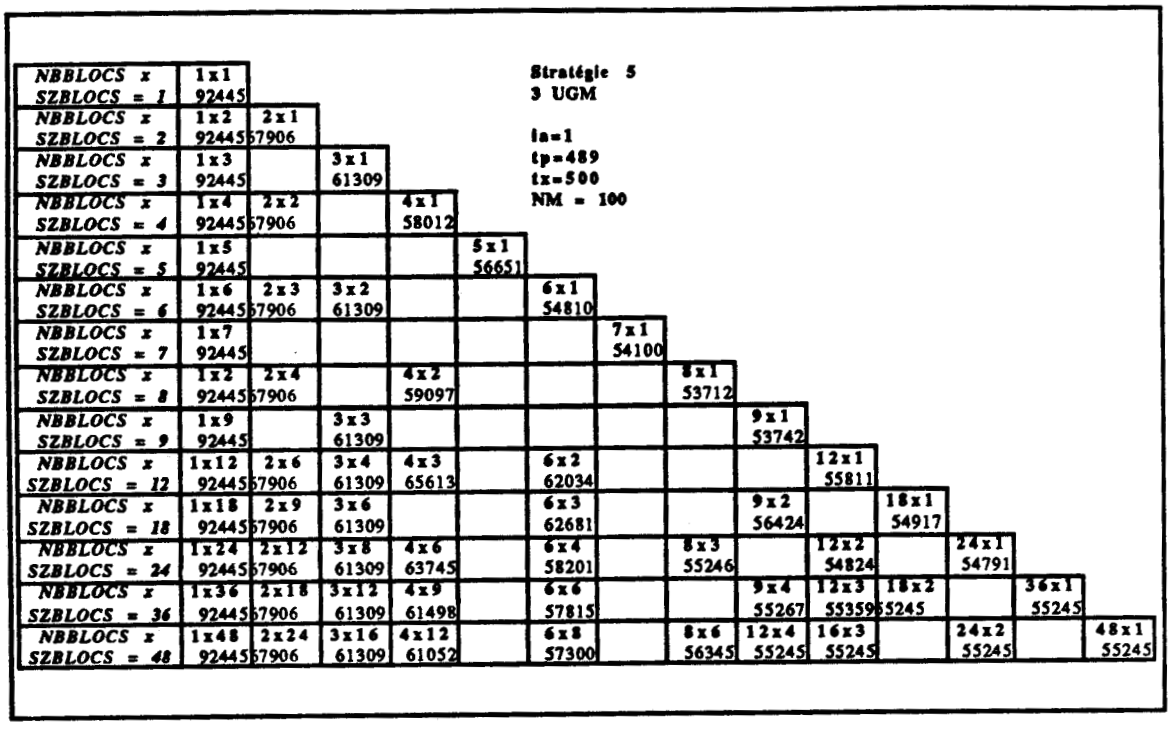
- Nous ne considérerons que des petites tailles de mémoire (moins de 48 blocs). Tout d'abord à cause du coût de la mémoire dans la conception actuelle du site 0. Ensuite parce qu'une extrapolation sur un plus grand nombre de bloc nous a semblé viable.
- Nous resterons dans les conditions optimales de la section précédente: stratégie CSP, SCP ou INT, ratio d'overhead de 50%.

PRESENTATION

Les résultats sont présentés sous la forme d'un tableau: sur une ligne, nous avons placé l'ensemble des simulations qui demandent la même taille mémoire globale. Nous avons mis dans la même colonne les résultats qui ont étés établis à partir du même nombre de bloc. Ainsi, en progressant le long d'une ligne nous pouvons voir l'influence d'un découpage plus important de la mémoire alors qu'un parcours vertical nous montrera l'évolution qu'entraine une augmentation de la taille mémoire à nombre de blocs constant.

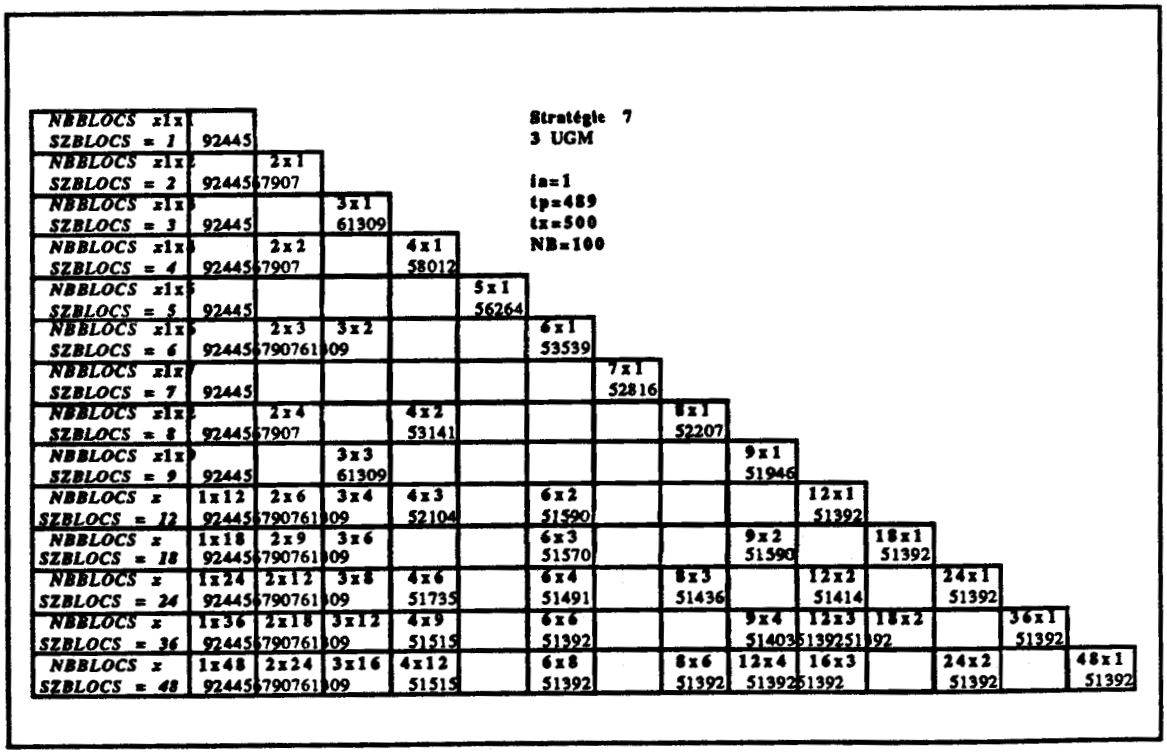
RESULTATS

Les valeurs brutes recueuilles en sortie de simulation montrent globalement le même profil. C'est pourquoi nous ne présenterons que deux tableaux. Un échantillon plus important des essais effectués est donné en annexe.



NBBLOCS x / SZBLOCS = 1	1x1
	92445
NBBLOCS x / SZBLOCS = 2	1x2 / 2x1
	92445 / 87906
NBBLOCS x / SZBLOCS = 3	1x3 / 3x1
	92445 / 61309
NBBLOCS x / SZBLOCS = 4	1x4 / 2x2 / 4x1
	92445 / 87906 / 58012
NBBLOCS x / SZBLOCS = 5	1x5 / 5x1
	92445 / 56651
NBBLOCS x / SZBLOCS = 6	1x6 / 2x3 / 3x2 / 6x1
	92445 / 87906 / 61309 / 54810
NBBLOCS x / SZBLOCS = 7	1x7 / 7x1
	92445 / 54100
NBBLOCS x / SZBLOCS = 8	1x2 / 2x4 / 4x2 / 8x1
	92445 / 87906 / 59097 / 53712
NBBLOCS x / SZBLOCS = 9	1x9 / 3x3 / 9x1
	92445 / 61309 / 53742
NBBLOCS x / SZBLOCS = 12	1x12 / 2x6 / 3x4 / 4x3 / 6x2 / 12x1
	92445 / 87906 / 61309 / 65613 / 62034 / 55811
NBBLOCS x / SZBLOCS = 18	1x18 / 2x9 / 3x6 / 6x3 / 9x2 / 18x1
	92445 / 87906 / 61309 / 62681 / 56424 / 54917
NBBLOCS x / SZBLOCS = 24	1x24 / 2x12 / 3x8 / 4x6 / 6x4 / 8x3 / 12x2 / 24x1
	92445 / 87906 / 61309 / 63745 / 58201 / 55246 / 54824 / 54791
NBBLOCS x / SZBLOCS = 36	1x36 / 2x18 / 3x12 / 4x9 / 6x6 / 9x4 / 12x3 / 18x2 / 36x1
	92445 / 87906 / 61309 / 61498 / 57815 / 55267 / 55359 / 55245 / 55245
NBBLOCS x / SZBLOCS = 48	1x48 / 2x24 / 3x16 / 4x12 / 6x8 / 8x6 / 12x4 / 16x3 / 24x2 / 48x1
	92445 / 87906 / 61309 / 61052 / 57300 / 56345 / 55245 / 55245 / 55245 / 55245

Fig. III-18. Résultat avec mémoire (Stratégie SCP).



NBBLOCS x1x / SZBLOCS = 1	92445
NBBLOCS x1x / SZBLOCS = 2	2x1
	92445 / 87907
NBBLOCS x1x / SZBLOCS = 3	3x1
	92445 / 61309
NBBLOCS x1x / SZBLOCS = 4	2x2 / 4x1
	92445 / 87907 / 58012
NBBLOCS x1x / SZBLOCS = 5	5x1
	92445 / 56264
NBBLOCS x1x / SZBLOCS = 6	2x3 / 3x2 / 6x1
	92445 / 790761 / 309 / 53539
NBBLOCS x1x / SZBLOCS = 7	7x1
	92445 / 52816
NBBLOCS x1x / SZBLOCS = 8	2x4 / 4x2 / 8x1
	92445 / 87907 / 53141 / 52207
NBBLOCS x1x / SZBLOCS = 9	3x3 / 9x1
	92445 / 61309 / 51946
NBBLOCS x / SZBLOCS = 12	1x12 / 2x6 / 3x4 / 4x3 / 6x2 / 12x1
	92445 / 790761 / 309 / 52104 / 51590 / 51392
NBBLOCS x / SZBLOCS = 18	1x18 / 2x9 / 3x6 / 6x3 / 9x2 / 18x1
	92445 / 790761 / 309 / 51570 / 51590 / 51392
NBBLOCS x / SZBLOCS = 24	1x24 / 2x12 / 3x8 / 4x6 / 6x4 / 8x3 / 12x2 / 24x1
	92445 / 790761 / 309 / 51735 / 51491 / 51436 / 51414 / 51392
NBBLOCS x / SZBLOCS = 36	1x36 / 2x18 / 3x12 / 4x9 / 6x6 / 9x4 / 12x3 / 18x2 / 36x1
	92445 / 790761 / 309 / 51515 / 51392 / 51403 / 51392 / 51392 / 51392
NBBLOCS x / SZBLOCS = 48	1x48 / 2x24 / 3x16 / 4x12 / 6x8 / 8x6 / 12x4 / 16x3 / 24x2 / 48x1
	92445 / 790761 / 309 / 51515 / 51392 / 51392 / 51392 / 51392 / 51392 / 51392

Fig. III-19. Résultats avec mémoire (Stratégie INT).

ANALYSE SOMMAIRE.

Une analyse globale des valeurs contenues dans les tableaux permet de dresser une 'cartographie' grossière des performances selon la taille et le nombre de blocs. Celle-ci est représentée dans la figure III-21: une zone plus sombre correspond à de faibles performances, inversement, une zone claire indique une configuration efficace.

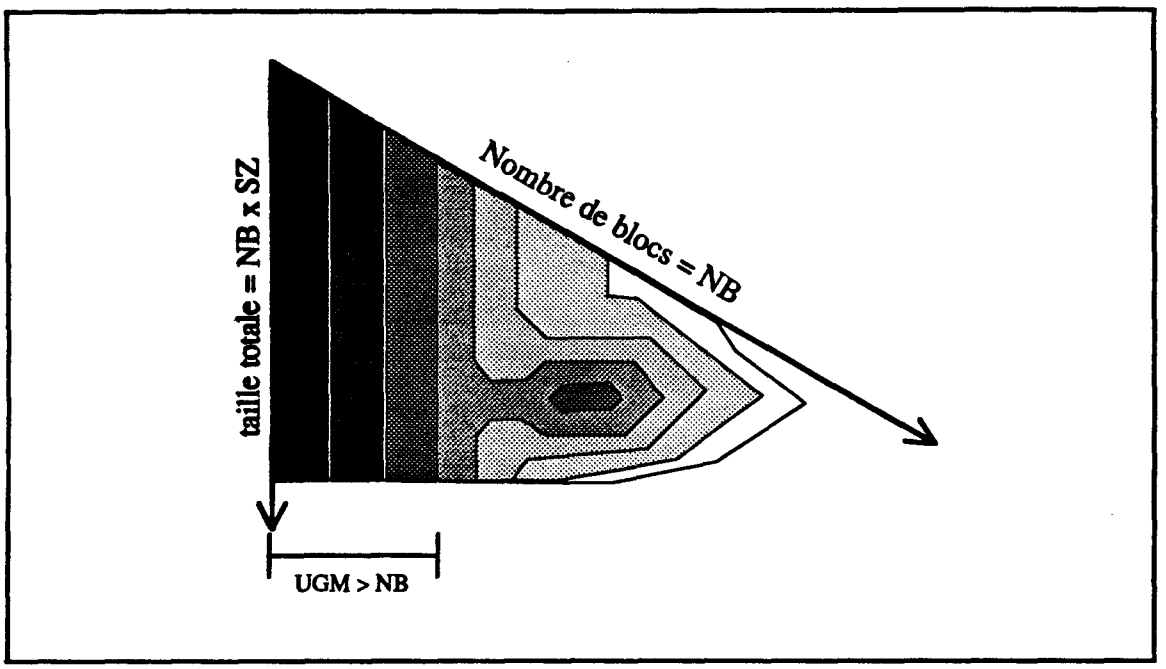


Fig. III-20. Représentation graphique.

On peut, au premier coup d'œil, distinguer deux zones:

- Une première zone le long de l'axe gauche regroupe les configurations pour lesquelles les durées de simulation ne dépendent pas de la taille des blocs. Cette caractèristique est révèlée par des bandes verticales de valeurs identiques. En fait fait cette zone regroupe tous les points pour lesquels le nombre de blocs (NBBLOCS) est inférieur ou égal au nombre d'UGM (UGM).

- La deuxième zone est moins nettement marquée. Elle présente une décroissance globale vers la droite (i.e. vers un plus grand découpage à taille mémoire égale). Par contre, selon l'autre axe (taille des blocs), elle est d'abord croissante jusqu'à une 'ligne de crête' approximativement

horizontale puis décroit jusqu'au minimum. Nous appellerons TMC (taille mémoire critique), la valeur de l'ordonnée de cette ligne de crête. Sur les expérimentations que nous avons effectuées (différentes taille de jeux d'essais, différents nombres d'UGM, plusieurs stratégies), TMC semble bizarrement constant (environ 18). Cette bosse est de moins en moins marquée à mesure que l'on décale vers la droite (plus grand nombre de blocs) et finit par disparaitre complètement. La description de cette zone peut être affinée par trois remarques:

→ Lorsque l'on simule la stratégie INT, la ligne de crête est nettement moins marquée. C'est ainsi que la durée de simulation ne décroit plus qu'exceptionnellement lorsqu'on augmente la taille des blocs. La décroissance vers la droite reste toujours valable.

→ La description géographique des tableaux pourrrait laisser penser que les propriétés mises en évidences (ligne de crête horizontale, valeur de taille mémoire critique) sont précises et bien établies. Il n'en est rien; les restrictions que nous nous sommes imposées, de même que le caractère discret des paramètres a restreint le nombre de simulations effectuées pour un tableau (70 simulations dont la moitié seulement pour la deuxième zone). La description de cette seconde zone ne peut alors être que qualitative.
En particulier, la valeur de TMC est donnée sous toute réserve: elle a été établie sur la base de 6 simulations par tableau (les 6 maximum relatifs selon les verticales).

→ Les valeurs du tableau, quand elles sont proches du minimum calculé (c'est à dire à droite et/ou en dessous d'une courbe qui part tangente au coté oblique du tableau et qui termine horizontalement vers la gauche) sont assez chaotiques. Nous avons consideré que toutes ces valeurs sont égales (soit une tolérance de ±5%).

INTERPRETATION ET JUSTIFICATIONS

UGM ≥ NBBLOCS

Les performances sont , dans cette zone, effectivement limitées par l'impossibilité pour PN d'anticiper la préparation de modules à cause de manque de blocs mémoire. Le comportement de la durée de simulation peut alors être caractérisé par les propositions suivantes:

→ Indépendance par rapport au nombre d'UGM et par rapport à la taille des blocs.
Il est inutile d'ajouter des UGM ou d'augmenter la taille des blocs. En effet, quelque soit le nombre d'UGM, PN ne peut anticiper assez de préparations de modules. Ceci est dû en particulier à la gestion de mémoire (du fait qu'un bloc contenant un module chargé est considéré comme inaccessible par PN)

→ Indépendance par rapport à la stratégie.
Les stratégies testées défavorisent toutes systématiquement la préparation vis à vis des autres préparations. Or le manque de performance est justement dû à un retard dans la préparation. Effectivement, dans ce cas précis, les stratégies que nous avons écartées se révèlent meilleures.

UGM ≤ NB

Dans ce cas, les limitations de performances sont dues uniquement au nombre d'UGM. En effet, il est possible d'associer indéfectiblement une UGM à un bloc mémoire. Les blocs supplémentaires, ne sont pas utilisés. D'où l'indépendances des performances par rapport au nombre de blocs.

CONCLUSION.

Cette étude, malgré l'imprécision des résultats, a permis de mettre en évidence deux choses:

Premièrement, la taille mémoire installée ne doit pas forcément être importante pour obtenir une bonne efficacité: La plus grosse partie des performances est déjà atteinte avec six blocs pouvant stocker chacun un module.

Deuxièmement, la stratégie INT se distingue des autres stratégies par une meilleure gestion de la mémoire de modules, qui permet

d'augmenter la taille de cette mémoire sans crainte de voir s'affaiblir les performances.

III.D. CONCLUSION

La simulation a permis d'établir deux sortes de résultats:

- la validation de certains choix à effectuer lors de l'écriture des logiciels:
  - Lors de l'élaboration de la stratégie de PN, n'effectuer des préparations qu'en dernier recours, i.e. quand il n'y a rien d'autre à faire.
  - La gestion de la mémoire de bloc (en termes d'allocation d'un bloc à un module lors de sa préparation) peut être faite grâce à une politique d'allocation tournante.
- Quelles sont les caractéristiques matérielles et logicielles qui permettent de tirer le mieux parti de l'organisation originale du site 0.
  - Le rendement du matériel est d'autant meilleur que le pourcentage d'overhead est proche de 100% (équilibre des charges sur les deux processeurs).
  - Si les conditions précédentes sont respectées, deux UGM sont suffisantes
  - L'amélioration obtenue au moyen de l'augmentation du nombre de blocs est minime au delà du sixième bloc mémoire.

La plupart de toutes ces conclusions rejoignent l'intuition. Quand toutes ces conditions sont réunies, on peut alors mettre en évidence un fonctionnement en 'bascules inverses' selon les deux processeurs: les deux processeurs utilisent alternativement chacun une UGM (PC pour une exécution, PN pour une sauvegarde, puis un chargement). Les charges des deux processeurs étant égales, il terminent simultanément et s'échangent leur UGM. On se retrouve alors dansl'état initial. Ce mode de fonctionnement utilise parfaitement la symétrie de l'organisation matérielle.

IV. IMPLANTATION ET MESURES SUR LE SITE ZERO.

Quand ont été posés les principes généraux de l'architecture OMPHALE d'une part, du site zéro d'autre-part, il est naturel de s'interroger sur l'adéquation de l'un à l'autre. La réponse la moins contestable – mais aussi la plus lourde – est sans doute l'évaluation complète des performances de l'ensemble. Nous présentons dans ce chapitre une implantation de OMPHALE sur le site zéro suivi de quelques éléments de mesure d'une implantation particulière.

IV.A. IMPLANTATION

IV.A.1. LA CHAINE DE DEVELOPPEMENT

Le site zéro étant une machine nue, il ne pouvait bien évidemment fournir aucun outil de développement. Nous avons donc utilisé le système MULTICS auquel nous avons accès pour l'écriture et la préparation des applications et des utilitaires, tandis que le site zéro ne servait qu'à la mise au point et à la validation des programmes. Il se posait alors les problèmes liés de choix des langages et au rassemblement des outils.

LANGAGES UTILISES

La gamme de logiciels à écrire comprenait aussi bien des programmes de manipulation du matériel que des parties de système d'exploitation. Aussi nous a t-il semblé souhaitable de choisir deux langages de niveaux différents:

- Un assembleur Z8001 pour le bas niveau que nous utiliserons ponctuellement – soit pour des objectifs difficiles à atteindre voire impossible à réaliser autrement (traitement des déroutements par exemple), soit pour des raisons d'efficacité –.
- Un langage de haut niveau pour les logiciels plus complexes permettant des constructions plus importantes à travail égal. MODULA2 est alors un bon compromis entre un haut niveau d'expression d'algorithme et le bas niveau des travaux que le système doit effectuer.

LES OUTILS UTILISES

ASSEMBLEUR.

Le système MULTICS nous proposait un méta-assembleur GAGE [Bekkers84] accompagné d'un éditeur de liens. Cet assembleur croisé a été utilisé, pour développer le noyau de multiprogrammation EXO (gestion des contextes, gestion des interruptions) ainsi que pour piloter de manière efficace certains dispositifs matériels.

MODULA2

L'obtention du compilateur MODULA2 a été nettement plus difficile. Nous sommes parti d'un compilateur (Version de WIRTH avec SMILER) qui fournissait du code LSI-11 pour un PDP11. Ce compilateur fut adapté pour produire du code Z8001. D'autre-part, il a fallut, pour intégrer ce compilateur dans la chaîne de développement, écrire un utilitaire de conversion entre le format du fichier de sortie du compilateur et le format d'entrée de l'éditeur de lien GAGE. Ces travaux furent réalisés par ATAMENIA et FOURET [Atamenia86] sous la direction de C. CARREZ. Nous avons ainsi pu disposer d'un compilateur croisé pour le site zéro.

UTILITAIRES DIVERS

RELOGEUR

La création du code segmenté à partir des fichiers – structurés en sections – produits par l'éditeur de lien GAGE est assuré par un relogeur. Ceci permet :

- La maîtrise complète de la gestion mémoire rendue nécessaire par le découpage en blocs de la mémoire de module.
- Une édition de lien 'à la main' entre EXO et les programmes écrits en MODULA2.

TELE-CHARGEMENT ET MISE AU POINT.

Le code final peut enfin être téléchargé via une liaison série sur le site zéro. Une précieuse aide à la mise au point existe sous la forme d'un émulateur Z8000 (ZSCAN).

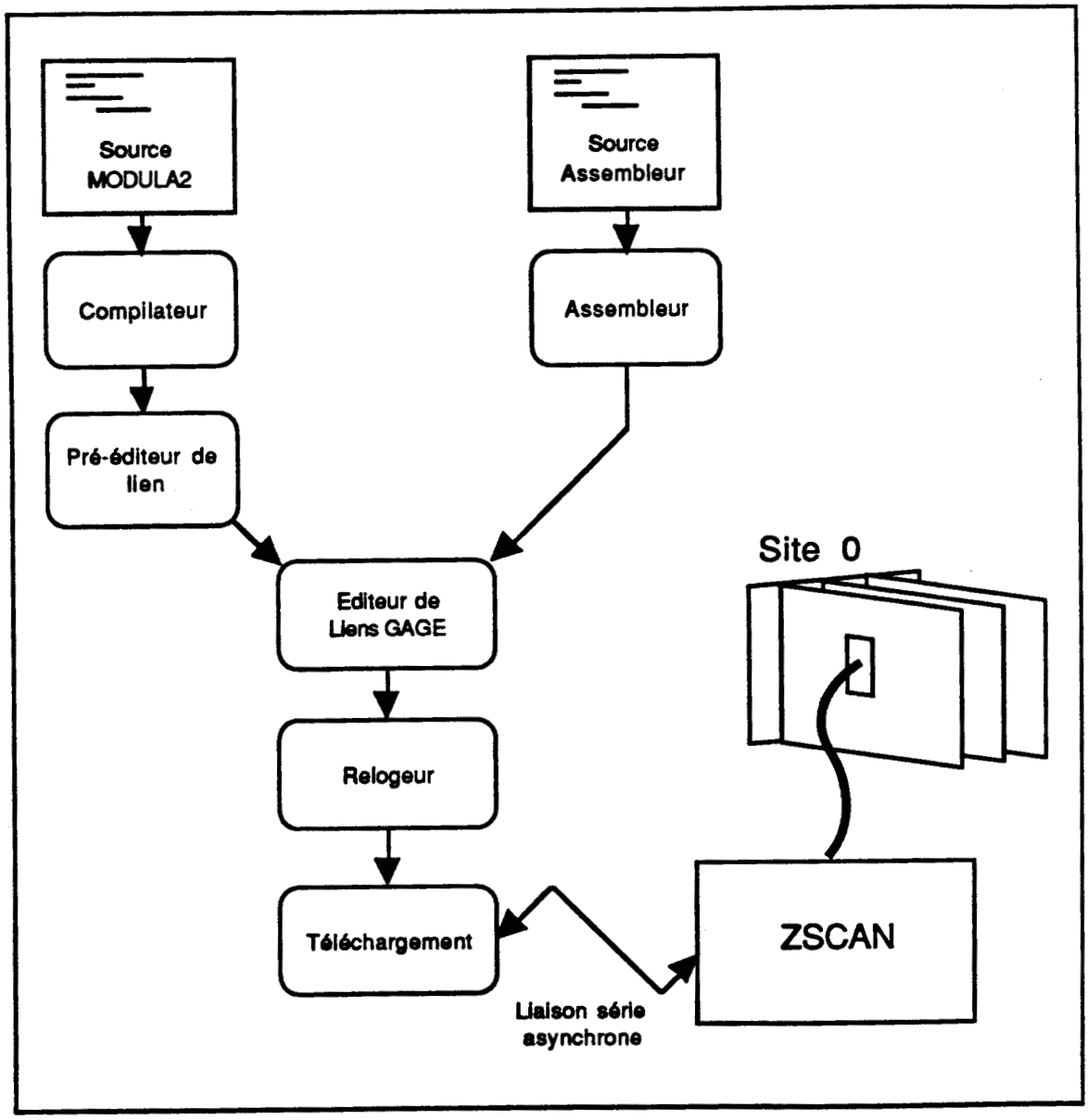


Fig. IV-1. Vue globale de la chaîne de développement.

IV.A.2. LES CHOIX DE L'IMPLANTATION

L'idée principale qui nous a guidés lors de l'implantation d'OMPHALE sur le site zéro a été de découper les activités du site en deux parties que nous avons assignées à chacun des processeurs PN et PC. L'objectif étant d'une part, de bénéficier du parallélisme apporté par le matériel et, d'autre part d'utiliser la protection mémoire fournie par les UGM (disponible uniquement du coté PC).

Il restait alors :

- La définition des moyens de communications d'un processeur à l'autre (utilisation de la mémoire de blocs et des UGM).
- La conception du fonctionnement local des deux processeurs. Bien que ces deux organisations soient indépendantes, il était évident que nous avions intérêt à réutiliser les mêmes outils des deux cotés.

REPARTITION PAR PROCESSEUR.

Nous donnerons dans cette section la répartition des calculs entre les processeurs selon deux parties:

- La première partie se compose de toutes les opérations 'système' telle que le transfert des messages d'un module à l'autre ou l'ordonnancement des modules. Cette partie est assignée au processeur PN .
- La deuxième partie comprend l'exécution du code de chaque module. Elle est à la charge de PC.

Cette dichotomie doit être nuancée par deux remarques pour que l'implantation aboutisse:

– Cette répartition ne concerne que les tâches globales au site (exécution des modules, gestion des environnements, liaisons avec les autres sites). Il reste malgré tout quelques opérations forcément locales à chacun des processeurs; l'exécutif temps réel EXO en est l'exemple le plus caractéristique.

- Certaines opérations 'système' devront être exécutées localement par le processeur PC sous peine d'inefficacité. Nous verrons par exemple qu'un envoi de message se décompose en deux fractions 'usager' et 'système'. Un respect strict de la régle énoncée plus haut imposerait de changer de processeur à chaque message, multipliant le nombre global de changements de contexte.
  Toute ces opérations se satisfont de données (système ou non) liées au module et stockées dans le bloc mémoire qui lui est associé. L'accès par PC à une donnée du système exclusivement possédée par PN est matériellement impossible.

COMMUNICATION ENTRE LES PROCESSEURS.

Dans la version ultime, le processeur PC aurait uniquement une fonction calculatoire. Les entrées-sorties ou les communications avec les autres sites seraient assurées par PN . Ainsi, les seuls éléments requis par PC seraient le module en cours d'exécution et les éléments indispensables à son exécution. C'est-à-dire:

- Le code du module.
- Les données usager du module (variables , pile).
- Les données système du module (environnement, zone d'état du processus lié au module, éléments de la gestion mémoire du module, contenu du message à traiter).
- Les zones d'émission et de réception du module.

Par ailleurs, la création de module au niveau du noyau (détermination du nom du module, création des quais) ne pouvait être bien faite que par PN . Il était donc impératif que PN puisse communiquer des modules nouvellement créés à PC.

En retour, après l'exécution d'un module, PN doit pouvoir récupérer:

- Les mêmes éléments car ils peuvent être demandés pour une nouvelle exécution (PC ne peut pas les mémoriser durablement).

– La zone d'émission qui contient les messages résultant de l'exécution précédente.

La définition du matériel nous imposait une contrainte importante: La mémoire locale de PC, peu importante, ne nous laisse pas la possibilité d'y conserver des données attachées à un module. En revanche, la mémoire de module nous permet un transfert instantané de grandes quantités d'information entre les deux processeurs. C'est donc là que nous mettrons les modules et leurs données. Une exception doit être mentionnée: la description de la mémoire vu par le processus sous forme de segments sera stockée et transférée à l'intérieur des registres de l'UGM.

Ainsi, un module sera complètement connu et manipulé par l'ensemble (bloc de mémoire + registres UGM). Notons au passage que ceci n'interdit pas la coexistence de plusieurs modules dans un seul bloc.

STRUCTURE GLOBALE.

Nous récapitulons ici les éléments symétriques du site zéro. Notons que si la mise en commun de ces éléments est essentielle en ce qui concerne la facilité de développement, elle n'est nullement impérative.

Un pré-requis à la factorisation du logiciel de l'ensemble du site est une certaine identité des aspects matériels et fonctionnels des deux moitiés. Celle-ci est en grande partie respectée par la définition du site zéro. Nous retrouvons en effet de chaque coté:

- Le même processeur Z8001.
- Le même protocole de commande pour le contrôle de la circuiterie de partage (demande par entrée-sortie et déroutement en cas d'impossibilité).
- La possibilité de structurer le logiciel attaché à chaque processeur en couches.

En revanche, d'autres aspects du matériel, comme le chemin d'accès à la mémoire de module, ou encore le rôle de chaque processeur dans la gestion des modules, diffèrent selon les deux cotés. Heureusement, ces

aspects n'ont pas du être pris en compte lors de la conception des couches basses du logiciel.

La figure suivante montre la structure logicielle globale du site. Nous y avons montré les aspects symétriques du matériel surmonté par une couche dupliquée d'exécutif temps réel (EXO). Cette couche est elle même la base du noyau NERF. A partir de cette couche, les éléments sont différents selon les cotés, d'où la distinction entre NERF-N et NERF-C. Le logiciel baptisé MODULE est exécuté uniquement par PC. Dans ce schéma, nous avons mis en valeur la similitude de la circuiterie de partage et la dissemblance des chemins d'accès mémoire entre les deux cotés et leur impact sur le logiciel.

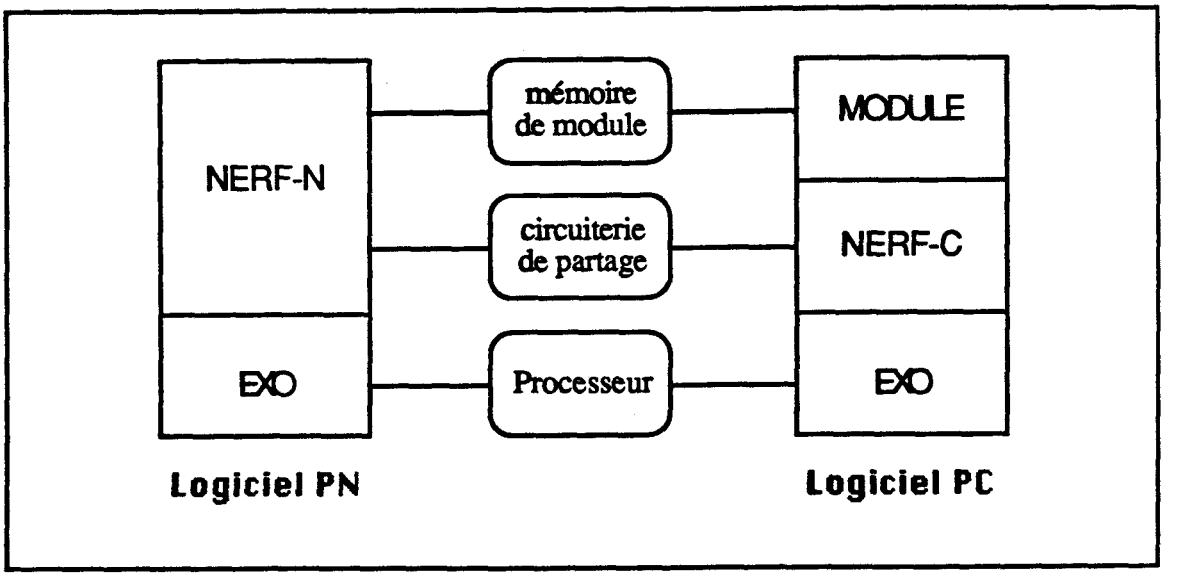


Fig. IV-2. Structure logicielle de l'implantation.

EXO: EXECUTIF TEMPS REEL.

Comme il a été mentionné plus haut, EXO [Lebail86] existe en deux exemplaires identiques sur le site zéro. Son rôle est de gérer un nombre fixe de processus. EXO offre aux processus qu'il gère des primitives:

- De gestion de processus (`DefineProc`, `UnDefineProc`, `Stop`, `SetPointers`).

- De synchronisation par moniteurs[Hoare74] , variante Mesa[Mesa] . (`EnterMonitor`, `ExitMonitor`, `Signal`, `Wait`, `SignalExit`). Un processus exécutant une primitive `Signal` qui réveille un processus bloqué ne perd pas le processeur au profit du processus réveillé.

- De gestion des événements matériels (`WaitTrap`, `WaitNMI`).

SYNCHRONISATION DES LOGICIELS.

La synchronisation entre les logiciels de PN et PC est assurée par le mode opératoire de la circuiterie de partage des UGM.

Par exemple, lorsque PC est disponible, il demande une UGM contenant le descriptif d'un module prêt à être exécuté. Si il n'en existe pas – parce que PN n'a pas eu le temps d'en préparer – , la demande non satisfaite provoquera un déroutement qui sera interprété par le logiciel de PC. Il y a donc là synchronisation des deux cotés du site. Cette synchronisation, pour rudimentaire qu'elle soit (PC et PN disposent respectivement de deux commandes et quatre commandes), suffit à un fonctionnement correct du site.

DU COTE N.

Au dessus d'EXO, coté N, nous trouvons NERF-N, la partie du noyau attribuée au processeur N. Son rôle est:

- recueillir les messages émis par les modules.

- acheminer ces messages jusqu'à leur destinataire local (soit un module du même site, soit la station de transport).

- gérer l'ensemble des modules du site, c'est-à-dire, pour chaque module:

  - Garder ses données (état des quais, environnement, variables).

  - Maintenir les files d'attente des messages arrivés.

  - Préparer l'exécution déclenchée par l'arrivée d'un message.

Rappelons que pour cela, NERF-N, en plus de l'intégralité des données du module (qui voyage de PC à PN), dispose de structures de données qui lui sont propres.

Cette partie logicielle n'a pas pu être développée plus avant par manque de temps et par l'état inachevé du site zéro. Une version similaire a été cependant validée sur une implantation d'OMPHALE sur SPS7. Sa faisabilité est donc assurée.

DU COTE C.

Sur le processeur PC, EXO sert de base à NERF-C. Cette portion du noyau est beaucoup moins ambitieuse: les travaux 'système' les plus lourds sont assurés par PN. Il reste à PC uniquement deux fonctions:

- Contrôler l'enchaînement de l'exécution des modules, tâches brièvement décrite par le cycle: demande d'UGM, en cas de satisfaction, lancement du processus usager qui exécute le code du module. En fin d'exécution, le processus usager rend le contrôle à NERF-C qui relâche l'UGM en vue du traitement final par PN.

- Réaliser le petit nombre d'opérations système dont nous avons vu que l'exécution était possible et préférable sur PC – à savoir: gestion de la zone d'émission et de l'environnement –. Ces opérations sont toutes déclenchées par le processus usager et par le biais de "SYSTEM CALL".

FONCTIONNEMENT DU SITE

Nous allons décrire ici les opérations qui sont impliquées dans la réalisation d'une période d'activité d'un module. Nous prendrons comme exemple, l'activité déclenchées chez le module `PILE` (décrite dans le chapitre I) par la réception d'un message `M`.

1) Quelque soit son site d'origine, le message `M` est pris en compte par PN.

2) Ce message est mis dans la file d'attente des messages associée au quai destinataire (en l'occurence `Depiler`). Ces files d'attente sont des structures de données locales à PN. Si le quai est ouvert et que le module est en attente, l'action suivante est déclenchée. Sinon, le message est simplement stocké dans l'attente de l'ouverture du quai.

3) Demander l'allocation d'une UGM à la circuiterie de partage. Nous supposerons qu'une UGM est disponible.

4) PN prépare l'exécution du module. Cela consiste à

   - Retirer les liens du message et les ajouter à l'environnement du module. Ces liens sont remplacés par leur nom local.

   - Placer le message dans une zone accessible par le module.

   - Charger les registres de l'UGM avec les valeurs qui décrivent l'espace mémoire associé au module. Cet espace est contenu totalement dans un bloc.

5) Rendre l'UGM disponible pour PC. Il s'agit d'une commande 'fin de chargement' destinée à la circuiterie de partage.

6) A la suite de ces opérations et selon son activité (pas forcément immédiatement après), PC (plus précisément NERF-C) va demander l'acquisition d'une UGM chargée.

7) Ayant obtenu satisfaction, NERF-C va lancer l'exécution du processus usager lié au module. Ceci est fait par l'intermédiaire d'EXO.

8) Les actions du processus usager sont données par la programmation du module `PILE` du chapitre I. Nous remarquerons en vrac:

   - La lecture du message reçu.

   - L'envoi d'un message (appel d'une primitive système). Cet appel est entièrement traité par PC. Pour cela, PC a besoin d'accéder à des données liées au module mais non visibles par le processus usager (comme ici la zone d'émission). Nous pourrions dire la même chose au sujet des primitives de manipulation de lien. De plus, ces appels sont synchrones.

   - Pour finir, le processus usager, exécute une primitive `WAIT()`. L'exécution de cette primitive déclenche l'arrêt du processus usager et le réveil de NERF-C.

9) NERF-C rend alors l'UGM (commande de fin d'exécution) à la circuiterie de partage.

10) PN – à son heure – reprend, lors de sa prochaine commande 'début de sauvegarde', le contrôle de l'UGM, donc du module, et peut alors traiter la zone d'émission du module, c'est-à-dire en extraire les messages créés lors de cette période d'activité.

STRUCTURE MEMOIRE D'UN MODULE.

Outre le problème du choix des structures de données du noyau qui ne sont pas notre propos actuel, le bon fonctionnement d'un processus dans un espace mémoire segmenté comme celui du Z8001 demande bien évidemment la définition des segments. Nous nous intéresserons ici à la segmentation d'un module. Celle-ci doit obéir à des impératifs variés:

- L'espace mémoire du module doit tenir en entier dans un bloc mémoire.
- Le code généré par le compilateur MODULA2 suit des régles dont il faudra tenir compte.
- Le principe du moindre privilège nous enseigne que le fonctionnement du logiciel sera d'autant plus fiable que les opérations sur les objets (donc sur la mémoire) seront restreintes au minimum nécessaire. Nous devrons donc, pour chaque zone mémoire, faire l'inventaire des accès que le processus usager devra y faire.

CONTRAINTES IMPOSEES PAR LE COMPILATEUR.

Arrivé à ce stade de la discussion, il est important de distinguer la notion de MODULE omphale (environnement d'exécution), de la notion de module utilisée en MODULA2 (portion du texte source). Nous appellerons module source cette dernière notion. Ainsi un MODULE OMPHALE résultera de l'assemblage de plusieurs modules sources.

L'examen de la configuration mémoire des programmes générés par le compilateur révèle trois mode d'accès:

- Le code lui-même est contenu dans un segment.

- Les données locales à une procédure sont mises dans la pile. Le segment attribué à la pile doit donc être d'autant plus important. D'autre part, cette technique présente l'inconvénient d'une complexité d'accès qui pénalise beaucoup le temps de calcul.

- Les données globales à un module source – statiques – sont placées dans un segment de données associé au module. Dans le résultat final nous pouvons donc avoir un segment par module source. Inversement, nous ne pouvons pas forcer deux variables globales de deux modules sources différents dans le même segment.

Globalement, nous suivrons les régles suivantes:

- Utiliser plutôt les variables globales par soucis d'efficacité surtout dans les modules source système.

- Grouper les données par segment (en fonctions des droits requis pour un fonctionnement correct du logiciel). Chacun de ces segments regroupera les variables globales d'**un** module source.

Rappelons que l'attribution des numéros de segments est faite au moment de l'édition de lien.

SEGMENTATION CHOISIE

Le recensement des données et parties de code et leur regroupement selon les accès requis et leur durée de vie fait apparaître un découpage en 7 segments:

- La mémoire privée PC, située en segment 0 à cause de la conception matérielle du site. Cette mémoire comprend le code et les données de l'exemplaire (local à PC) d'EXO et de NERF-C . Ce segment ne peut être accédé qu'en mode système mais avec tous les droits (lecture, écriture, exécution).

- Le code du module destiné à être exécuté par le processus en mode usager.

- Les données statiques du module. Ce sont les variables globales du module usager.

- La pile dont la gestion est invisible au programmeur du module, car assurée par le compilateur MODULA2. Cette pile contient, les variables locales aux procédures, les adresses de retour, les paramètres, la zone de display, etc...

Outre ces zones qui ne sont pas caractéristiques de l'architecture OMPHALE, on peut dénombrer trois autres zones déduites du fonctionnement de OMPHALE.

- La zone de communication contient les variables qui permettent de faire communiquer le processus usager et NERF-C (communication par mémoire commune). Une alternative à cette zone serait de faire communiquer ces deux logiciels par appel de procédure (et passage de paramètres dans les registres). L'étude du coût en temps de cette technique ainsi que l'absence de nécessité de protéger ces variables nous a fait préférer la première solution.
  Cette zone est accessible en lecture et en écriture que l'on soit en mode système ou non.

- La zone d'émission sert à stocker tous les messages émis durant la période d'activité courante. Elle est entièrement gérée par les procédures d'envoi et d'annulation de message de NERF-C (elles même appelées par le processus usager).
  Seul NERF-C (processus en mode système) doit avoir accès à cette zone.

- L'environnement du module OMPHALE est situé dans un segment particulier. Cette zone, tout comme la zone d'émission, ne doit être manipulée que par les procédures de NERF-C mises à disposition du processus usager (duplication de lien, restriction des droits etc...). Il reste cependant préférable de dissocier les deux zones; d'une part parce que le coût de cette séparation est nul, d'autre-part parce que la «durée de vie» des deux zones est différente. La zone d'émission, une fois traitée par PN est vide et peut donc être purgée. L'environnement lui, doit être conservé d'une période d'activité à l'autre.

La figure suivante représente l'aspect de la mémoire d'un module, mettant en évidence le découpage en segments, la localisation physique de chaque segment et la visibilité qu'a le processus usager de cette mémoire.

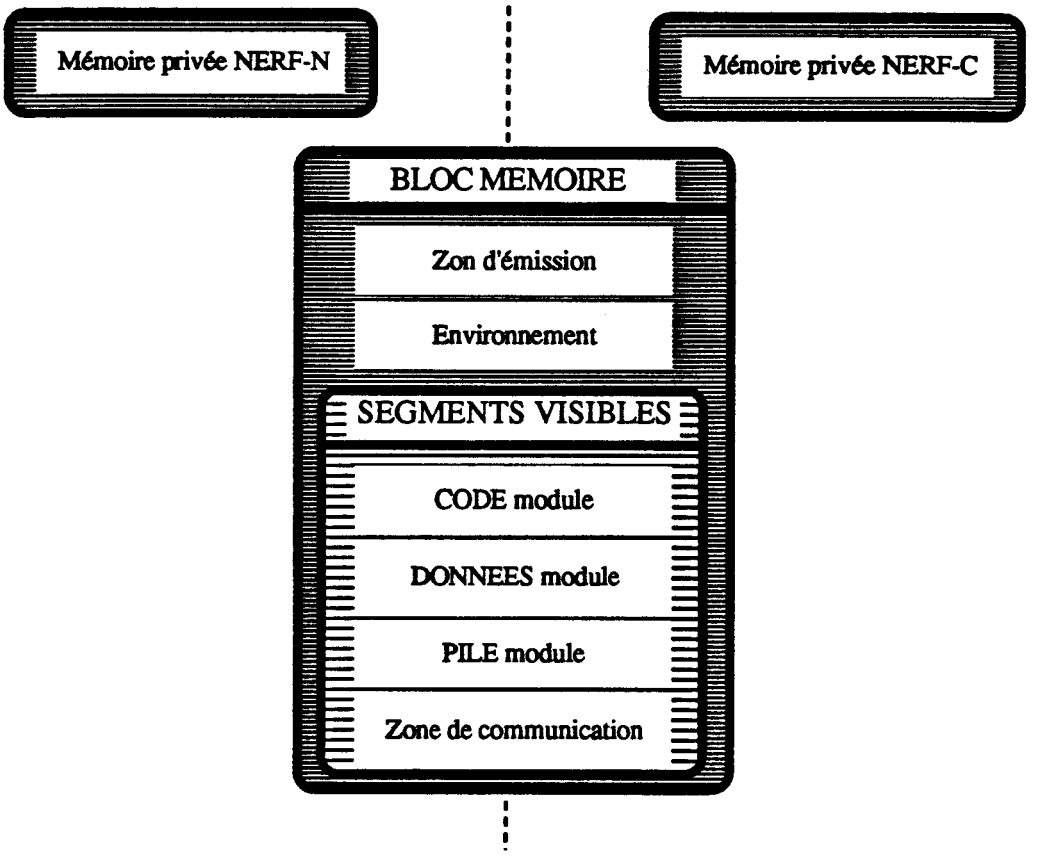


Fig. IV-3. Structure mémoire d'exécution d'un module.

Une autre représentation, moins imagée mais plus complète de cette mémoire est le tableau ci-dessous. On y trouve, pour chaque segment, les droits d'accès selon les deux modes ainsi que sa durée de vie.

Numéro Segment	Rôle	Durée de vie	Accès Usager	Accès Système
0	Mémoire privée PC	spécial	-	R/W/X
1	Code usager	rem.	X	-
2	Données statiques	rem.	R/W	R/W
3	Pile usager	rem.	R/W	R/W
4	Zone de Communication	act.	R/W	R/W
5	Zone d'émission	act.	-	R/W
6	Environnement	rem.	-	R/W

signification des abréviations

rem.	Rémanent: le segment doit être gardé intact d'une période d'activité à l'autre.
act.	Une période d'activité: Ce segment est reconstruit ou vidé à chaque période d'activité du module.
spécial	La mémoire privée de PC contient les données de NERF-C et doit donc être conservée. Elle n'est cependant pas liée au module.
R	Droit de lecture.
W	Droit d'écriture.
X	Droit d'exécution.

Fig. IV-4. Différents segments.

LES COUCHES D'EXECUTION

Indépendamment de la structure mémoire d'un module, il est intéressant d'expliciter clairement et de manière synthétique le «trajet» suivi par le processus à travers les différentes couches d'exécution au cours d'un appel système. Ces couches sont au nombre de quatre:

- La couche utilisateur dirige le comportement du module au moyen des primitives fournies par l'architecture OMPHALE. Le terme «utilisateur» est ici à prendre au sens large: cette couche se trouve bien sûr également dans les modules qu'on pourrait appeler système (le gérant de module par exemple). Ce qui caractérise cette couche est un niveau de privilège minime réduit à ses variables propres et à la zone de communication.
  Nous incluons dans cette couche les modules sources de description de l'architecture, à savoir:

  - Le module OMPHALE.

  - Les modules créés par le préprocesseur pour aider à la gestion des formats de message.

- La couche de transit entre les modes système et usager du processeur. Cette couche intervient pour restaurer un bon environnement aux routines qui s'exécutent en mode système. Hormis l'instruction centrale "SYSTEM CALL", elle opère principalement des manipulations de pile.

- La couche NERF-C représente l'ensemble des primitives qui gèrent l'environnement et la zone d'émission en mode système. Cette couche suffit à la plupart d'entre elles. Dans ce cas, après la réalisation effective de l'opération, le processus peut remonter jusqu'à la couche utilisateur. L'exception à cette règle est le fait de la primitive WAIT. Cette dernière demande l'intervention de la dernière couche.

- La couche EXO gère l'ensemble des processus du côté C. En particulier, lors de l'opération WAIT, EXO suspend le processus associé au module et relance le processus maître de PC qui regarde la disponibilité de nouveaux modules dans les UGM. EXO a déjà été détaillé plus haut.

LE DECOR DE PROGRAMMATION.

Après la description du fonctionnement interne du système, nous désirons expliciter les éléments de programmation déduits des choix fixés plus haut. Ces éléments sont des procédures ou des variables introduits par le biais de modules MODULA2 mis à la disposition de l'utilisateur.

MECANISME GENERAL D'APPEL DES PRIMITIVES.

Le module source baptisé OMPHALE permet d'appeler les primitives fournies par NERF-C depuis un source MODULA2. Les procédures correspondantes de ce module sont implantées par une simple instruction «SYSTEM CALL» qui place le processus en mode système, lui donnant ainsi le droit de manipuler les zones mémoire réservées (environnement, zone d'émission). Le contrôle de validité des paramètres est fait dans la routine appelée.

GESTION DE L'ENVIRONNEMENT.

les primitives de gestions de l'environnement rendent possible la manipulation des liens d'un module. Il s'agit ici d'encapsuler l'environnement dans un ensemble de procédures qui assurent son

intégrité. Ces procédures sont au nombre de trois. Toutes trois admettent comme paramètres des types simples.

GESTION DES MESSAGES.

Là aussi, il s'agissait de protéger la zone d'émission d'une manipulation intempestive par le module proprement dit. La conception de ces procédures est cependant plus complexe que pour la gestion de l'environnement. Les objets à traiter sont en effet de types non connus à priori (il importe de laisser le programmeur libre du contenu, donc du format des messages échangés par ses modules). La difficulté consistait donc:

- A tourner le typage demandé par MODULA2 en gardant si possible la protection qu'il apporte.

- A déterminer une méthode de description du format pour permettre au système de traiter correctement les messages. Ce dernier doit non seulement acheminer une valeur binaire de longueur variable vers le destinataire, mais aussi, quand ce message contient des liens, les ôter (ou les ajouter) à l'environnement des modules émetteur (respectivement destinataire).

La solution proposée préconise un découpage selon deux niveaux:

- Utilisation de procédures typées par les modules. Ceci nécessite l'écriture d'un jeu de primitives par format de message. A fortiori, ceci impose une déclaration explicite des messages et de leur structure, d'où les déclarations de format introduites par le langage OMPHALE.

- Ces procédures utilisent des outils de plus bas niveau fournis par le système et qui permettent:

  - La description d'un format de message (primitive OFFSET).

  - L'envoi d'un message non typé (primitive SYSTSEND utilisant le type WORD du MODULA2).

Ces deux procédures utilisent le mécanisme général décrit plus haut.

A l'écriture, le travail qu'impose ce découpage est largement réduit par l'utilisation d'outils automatiques (préprocesseur). En revanche, à l'exécution, cette solution est assez lente. En effet, la chaîne des appels MODULA2 nécessaires à un envoi de message consomme un temps non négligeable. Une forme optimisée de cette solution demande le développement d'un compilateur spécifique.

LA PRIMITIVE *WAIT*.

L'aisance de programmation d'un module impose de considérer cette action comme une primitive. Un module qui doit attendre un message particulier dans un traitement critique (qui ne doit pas être interrompu par l'arrivée d'autres messages) est ainsi écrit plus simplement et sans artifices. Néanmoins, cette primitive marque bien la fin de la période d'activité du module. A cette exception près, l'implantation de cette primitive ne diffère pas du modèle général.

LA ZONE DE COMMUNICATION.

Devant le coût en efficacité de l'appel de procédure MODULA2, nous avons essayé d'établir le maximum de communication entre NERF-C et le module par l'intermédiaire de variables communes. Ceci n'est bien entendu possible que si les protections demandées par ces variables peuvent s'exprimer en termes de droits d'accès, excluant toute procédure complexe (l'indirection d'un nom local vers lien par exemple). Cette démarche concerne deux familles de variables:

- **Les variables destinées au module** sont positionnées par NERF-N pour être éventuellement consultées par le module. Ces valeurs ne seront plus utilisées par le noyau et peuvent être modifiées par le module. Il s'agit:
  - RECEIVEDMESSAGE: Contenu du message qui a déclenché la période d'activité du module .
  - RECEIVINGSERVICE: Désignateur du service sur lequel ce message a été reçu.
- **Les variables système librement manipulables par le module.** Cette famille est réduite à la seule variable OPENEDSERVICES. Ce mot de 16 bits

sert au module pour indiquer quels seront les quais ouverts entre les périodes d'activités.

Toutes ces variables sont regroupées dans la zone de communication. Pour mémoire, cette zone est le segment 4 du module. Elle est par conséquent décrite comme l'ensemble des variables globales d'un module source particulier.

VUE GLOBALE.

Une vue globale du fonctionnement d'un module peut être représentée dans la figure suivante:

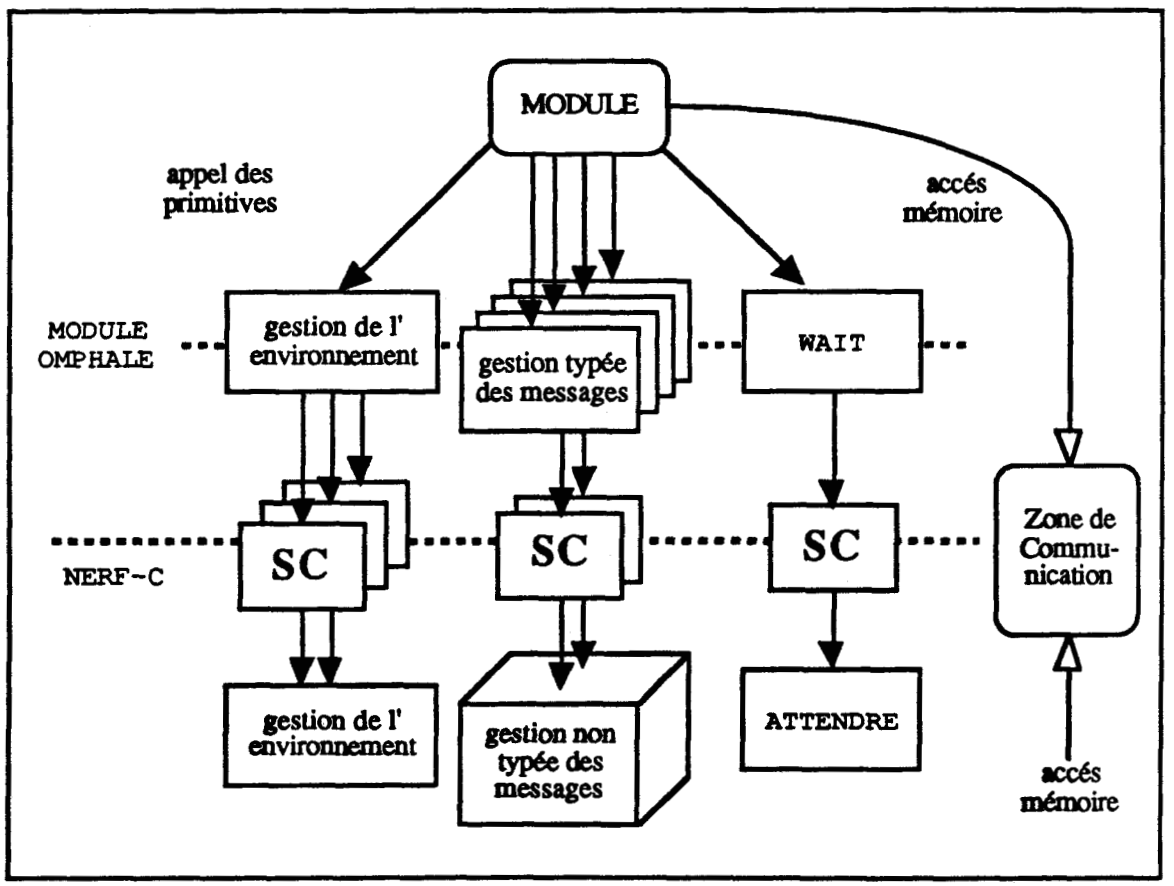


Fig. IV-5. Vue globale de l'implantation.

IV.A.3. L'IMPLANTATION

Il s'agit ici d'indiquer certains détails d'implantation qui ne présentent pas un intérêt capital quant à la faisabilité ou même au

fonctionnement de l'architecture mais qui influent grandement sur les performances que nous allons mesurer.

CARACTERISTIQUES GENERALES

LA COUCHE DE TRANSIT.

La nécessité de manipuler la pile a imposé l'utilisation de l'assembleur comme langage d'écriture de cette couche logicielle.

NERF-C.

Les algorithmes des primitives de NERF-C ont été écrits en MODULA2. Ceci pour obtenir une lisibilité correcte. Ces primitives compilées ont servi de base aux chiffrages des durées d'exécution. Elles ont été optimisées, souvent au dépend de la propreté de programmation. Cette remarque peut être illustrée par deux exemples:

- Après avoir comparé les méthodes d'accès utilisées par les programmes compilés, nous avons systématiquement déclaré les variables globalement à un module.

- La structure du tableau qui représente l'environnement n'est pas cachée à l'intérieur du module ENVI. Qui plus est, les modules de NERF-C utilisent le type WORD et la fonction ADR (permet d'obtenir l'adresse d'une variable).

Nous avons, procédé à la réécriture de certains de ces modules en assembleur pour évaluer la perte de temps due à l'utilisation de MODULA2. Nous discuterons de cet aspect lors de l'analyse critique des mesures.

L'ENVIRONNEMENT.

L'environnement reprend le principe des C-listes[Levy84] : il se compose d'une table de liens. Le module accède à un lien en mentionnant son nom local qui correspond à l'indice du lien dans la table.

Nous n'avons pas introduit de structure supplémentaire pour la gestion de l'espace libre dans cette table: la recherche d'une position libre est faite par balayage séquentiel.

REPRESENTATION D'UN LIEN.

Un lien est une structure de quatre champs utiles.

- Un pointeur vers le module cible. L'encombrement de ce champ (4 octets) importe peu ici.

- Les droits sur chacun des 16 services du lien. Il existe trois possibilités pour ce droit: aucun droit, utilisation unique et utilisation illimitée. Nous avons choisi la représentation la plus proche possible des opérations fournies au programmeur: deux mots de 16 bits (1 bit par service) indiquent respectivement «utilisation unique» ou «aucun droit». Normalement, pour un service, les deux bits ne peuvent être simultanément à 1.

- Un booléen indique si le lien est duplicable.

Il existe en outre des champs destinés à la gestion de la table.

- L'état de la position de la table peut prendre les valeurs suivantes:

  – Libre : La position ne contient aucun lien.

  – Utilisé : La position contient un lien.

  – Envoyé : La position contenait un lien qui a été inclus dans un message durant la période d'activité courante. Ce lien ne peu donc pas être détruit et à fortiori, la position ne peut pas être réutilisée. Ce lien peut éventuellement redevenir utilisable, suite à la destruction du message qui le contient.

  – Verrouillé : Permet d'interdire la destruction de liens critiques comme le «l'auto-lien».

- Un numéro de version a été aussi ajouté au lien. Sa justification est la suivante: Une position peut être réutilisée. Un même indice dans l'environnement peut donc, à des moments distincts de la durée de vie du module, représenter des liens complètement différents. Si le nom local d'un lien est uniquement cet indice, il a possibilité de confusion (par exemple le programmeur peut mémoriser ce nom local dans une variable et le réutiliser plus tard, en oubliant que le lien désigné a été

détruit. Ce nom local peut alors correspondre à un autre lien). Une parade à ce problème est d'accoler au nom local de lien un numéro de version du nom local qui est incrémenté à chaque réutilisation de la position.
Une remarque est à faire au sujet de ce numéro de version: Ce mécanisme n'est pas partie intégrante de l'architecture OMPHALE. L'absence de ce mécanisme ne diminue pas le niveau de protection offert par le système. Il doit, à la place, être vu comme une aide à la mise au point des modules au même titre qu'un contrôle de bornes d'indice dans un tableau ou que l'état «verrouillé» d'un lien.

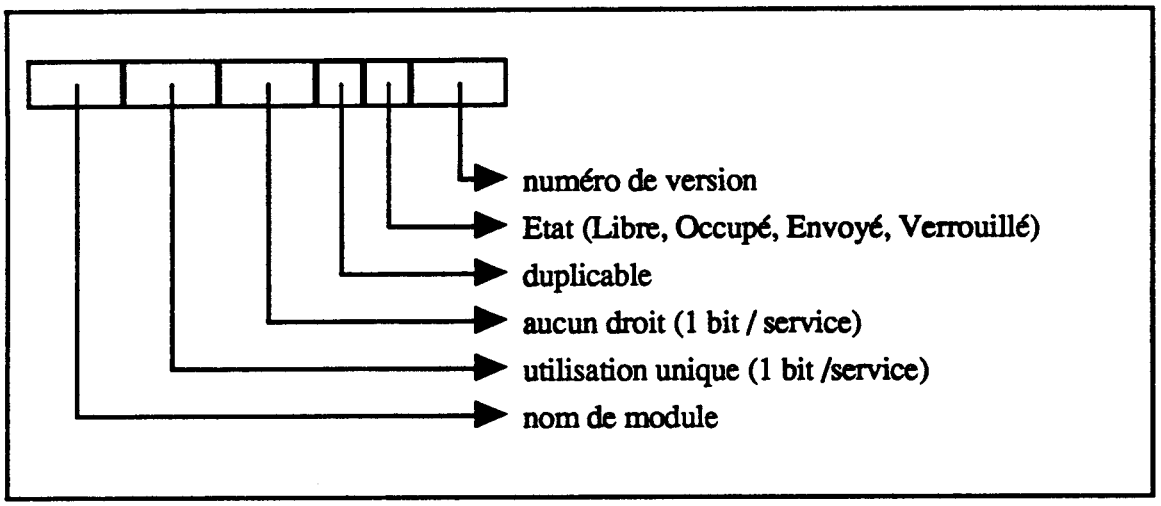


Fig. IV-6. Représentation d'un lien.

LA ZONE D'EMISSION.

Deux décisions indépendantes sont à prendre pour définir le fonctionnement de cette zone.

- Comment est géré l'espace mémoire de cette zone? en particulier, comment est stocké un message?

- Donner la l'inventaire des primitives permettant le traitement et l'envoi des messages (nous avions vu que ce traitement impliquait la description du format du message).

GESTION DE LA ZONE D'EMISSION.

Cette version expérimentale de OMPHALE est réduite aux opérations que nous croyons les plus courantes dans la programmation des modules. Nous avons donc négligé les primitives d'annulation d'envoi de message (UNSEND et UNSENDALL). Tout naturellement, nous avons essayé d'optimiser la gestion de la zone d'émission pour l'opération d'envoi, au détriment des primitives non implantées. De ce fait, cette gestion est des plus rudimentaire: les messages sont concaténés dans la zone d'émission supposée suffisamment vaste. La procédure de récupération des messages annulés en est plus difficile mais reste possible, Un pointeur désigne le haut de la zone utilisée selon la figure suivante:

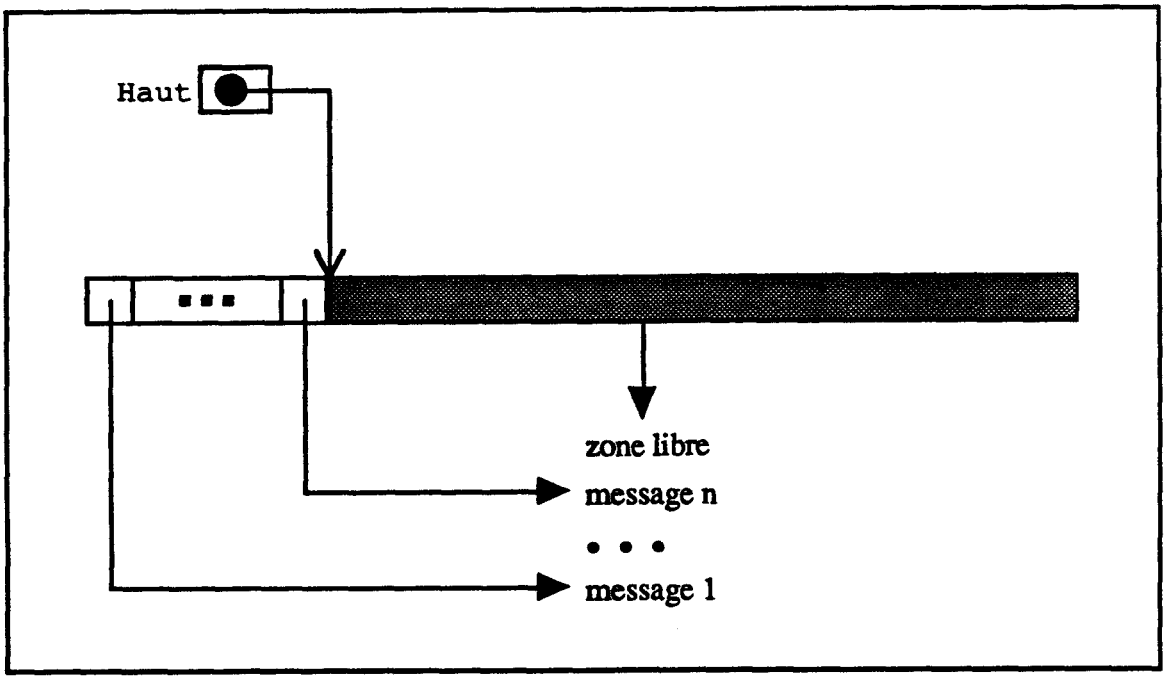


Fig. IV-7. Gestion de la zone d'émission.

La structure de stockage utilisée pour les messages est induite par deux remarques:

- Les messages sont de longueur variable. Il faut donc mémoriser la taille du message dans la structure choisie.
- Seul parmi les éléments inclus dans le message,les noms locaux de modules doivent être traités par NERF-C. Tandis qu'une recopie binaire

des autres éléments est suffisante, la présence d'un nom local de lien impose une traduction (via l'environnement du module) pour émettre effectivement le nom unique de module correspondant.
Cette traduction demande des structures de données supplémentaires. La taille de ces noms est fixe. Il est donc suffisant de connaître le nombre et la position des liens dans le message: les noms locaux sont traduits en lien et concaténés lors de l'envoi du message tandis que les autres éléments du message sont simplement recopiés. Nous avons borné à 10 le nombre de liens par message.

La structure adoptée est la suivante:

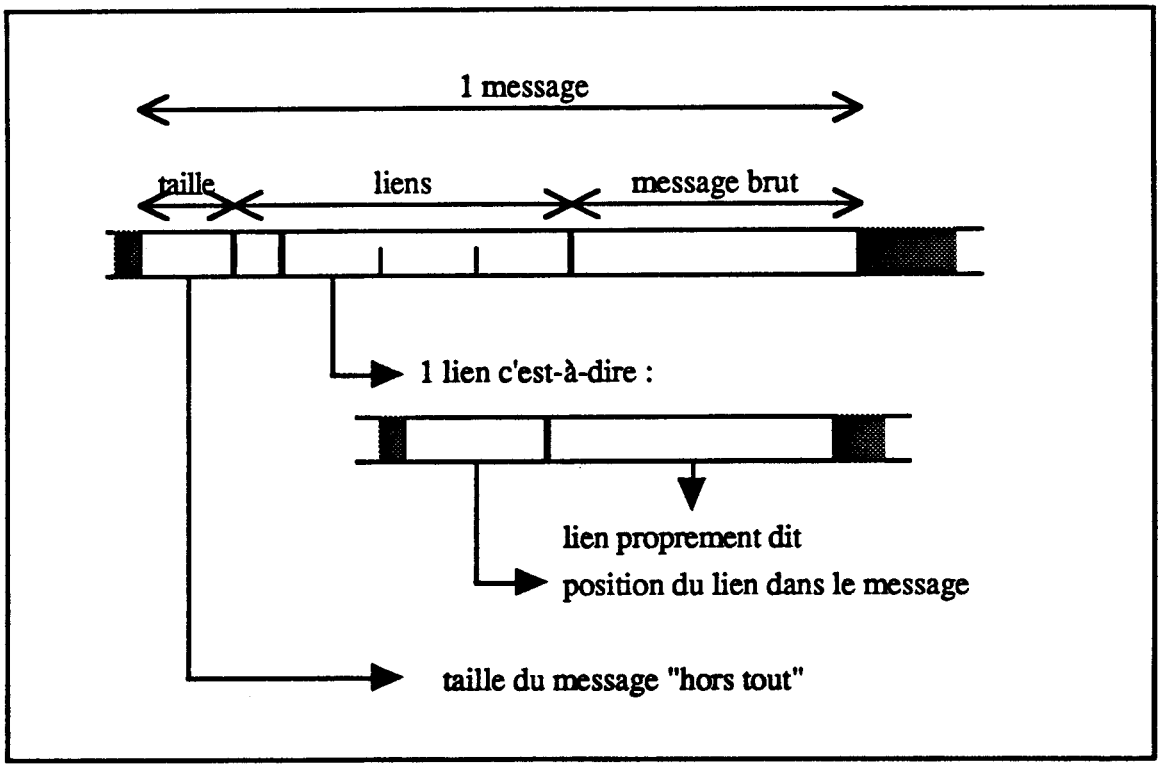


Fig. IV-8. Exemple de message comportant trois liens.

IV.B. EXEMPLE CHOISI

Nous nous intéressons plus précisément aux performances de l'action `EMPILER` à travers la pile que nous avons écrite. La description détaillée de cette opération nous permettra à la fois d'illustrer l'implantation et de fournir le support de nos mesures.

Nous appellerons:

APPLICATION l'ensemble des modules qui réalisent la pile (i.e. `Pile` et `Element`).

ACTIVITE La période d'activité déclenchée chez un des modules concerné par la réception d'un message.

OPERATION Toutes les actions provoquées par la demande d'empilage d'un entier. En particulier, l'opération comportera plusieurs phases d'activité et les travaux effectués par le système d'exploitation (NERF-C, NERF-N ET EXO),

IV.B.1. JUSTIFICATION

Le choix de l'opération `EMPILER` est la conséquence des objectifs fixés:

- Avoir un exemple réduit afin de minimiser, et les erreurs de chiffrage, et la complexité de l'interprétation.
- Prendre une opération sur laquelle le facteur d'échelle (ici la taille d'un élément de pile) influe. L'écriture de l'application ne nous fournit pas d'activité dont la durée est directement sujette à la taille d'élément choisie. Nous avons, à la place, utilisé l'opération `EMPILER` qui est influencée indirectement: une modification de la taille d'un élément change la durée de l' opération en accroissant le nombre de messages échangés, donc le nombre total d'activités impliquées dans l'opération.

Toujours pour mettre en évidence le facteur d'échelle, nous avons travaillé sur un nombre significatif d'opérations élémentaires: soit 108[1] opérations d'empilage.

---

[1] Il nous faut un nombre qui ait suffisamment de facteurs premiers (pour avoir des calculs entiers): 108 = 2 x 2 x 3 x 3 x 3

IV.B.2. FONCTIONNEMENT MACROSCOPIQUE

Le fonctionnement macroscopique, c'est à dire en terme d'enchaînement de périodes d'activité des modules découle directement de la conception de l'application et est indépendante de l'implantation.

L'opération décrite peut demander l'intervention de trois modules:

- Le module pile prend en compte la requête. Ce module demande éventuellement la création d'un nouvel élément. Pour cela, il fait appel au module «gérant des modules».
- Le gérant des modules crée un nouvel élément.
- L'élément sommet reçoit alors la valeur à empiler.

ENCHAINEMENT DES ACTIVITES

Les activités déclenchées par l'opération sont représentées dans la figure suivante. Le flux d'activités peut prendre deux chemins selon le taux de remplissage de l'élément sommet.

Si l'élément sommet n'est pas rempli, l'opération se décompose en deux activités: réception de la demande par le module PILE et transmission à l'élément sommet d'une part, Empilage effectif de l'entier dans l'élément sommet ELEMENT d'autre-part.

Lorsque l'élément sommet est saturé, le module doit créer un nouveau sommet: il demande cette création au gérant des modules, reçoit en échange un lien vers le nouvel élément (ceci par autre activité), et assure le chaînage des éléments entre eux. Le flux d'activité peut alors reprendre le schéma général.

Globalement, on peut dénombrer 5 activités nécessaires à l'empilage qui sont lancées selon le schéma suivant (les activités non mises en jeu ne sont pas représentées):

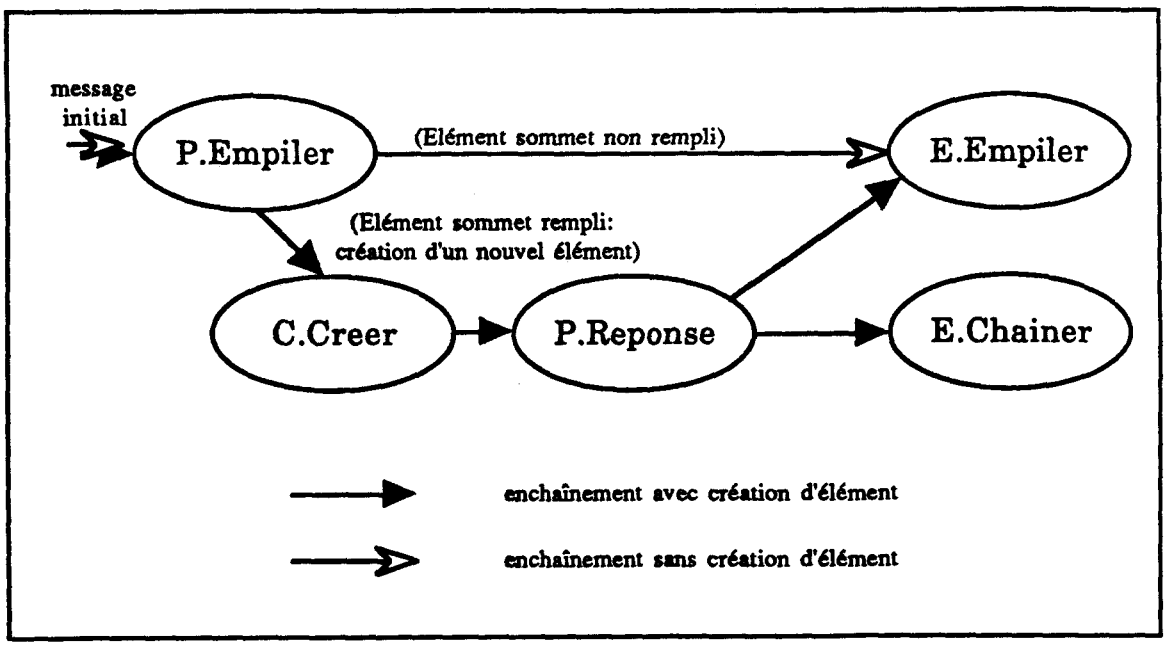


Fig. IV-9. Activités nécessaires à la réalisation de l'opération d'empilage.

Deux remarques sont à faire sur la description précédente:

- Nous supposons la démarche suivante de la part du module P lors de la création d'un nouvel élément: Le module envoie au gérant de modules (C) le nom du «prototype» dont il veut créer un exemplaire (ELEMENT), puis se met en attente d'une réponse de la part de C. Cette réponse est en fait un lien vers le module élément nouvellement créé.
  Ce fonctionnement implique deux périodes d'activité pour le module P lors de cette opération. La non réalisation du gérant de module nous empêche de détailler plus avant son fonctionnement.

- Même si le schéma ci-dessus suggère que les activités `Chainer` et `Empiler` du module **E** peuvent être effectuées en parallèle (et le fonctionnement du module le permet réellement), le schéma d'exécution OMPHALE interdit le parallélisme de deux activités d'un même module.

IV.B.3. FONCTIONNEMENT MICROSCOPIQUE

Nous allons représenter les différents niveaux de fonctionnement par rapport à une des activités de l'opération, à savoir l'activité de **P** quand il reçoit une demande d'empilage. Ce sont, du plus visible au plus caché:

- Le programme dit «usager» décrit par le concepteur du module. Ce programme utilise les primitives fournies par le module source OMPHALE.

- Le module source OMPHALE cache en fait NERF-C et l'interface nécessaire à l'exécution de NERF-C: Appel MODULA2, passage en mode système. Nous avons décomposé NERF-C en deux parties qui ne font pas appel aux mêmes mécanismes:

  - NERF-Cs: Les appels synchrones sont des appels directs des primitives. Une fois la primitive effectuée, le contrôle repasse au niveau usager. Le processus appelant ne perd pas le processeur.

  - NERF-Ca: Lors de l'appel asynchrone WAIT qui clôt une période d'exécution d'un module OMPHALE, il y a changement du processus actif: Cet appel cache vis à vis du concepteur du module le fonctionnement d'OMPHALE, c'est-à-dire, le retour au processeur maître via EXO, la libération de l'UGM, la recherche d'une UGM libre, etc... Le processus réveillé à ce moment est appelé NERF-Ca.

- EXO qui commute les processus et effectue les changements de contexte.

- Nous avons également figuré NERF-N pour toutes les opérations de chargement, des UGM, gestion des files d'attentes, réveil des modules, traitement de la zone d'émission.

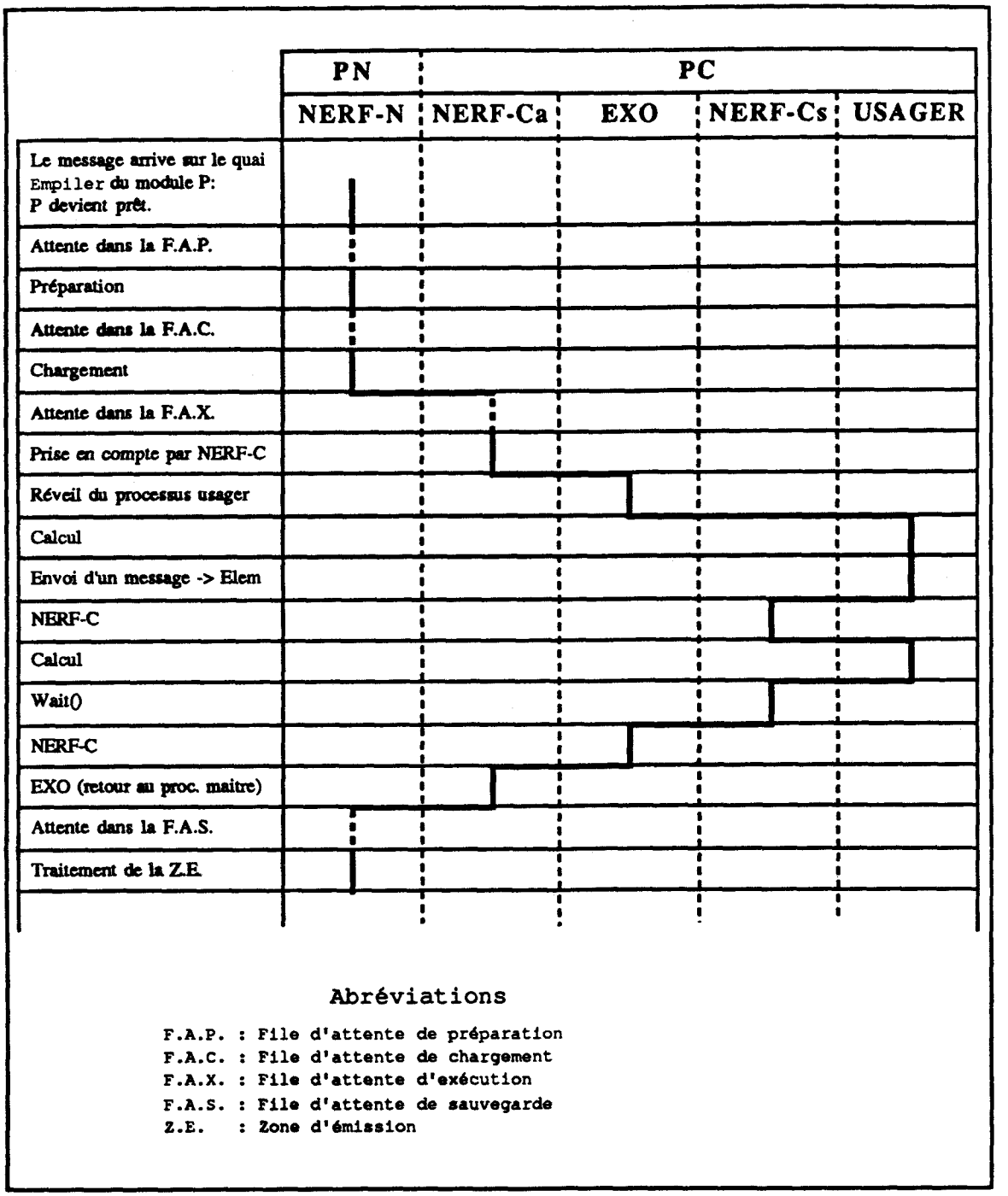


Fig. IV-10. Détail d'une période d'activité. Le tracé représente le passage du flux de calcul lié à une période d'activité d'un module entre les différentes couches logicielles. La colonne de gauche s'exécute sur PN. Les autres colonnes s'exécutent sur PC. On peut également mettre en évidence les deux processus: NERF-Ca en deuxième colonne et le processus usager qui englobe les deux dernières colonnes.

IV.C. MESURES.

Les mesures que nous mentionnons ici sont simplement l'évaluation de la durée d'exécution de l'application. De la même manière que pour la description de l'exemple, nous pourrons mettre en évidence des mesures macroscopiques (pour l'ensemble de l'application) et dépendantes du degré de granularité de l'application et microscopiques (pour une période d'activité) que nous utiliserons pour calculer le taux d'overhead lié au fonctionnement d'OMPHALE sur le site zéro. Il peut également sembler intéressant de mettre en évidence le temps passé dans chaque niveau de fonctionnement (c.f. description microscopique).

IV.C.1. LE PRINCIPE DES MESURES.

Le principe des mesures est on ne peut plus simple. Il suffit de se procurer d'une manière ou d'une autre le résultat final du développement (instructions assembleur), puis de se reporter à la documentation du processeur [Zilog81] pour connaître la durée d'exécution de chaque instruction (en nombre de cycles machine). Ce mode opératoire n'est praticable qu'à petite échelle, quand le nombre et la complexité des logiciels mis en œuvre sont faibles.

La construction modulaire du logiciel nous a conduit à effectuer ces calculs sur le même modèle (nous mesurons la durée de chaque procédure afin d'en déduire la durée des procédures appelantes). Trois étages peuvent alors être mis en évidence:

- Evaluation par procédure. Le chiffrage s'effectue à partir d'un des résultats de la compilation.

- Evaluation par primitive. Les deux primitives (WAIT et SEND) utilisées sont évaluées. Le résultat s'obtient en additionnant les procédures utilisées sur l'ensemble des plans de programmation (qui sont décrits ci-après). Il s'agit donc ici de détailler quelle sont les procédures appelées et combien de fois elles le sont. Notons que, pour le calcul de la primitive SEND, ce calcul est paramètré par le nombre de liens et par la taille du message.

- Evaluation par activité. Les quatres activités considérées sont les activités de chacun des modules PILE et ELEMENT pour deux cas (création ou non d'un nouvel élément). Là encore, on ajoute les durées des logiciels spécifiques à l'activité et des primitives utilisées.

CHIFFRAGES EFFECTUES

Nous donnerons trois sortes de résultats. Deux d'entre elles portent sur le niveau microscopique comme détaillé dans la section précédente et propose donc le chiffrage d'une activité. La dernière permet d'évaluer la durée d'exécution d'une opération d'empilage et permet de mettre en évidence l'influence du degré de granularité de l'application.

CHIFFRAGES PAR PLAN.

La description microscopique laisse apparaître quatre niveaux de fonctionnement: USAGER, NERF-Ca, NERF-Cs et EXO. Nous irons plus loin en proposant huit plans de programmation dont la superposition constitue l'ensemble du logiciel. En effet, un niveau de fonctionnement peut être lui-même décomposé en plusieurs plans de programmation : Le niveau USAGER comporte à la fois du source OMPHALE (ce qui écrit par le programmeur), du source ajouté par le préprocesseur et le source du module prédéfini OMPHALE. De la même manière, le routage des "SYSTEM CALL" constitue un source à part entière. Le découpage sera alors le suivant:

Niveau de Fonctionnement	Etage de Programmation		Langage
USAGER	A	Source OMPHALE	MODULA2
	B	Préprocesseur	MODULA2
	C	MODULE OMPHALE	MODULA2
NERF-Cs	D	Modula2-SC-Modula2	ASSEMBLEUR
	E	MODULES NERF-C	MODULA2
EXO	F	SC	ASSEMBLEUR
	G	EXO	ASSEMBLEUR
NERF-Ca	H	Processus NERF-C	MODULA2

Fig. IV-11. Niveaux de fonctionnement et plans de programmation

Notons que le plan D reprend la totalité du plan F en y ajoutant les manipulations nécessaires au bon fonctionnement de la pile.

CHIFFRAGE PAR CATEGORIE.

Nous avons distribué l'ensemble des instructions produites selon quatres catégories numérotées de I à IV:

- Les instructions de manipulation de pile (I). Le compilateur génère un nombre non négligeable d'instructions pour gérer la pile qui contient entre autres, les variables locales, les paramètres et les zones de display. C'est principalement dans cette catégorie d'instruction que nous avons regretté l'absence d'optimisation du code généré par le compilateur.

- Les instructions dites «efficaces» (II), qui correspondent au sens sémantique des instructions sources (par exemple, la mise à zéro d'une variable ou l'évaluation d'une expression).

- Les instructions d'accès (III). La décalque des structures de données de haut niveau (tableaux, structures) sur le support matériel d'une mémoire adressable implique le réalisation d'un certain nombre de

calculs (quand les modes d'adressage fournis par le processeur s'avèrent insuffisants). Nous avons comptabilisé à part le coût en temps de ces instructions.

- Les instructions de contrôle (IV). Sont les sauts (calculés, conditionnels ou non) ainsi que les appels de sous-programmes (CALL et SC).

La répartition des instructions selon ces catégories est bien sûr discutable. Ainsi, l'instruction d'empilage d'un paramètre utilisant un mode d'adressage complexe pourrait être classée indifféremment dans les catégories I et III. Mais le compilateur nous a beaucoup aidé en produisant un code très structuré (car peu optimisé). Par ailleurs, la distinction instructions d'accès/ instructions efficaces n'est absolument pas fondamentale car trop liée au niveau d'abstraction du langage (considérer la ligne modula2 suivante).

```
Tableau [2*I] := 0 ;
```

Cette décomposition permet néanmoins de mettre en évidence la proportion d'instructions "utiles" ou "parasites" d'un programme.

CHIFFRAGE PAR OPERATION.

Nous terminerons par une évaluation globale des éléments connus du système pour une série de 108 opérations d'empilage en fonction de la taille d'un élément. Ceci nous donnera une estimation du «coût» de la modularité.

IV.C.2. LES MESURES.

Hormis l'évaluation triviale de la durée des procédures (plan A). Il nous faut détailler l'étude des primitives WAIT et SEND.

ETUDE DES PRIMITIVES

La décomposition des deux primitives utilisées par les activités considérées est nécessaire au calcul des primitives. Il s'agit en l'occurence d'énumérer l'ensemble des sources mis en œuvre par ces primitives (par plan de programmation et, pour la primitive SEND, de factoriser ces décompositions.

ETUDE DU WAIT.

La décomposition du WAIT est complètement linéaire, au moins jusqu'à l'étage NERF-Cs: Un appel à cette primitive se traduira à chaque fois par l'appel à une primitive de l'étage inférieur. Cet enchaînement devient plus complexe au niveau EXO. En effet plusieurs appels EXO sont nécessaires pour réveiller le processus NERF-Ca, puis un autre processus USAGER. Nous réduirons la complexité de cette étude en ne donnant que des chiffres globaux par plan. Nous obtenons alors la figure suivante (les lettre entre parenthèses désignent les plans de programmation, les flèches indiquent les appels procéduraux):

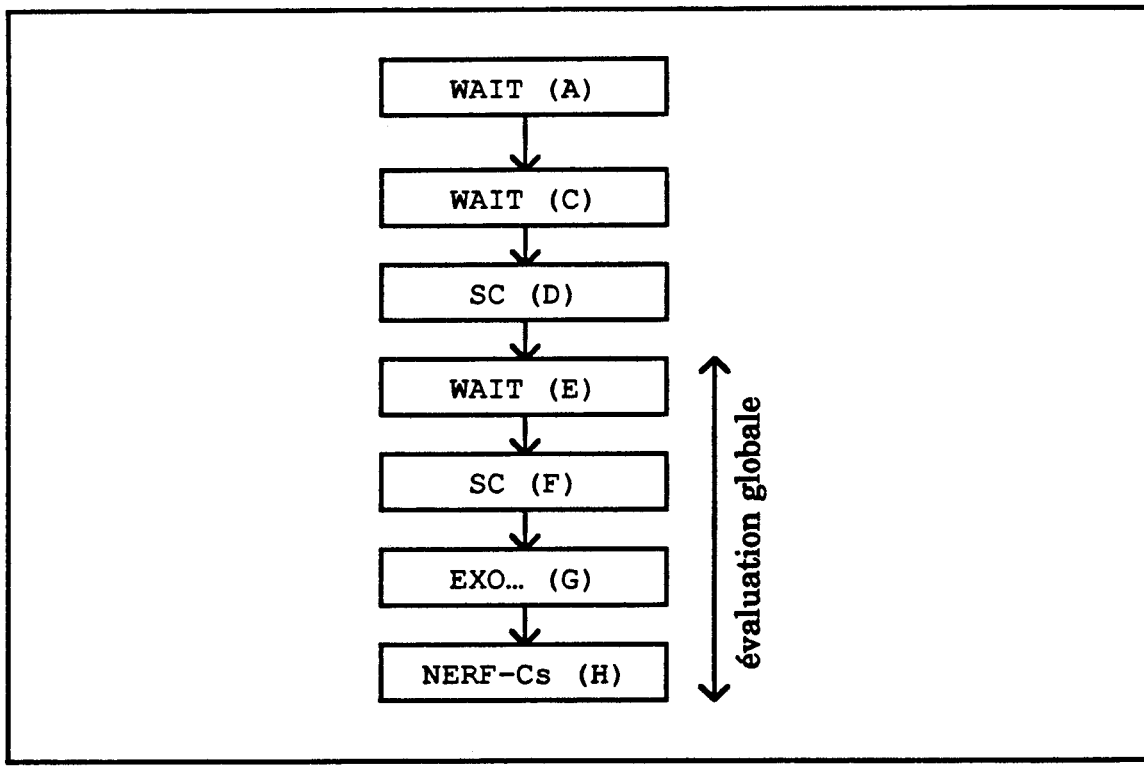


Fig. IV-12. Déroulement de la primitive WAIT.

ETUDE DU SEND.

Le déroulement de la primitive SEND est plus complexe[1]. Non seulement, un nombre d'étages plus important est mis jeu (ici le préprocesseur intervient), mais aussi l'accomplissement de cette primitive demande un nombre d'itérations dépendant de deux paramètres:

- **L**: Le nombre de liens inclus dans le format du message. Rappelons que les liens d'un message imposent un traitement spécifique (recherche du lien dans l'environnement et ajout au message).
- **M**: La taille du message.

Nous avons représenté dans la figure qui suit, la structure de ce déroulement. Certain nœuds sont pondérés car ils sont exécutés plusieurs fois.

Les éléments suivants sont représentés (avec mention de leur plan de programmation entre parenthèses):

- SEND (A) : Appel de la primitive générique d'envoi de message par l'utilisateur.
- FORMAT.SEND (B) : L'appel est remplacé par le préprocesseur, par un appel typé qui, en plus de l'envoi effectif, précise le format du message.
- OFFSET (C) : Cet appel sert à indiquer à NERF-C la position d'un nom local de lien dans le message.
- SC (D) : Couche de séparation mode usager– mode système utilise le SystemCall du Z8001.

---

[1]Nous parlons ici de la primitive SEND au niveau le plus haut de la programmation (langage OMPHALE). Cette primitive "générique" est transformée par le préprocesseur selon le format du message qui doit être transmis au noyau d'une manière ou d'une autre.

- OFFSET (E) : Mémorisation de la position d'un lien dans un tableau local.

- SYSTSEND (C) : Partie visible de l'envoi atypique de message. fait partie du module OMPHALE.

- SYSTSEND (E) : Implantation de SYSTSEND. Gére la zone d'émission et prend en compte le tableau mis à jour dans OFFSET pour traduire les nom locaux de lien.

- CONCAT (E) : Procédure utilitaire de concaténation sans interprétation de mots binaires dans la zone d'émission. Cette procédure est appelée successivement pour écrire
  - Le nombre de lien du message (2 octets).
  - L'identificateur de message (2 octets).
  - Chaque lien réel (10 octets par lien).
  - Le contenu du message (M octets).

- GETLINK (E) : Du fait de la séparation entre la gestion de la zone d'émission et l'environnement, SYSTSEND doit appeler la procédure GETLINK pour obtenir le contenu réel d'un lien à partir de son nom local.

- VALIDE (E) : Appelé par GETLINK. Détermine si un nom local de lien est valide ou non.

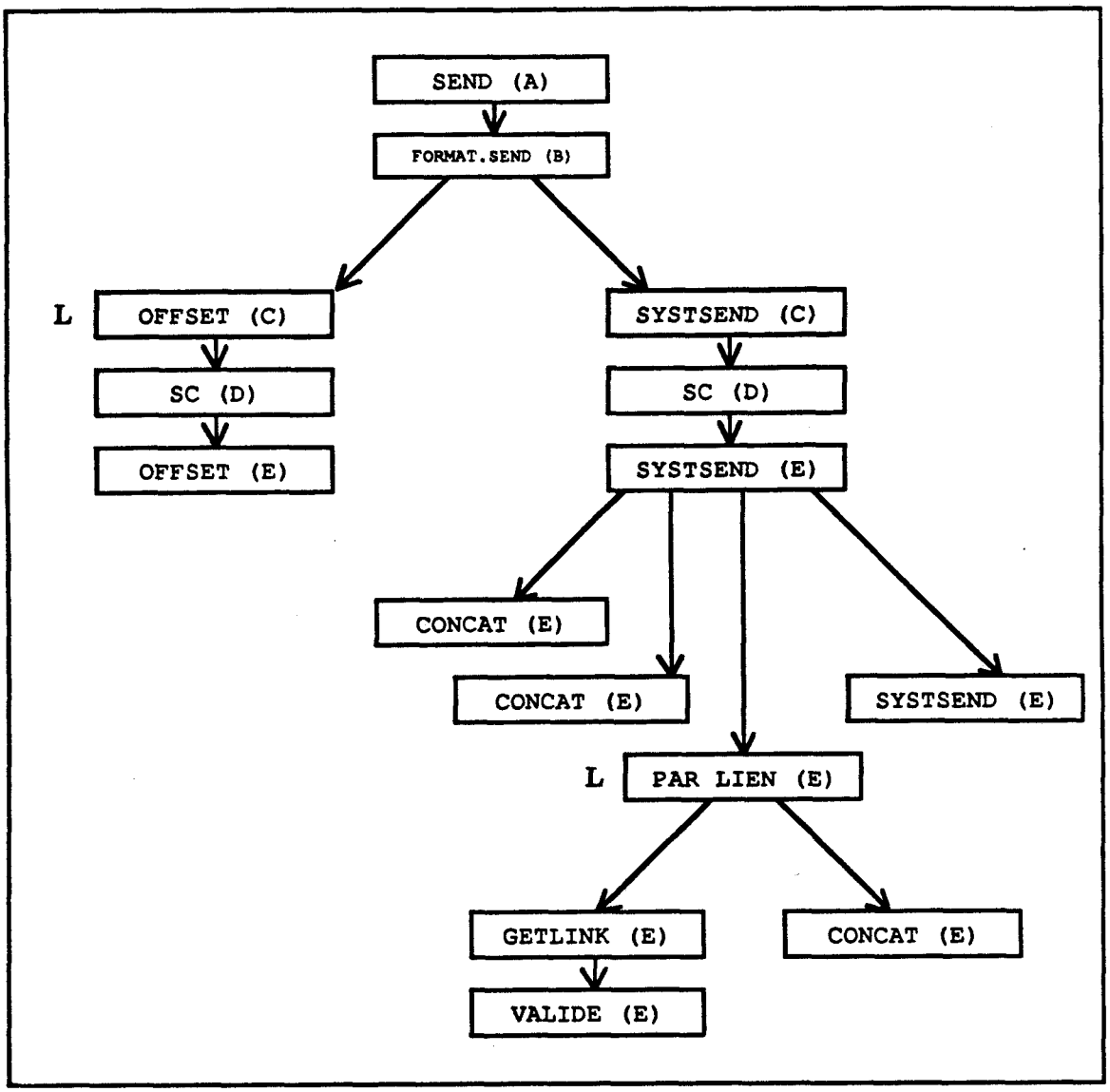


Fig. IV-13. Déroulement de la primitive SEND.

RESULTATS

Les résultats sont reportés dans les tableaux suivants. Toutes les mesures sont données en nombre de cycles. A chaque fois, les résultats se présentent sous la forme d'un polynôme à au plus trois coefficients (1, L : nombre de lien d'un message et M : taille d'un message). Ces tableaux présentent en une ou plusieurs lignes (une ligne par coefficient) chaque composant mesuré. Les cinq colonnes donnent la répartition par catégorie ainsi que le total. Pour certains composants, nous n'avons donné qu'un chiffre global.

MESURES PAR PLAN

LES CHIFFRES

Les durées d'exécution pour la primitive WAIT ainsi que la primitive SEND sont proposées par plan de programmation dans le tableau ci-dessous. Nous y avons aussi inclus l'application de la formule générale SEND (L, M) (L est le nombre de liens du message et M sa longueur en mots) aux différents formats utilisés par l'exemple. FX est le format de message supposé contenir un nom symbolique de module (pour la demande de création au gérante de module) sur 4 mots.

Plan	WAIT	SEND (L,M)					L=0 M=1 Fentier SEND	L=1 M=1 FLien SEND	L=0 M=4 FX SEND	LANG
A Source OMPHALE	92	203					203	203	203	M2
B Préprocesseur		332	+L*	108			332	440	332	M2
C Module OMPHALE	86	161	+L*	86			161	247	161	M2
D SC	266	266	+L*	266			266	532	266	ASS
E NERF-Cs	109	1426	+L*	2204	+M*	197	1623	3827	2214	M2
F SC	66						0	0	0	ASS
G EXO	4575						0	0	0	ASS
H NERF-Ca	535						0	0	0	M2
TOTAL	5729	2388	+L*	2664	+M*	197	2585	5249	3176	

Fig. IV-14. Chiffrage des primitives par plan de programmation.

ANALYSE

Les réflexions faites au vu de ces chiffres sont assez décevantes:

- La durée de la primitive WAIT est surtout passée à l'intérieur d'EXO (81%). Ce n'est pas tant l'importance d'EXO par rapport aux plans supérieurs (9,4% du temps total) qui est en cause mais plutôt par rapport au plan NERF-Ca (9,6 % du temps total). Les couches supérieures sont uniquement des couches de transit, alors que les fonctionnalités du WAIT sont essentiellement réalisées par les deux plans EXO et NERF-Ca. En revanche, l'importance d'EXO par rapport à NERF-Ca est surprenante. Ceci est expliqué par la conception d'EXO trop générale qui pénalise son fonctionnement. En d'autre termes, une optimisation d'EXO serait indispensable sur une version 'courante' du

site zéro. A titre de comparaison, une optimisation immédiate (par spécialisation/regroupement des appels à EXO) aurait permis de gagner 37 % du temps du plan EXO (26 % du temps total). A l'extrême, une version d'EXO qui ne prend en compte que deux processus fonctionnant en bascule demanderait 750 cycles au lieu de 4575 (soit une économie de 83 % sur EXO ou 68 % du temps total). Cette dernière ne pouvait pas être utilisée car elle ne gère pas les entrées-sorties de PC utilisées lors du développement.

- La primitive SEND ne fait appel ni à EXO, ni à NERF-Ca. On aurait donc pu donc espérer pour celle-ci des chiffres plus modérés. Là c'est l'écriture trop structurée de NERF-Cs et le manque d'optimisation du compilateur qui est en cause. Par exemple, la présence d'un lien supplémentaire dans le message ajoute un appel à OFFSET (E), GETLINK (E) et CONCAT (E) et tous les transits induits des plans supérieurs depuis le plan préprocesseur (B) jusqu'au plan NERF-Cs (E). Cette considération, jointe au chiffrage d'un appel de procédure MODULA2 (104 cycles sans compter les passages de paramètres) permet de mettre en évidence l'inefficacité de l'ensemble. Ces remarques portent surtout sur le plan E (NERF-Cs).

- Le coût d'envoi d'un lien supplémentaire dans le message (148 % de la durée d'un envoi vide) est également expliqué par la remarque précédente. C'est en effet le traitement des liens qui demandent le plus d'intermédiaires (par la nécessité d'aller rechercher le contenu du lien dans l'environnement à partir du nom local).

- Une version simplifiée[1] de SEND (E) a été écrite en assembleur. Son chiffrage a donné une valeur de `241 + 391 x L + 9 x M`. La même procédure écrite en modula2 demanderait :
`713 + 1201 x L + 208 x M` (toujours en nombre de cycles). soit un gain de 66 % sur un appel vide, de 67 % sur un lien et de 96 % par mot (taille du message).

[1] C'est-à-dire sans aller effectuer la traduction nom local vers contenu du lien.

MESURES PAR CATEGORIE

LES CHIFFRES

		I	II	III	IV	TOTAL	
WAIT (C)		34	0	0	52	86	WAIT - MODULE OMPHALE
WAIT (E)						109	SC WAIT et suite
F.SEND (B)	x1	253	0	46	33	332	Construit par préprocesseur:
	xL	88	0	0	20	108	envoi de message typé
OFFSET (C)		34	0	0	52	86	appelé par F.SEND (B)
OFFSET (E)		34	38	52	13	137	implantation de OFFSET (C)
SYSTSEND (C)		109	0	0	52	161	SEND - MODULE OMPHALE (atypique)
SYSTSEND (E)	x 1	435	208	89	33	765	implantation de SYSTSEND (C)
	x L	245	68	46	90	449	
CONCAT (E)	x1	25	21	19	24	89	appelé par
	xM	0	54	12	131	197	SYSTSEND (E)
GETLINK (E)		166	127	492	69	854	appelé par SYSTSEND (E)
VALIDE (E)		103	85	247	43	478	appelé par GETLINK (E)

Fig. IV-15. Mesure des procédures.

		I	II	III	IV	TOTAL
SEND(L,M)	x1	872	271	192	190	1791
	xL	695	393	868	442	2664
	xM	0	54	12	131	197
FEntier.SEND		872	325	204	321	1988
FLien.SEND		1567	718	1072	763	4652
FX.SEND		872	325	204	321	1988

Fig. IV-15. Mesure des primitives SEND.

		I	II	III	IV	TOTAL
PILE S.C.	P.A	161	115	0	96	372
	P.B	212	57	0	26	295
	P.C	362	564	437	386	1323
	P.D	34	0	0	52	195
	TOT	769	736	437	560	2185
PILE A.C.	P.A	161	115	0	96	372
	P.B'	471	153	12	66	702
	P.C	362	564	437	386	1323
	P.D	34	0	0	52	195
	P.E	362	726	473	779	1914
	P.F	34	0	0	52	195
	P.G	1057	957	1305	828	3987
	TOT	2481	2515	2227	2259	8688
ELEM 1er E	E.A	161	17	0	60	238
	E.B	34	0	0	52	195
	E.C	34	26	0	13	73
	E.D	144	87	0	76	307
	E.E	38	66	54	13	171
	E.F	34	0	0	52	195
	TOT	445	196	54	266	1179
ELEM	E.D	0	0	0	0	0
	E.E	38	66	54	13	171
	E.F	34	0	0	52	195
	TOT	72	66	54	65	257

Fig. IV-16. Mesure des activités par catégorie. Les notations M.X qui indiquent le découpage d'une activité du module M, sont précisées figures IV-17 et IV-18.

ANALYSE.

Les mesures par catégories (I: gestion de pile, II: instructions efficaces, III: calculs d'adresses, IV: structures de contrôle) n'ont été effectuées que sur les plans MODULA2: Le même calcul sur un source écrit en assembleur donne un taux presque nul pour la catégorie I et très réduit sur la catégorie III. En effet, l'utilisation optimisée des registres diminue radicalement le besoin d'une pile et l'adressage indirect permet de réduire les calculs d'adresse à une valeur raisonnable. De fait cette section teste la qualité du code généré par le compilateur.

Dès que l'on condidère le niveau global (Fig. IV-15), on constate un équilibre approximatif entre les quatres catégories. Une seule exception est à mentionner. Le coefficient M de l'envoi de message n'a aucune instruction en catégorie I (gestion de pile). Ceci s'explique par le fait que le nombre de plans traversés est indépendant de la taille du message.

Une indication supplémentaire peut cependant être obtenue: nous avons classé les instructions d'appel de routine (CALL) dans la catégorie IV

(structures de contrôle). On révèle alors les procédures qui servent à un changement de plan par le fait que le nombre d'instructions II (instructions efficaces) y est nul.

MESURES MACROSCOPIQUES (PAR ACTIVITE)

LES CHIFFRES

Les chiffres sont présentés dans les quatres tableaux ci-dessous. Ceux-ci permettent de chiffrer la durée de calcul d'un module dans un des deux cas d'opération (avec ou sans création de pile): ainsi le calcul effectué par le module `PILE` est plus long si il est nécessaire de créer un nouvel élément (activité `P.REPONSE` en plus de l'activité `P.EMPILER`). Il en est de même pour le module `ELEMENT` : il y a chaînage de l'ancien élément sommet par l'activité `E.Chainer` (en plus de `E.EMPILER`). Sur la figure IV-9, la différence entre les cas n'apparait que par la description de nouvelles activités (`P.Reponse` et `E.Chainer`). Par ailleurs, il nous faut rappeler que l'évaluation de l'activité C.Créer n'a pas été faite.

Chaque tableau posséde deux entrées:

- Les différents plans de programmation apparaissent horizontalement.
- Pour détailler les mesures des activités, nous avons découpé chaque activité en séquences. Chacune de ces séquences correspond à une verticale et est définie dans le paragraphe suivant.

DECOUPAGE DES ACTIVITES.

Module PILE.

En fonctionnement normal, le module `PILE` est dans une boucle infinie qui est constituée de "attente de message", "lecture du quai récepteur" puis aiguillage sur la procédure de traitement en fonction de la valeur lue. La durée d'exécution de cette boucle (hormis l'attente de message évaluée à part) qui est appelée "1 tour" est notée **P.A.**

La procédure `EMPILER` qui est appelée après la réception d'une demande d'empilement est notée **P.B**. Cette procédure doit elle-même envoyer un message par la procédure `Fentier.SEND` (notée **P.C**). Il ne

reste plus alors qu'à utiliser la primitive WAIT (appel noté **P.D**) dans la boucle infinie. Ces quatres temps forment l'activité P.Empiler.

Quand une création de module est nécessaire, Il faut chiffrer les actions supplémentaires:

- Envoi de la demande de création (**P.E**) au gérant de modules.
- Attente de la réponse (**P.F**).
- Envoi du lien vers l'ancien sommet vers le nouvel élément (**P.G**)
  Ces trois actions constituent l'activité `P.Reponse`.

Module ELEMENT.

Là aussi, nous avons évalué une boucle infinie (**E.D**) qui aiguille lors d'un empilement vers une procédure Empiler (**E.E**) et qui, une fois cette procédure achevée, attend un autre message (**E.F**). Les temps E.D, E.E et E.F forment l'activité `E.Empiler`.

La création d'un module est également plus coûteuse pour ce module: outre la phase d'initialisation (**E.A**) et l'établissement du chaînage (**E.C**), il y a l'appel de la primitive WAIT (**E.B**). Les temps E.A, E.B et E.C constituent l'activité `E.Chainer`.

Plan		PILE (sans création) P.A	P.B	P.C	P.D	TOTAL	LANG
A	Source OMPHALE	372	295			667	M2
B	Préprocesseur			332		332	M2
C	Module OMPHALE			161	86	247	M2
D	SC			266	266	532	ASS
E	NERF-Cs			1229	109	1338	M2
F	SC				66	66	ASS
G	EXO				4575	4575	ASS
H	NERF-Ca				535	535	M2
TOTAL		372	295	1988	5637	8292	

P.A	1 tour	
P.B	ProcEmpiler	
P.C	Fentier.SEND	
P.D	WAIT	
P.B'	ProcEmpiler	
P.E	FX.Send	
P.F	WAIT	
P.G	FLien.SEND	

		PILE (avec création) P.A	P.B'	P.C	P.D	P.E	P.F	P.G	TOTAL	LANG
A	Source OMPHALE	372	702						1074	M2
B	Préprocesseur			332		332		440	1104	M2
C	Module OMPHALE			161	86	161	86	247	741	M2
D	SC			266	266	266	266	532	1596	ASS
E	NERF-Cs			1229	109	1820	109	3433	6700	M2
F	SC				66		66		132	ASS
G	EXO				4575		4575		9150	ASS
H	NERF-Ca				535		535		1070	M2
TOTAL		372	702	1988	5637	2579	5637	4652	21567	

Fig. IV-17. Mesure des activités de PILE par plan.

		ELEM (Premier élément) E.A	E.B	E.C	E.D	E.E	E.F	TOTAL	LANG
A	Source OMPHALE	238		73	581	171		1063	M2
B	Préprocesseur							0	M2
C	Module OMPHALE		86				86	172	M2
D	SC		266				266	532	ASS
E	NERF-Cs		109				109	218	M2
F	SC		66				66	132	ASS
G	EXO		4575				4575	9150	ASS
H	NERF-Ca		535				535	1070	M2
TOTAL		238	5637	73	581	171	5637	12337	

		ELEM E.D	E.E	E.F	TOTAL	LANG
A	Source OMPHALE	581	171		752	M2
B	Préprocesseur					M2
C	Module OMPHALE			86	86	M2
D	SC			266	266	ASS
E	NERF-Cs			109	109	M2
F	SC			66	66	ASS
G	EXO			4575	4575	ASS
H	NERF-Ca			535	535	M2
TOTAL		581	171	5637	6389	

E.A	Initialisation
E.B	WAIT
E.C	Chainer
E.D	1 tour
E.E	Empiler
E.F	WAIT

Fig. IV-17. Mesure des activités de ELEMENT par plan.

ANALYSE

L'analyse des mesures macroscopiques (par activité) découle logiquement de ce qui précède (section mesures par plan): une grande partie des durées d'exécution sont consommées par les procédures des plans 'systèmes' que sont NERF-Cs et EXO (généralement 70 % pour l'ensemble des deux plans). Les durées d'exécution par activité varient entre 6000 et 14000 cycles:

P.Empiler (A.C.)	9388
P.Empiler (S.C.)	8981
P.Reponse	14154
E.Empiler	6481
E.Chainer	6040

Fig. IV-18. Temps d'exécution par activités.

MESURES MACROSCOPIQUES (SUR 108 OPERATIONS).

LES CHIFFRES

	OPER. A.C.	OPER. S.C.	DELTA	DELTA PRCT
A Source OMPHALE	2137	1419	718	51%
B Préprocesseur	1104	332	772	233%
C Module OMPHALE	913	333	580	174%
D SC	2128	798	1330	167%
E NERF-Cs	6918	1447	5471	378%
F SC	264	132	132	100%
G EXO	18300	9150	9150	100%
H NERF-Ca	2140	1070	1070	100%
TOTAL	33904	14681	19223	131%

Fig. IV-19. Mesures globales des activités des modules PILE et ELEMENT avec ou sans création d'élément supplémentaire (en milliers de cycles). DELTA donne la différence entre les deux premières colonnes; DELTA PRCT donne le ratio $\frac{\text{DELTA}}{\text{OPER A.C.}}$.

	TAILLE	1	2	3	4	5	6	9	54	108
	CREATIONS	108	54	36	27	21,6	18	12	2	1
A	Source OMPHALE	231	192	179	173	169	166	162	155	154
B	Préprocesseur	119	78	64	57	53	50	45	37	37
C	Module OMPHALE	99	67	57	52	48	46	43	37	37
D	SC	230	158	134	122	115	110	102	89	88
E	NERF-Cs	747	452	353	304	274	255	222	167	162
F	SC	29	21	19	18	17	17	16	15	14
G	EXO	1976	1482	1318	1235	1186	1153	1098	1007	997
H	NERF-Ca	231	173	154	144	139	135	128	118	117
TOTAL		3662	2624	2278	2105	2001	1932	1816	1624	1605
RAPPORT		100%	72%	62%	57%	55%	53%	50%	44%	44%

Fig. IV-20. Mesures pour 108 opérations en fonction de la taille d'un élément (les valeurs sont données en milliers de cycles).

Le tableau de la figure IV-20 a été calculé en considérant les durées cumulées des activités des modules PILE et ELEMENT avec ou sans création d'élément nouveau montrées figure IV-19 (notées respectivement OPER A.C. et OPER S.C.). Pour 108 opérations d'empilage et pour une taille de module T, il y a $\frac{108}{T}$ opérations avec création d'élément et 108 - $\frac{108}{T}$ opérations sans création d'élément. La combinaison de ces deux durées nous donne une fonction linéaire entre la durée globale (pour les 108 opérations) et le nombre de créations. la pente (strictement positive) de cette fonction est déduite, pour chaque plan, de la différence:

```
DELTA = OPER A.C. - OPER S.C.
```

ANALYSE

On constate que, sur les activités étudiées[1], le coût supplémentaire dû à un plus grand nombre de créations est surtout présent dans les plans de programmation de bas niveau (Fig. IV-19) : Non seulement ces plans sont les plus gourmants et donc toute augmentation relative porte plus – en valeur absolue – sur ces niveaux (par exemple le plan EXO passe de 9150 à 18300), mais aussi le travail supplémentaire consiste principalement en envois de message et attentes de réponse. C'est ainsi que, alors qu'on observe une augmentation relative de 131% sur la globalité de l'opération,

[1]L'activité de création propre au gérant de module n'a pas été considérée.

on peut mettre en évidence une augmentation de 378% dans le plan NERF-Cs et de 100% au niveau EXO..

Le choix de l'exemple doit être justifié. En effet, la programmation des module de l'application "pile d'entiers" dégénère assez vite en transmission de message, sans calculs ni accès à des structures de données. Il semble, à première vue que cette caractéristique est très courante en programmation objet, tout au moins pour les applications où cette méthodologie est adaptée. L'accent est alors mis sur les déficiences du système d'exploitation (coût d'un changement de contexte, inefficacité d'EXO). L'intérêt essentiel de cet exemple est surtout sa simplivité et sa variabilité quelque soit son réalisme. Il nous a manqué ici des mesures (au sens du génie logiciel) sur les programmes orientés objet.

Néanmoins, malgré une conception maladroite du système d'exploitation, la durée globale décroît assez rapidement pour arriver à la moitié de la valeur initiale pour une taille d'élément de 9 entiers (c'est une hyperbole). En d'autre termes, cela signifie que si l'architecture offre un doublement de la puissance de calcul, elle permet une granularité 9 fois plus forte dans l'écriture des modules.

IV.C.3. ESTIMATION DES TEMPS D'EXECUTION OPTIMISES

Les mesures établies précédemment pour rigoureuses qu'elles soient, n'en sont pas moins largement entachées par les défauts de conception et les imperfections de la chaîne de développement. Il semble alors intéressant d'estimer ce que donnerait une bonne optimisation du logiciel système du coté PN et du coté PC.

Nous estimerons les temps d'exécution liés à chaque période d'activité en fonction de trois paramètres:

- X: Le nombre de messages envoyés durant la période d'activité.
- L: Le nombre total de liens envoyés dans l'ensemble des messages.
- M: La taille cumulée de l'ensemble des messages (en octets)

COTE PC

Comme il a déjà été dit, les logiciels système du coté PC peut être optimisé dans les plans NERF-Cet EXO .

il est possible de réduire la durée d'exécution de NERF-C en utilisant moins les outils de structuration du langage MODULA2 qui sont implantés inefficacement par le compilateur. Ceci avait partiellement été fait en déclarant globalement des variables qui pouvaient être locales à une procédure. Un pas de plus dans cette direction est possible en développant certaines procédures en ligne (pour éviter de consommer du temps en passages de paramètres et gestion de pile).

Nous pouvons également gagner du temps en privilégiant le transfert d'informations entre le plan utilisateur et le plan NERF-C au moyen de structures de données par opposition à l'utilisation des paramètres: Par exemple la technique qui consiste à appeler une procédure OFFSET pour chaque lien présent dans le message afin d'indiquer sa position est malvenue. Il eut mieux fallu transmettre un tableau qui contient la totalité de ces informations au moment de l'envoi effectif. L'ensemble de l'envoi utilise alors un seul appel système. On peut ainsi économiser L appels de procédure (du plan B au plan E), soit L * 3 changements de plan.

EXO peut lui aussi être optimisé en considérant que sa seule fonction sur PC est de commuter deux processus seulement. Cette hypothèse a été chiffrée et fourni un résultat de 750 cycles dans le plan E pour la primitive WAIT.

En suivant ces règles, nous estimons la durée d'exécution de la partie système d'une période d'activité à

$$1800 + 2200 * X + 500 * L + 20 * M$$

Les durées d'activité "utilisateur" sont évidemment dépendantes de l'application mais restent de l'ordre de quelques centaines de cycles.

COTE PN

Des considérations sur le fonctionnement de NERF-N et la comparaison avec ce que nous avons évalué de NERF-C vont nous permettre d'apprécier

les temps d'exécution de NERF-N. Nous allons d'abord décrire la structure de données de NERF-N. Puis énumérer l'ensemble des actions qui doivent être réalisées par PN au début et à la fin d'une période d'activité d'un module. Ceci nous permettra d'estimer le temps d'exécution de NERF-N.

STRUCTURE DE DONNEES.

Les structures de données de NERF-N sont essentiellement consituées de files d'attente de module, d'une zone de description des modules, pour chaque quai d'un module, d'une file d'attente des messages. Pour chaque file, il y aura un pointeur vers la tête de file, un pointeur vers la fin de file, et pour les files d'attente de message, un compteur de messages. La zone de réception où doit se trouver l'ensemble des messages en attente de traitement est gérée en blocs de taille fixe.

TRAITEMENTS REALISES PAR PN.

Les traitements réalisés par PN sont éclatés en deux parties: les traitements en fin d'exécution d'un module et les traitements au réveil d'un module. Les traitements en fin d'exécution sont eux-même classés selon deux groupes: le traitement de la zone d'émission et le traitement de la zone de communication.

Le traitement de la zone d'émission consiste en :

- Recherche du module courant en tête de FAX (file d'attente d'exécution).
- Calcul de l'adresse de la zone de description de module.
- Recherche de l'adresse de la zone d'émission.
- Allocation tampon pour le message.
- Recopie du message.
- Recherche du numéro du quai destinataire.
- Calcul de l'adresse du message suivant.
- Mise en File d'attente du message envoyé.
- Mise a jour du compteur de message de la file.
- Test si quai ouvert et compteur non nul.

pour chaque message

- Eventuellement Eligibilité du module:
  - Inscription `ReceivingService`.
  - Mise en F.A.P. (File d'attente de préparation)

Le traitement de la zone de communication se décompose de la manière suivante:

- Recopie de l'état des quais (Zone de Communication vers la zone de description du module).
- Eclatement de l'état des quais (Les bits de chaque service sont éclatés vers le compteur correspondant).
- A chaque ouverture de quai, contrôle de l'éligibilité du module.

Au réveil d'un module, NERF-N doit effectuer les actions suivantes:

- Retrait du module de la F.A.P. (file d'attente de préparation)
- Récupération du n° de quai récepteur.
- Recherche du message reçu.
- Pour chaque lien du message:
  - Inscription dans l'environnement .
  - Calcul du nom local de lien
  - Remplacement du nom local de lien dans le message
- Recopie du message dans la zone de communication.
- Libération du tampon.
- Calcul de la segmentation du module.
- Mise du module en en F.A.C. (file d'attente de chargement)

Ces opérations sont, quand une U.G.M. est libre,suivies du chargement:

- Retrait F.A.C.
- Chargement U.G.M.
- Mise en F.A.X.

DUREE D'EXECUTION DE NERF-N.

Du point de vue global, chaque message donne lieu à une période d'activité pour un module. Nous pouvons donc, pour le post-traitement d'un module, considérer qu'il y a un réveil de module par message. En

revanche, il faut multiplier la durée de pré-traitement par le nombre de message émis durant la phase d'activité.

Une évaluation grossière de l'ensemble des opérations décrites nous donne alors une charge de $500 + 1000 * X + 500 * L + 40 * M$ cycles par période d'activité.

ANALYSE

X=	L=	M=		TOTAL	
0	0	0	0	500	Service sans réponse
			0	1800	
1	1	10	400	2400	Question; Réponse par 1 lien
			200	4500	
1	5	50	2000	6000	
			1000	6500	
5	5	50	2000	10000	Envoi parallèle d'un lien/ 5 modules
			1000	15300	
1	10	100	4000	10500	
			2000	9000	

Paramètres pour une période d'activité
X = Nombre de messages émis
L= Nombre cumulé de liens émis
M = Taille cumulée des messages émis (en mots)

Fig. IV-21. Durées d'exécution comparées pour quelques cas typiques.

En résumé, on peut estimer les durées d'exécution par période d'activité du module pour quelques cas caractéristiques (dans un contexte de programmation orientée objet).Il reste à inclure dans l'estimation de PN les durées des tâches que seul ce dernier est autorisé à effectuer comme par exemple les recopies dans ou vers un bloc mémoire soit pour gérer cet espace, soit pour créer les images mémoire des modules. Somme toute, l'équilibre de charge entre les deux processeurs est suffisant pour justifier l'existence du processeur PN. L'adjonction d'un deuxième processeur de calcul est envisageable mais est à mettre en rapport avec l'accroissement de complexité qu'elle apporte (chemin de données

supplémentaire pour les U.G.M. et ajout de commutateurs à chacun des blocs mémoires).

IV.D. CONCLUSION.

Les études menées dans ce chapitre ont abouti à des conclusions très variées, à la fois sur les outils employés et sur les résultats obtenus.

LES OUTILS ET METHODES EMPLOYES.

Il est courant d'évaluer, après-coup les qualités et les défauts du travail que l'on vient de terminer. Cette démarche est ici d'autant plus justifiée qu'elle conduit à une analyse critique des résultats obtenus. Les imperfections mise en évidence sont nombreuses:

- L'utilisation d'un compilateur non optimisé n'a pu être que partiellement contrée par une programmation appropriée.
- La multiplicité des couches logicielles augmente fortement les durées d'exécution. Cette remarque est valable aussi bien pour EXO que pour NERF-C.
- La technique employée pour le contrôle de type des messages est peu judicieuse car elle est procédurale (il y a un appel de procédure par lien dans le message et par envoi). Elle amplifie ainsi les effets de la remarque précédente.
- L'utilisation de deux langages différents pour la programmation du système diminue les performances du logiciel par le simple fait de devoir assurer la compatibilité entre ces couches.

A notre décharge, il faut dire que nous avons complètement constitué la chaîne de développement à partir des quelques éléments disparates qui nous étaient accessibles. La rapidité de mise en œuvre a souvent primé sur la qualité du résultat obtenu.

Le principal problème réside finalement dans le compromis qu'il faut trouver entre des démarches générales faciles à mettre en œuvre mais peu optimisables et des développements spécifiques qui apportent une efficacité maximale au prix d'un temps de développement long. Le

contexte de travail habituel des laboratoires permet rarement de préférer la seconde solution, la rentabilité des longs développement ne pouvant être assurée qu'en contexte industriel. Les travaux d'évaluation en sont d'autant plus difficiles.

LES RESULTATS OBTENUS.

L'implantation d'OMPHALE sur le site zéro a tout d'abord permis de montrer le sur-dimensionnement des unités de gestion mémoire. Dans le contexte étudié, leur seul rôle est d'assurer la protection de la mémoire par rapport à un processus unique. Un faible nombre de segments est alors suffisant. Par opposition, de nombreux segments sont utiles lors qu'il faut permettre le partage de données entre plusieurs processus.

Les mesures effectuées auraient pu être largement hypothéquées par les réflexions faites quant-aux outils et méthodes employés. L'expérience acquise lors de ces mesures d'une part, la comparaison avec une autre implantation d'OMPHALE d'autre-part, nous ont toutefois permis d'estimer les temps d'exécutions après optimisation. Ces estimations portent sur un exemple dont la granularité est de l'ordre de grandeur de celle des langages orientés objets: seul les données élémentaires sont traitées par le matériel; les données plus complexes sont encapsulées dans des modules qui communiquent par messages.

Malgré ces réserves, les valeurs calculées sont encourageantes: les temps des traitements attribués à chaque processeur sont approximativement équivalents. Un léger déséquilibre subsiste (PC est quelque peu plus chargé que PN) et pourrait inciter à l'installation d'un deuxième processeur de calcul. Cette dissymétrie ne remet nullement en cause l'organisation matérielle du site zéro.

CONCLUSIONS

Après une brève description de l'architecture omphale, nous avons détaillé le fonctionnement d'une organisation matérielle destinée à l'exécution d'un système constitué de nombreux modules communicants par messages. Un prototype (site zéro) a été réalisé et a permis la validation de l'architecture.

RESULTATS OBTENUS

PAR LA SIMULATION

En première approche, une simulation nous a permis d'affiner certains paramètres et de connaître les caractéristiques que doivent présenter le logiciel pour une exécution efficace sur le site zéro. De manière annexe, cette simulation nous a également permis de soulever, puis de résoudre quelques questions quant à l'écriture du logiciel système.

Pour ce qui est des paramètres matériels, la conclusion est qu'en fonctionnement général et en choisissant la meilleure stratégie (en ce qui concerne le travail de PN), il n'est pas utile de disposer de plus de deux U.G.M. et de plus de six blocs mémoires. Dans tous les cas de figure, la rentabilité maximale du matériel est atteinte quand les charges des deux processeurs PN et PC sont équilibrées. Dans les meilleures conditions, on peut espérer doubler les performance d'un monoprocesseur. Nous estimons que l'amélioration obtenue dans les conditions ordinaires, est de l'ordre de 1,5 fois les performances d'un monoprocesseur.

Nous avons observé également que les meilleures stratégies à appliquer par PN sont celles qui défavorisent la préparation des modules en faveur des opérations de chargement ou de sauvegarde des UGM. L'application de la stratégie INT qui demande plus de matériel, apporte un peu plus de stabilité dans les performances (par rapport aux caractéristiques de la mémoire) et très peu d'efficacité supplémentaire.

PAR LES MESURES

En seconde approche, nous avons écrit une partie du noyau omphale et conçu un environnement de programmation omphale pour le site zéro. Les mesures effectuées, bien que partielles, ont démontré l'intérêt que présente l'organisation matérielle par rapport à l'architecture omphale en particulier, aux systèmes à communication par message en général.

La première conclusion qui s'impose des mesures effectuées est une réflexion critique sur l'ensemble des méthodes et des outils utilisés pour l'implantation d'OMPHALE sur le site zéro.

Une fois ce stade dépassé, nous avons estimé les temps d'exécution au mieux à partir des temps connus. Le tout dans un contexte de langage orienté objet (peu d'émission de messages et de liens par phase d'activité d'un module et faible durée du calcul utilisateur). On arrive à un rapport de charge entre PN et PC qui varie entre 1 et 3 (PC peut être 3 fois plus chargé que PN).

CONFRONTATION.

Les résultats obtenus par la simulation d'une part, par les mesures d'autre-part ne coïncident pas exactement. Dans le cas le plus défavorable, le gain de temps apporté par l'organisation matérielle serait de 75 % (1,3 les performances d'un monoprocesseur). Cependant, un résultat important est que les charges des deux processeurs sont du même ordre de grandeur.

Un réajustement peut être effectué de deux manières:

Un rééquilibrage des charges dans l'implantation du systéme d'exploitation (migrations de fonctions systèmes locales de PC vers PN) est problématique: ce changement n'est opportun que si la durée d'exécution sur PC de l'appel est supérieure à celle du changement de contexte. Nous n'avons pas, dans l'implantation d'OMPHALE, d'exemple de telles fonctions.

L'ajout d'un deuxième processeur PC avec un autre bus BC et d'autres blocs mémoire semble la meilleure solution. C'est également la plus

couteuse en matériel et elle soulève quelques problèmes dans la répartition des modules entre les processeurs et la configuration de la mémoire de blocs.

REFLEXION SUR LES METHODES UTILISEES

Les méthodes envisageables pour l'évaluation d'une architecture sont complémentaires.

RESOLUTION ANALYTIQUE

La résolution analytique d'un système demande sa modélisation sous forme de réseaux de files d'attente. Les caractéristiques numériques (temps de services) doivent être estimées. Les résultats obtenus ont alors toute la généralité désirée. Cette méthode devient cependant inapplicable à mesure que le modèle obtenu se complexifie (en particulier si il présente des aspects algorithmiques).

LA SIMULATION

La simulation supporte des modèles plus complexes, ce qui permet de tester très rapidement un plus grand nombre d'hypothèses (par l'établissement de modèles plus généraux). La difficulté est, à ce niveau, d'interpréter et d'ordonner l'ensemble des résultats obtenus. C'est à cette difficulté que la méthode doit sa limite. Notons que cette méthode demande aussi une évaluation statistique des caractéristiques des objets testés.

La simulation nous a donné rapidement des résultats globaux sur le comportement de l'ensemble matériel/logiciel du site zéro. Elle a permis de préciser la configuration optimale de l'ensemble, à partir d'un minimum d'hypothèses. Cependant, les résultats obtenus restent globaux: Il est inintéressant d'évaluer par cette méthode un exemple réel.

IMPLANTATION ET MESURES

La dernière méthode utilisée correspond plus à une démarche ascendante: à partir du matériel, implanter le système, écrire une application puis mesurer les durées d'exécution du code obtenu. Ceci a

l'avantage du réalisme: tout programmeur connait la différence qui existe entre un logiciel prévu et sa réalisation. Les chiffres ainsi obtenus ne sont évidemment pas contestables. Ce réalisme apporte d'autres inconvénients: Il s'agit de l'étude d'un cas particulier (l'implantation et l'application) que l'on espére représentatif des mises en œuvre réelles.

Les précautions qui ont étés prises lors du dernier chapitre montrent bien les défauts de cette méthode. Nous avons dû estimer les gains apportés par une bonne optimisation du logiciel et nous avons justifié le choix de l'application. Les résultats obtenus sont liés à l'implantation d'une architecture matérielle et à une application. Cette méthode permet quand même de se faire une idée assez précise des caractéristiques du logiciel.

CONCLUSION.

Globalement la démarche peut être résumé par les points suivants: A partir du choix d'une architecture logicielle et d'une application, nous avons pu évaluer par la mesure certaines caractéristiques logicielles. Ces caractéristiques sont reprises avec le modèle matériel pour chiffrer l'augmentation des performances dues à l'architecture matérielle.

REFLEXIONS SUR L'ARCHITECTURE MATERIELLE

Nous allons donner ici une description plus fonctionnelle de l'architecture matérielle. Ce raisonnement en termes de fonctions plus qu'en termes de matériels nous permettra de généraliser l'architecture étudiée et de considérer ses limites.

PRE-REQUIS SUR LE LOGICIEL.

Il est bien entendu que l'on utilise, lors de la conception de l'organisation matérielle, une caractéristique particulière de l'architecture OMPHALE. En effet, l'ensemble du travail logiciel peut être découpé en deux parties indépendantes (noyau et utilisateur). Cette propriété est loin d'être universelle.

DECOUPAGE FONCTIONNEL DU SITE.

INVENTAIRE DES FONCTIONS

Nous avons découpé l'ensemble des travaux du système en trois fonctions essentielles:

- Exécution des programmes "utilisateur" dans un contexte restreint. Cette fonction sera assurée par ce que nous appellerons des "processeurs d'exécution". Le résultat de cette exécution est de deux formes:
  - Modification du contexte local.
  - Envoi d'un certain nombre de messages.
- Réception et routage des messages. Un message à destination d'un module peut modifier l'état de ce module (i.e. le rendre prêt à être exécuté). Cette tâche sera assurée par un "gérant de message".
- Gestion des contextes. L'association contexte-processeur d'exécution peut être figée (processeur spécialisé) ou variable (processeurs banalisés). Dans ce dernier cas, les considérations de protection d'une part, la nécessité d'assurer la connexion physique entre les processeurs d'exécutions d'autre-part, imposent d'établir un dispositif matériel qui assure le changement de contexte.

SCHEMA SYNOPTIQUE.

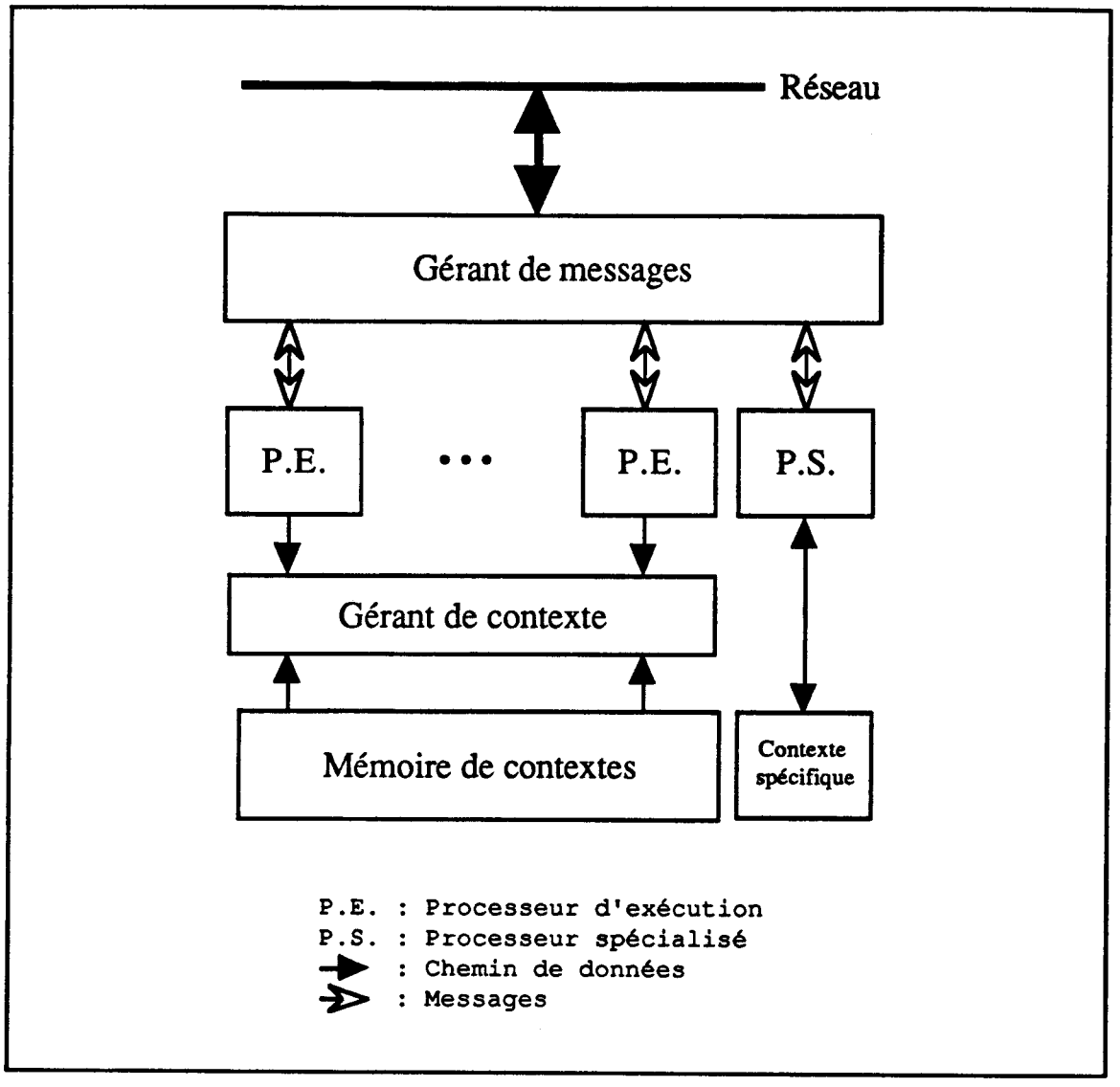


Fig. V-1. Schéma synoptique.

Le gérant de message reçoit les messages résultant de l'exécution d'un module. Il les stocke dans sa mémoire locale. Un message émis peut rendre exécutable un module. Dans ce cas, il fournit simultanément à l'un des processeurs susceptible d'exécuter ce module le contenu du message et un désignateur de module (donc de contexte). Quand le processeur est disponible, il transmet l'identificateur de contexte au gérant de contexte qui configure le matériel pour établir la liaison entre le processeur d'exécution et la zone correspondante de la mémoire de

contextes. On peut aussi envisager des processeurs d'exécution spécialisés (par exemple liés à des dispositifs d'entrée-sortie éventuellement avec une mémoire de contexte propre).

REALISATION MATERIELLE.

- La réalisation du gérant de messages ne pose pas de problème fondamental. La simplicité des fonctions à réaliser laisse présager une intégration matérielle facile.

Pour la réalisation du gérant de contexte, deux problèmes se posent:

- Obtenir un mécanisme de traduction d'adresse qui permette à un processeur d'accéder à un élément du contexte à l'aide de l'identificateur de contexte. Ce mécanisme peut être couplé à un dispositif de protection. On utilisera alors une unité de gestion mémoire.
  Il n'est pas utile de disposer d'un grand nombre de segments. En fait, il est tout à fait possible de reprendre un mécanisme comme celui du SM90 où le numéro de segment est accolé à un identificateur de processus.

- Concevoir une mémoire permettant un accès simultané à des contextes différents depuis chacun des processeurs. Nous envisageons trois possibilités.

  - Mémoire commune et arbitrage des conflits d'accès. Cette solution introduit un goulot d'étranglement pour les performances au niveau de l'accès à la mémoire qui a été bien étudié dans le cas des multiprocesseurs..

  - Mémoire locale double accès. Une mémoire double accès locale à chaque processeur peut être chargée et déchargée par un processeur spécialisé qui assure le va-et-vient des modules vers une zone de va-et-vient commune.

  - Mémoire de blocs. C'est, en terme de visibilité de la mémoire depuis les processeurs d'exécution, une solution intermédiaire entre les deux précédentes: la mémoire est découpée en blocs.

Les connexions entre processeurs d'exécution et blocs mémoire est faite dynamiquement. Cette technique permet d'éviter à la fois les transferts mémoire et le goulot d'étranglement d'une mémoire commune. La complexité matérielle (un commutateur par processeur d'exécution) est plus grande dans cette solution.

On peut décrire le site zéro par rapport à la classification précédente:

- La présence d'un seul processeur d'exécution disposant d'un adressage protégé constitue la base même de l'architecture matérielle.
- Le gérant de messages est entièrement émulé par NERF-N exécuté par le processeur PN.
- La gestion des contextes est assurée par l'ensemble composé du logiciel NERF-N (qui a donc un double rôle) et des matériels que sont la circuiterie de partage et les UGM. L'unicité du processeur d'exécution permet d'emploiyer la dernière technique (mémoire de bloc). Cette combinaison de matériels et son utilisation pour lier dynamiquement un bloc mémoire à PC constitue l'originalité de cette machine.

La présente étude valide cette organisation matérielle en termes de performances.

ANNEXE A.
PREPROCESSEUR OMPHALE.

Avant-propos: Pour la concision de l'exposé, nous appellerons *Omodule* un module OMPHALE et *Mmodule* un module modula2.

mise en oeuvre du preprocesseur

B.1. Commande

Le préprocesseur s'appelle par la commande ***ppo***. qui admet un seul argument qui est le nom du fichier qui contient le source OMPHALE (ordinairement avec l'extension .omp).

B.2. Fichiers engendres

Deux cas peuvent se présenter:

Le source est une description de format de message.

Le préprocesseur génère une paire de fichiers correspondant aux parties définition et implémentation du Mmodule résultat. Le nom de ce Mmodule est le nom même du format.

Exemple: Le format FRM donnera les fichier FRMdef.def et FRM.mod ; deux fichiers source du Mmodule FRM.

Le source est une description de Omodule XXX.

Le préprocesseur donnera deux paires deux fichiers:

la définition et l'implémentation du Mmodule de nom XXX. Ce Mmodule est le segment 1 sur le site zéro. Ces fichiers sources ont pour nom XXX1def.def et XXX1.mod.

La définition et l'implémentation du Mmodule de nom XXX4 pour le segment 4 du site zéro. Les fichiers sources ont pour nom XXX4def.def et XXX4.mod.

B.3. Ordre de compilation

L'ordre de compilation des différents modules crées par le préprocesseur est déduit des regels suivantes:

Tous les modules crées utilisent le module OMPHALE. Il faut donc avant toute chose compiler OMPHALEdef.def.

Les modules résultant de formats n'utilisent que OMPHALEdef.def

Les modules résultants de modules OMPHALE sont à compiler dans l'ordre segment4 puis segment1.

Lorsqu'un format ou un Omodule importe explicitement des objets d'un Mmodule écrit par ailleurs ou lorsqu'un Omodule déclare utiliser soit un format, soit un autre Omodule, le Mmodule fournisseur doit être compile avant le Mmodule client. Dans le dernier cas, remarquons que seul le segment1 d'un Omodule exporte des objets (noms de quais, etc...).

B.4. D.Exemple de session.

Ce chapitre donne un exemple de session sans erreur dans l'environnement 'classique' modula2 sur multics auquel on a rajouté un répertoire **omp** qui contiendra les sources omphale. On suppose également que l'on a à disposition les abréviations pour compiler les modules de définition (**MCD**) et d'implémentation (**MCI**).

A. Compilation d'un format

Soit le format FORMEX décrit par le source formex.omp. La compilation de ce format se fera de la façon suivante:

appel du préprocesseur;
génére les fichiers
FORMEXdef.def et
FORMEX.mod

```
ppo formex.omp
```

recopie dans le répertoire
source modula2

```
mv FORMEXdef.def <src>==
mv FORMEX.mod <src>==
```

compilation modula2

```
MCD FORMEX
MCD FORMEX
```

B. Compilation d'un module

Soit le module de nom MODUL décrit par le source modul.omp. La session est effectuée ainsi:

appel du préprocesseur;
génére les fichiers
MODUL1def.def, MODUL1.mod,
MODUL4def.def et
MODUL4.mod.

```
ppo modul.omp
```

recopie dans le répertoire src

```
mv MODUL1def.def <src>==
mv MODUL1.mod <src>==
mv MODUL4def.def <src>==
mv MODUL4.mod <src>==
```

compilation du segment 4

```
MCD MODUL4
```

```
                                        MCI MODUL4
```

compilation du segment 1

```
                                        MCD MODUL1
                                        MCI MODUL1
```

Syntaxe des programmes OMPHALE

A cause du compilateur modula2, un nom de format ou de module est limite à 7 lettres.

C.1. Elements communs

Iliste : Ensemble de listes d'importation explicite selon la syntaxe modula2.

```
FROM TOTO IMPORT TUTU, TITI ;
```

Uliste : Ensemble de déclarations d'utilisation de format ou de modules externes.

```
USES FORMAT f1, f2 ;
USES MODULE m1, m2, m3;
USES PROTOTYPE  p1, p2 ;
USES PUBLIC p11 ;
```

Dans l'état actuel du préprocesseur, on ne distingue pas les MODULES, PROTOTYPES et PUBLIC (cette distinction était faite pour génèrer automatiquement les procédures de création des MODULES (privés) ou de recherche des modules PUBLIC; ceci n a pas encore été réalisé).

Sliste : Ensemble de listes de services ou de quais. On peut - à titre purement documentaire - donner entre parenthèses les formats des messages attendus sur ce quai.

```
SERVICE   q1 (f1, f2), q2 ;
```

CTliste : Ensemble de déclarations de constantes ou de type (par de variables ni de procédures) selon la syntaxe modula2.

```
CONST a = 5 ;
TYPE ta = (x, y, z) ;
```

Eliste : UNE liste d'exportation selon la syntaxe modula2

```
EXPORT QUALIFIED a, ta ;
```

Elistes et CTlistes permettent de définir et d'exporter les types nécessaires à l'utilisation d'un format de message.

C.2. Format

Un source de format consiste en (dans l'ordre)

l'entête du format

```
LINKED FORMAT f ;
```

ou

```
FORMAT f ;
```

Une Iliste facultative

```
FROM x IMPORT y ;
```

Une Eliste facultative

```
EXPORT QUALIFIED a, ta;
```

Une CTliste facultative

```
CONST a = 5 ;
TYPE ta = [1..a] ;
```

Une déclaration de structure similaire à une déclaration de record modula2 mais sans variante et avec des types de champs qui sont en un seul mot;

```
STRUCTURE
c1 : CARDINAL ;
c2 : ta ;
END ;
```

Un épilogue (rappelle le nom du format)

```
END f .
```

C.3. MODULE

Un Omodule peut être constitue uniquement d'une partie spécification ou posséder les deux parties spécification et implémentation. Dans tous les cas,

il commencera par un entête de module

```
MODULE m ;
```

et finira par un épilogue

```
END m .
```

i. Partie specification

Cette partie débute par le mot clé :
Il y aura ensuite

```
SPECIFICATION
```

Une Iliste facultative

```
FROM f IMPORT a, ta ;
```

Une Uliste facultative

```
USES FORMAT f1, f2 ;
USES MODULE m ;
```

Une Sliste facultative

```
SERVICE s1, s2 ;
```

Une Eliste facultative

```
EXPORT QUALIFIED b ;
```

Une CTliste facultative

```
TYPE a = ARRAY ta OF CARDINAL ;
```

ii. Partie Implementation

Cette partie débute par le mot clé:
suivie de

```
IMPLEMENTATION
```

Une Iliste facultative

```
FROM x IMPORT y ;
```

Une Uliste facultative

```
USES MODULE b ;
```

Une Sliste facultative

```
SERVICE s3, s4 ;
```

Une Mliste facultative.
Ensemble de déclarations de variables de type message

```
MESSAGE
  af1, bf1 : f1 ;
af2 : f2 ;
```

Une CTVPliste facultative. Ensemble de déclarations de constantes, types, variables et procédures selon la syntaxe modula2.

```
CONST
   b = a + 4 ;
TYPE x = ........
PROCEDURE P1 ;
....
END P1 ;
```

Le corps du module équivalent de la partie initialisation d'un module modula2.

```
BEGIN
....
```

Rappel: n'oublions pas l'épilogue

```
END  m .
```

les particularites du langage

Objet importes.

Les éléments qui sont importes depuis des format ou des Omodules doivent être préfixes par le nom du module fournisseurs (qualification de l'identificateur). En particulier pour les noms de services des autres modules.

Procédure SEND et RCALL

utilisables et dans les corps du module, et dans les procédures. le premier paramètre formel doit être impérativement un nom de message (de la Mliste). Le préprocesseur utilise cette déclaration pour qualifier le nom de procédure avec le nom du format.

Exple:

MESSAGE mx : mf ;

...

SEND (mx,.... deviendra mf.SEND (mx,....

les elements predeclares du langage

Ces éléments sont soit importes automatiquement du Mmodule OMPHALE, soit déclare automatiquement par le préprocesseur.

E.1. Les elements construits par le preprocesseur.

i. Les variables du segment 4.

Ces variables sont propres à chaque Omodule (voir chaque exemplaire de Omodule). Elles sont rémanentes i.e. leurs valeurs sont conservées d'une phase d'activité à l'autre.

SYSTEMRESULT

Compte-rendu des primitives systèmes. du type énumère SYSTEMERRORS. vaut NOERROR si tout s'est passe correctement.

OPENEDSERVICES

de type BITSET ou encore SET OF SERVICEID. Un 1 à une position indique que le quai correspondant est ouvert. Cette variable peut être manipulée par les opération ensemblistes du modula2.

RECEIVEDMESSAGE

est une variable d'un type record avec variante. chaque variante permet de simuler un format de message. Il y a donc autant de variante que de format utilises par le module

RECEIVINGSERVICE

de type SERVICEID. Identifie le service qui a reçu le message qui a déclenche la phase d'activité actuelle. Sa valeur est indéfinie lors de la phase d'initialisation (première phase d'activité).

ii. Les procedures associees aux formats.

SEND (X, S, M) : MID ;

Envoie un message X, vers le service S d'un module M, Retourne un identificateur de message Mid.

RCALL (X, S, M) :

Envoie un message X vers le service S du module M et attend la reponse (et seulement la reponse). Cet appel est bloquant.

E.2. Les elements du module OMPHALE

i. Les types systèmes

SERVICEID

Un entier de 1 à 15. Le type SET OF SERVICEID est équivalent au type BITSET.

MODULEID

Un nom local de lien, donc un index dans la table des liens de l'environnement.

MESSAGEID

Ce type de valeur, retourné par la primitive d'envoi de message identifie le message envoyé et permet ainsi une destruction sélective de ceux-ci par UNSEND.

ii. Les procedure internes de message

SYSTSEND (X, S, M) : MID

Envoi de message non typé.

SYSTRCALL (X, S, M,T)

RCALL système. sauvegarde la variable OPENEDSERVICES dans la variable paramètre T

OFFSET (P)

Permet d'indiquer au système le nombre et la position (P) de chacun des noms locaux dans un format de message. Chacun d'eux sera transformé en lien à l'émission et reconverti automatiquement en nom local à la réception.

UNSEND (MID)

Le message identifié par Mid est repris. Les liens qui y sont inclus sont recupérés et remis dans l'environnement.

UNSENDALL

Detruit tout les messages emis durant la phase d'activité en cours. Les liens qui y sont inclus sont remis dans l'environnement.

iii. Les procEdures de manipulation de lien

DUPLICATELINK (S,D,B)

Permet de dupliquer un lien Source. La copie obtenue aura le nom local Destination. Le boolean B indique si la copie est elle-même duplicable ou non.

SETNOUSE (X,L)

Rend les services di lien L indiqués dans X (BITSET) inutilisables.

SETONEUSE (X,L)

Rend les services di lien L indiqués dans X (BITSET) à utilisation unique.

iv. La primitive WAIT

WAIT

Termine la phase d'activité du module. Bloquant.

v. Les constantes

ANSWER

Quai réservé aux réponses d'appel procédural à distance RCALL. De type SERVICEID.

MYSELF

Nom local , dans un module, du lien qui désigne le même module.

5. MESSAGES D'ERREUR

La numérotation est complétement anarchique. Si vous avez un numéro d'erreur qui n'est pas dans cette liste, il s'agit soit d'un bug du préprocesseur, soit d'une erreur dans la doc... Contactez JMPlace au LIFL.

1.....Fin de format mal formée (manque nom format)
13.....Fin de module mal formée (manque nom module)
14.....Le type d'un message doit être un format
16.....Fin de procédure mal formée (manque non procédure)
202.....Nom du module fournisseur attendu.
203.....IMPORT attendu dans liste d'importation
301.....STRUCTURE attendu dans la dexcription du format.
304.....Mot clé rencontré au lieu d'un nom de champ.
305.....: Attendu dans la définition des types de champ
307.....; attendu (un type de champ message doit etre décrit par un seul mot)
308.....END attendu (Fin de description de structure)
309.....; attendu (Fin de description de structure)
403.....BEGIN attendu (Début de procédure).
404.....END attendu (Fin de procédure).
405.....Mot clé rencontré au lieu d'un nom de procédure (Fin de procédure)
406.....; attendu (Fin de procédure)
502.....FORMAT attendu (entête de format)
503.....mot clé rencontre au lieu du nom de format
504.....; attendu (entête de format)
504.....END attendu (Fin de format)
505.....Mot cle rencontré au lieu du nom du format (Fin de format)
506..... attendu (Fin de format)
2901.....MODULE attendu (entête de module)
2902.....nom de module attendu (entête de module)
2903.....; attendu (entête de module)
2904.....END attendu (Fin de module)
2906.....nom de module attendu (Fin de module)
2907..... attendu (Fin de module)
5105.....FORMAT, MODULE, PUBLIC, ou PROTOTYPE attendu après USES
5200.....mot clé trouve dans liste d'import
5201.....Une virgule doit séparer les éléments importés
5302.....une virgule doit séparer les noms de service
5304.....mot cle rencontre dans Sliste au lieu d'un nom de service
5307.....nom de format attendu
5308.....une virgule doit séparer les noms de format
5309.....nom de format attendu
5310.....) attendue .
7001.....BEGIN attendu (Corps du module)
8001.....: attendu (déclaration de message)
8002.....mot cle rencontre au lieu de nom de message
8003.....nom de format attendu (déclaration de message)
8004.....; attendu après le nom de format
8005.....une virgule doit séparer les noms de message
8501.....QUALIFIED attendu (exportation qualifiée obligatoire)
8502.....mot cle rencontre dans liste d'export
8503.....Une virgule doit séparer les éléments exportés
9002.....parenthèse gauche attendue après SEND ou RCALL.
9003.....le premier parametre de SEND ou RCALL doit etre un message
9004.....End attendu

ANNEXE B.
PROGRAMMATION DES MODULES PILE.

```
MODULE Elem ;

SPECIFICATION
USES FORMAT FEntier, FLien, FQui ;
SERVICE Empiler, Depiler, Chainer, Suivre ;

IMPLEMENTATION
CONST
        TailleElement = 10 ;
TYPE
        IndiceElement = [0..TailleElement] ;
VAR
        ElementSuivant : MODULEID ;
        PileLocale : ARRAY IndiceElement OF INTEGER ;
        SommetLocal : IndiceElement ;
        Dummy : MESSAGEID ;

PROCEDURE ProcEmpiler() ;
VAR
        Valeur : INTEGER ;
BEGIN
        Valeur := RECEIVEDMESSAGE.FEntier.Entier ;
        PileLocale [SommetLocal] := Valeur ;
        INC(SommetLocal) ;
END ProcEmpiler ;
```

```
PROCEDURE ProcDepiler() ;
VAR
        Valeur : INTEGER ;
MESSAGE
        Retour : FEntier ;
VAR
        ModuleRetour : MODULEID ;
        QuaiRetour : SERVICEID ;
BEGIN
        ModuleRetour := RECEIVEDMESSAGE.FQui.NomModule ;
        QuaiRetour := RECEIVEDMESSAGE.FQui.NomQuai ;
        DEC(SommetLocal) ;
        Valeur := PileLocale [SommetLocal] ;
        Retour.Entier := Valeur ;
        Dummy := SEND (Retour, ModuleRetour, QuaiRetour) ;
END ProcDepiler ;

PROCEDURE ProcChainer() ;
BEGIN
ElementSuivant := RECEIVEDMESSAGE.FLien.NomModule ;
END ProcChainer ;

PROCEDURE ProcSuivre() ;
MESSAGE
```

```
        ReponseSuivre : FLien ;
    VAR
        ModuleDemandeur : MODULEID ;
        QuaiDemandeur : SERVICEID ;
    BEGIN
        ModuleDemandeur := RECEIVEDMESSAGE.FQui.NomModule ;
        QuaiDemandeur := RECEIVEDMESSAGE.FQui.NomQuai ;
        ReponseSuivre.NomModule := ElementSuivant ;
        Dummy := SEND(ReponseSuivre, ModuleDemandeur,
QuaiDemandeur) ;
        OPENEDSERVICES := {} ;
    END ProcSuivre ;
    BEGIN
        OPENEDSERVICES := { Chainer } ;
        WAIT() ;
        ProcChainer() ;
        LOOP
            OPENEDSERVICES := {Empiler, Depiler, Suivre} ;
            WAIT() ;
            CASE RECEIVINGSERVICE OF
                Empiler : ProcEmpiler()
                Depiler : ProcDepiler()
                Chainer :
                Suivre : ProcSuivre()
            END ;
        END
    END Elem .
```

```
    FORMAT FEntier ;
        STRUCTURE
        Entier : INTEGER ;
        END ;
    END FEntier .
```

```
    FORMAT FLien ;
        STRUCTURE
        NomModule : MODULEID ;
        END ;
    END FLien .
```

```
    FORMAT FQui;
        STRUCTURE
        NomModule : MODULEID ;
        NomQuai : SERVICEID ;
        END ;
    END FQui .
```

```
    MODULE Pile ;
    SPECIFICATION
    USES FORMAT FEntier, FLien, FQui ;
    SERVICE Empiler(FEntier), Depiler(FQui) ;
```

```
IMPLEMENTATION
USES MODULE Elem ;
SERVICE Reponse ;
MESSAGE
     Transmis : FQui ;
     Chainage : FLien ;
     DemandeSuivant : FQui ;
     ValeurTransmise : FEntier ;

CONST
     TailleElement = 10 ;
TYPE
     CapaciteElement = [0..TailleElement] ;
VAR
     ElementSommet : MODULEID ;
     Utilise : CapaciteElement ;
     Dummy : MESSAGEID ;

PROCEDURE CreerElement() : MODULEID ;
BEGIN
(* PROCEDURE NON DETAILLEE CAR ELLE FAIT REFERENCE *)
(* AU CORTEX - GERANT DE MODULE EN PARTICULIER     *)
END CreerElement ;

PROCEDURE DetruireElement() ;
VAR
     SauvegardeEtatQuai : BITSET ;
       AncienSommet : MODULEID ;
     Clone : MODULEID ;
BEGIN
     AncienSommet := ElementSommet ;
     DUPLICATELINK (MYSELF, Clone, FALSE) ;
     DemandeSuivant.NomModule := Clone ;
     DemandeSuivant.NomQuai := Reponse ;
     SauvegardeEtatQuai := OPENEDSERVICES ;
     OPENEDSERVICES := { Reponse } ;
     Dummy := SEND (DemandeSuivant, AncienSommet, Elem.Suivre) ;
     WAIT() ;
     OPENEDSERVICES := SauvegardeEtatQuai ;
     ElementSommet := RECEIVEDMESSAGE.FLien.NomModule ;
(*   DESTROYLINK (AncienSommet) ; NON IMPLEMENTE DANS LE
PREPROCESSEUR *)
END DetruireElement ;

PROCEDURE ProcEmpiler() ;
VAR
     NouveauSommet : MODULEID ;
BEGIN
     ValeurTransmise.Entier := RECEIVEDMESSAGE.FEntier.Entier ;
     IF Utilise = TailleElement THEN
       NouveauSommet := CreerElement() ;
       Utilise := 0 ;
       Chainage.NomModule := ElementSommet ;
       Dummy := SEND (Chainage, NouveauSommet, Elem.Chainer) ;
       ElementSommet := NouveauSommet ;
     END ;
     Dummy := SEND (ValeurTransmise, ElementSommet, Empiler) ;
END ProcEmpiler ;

PROCEDURE ProcDepiler() ;
```

```
VAR
     Demandeur : MODULEID ;
BEGIN
     Transmis.NomModule := RECEIVEDMESSAGE.FQui.NomModule ;
     Transmis.NomQuai := RECEIVEDMESSAGE.FQui.NomQuai ;
     IF (Utilise = 0) THEN
       DetruireElement() ;
     END ;
     IF (ElementSommet <> 0) THEN
       Dummy := SEND (Transmis, ElementSommet, Depiler) ;
       DEC (Utilise) ;
     END ;
END ProcDepiler ;

BEGIN
     ElementSommet := 0 ; (* MISE A NIL *)
     Utilise := 0 ;
     LOOP
       OPENEDSERVICES := {Empiler, Depiler} ;
       WAIT() ;
       CASE RECEIVINGSERVICE OF
          Empiler : ProcEmpiler() |
          Depiler : ProcDepiler()
       END ;
     END ;
END Pile .
```

ANNEXE C.
CIRCUITERIE DE PARTAGE DES UGM.

DISTRIBUTED HARDWARE FOR VLSI COMPONENT SHARING

Eric DELATTRE
Jean Marc GEIB
Jean Marie PLACE

présenté à :

EUROMICRO 85
Bruxelles

3 - 6 septembre 1985

DISTRIBUTED HARDWARE FOR VLSI COMPONENT SHARING

Eric DELATTRE, Jean-Marc GEIB and Jean-Marie PLACE

Laboratoire d'Informatique Fondamentale de Lille (UA CNRS 369)
UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE I
Bâtiment M3
59655 VILLENEUVE D'ASCQ CEDEX - FRANCE
Tél. : 20/91.92.22 poste 2813

Two decentralized hardware resource sharing mechanisms are described. Both are suited to multi-micro computers with specialized processors. The sharing mechanisms apply to several identical resources which may be described by a cyclic graph.
The sharing mechanisms described can be implemented by adding an electronic management circuit to each resource. Processor requests are broadcast to each and every circuit. Each circuit executes the same algorithm. As a result, at most one circuit accepts the request, acknowledging the acceptance to the requestor.
The first mechanism satisfies requests in their order in which they are presented. It consists in the circulation of tokens (the ability to accept a request) along a ring to which resources are connected.
While remaining fair, the second mechanism disregards the order of requests. Two rings - called preference and request ring - carry the information that allow a circuit to decide whether or not it will accept a given request.
Focusing on the integration of management circuits a limited number of connections is sought.

The dramatic advances of VLSI technology make feasible innovative multi-micro computer architectures [1]. Future hardware architectures will be a mixture of both "vertical migration" of high-level functions into the microcode of a CPU, and "horizontal migration" of function out the CPU into dedicaced processors. Thus, tomorrow's computers will be built around VLSI specialized units. Computer designers are faced with the problem of the sharing of these units. The paper addresses a sub-class of this general sharing problem.

This study is based on the OMPHALE project, a fault-tolerant, highly modular and highly parallel distributed computing system [2].

1. INTRODUCTION

1.1. Resource features and scheduling policy

Suppose a system consists of p processors and r hardware resources. The design of a hardware mechanism which permits the dynamic sharing of the r resources among the p processors is desired. The present section defines a sub-class of the resource sharing issue which is addressed throughout the paper.

The resource features and the scheduling policy are listed below.

1) Identical resources

All resources are identical. Any two resources are interchangeable. The total number of resources is unspecified.

2) Cyclic state graph

Each resource exhibits the same cyclic graph. The graph may consist of g states and g transitions. Transition # i moves the resource from state # i to state # i+1 mod g. State # 0 is the initial state.

3) Specialized processors

To gain access to a resource, a processor issues a command. A successful command is acknowledged by the selected resource and triggers a state transition at the resource. There is a one to one correspondance between the commands and the transitions. To gain a transition # i, a processor issues a command # i. Thus, there are g different commands.

The processors are specialized. Each command may be issued by only one processor, the command "requestor". Consider processor # k : Let Q_k be the set of the commands whose processor # k is the requestor and let C be the set of the commands. Then the set defined by $\{Q_0, Q_1, \ldots, Q_{p-1}\}$ is a partition of set C.

4) Ordering of the transitions

When a requestor issues a command # i, it must receive one of the two following answers from the resources :

- NO ACKNOWLEDGMENT. That is to say, if there is no resource in the state # i, the transition # i cannot be performed by any resource. The command is not acknowledged.

- or SINGLE ACKNOWLEDGMENT. If there exists at least one resource in the state # i, only one resource in the state # i must perform transi-

tion # i and must acknowledge the command.

Consider the sequencing order of the following events : successful initial command # 0 is issued by the corresponding requestor. This order is application-dependant. For instance, the order of commands # 0 is derived from the ready process queue. Depending on the application, the order of successful commands # i (i=1,..., g-1) must be either the same as the order of successful commands # 0 or different than. The case of the preservation of the order is called the "sequencing case". The other is called the "non-sequencing case".

1.2. A case study : Memory Management Unit Sharing

The resources are Zilog Z8010 Memory Management Units (MMU) [3]. The requestor processors are Zilog Z8001s [4] which are characterized by a segmented addressing scheme. MMU chips translate a logical address generated by the microprocessor into a physical memory address. The address space for each accessible segment of the Z8001 is limited by data inside MMU registers : that is the value of the base address in memory, and the length in multiples of 256 bytes.

A process context switching consists of

- saving the MMU register content representing the old process address space,
- and then loading the MMU registers with values representing the new process address space.

A bi-processor has been designed [5] for the purpose of fast context switching (see figure 1). The MMUs are arranged in a pool. A MMU exhibits a cyclic graph having the following six states (see figure 2) : free, loading, loaded, executing, executed and saving. Processor # 0 executes system software and issues start loading, stop loading, start saving ans stop saving commands, while processor # 1 runs user processes and issues start executing and stop executing commands.

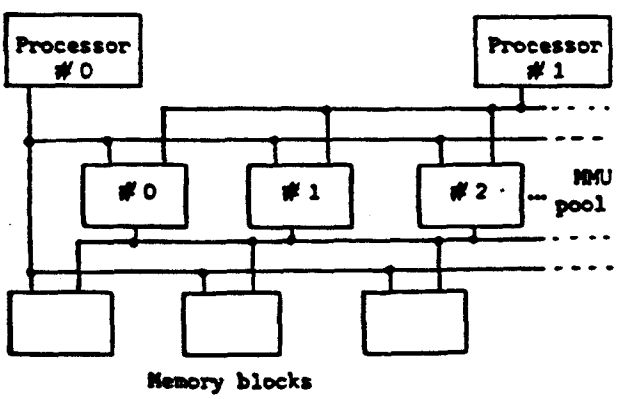


Figure 1 : A bi-processor with fast context switching.

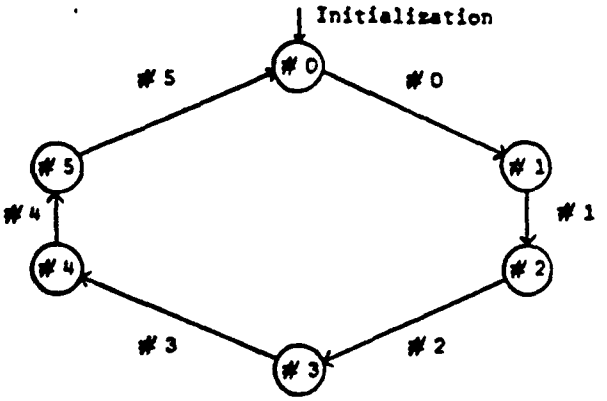


Figure 2 : The MMU cyclic graph.

States

0 : free
1 : loading
2 : loaded
3 : executing
4 : executed
5 : saving

Transitions (successful commands)

0 : start loading
1 : stop loading
2 : start executing
3 : stop executing
4 : start saving
5 : stop saving

Resource sharing features are the following :

- the number of MMU in the pool is unspecified (but a minimum number of 2 is required),
- each MMU exhibits the same six state cyclic graph,
- the processors are specialized,
- the architecture may or may not preserve the application-dependant sequencing order of the commands.

2. DRAWBACKS OF CENTRALIZED MECHANISMS

One can design a single centralized automaton which solves the sharing problem, by combining every resource cyclic automaton. Such a design is feasible, but offers the following drawbacks :

- POOR EXTENSIBILITY. To add a requestor or a resource, one must redesign the whole automaton and reprogram the automatism, which will

be over-sized, in order to avoid a redesign of the hardware.

- POOR RELIABILITY. A centralized automatism is prone to hardware failures.
- TOO MANY CONNECTIONS. The automatism board is complex, supporting all the control lines and all the data path selection lines. There are too many connections.

Example :

Refer to the bi-processor described in section 1.2. If the MMU pool contains 2 resources, the automaton is composed of 31 states. If the pool contains 4 MMUs, the composition of the automaton exceeds 200 states.

Distributed mechanisms which solve the sharing issue are detailed in the rest of the paper.

3. COMMANDS AND MECHANISM DISTRIBUTION

3.1. Commands

There are several means by which p processors can be connected to r resources : bus, ring, crossbar switch, mesh network,... These usual solutions are rejected because they raise the issue of sharing the switching mechanism itself in place of the resources.

Each command is broadcast by its requestor processor to each and every resource via a command line. Each resource senses all the commands lines. When some resource accepts a command, it acknowledges the requestor via a acknowledgment line. Thus, there exists a total number of g+p specialized lines between the processors and the resources.

3.2. Data path switching

Data path switching in not pursued in this paper, as the conventional high-Z circuitry is applicable.

3.3. Mechanism distribution

Each resource is provided with an identical sharing circuit. Each sharing circuit exhibits the same behavior and accepts the same data, the commands.

4. NON-SEQUENCING CASE

4.1. Request chain

Request chains are built by connecting the resource # j to the resource # j+1 ($j < r-1$), one chain for each state (there are g request chains). With such a construction, resource # 0 is assigned the highest priority, whereas resource # r-1 has the lowest.

A sharing circuit is composed of g request circuits and one wired automaton that dispatchs the state of the resource to the request circuits.

While in the state # i, the resource # j inhibits the response of lower priority resources by reseting the i-th request chain. When a request # i occurs, one resource, if any, triggers this transition : namely, the one which has the highest priority among those in the state # i.

This solution is not a perfect one. The sequence of the requests may happen to be such that the resource # 0 will accept more requests than other resources, disturbing the system. In order to solve this problem, one can try to change dynamically the priorities among resources.

4.2. Request ring and preference ring

To change priorities among resources, request chains are converted into rings (resource # j is connected to resource # j+1 mod r) and a preference ring similar to the request ring is added for each resource state.

The resource with the highest prority is said to "own" the preference. Fairness is enforced by moving the preference along the ring. The request ring continues to inhibit lower priority resources.

A resource owning the preference is characterized by having :

a) The highest priority for responding to the corresponding request, and

b) The lowest priority for being granted the preference at the next change.

4.2.1. Implementation

We now detail the behavior of a sharing circuit for a given request.

The circuit has one input and one output for each ring. The name and meaning of these are listed below.

We consider two internal logical variables (hardware flip-flops).

$S_{i,j}$ $\Longleftrightarrow$ the resource # j is in the state # i.

$P_{i,j}$ $\Longleftrightarrow$ the resource # j owns the preference # i.

The name and meaning of the connections to the rings are as follows :

- Request ring

Input :	$Spred_{i,j}$	$\Longleftrightarrow$	There is no resource with higher priority than the resource # j in the state # i.
Output :	$Ssucc_{i,j}$		Computed from $Spred_{i,j}$, $S_{i,j}$ and $P_{i,j}$ (see section 4.2.4).

- Preference ring

Input :	$Ppred_{i,j}$	$\Longleftrightarrow$	There is a resource with higher priority asking for the preference.
Output :	$Psucc_{i,j}$		Computed from $Ppred_{i,j}$, $S_{i,j}$ and $P_{i,j}$ (see section 4.2.4).

4.2.2. *Processing a request*

When a processor issues a request, two cases may arise :

- No resource is in the state # i ; the request is refused.
- One or more resources may realize the requested transition. The one with the higher priority will be choosen i.e. the one that verifies the predicate $S_{i,j}$ AND [$P_{i,j}$ OR $Spred_{i,j}$] (The resource is in the state # i and it has the preference or else no resource with higher priority can realize the transition).

4.2.3. *Preference moving*

The movement of preference along the preference ring from resource # j to resource # h may be viewed as two distinct (thought concurrent) operations.

1) The resource # j that owns the preference loses it when another resource demands the preference. ($P_{i,j}$ is reset if $P_{i,j}$ AND $Ppred_{i,j}$), and
2) Only one resource can gain the preference, namely that resource in the state # i with the highest priority (excluding the one which actually owns the preference). The resource gaining the preference does so by satisfying the following : NOT [$P_{i,h}$] AND NOT [$Ppred_{i,h}$] AND $S_{i,h}$

4.2.4. *Generation of Psucc and Ssucc*

We consider the rings # i. The logical equations of Ssucc and Psucc are as follows :

$Ssucc_{i,j}$ = NOT[$S_{i,j}$] AND [$P_{i,j}$ OR $Spred_{i,j}$],

$Psucc_{i,j}$ = NOT[$P_{i,j}$] AND [$S_{i,j}$ OR $Ppred_{i,j}$].

When a resource is not in the state # i, the resource merely transmits Spred to Ssucc and Ppred to Psucc.

If in the state # i, the resource automatically reset $Ssucc_{i,j}$ showing that it is ready to accept the request to all resources with lower priority.

4.2.5. *Local behavior of a resource with respect to a request*

When a request # i arises, there are four cases possible for a resource.

a) Some state other than # i without preference.

The resource is not able to accept the request nor is it able to manipulate the preference because it doesn't have it. No reaction.

b) State # i without preference.

The resource may or may not accept the request, depending on Spred. In addition, it will receive the preference if it has the highest priority excluding the resource that owns the preference. Thus, we can cite three possible cases :

b.1) The resource accepts the request and receives the preference.

b.2) The resource only receives the preference.

b.3) No change.

c) State # i with preference.

The resource accepts the request. Furthermore, it may lose the preference if any other resource asks for it (this fact is known by Ppred).

d) Some state other than # i with preference.

The resource will not accept the request. It may however give its preference to any other resource which wants it.

It should be noted that the acceptance of a request changes the state of the accepting resource from # i-1 mod g to # i, making $S_{i,j}$ true and afterward making $Ssucc_{i,j}$ and $Psucc_{i,j}$ true.

4.3. Sharing circuit

A sharing circuit consists of one automaton and g request circuits. The automaton signals the g states, S_0 to S_{g-1}.

The request circuits are connected to request and preference rings. The connections of a request circuit can be :

- Spred - input from the request ring.
- Ssucc - output to the request ring.
- Ppred - input from the preference ring.
- Psucc - output to the preference ring.
- C - input from where commands are received.
- S - input from where the state of the resource is known.
- Ck - output which triggers a state transition for the resource.

The flip-flop which retains the value of $P_{i,j}$ is internal to the request circuit.

When a request # i is accepted by the request circuit of the resource # j, a pulse is generated on the Ck output. This pulse triggers the change of resource state from state # i to state # i+1 mod g. Of course, this change is reflected on the signal S_i and $S_{i+1 \bmod g}$.

One schema of the request circuit can be :

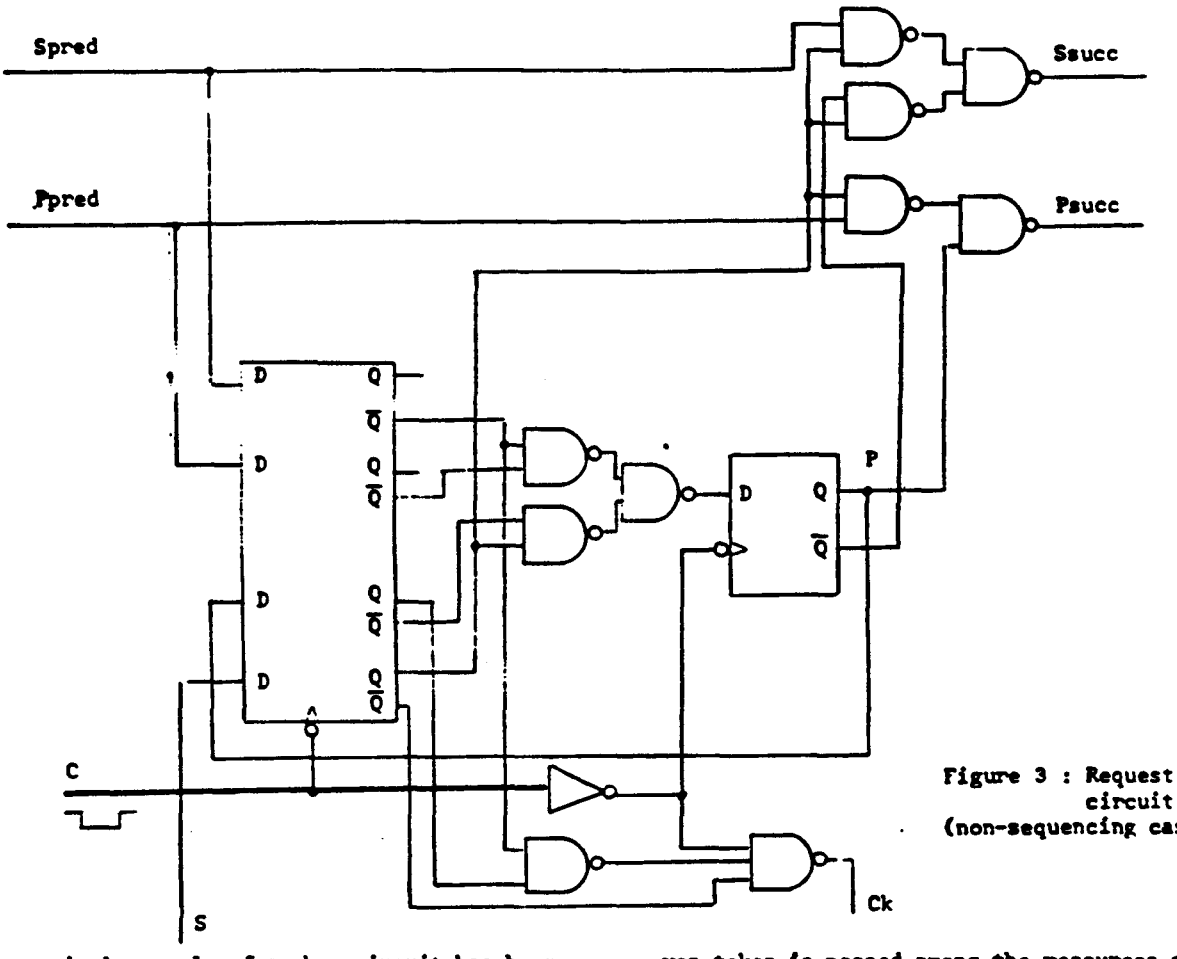


Figure 3 : Request circuit (non-sequencing case).

A practical example of such a circuit has been designed in the frame of MMU sharing presented in section 1.2. The automaton is realized simply, by a parallel load shift register. Any pulse from any Ck will provoke a one position shift.

On system start-up, all resources have to be put in the state S_0. One of them should obtain all the preferences. This operation, like the generation of acknowledge signals, is really no a problem.

The sharing circuit has been designed with some simplified request circuits, based on the knowledge that only one resource at a time may accept corresponding requests (namely commands # 1, # 3 and # 5). Negative logic is used.

5. SEQUENCING CASE

In the case of sequencing, the resource's order of responding must be the same, whichever is the request number considered. For example, if, for the request # 0, the order is "resource # 0 then # 1 then # 2", the same order should be conserved for request # 1 or # 2 or any other.

5.1. Token passing

There exists one token per resource state. A given token is passed among the resources connected on a ring. A resource in the state # i triggers a transition if it owns the single token # i when a command # i occurs. This resource then grants the token to the next resource on the ring.

5.2. Sequencing enforcement

We note a → b to mean "a occurs before b".

The state at system start-up is defined as follows :

- Every resource is in the state # 0.
- The resource # 0 owns all the tokens.

Our mechanisms is based on three properties which are verified by any resource # j :

a) A given transition can be triggered by resource # j only if the resource owns the corresponding token.
b) A transition # i triggers the move of the token # i from the responding resource # j to the next resource (numbered # j+1 mod r).
c) The executing resource # j can execute transition # i if and only if it is in state # i. That is to say, resource # j must have just previously executed # i-1 (with the exception where # i is the system start-up state).

From the transitivity of the relation $\rightarrow$, we can show three results to be always true :

1) A resource will have triggered as many, or more transitions than a higher ranking resource.

2) A given transition will be triggered in the same order by each resource.

3) Any resource will receive tokens according to the cyclic order 0, 1,..., g-1, 0, 1,..., etc.

Moreover, a property that simplifies realization can be obtained by recurrence :

If a resource is in the state # i and if it owns z tokens, then those tokens are tokens # i, i+1,..., i+z-1 mod g.

5.3. Sharing circuit

Each resource is fitted with a sharing circuit. These circuits are connected along only one ring which is used to transmit the tokens. This is predicated on the fact that tokens always move in the same order.

The sharing circuit associated with the resource # j has one token receiving input RT and one token sending output ST.

Like the non-sequencing case, the automaton in the sharing circuit is materialized as a parallel load shift register that delivers the signals S_0 to S_{g-1}. We take S_x to mean that the resource is in the state # x.

As only one ring is used, only a counter-decounter is needed to memorize which tokens are owned by the resource (In fact, the only information needed to decide whether or not a transition will be triggered is to answer the question "is there any token at all ?"). See figure 4.

The system start-up state is characterized by the following conditions :

Resource # 0 owns every token (they are g of them).
Resource # j for j > 0 have no tokens.

Each resource is in the initial state # 0.

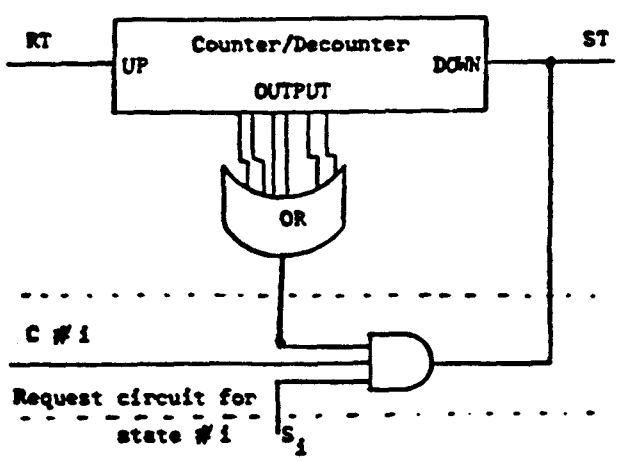


Figure 4 : Sharing circuit (sequencing case).

6. FEASIBILITY ISSUES

The feasibility of integrating the sharing circuit is dependant of the number of connections required and of the complexity of the mechanism. These issues are examined in the following sections.

6.1. Connections

The system consists of p processors and r resources. Each resource is characterized by a g state cyclic graph.

In the non-sequencing case, the number of required connections per sharing circuit is as follows :

- g connections for the command lines,
- p connections for the acknowledgment lines,
- 4 connections per state for the request ring and the preference ring (input and output).

The total number of lines is therefore $g + p + 4*g$

In the MMU sharing case $g = 6$ and $p = 3$ (maximum number). In the non-sequencing case, the number of connections is 33. In the sequencing case, 11 connections are required.

6.2. Complexity

The number of required elementary gates per sharing circuit is g in the non-sequencing case and 3*g in the sequencing case.

The number of required memory bits is 6*g in the non-sequencing and $\log_2 g$ in the sequencing case.

6.3. Conclusion

The sharing circuit is not complex. A small number of connections is required. The integration of the sharing circuit seems feasible.

7. FUTURE WORK

Planned future research includes the following issues :

- non-cyclic resource graph.

The mechanism of request and preference rings remains unchanged, but the request circuit can't be the same for every state : When in a state, a resource is able to respond at more than one request. The automaton has to be designed the usual way. The major drawback is that each problem will need its own specific study. The integration may not be appropriate.

- Multiple requestors

One can generalize the non sequencing case to multiple requestors. For example, a given request may be issued by several processors. The sharing circuit may be the same as that described, provided that collisions (simultaneous issues of the same request) are effectively dealt with.

Two mechanisms may be considered : prevention or detection. Both mechanism preserve the generality of the sharing circuit.

ACKNOWLEDGEMENTS. We would like to thank C. Carrez for his careful review of previous drafts of this paper and D. Johnston for his improvements of our English syntax.

REFERENCES

[1] W.K. GILOI, P. BEHR
"Hierarchical function distribution - a design principle for advanced multicomputer architectures".
Proc. of the 10th Annual Symp. on Computer Architecture, 1983.

[2] J.M. GEIB, E. DELATTRE
"A distributed and protected system, which maintains modularity up to execution".
Proc. of the IFIP WG 10.3 Workshop "Concurrent Languages in distributed Systems", Bristol, March 1984.

[3] Zilog CORP
"Z8010 - Z8000 Z-MMU Memory Management Unit - Product specification".
March 1981.

[4] Zilog CORP
"Z8000 CPU - Technical Manual".
August 1980.

[5] E. DELATTRE
"Architecture bi-processeur dotée d'un adressage segmenté protégé et bâtie autour du microprocesseur Z8000".
Pub. # 26, Laboratoire d'Informatique Fondamentale de Lille, Mars 1983.

ANNEXE D.
PROGRAMME DE CHARGEMENT °D'UNE MMU.

```
          NAME      'pmmu1'
PMMU1_    SEGMENT   1
          ORG       5000H
!*************************************************************************
! Programmation des MMU - E. Delattre - 3/6/87
!
! Modifications pour tests des commandes MMU par F. Hemery  - 11/6/87


          GLOBAL    SAR,DESC,PROG,TR,COMC,COMX,COMS,PROG1,PROG2,PROG3
          GLOBAL    PROG4,PROG5,LECT

          INCLUDE   70        ! Ports des MMU

MMU1      EQU       2H
MMU2      EQU       4H

MMU       EQU       MMU1      ! MMU a programmer

PROG      EQU       *
          SOUT      #PRESET+MMU,R1      ! RESET

! MODE : Id MMU = 000 - NMS = 0 (Don't care) - MST = 0 No Multiple Segment
!  Table - URS = 0 Lower Ranger Select - TRNS = 0 No Translate - MSEN = 1
!  Master Enable ---> 1000 0000 = 80H
          LD        R1,#80H*8
          SOUT      #PMODE+MMU,R1

          SOUT      #PSETCPUI+MMU,R1    ! Set All CPUI Flags
          SOUT      #PSETDMAI+MMU,R1    ! Set All DMAI Flags

          LD        R1,#0               ! Progr. descripteur #0
          SOUT      #PRWSAR+MMU,R1
          LD        R1,#LIMIT
          SOUT      #PRWDSC+MMU,R1
          LD        R1,#0FFH*8
          SOUT      #PRWD+MMU,R1
          LD        R1,#ATTRIBUTE       ! Attribute field
          SOUT      #PRWDSC+MMU,R1
          LD        R1,#12H*8           ! xx01 0010
!                           DIRW=0, DMAI=1, EXC=0, CPUI=0 Cpu enable,
!                           SYS=1 System only, Rd=0
          SOUT      #PRWD+MMU,R1

          LD        R1,#23H*8           ! Progr. descripteur #23H
          SOUT      #PRWSAR+MMU,R1
!         Adresse de base = 0011 0101 1010 * 8 = 035AH * 8
!         = 1 1010 1101 0000 = 1AD0H
          LD        R1,#H_BASE_ADDRESS
          SOUT      #PRWDSC+MMU,R1
          LD        R2,#3H*8            ! 03H * 8
          SOUT      #PRWD+MMU,R2
          LD        R1,#L_BASE_ADDRESS
          SOUT      #PRWDSC+MMU,R1
          LD        R3,#5AH*8           ! 5AH * 8
```

```
          SOUT      #PRWD+MMU,R3
          LD        R1,#LIMIT
          SOUT      #PRWDSC+MMU,R1
          LD        R1,#07H*8           ! Limite 7*8 octets !!!
          SOUT      #PRWD+MMU,R1
          LD        R1,#ATTRIBUTE
          SOUT      #PRWDSC+MMU,R1
          LD        R1,#10H*8           ! xx01 0000
!                           DIRW=0, DMAI=1, EXC=0, CPUI=0 Cpu enable,
!                           SYS=0, RD=0
          SOUT      #PRWD+MMU,R1
et1       JR        et1

!************************ Ecriture d'une base adresse *************
!
PROG1     EQU       *
          LD        R1,#05H*8           ! DESCRIPTEUR 05H
          SOUT      #PRWSAR+MMU,R1
          LD        R1,#05H*8           ! H_BASE ADDRESS
          SOUT      #PRWBF+MMU,R1       ! ECRITURE DE B_A DANS SAR 05H
          LD        R1,#55H*8           ! L_BASE ADDRESS
          SOUT      #PRWBF+MMU,R1       ! ECRITURE DE B_A DANS SAR 05H
FPROG1    JR        FPROG1

!****************** Ecriture de la B_A avec incrementation ***********
!
PROG2     EQU       *
          LD        R1,#10H*8           ! DESCRIPTEUR 10H
          SOUT      #PRWSAR+MMU,R1
          LD        R1,#0AH*8           ! BASE ADDRESS
          SOUT      #PRWBFI+MMU,R1      ! ECRITURE DE B_A DANS SAR 10H
          LD        R1,#0AAH*8          ! BASE ADDRESS
          SOUT      #PRWBFI+MMU,R1      ! ECRITURE DE B_A DANS SAR 10H
          LD        R1,#0BH*8           ! BASE ADDRESS
          SOUT      #PRWBFI+MMU,R1      ! ECRITURE DE B_A DANS SAR 11H
          LD        R1,#0BBH*8          ! BASE ADDRESS
          SOUT      #PRWBFI+MMU,R1      ! ECRITURE DE B_A DANS SAR 11H
FPROG2    JR        FPROG2

!****************** ecriture d'une limite avec incrementation *******
!
PROG4     EQU       *
          LD        R1,#10H*8           ! DESCRIPTEUR 10H
          SOUT      #PRWSAR+MMU,R1
          LD        R1,#07H*8           ! LIMIT
          SOUT      #PRWLFI+MMU,R1      ! ECRITURE DE LA LIMIT DANS SAR 10H
          LD        R1,#06H*8           ! LIMIT
          SOUT      #PRWLFI+MMU,R1      ! ECRITURE DE LA LIMIT DANS SAR 11H
FPROG4    JR        FPROG4

!!****************** Ecriture d'un attribut avec incrementation ********
!
PROG5     EQU       *
          LD        R1,#10H*8           ! DESCRIPTEUR 10H
          SOUT      #PRWSAR+MMU,R1
          LD        R1,#10H*8           ! ATTRIBUT
          SOUT      #PRWAFI+MMU,R1      ! ECRITURE DE L'A DANS SAR 10H
          LD        R1,#10H*8           ! ATTRIBUT
          SOUT      #PRWAFI+MMU,R1      ! ECRITURE DE L'A DANS SAR 11H
FPROG5    JR        FPROG5

!********************** Lecture de la totalite d'un descripteur ********
!         Sorties : (RR2) = totalite du descripteur
!PROG6    EQU       *
!         LD        R1,10H*8            ! DESCRIPTEUR 10H
!         SOUT      #PRWSAR+MMU,R1      !
!         SINL      RR2,#PRWDI+MMU      ! LECTURE DU DESCRIPTEUR
```

```
!         SIN       R1,#PRWDSC+MMU      ! LECTURE DU COUNTEUR
!FPROG6   JR        FPROG6

!********************* Lecture de l'ensemble d'un descripteur ***********
!         Entree :  (R1) = descripteur * 8
!         Sortie :  (R5) = (H_Base_Address)
!                   (R6) = (L_Base_Address)
!                   (R7) = (Limit)
!                   (R8) = (Attribut)
PROG3     EQU       *
          SOUT      #PRWSAR+MMU,R1
          LD        R1,#00H
          SOUT      #PRWDSC+MMU,R1
          SIN       R5,#PRWD+MMU
          SIN       R6,#PRWD+MMU
          SIN       R7,#PRWD+MMU
          SIN       R8,#PRWD+MMU
FPROG3    JR        FPROG3


!****************** lecture de SAR et DSC *********************************
!         Sortie :  (R7) = (SAR)
!                   (R8) = (DSC)
SAR       SIN       R7,#PRWSAR+MMU
          SIN       R8,#PRWDSC+MMU
et2       JR        et2

!****************** lecture d'un octet d'un descripteur *****************
!         Entree :  (R1) = #descripteur * 8
!                   (R2) = *dsc * 8
!         Sortie :  (R3) = octet
DESC      SOUT      #PRWSAR+MMU,R1
          SOUT      #PRWDSC+MMU,R2
          SIN       R3,#PRWD+MMU
et3       JR        et3

!****************** commande de traduction d'adresses *******************
! MODE : Id MMU = 000 - NMS = 0 (Don't care) - MST = 0 No Multiple Segment
!  Table - URS = 0 Lower Ranger Select - TRNS = 1  TRANSLATE - MSEN = 1
!  Master Enable ---> 1100 0000 = C0H
TR        LD        R1,#0C0H*8
          SOUT      #PMODE+MMU,R1
et4       JR        et4

!************************ Commande C ************************************
!
COMC      OUT       #0FBEFH,R0
et5       JR        et5

!************************ Commande X ************************************
COMX      OUT       #0FBF7H,R0
et6       JR        et6

!************************ Commande S ************************************
COMS      OUT       #0FBDFH,R0
et7       JR        et7

!********************** lecture du mode register ************************
!              Sortie : (R7) = (Mode_Register)
LECT      EQU       *
          SIN       R7,#PMODE+MMU
et8       JR        et8

          END
```

ANNEXE E. : RESULTATS DE SIMULATION AVEC MEMOIRE.

Stratégie INT
3 UGM
Jeu d'essai 12
la=1
tp=489
tx=500

NBBLOCS x SZBLOCS = 1	1x1 263503												
NBBLOCS x SZBLOCS = 2	1x2 263503	2x1 194529											
NBBLOCS x SZBLOCS = 3	1x3 263503		3x1 173347										
NBBLOCS x SZBLOCS = 4	1x4 263503	2x2 194529		4x1 162940									
NBBLOCS x SZBLOCS = 5	1x5 263503				5x1 156876								
NBBLOCS x SZBLOCS = 6	1x6 263503	2x3 194529	3x2 173347			6x1 151551							
NBBLOCS x SZBLOCS = 7	1x7 263503						7x1 148722						
NBBLOCS x SZBLOCS = 8	1x2 263503	2x4 194529		4x2 150157				8x1 146618					
NBBLOCS x SZBLOCS = 9	1x9 263503		3x3 173347						9x1 146000				
NBBLOCS x SZBLOCS = 12	1x12 263503	2x6 194529	3x4 173347	4x3 146571		6x2 144061				12x1 142900			
NBBLOCS x SZBLOCS = 18	1x18 263503	2x9 194529	3x6 173347			6x3 141621			9x2 139963		18x1 138761		
NBBLOCS x SZBLOCS = 24	1x24 263503	2x12 194529	3x8 173347	4x6 139394		6x4 138651		8x3 137927		12x2 136999		24x1 136977	
NBBLOCS x SZBLOCS = 36	1x36 263503	2x18 194529	3x12 173347	4x9 137342		6x6 136977			9x4 137065	12x3 136977	18x2 136977		36x1 136977
NBBLOCS x SZBLOCS = 48	1x48 263503	2x24 194529	3x16 173347	4x12 137342		6x8 136977		8x6 136977	12x4 136977	16x3 136977		24x2 136977		48x1 136977

NBBLOCS x SZBLOCS = 1	**1x1** 263503												
NBBLOCS x SZBLOCS = 2	**1x2** 263503	**2x1** 194527											
NBBLOCS x SZBLOCS = 3	**1x3** 263503		**3x1** 173346										
NBBLOCS x SZBLOCS = 4	**1x4** 263503	**2x2** 194527		**4x1** 162940									
NBBLOCS x SZBLOCS = 5	**1x5** 263503				**5x1** 157877								
NBBLOCS x SZBLOCS = 6	**1x6** 263503	**2x3** 194527	**3x2** 173346			**6x1** 155026							
NBBLOCS x SZBLOCS = 7	**1x7** 263503						**7x1** 153489						
NBBLOCS x SZBLOCS = 8	**1x2** 263503	**2x4** 194527		**4x2** 183930				**8x1** 151288					
NBBLOCS x SZBLOCS = 9	**1x9** 263503		**3x3** 173346						**9x1** 151432				
NBBLOCS x SZBLOCS = 12	**1x12** 263503	**2x6** 194527	**3x4** 173346	**4x3** 188602		**6x2** 173249				**12x1** 153514			
NBBLOCS x SZBLOCS = 18	**1x18** 263503	**2x9** 194527	**3x6** 173346			**6x3** 184850			**9x2** 164682		**18x1** 150592		
NBBLOCS x SZBLOCS = 24	**1x24** 263503	**2x12** 194527	**3x8** 173346	**4x6** 192012		**6x4** 169050		**8x3** 173345		**12x2** 155228		**24x1** 151064	
NBBLOCS x SZBLOCS = 36	**1x36** 263503	**2x18** 194527	**3x12** 173346	**4x9** 181430		**6x6** 170818			**9x4** 160176	**12x3** 156786	**18x2** 155524		**36x1** 151200
NBBLOCS x SZBLOCS = 48	**1x48** 263503	**2x24** 194527	**3x16** 173346	**4x12** 186296		**6x8** 167172		**8x6** 161290	**12x4** 153156	**16x3** 153821		**24x2** 153418	**48x1** 153385

Stratégie SCP
3 UGM
jeu d'essai 12
la=1
tp=489
tx=500

Stratégie INT
5 UGM
jeu d'essai 11
ia=1
tp=489
tx=500

NBBLOCS x SZBLOCS = 1	1x1 92445											
NBBLOCS x SZBLOCS = 2	1x2 92445	2x1 67907										
NBBLOCS x SZBLOCS = 3	1x3 92445		3x1 61309									
NBBLOCS x SZBLOCS = 4	1x4 92445	2x2 67907		4x1 57962								
NBBLOCS x SZBLOCS = 5	1x5 92445				5x1 56184							
NBBLOCS x SZBLOCS = 6	1x6 92445	2x3 67907	3x2 61309			6x1 53499						
NBBLOCS x SZBLOCS = 7	1x7 92445						7x1 52776					
NBBLOCS x SZBLOCS = 8	1x2 92445	2x4 67907		4x2 57962				8x1 52167				
NBBLOCS x SZBLOCS = 9	1x9 92445		3x3 61309						9x1 51906			
NBBLOCS x SZBLOCS = 12	1x12 92445	2x6 67907	3x4 61309	4x3 57962		6x2 51180				12x1 51392		
NBBLOCS x SZBLOCS = 18	1x18 92445	2x9 67907	3x6 61309			6x3 51777			9x2 51623		18x1 51392	
NBBLOCS x SZBLOCS = 24	1x24 92445	2x12 67907	3x8 61309	4x6 57962		6x4 51689		8x3 51667		12x2 51414		24x1 51392
NBBLOCS x SZBLOCS = 36	1x36 92445	2x18 67907	3x12 61309	4x9 57962		6x6 51535			9x4 51425	12x3 51403	18x2 51392	
NBBLOCS x SZBLOCS = 48	1x48 92445	2x24 67907	3x16 61309	4x12 57962		6x8 51535		8x6 51590	12x4 51403	16x3 51392		24x2 51392

Stratégie SCP
5 UGM
jeu d'essai 11
ia=1
tp=489
tx=500

NBBLOCS x	1x1												
SZBLOCS = 1	92445												
NBBLOCS x	1x2	2x1											
SZBLOCS = 2	92445	67906											
NBBLOCS x	1x3		3x1										
SZBLOCS = 3	92445		61309										
NBBLOCS x	1x4	2x2		4x1									
SZBLOCS = 4	92445	67906		57962									
NBBLOCS x	1x5				5x1								
SZBLOCS = 5	92445				56184								
NBBLOCS x	1x6	2x3	3x2			6x1							
SZBLOCS = 6	92445	67906	61309			53499							
NBBLOCS x	1x7						7x1						
SZBLOCS = 7	92445						52776						
NBBLOCS x	1x2	2x4		4x2				8x1					
SZBLOCS = 8	92445	67906		57962				52167					
NBBLOCS x	1x9		3x3						9x1				
SZBLOCS = 9	92445		61309						51906				
NBBLOCS x	1x12	2x6	3x4	4x3		6x2				12x1			
SZBLOCS = 12	92445	67906	61309	57962		61814				51392			
NBBLOCS x	1x18	2x9	3x6			6x3			9x2		18x1		
SZBLOCS = 18	92445	67906	61309			63880			55021		51891		
NBBLOCS x	1x24	2x12	3x8	4x6		6x4		8x3		12x2		24x1	
SZBLOCS = 24	92445	67906	61309	57962		63297		57713		57296		52306	
NBBLOCS x	1x36	2x18	3x12	4x9		6x6			9x4	12x3	18x2		36x1
SZBLOCS = 36	92445	67906	61309	57962		58647			55159	53303	52535		52306
NBBLOCS x	1x48	2x24	3x16	4x12		6x8		8x6	12x4	16x3		24x2		48x1
SZBLOCS = 48	92445	67906	61309	57962		57634		55542	53156	53467		52306		52306

NBBLOCS x	**1x1**												
SZBLOCS = 1	92445												
NBBLOCS x	**1x2**	**2x1**											
SZBLOCS = 2	92445	67907											
NBBLOCS x	**1x3**		**3x1**										
SZBLOCS = 3	92445		61309										
NBBLOCS x	**1x4**	**2x2**		**4x1**									
SZBLOCS = 4	92445	67907		58012									
NBBLOCS x	**1x5**				**5x1**								
SZBLOCS = 5	92445				56264								
NBBLOCS x	**1x6**	**2x3**	**3x2**			**6x1**							
SZBLOCS = 6	92445	67907	61309			53539							
NBBLOCS x	**1x7**						**7x1**						
SZBLOCS = 7	92445						52816						
NBBLOCS x	**1x2**	**2x4**		**4x2**				**8x1**					
SZBLOCS = 8	92445	67907		53141				52207					
NBBLOCS x	**1x9**		**3x3**						**9x1**				
SZBLOCS = 9	92445		61309						51946				
NBBLOCS x	**1x12**	**2x6**	**3x4**	**4x3**		**6x2**				**12x1**			
SZBLOCS = 12	92445	67907	61309	52104		51590				51392			
NBBLOCS x	**1x18**	**2x9**	**3x6**			**6x3**			**9x2**		**18x1**		
SZBLOCS = 18	92445	67907	61309			51570			51590		51392		
NBBLOCS x	**1x24**	**2x12**	**3x8**	**4x6**		**6x4**		**8x3**		**12x2**		**24x1**	
SZBLOCS = 24	92445	67907	61309	51735		51491		51436		51414		51392	
NBBLOCS x	**1x36**	**2x18**	**3x12**	**4x9**		**6x6**			**9x4**	**12x3**	**18x2**		**36x1**
SZBLOCS = 36	92445	67907	61309	51515		51392			51403	51392	51392		51392
NBBLOCS x	**1x48**	**2x24**	**3x16**	**4x12**		**6x8**		**8x6**	**12x4**	**16x3**		**24x2**		**48x1**
SZBLOCS = 48	92445	67907	61309	51515		51392		51392	51392	51392		51392		51392

Stratégie INT
3 UGM

ia=1
tp=489
tx=500
NB=100

NBBLOCS x	1x1												
SZBLOCS = 1	92445												
NBBLOCS x	1x2	2x1											
SZBLOCS = 2	92445	67906											
NBBLOCS x	1x3		3x1										
SZBLOCS = 3	92445		61309										
NBBLOCS x	1x4	2x2		4x1									
SZBLOCS = 4	92445	67906		58012									
NBBLOCS x	1x5				5x1								
SZBLOCS = 5	92445				56651								
NBBLOCS x	1x6	2x3	3x2			6x1							
SZBLOCS = 6	92445	67906	61309			54810							
NBBLOCS x	1x7						7x1						
SZBLOCS = 7	92445						54100						
NBBLOCS x	1x2	2x4		4x2				8x1					
SZBLOCS = 8	92445	67906		59097				53712					
NBBLOCS x	1x9		3x3						9x1				
SZBLOCS = 9	92445		61309						53742				
NBBLOCS x	1x12	2x6	3x4	4x3		6x2				12x1			
SZBLOCS = 12	92445	67906	61309	65613		62034				55811			
NBBLOCS x	1x18	2x9	3x6			6x3			9x2		18x1		
SZBLOCS = 18	92445	67906	61309			62681			56424		54917		
NBBLOCS x	1x24	2x12	3x8	4x6		6x4		8x3		12x2		24x1	
SZBLOCS = 24	92445	67906	61309	63745		58201		55246		54824		54791	
NBBLOCS x	1x36	2x18	3x12	4x9		6x6			9x4	12x3	18x2		36x1
SZBLOCS = 36	92445	67906	61309	61498		57815			55267	55359	55245		55245
NBBLOCS x	1x48	2x24	3x16	4x12		6x8		8x6	12x4	16x3		24x2		48x1
SZBLOCS = 48	92445	67906	61309	61052		57300		56345	55245	55245		55245		55245

Stratégie SCP
3 UGM

la=1
tp=489
tx=500
NM = 100

Stratégie CSP
3 UGM
jeu d'essai 11
ia=1
tp=489
tx=500

NBBLOCS x	1x1											
SZBLOCS = 1	92445											
NBBLOCS x	1x2	2x1										
SZBLOCS = 2	92445	67882										
NBBLOCS x	1x3		3x1									
SZBLOCS = 3	92445		61291									
NBBLOCS x	1x4	2x2		4x1								
SZBLOCS = 4	92445	67882		58003								
NBBLOCS x	1x5				5x1							
SZBLOCS = 5	92445				56642							
NBBLOCS x	1x6	2x3	3x2			6x1						
SZBLOCS = 6	92445	67882	61291			54799						
NBBLOCS x	1x7						7x1					
SZBLOCS = 7	92445						54093					
NBBLOCS x	1x2	2x4		4x2				8x1				
SZBLOCS = 8	92445	67882		59095				53701				
NBBLOCS x	1x9		3x3						9x1			
SZBLOCS = 9	92445		61291						53733			
NBBLOCS x	1x12	2x6	3x4	4x3		6x2				12x1		
SZBLOCS = 12	92445	67882	61291	65612		62032				55802		
NBBLOCS x	1x18	2x9	3x6			6x3			9x2		18x1	
SZBLOCS = 18	92445	67882	61291			62680			57268		54911	
NBBLOCS x	1x24	2x12	3x8	4x6		6x4		8x3		12x2		24x1
SZBLOCS = 24	92445	67882	61291	63745		58198		55243		54820		54787
NBBLOCS x	1x36	2x18	3x12	4x9		6x6			9x4	12x3	18x2		36x1
SZBLOCS = 36	92445	67882	61291	61498		57815			55265	55358	55243		55243
NBBLOCS x	1x48	2x24	3x16	4x12		6x8		8x6	12x4	16x3		24x2	
SZBLOCS = 48	92445	67882	61291	61052		57300		56345	55243	55243		55243	

ANNEXE F. : PROGRAMMATION DE NERF-C

J.1. ENVI

```
    COMPILATION LISTING OF SEGMENT:
          >user_dir_dir>OMPHALE>JMPlace>modula2>src>ENVI.def
 SMILER-2  -  MODULA-2 CROSS COMPILER VERSION 2.      12/18/89  13:49:30

000001
000002 DEFINITION MODULE ENVI ;
000003  FROM SYSTEM IMPORT WORD ;
000004  EXPORT QUALIFIED
000005    Smoduleid,   Sserviceid,
000006    Sanswer,     Smyself,
000007    Smaxservice, Slien,
000008    DUPLICATELINK, SETNOUSE, SETONEUSE, GETLINK ;
000009  CONST Sanswer = 1 ;
000010 CONST Smaxservice = 15 ;
000011  CONST MaxVersion = 65535 ;
000012 CONST MaxEnvi = 2047 ;
000013  TYPE IndEnvi    = [0..MaxEnvi] ;
000014 TYPE IndVersion = [0..MaxVersion] ;
000015  TYPE Smoduleid = RECORD  (* Type des noms locaux de liens *)
000016                   Version : IndVersion ;
000017                   Position : IndEnvi ;
000018                   END ;
000019      Pmoduleid = POINTER TO Smoduleid ;
000020  VAR Smyself : Smoduleid (* := (0,0) *) ;
000021  TYPE Sserviceid  = [1..Smaxservice] ;  (* 15 Quais max par module *)
000022 TYPE ModuleUniqueName = CARDINAL ;
000023 TYPE Slien = RECORD
000024             OneUse : BITSET ;
000025             NoUse  : BITSET ;
000026             LinkTo : ModuleUniqueName ;
000027             Duplicable : BOOLEAN ;
000028             END ;
000029  PROCEDURE DUPLICATELINK (SOURCE : Smoduleid ;
000030                           VAR DEST : Smoduleid ;
000031                           DUPLICABLE : BOOLEAN) ;
000032  PROCEDURE SETONEUSE (QUAIS: BITSET; M: Smoduleid) ;
000033 PROCEDURE SETNOUSE (QUAIS: BITSET; M: Smoduleid) ;
000034 PROCEDURE GETLINK (M : Pmoduleid; VAR L : Slien) ;
000035 END ENVI .
```

VALIDE

```
000027         PROCEDURE VALIDE (M: Smoduleid) : BOOLEAN ;
000028        BEGIN  86
000029 000018 Position := M.Position ;                                    14
000030 000022 RETURN ((Position > 0)                          AND        378
```

```
000031 000022          (Position <= MaxEnvi)                              AND
000032 000022          (Environment [Position].Status = Used)             AND
000033 000022          (M.Version = Environment [Position].Version)) ;
000034         END VALIDE ;
```

SETONEUSE

```
000074          PROCEDURE SETONEUSE (QUAIS: BITSET; M: Smoduleid) ;
000075         BEGIN
000076 000018 SYSTEMRESULT := NoError ;
000077 000046 IF VALIDE (M)
000078 000046    THEN Environment [Position].OneUse :=
000079 000046              Environment [Position].OneUse + QUAIS ;
000080 000084    ELSE SYSTEMRESULT := BadLink ;
000081 00008C    END ;
000082         END SETONEUSE ;
```

SETNOUSE

```
000083          PROCEDURE SETNOUSE (QUAIS: BITSET; M: Smoduleid) ;
000084         BEGIN
000085 000018 SYSTEMRESULT := NoError ;
000086 000046 IF VALIDE (M)
000087 000046    THEN Environment [Position].NoUse :=
000088 000046              Environment [Position].NoUse + QUAIS ;
000089 000084    ELSE SYSTEMRESULT := BadLink ;
000090 00008C    END ;
000091         END SETNOUSE ;
```

GETLINK

```
000092         PROCEDURE GETLINK (M : Pmoduleid ; VAR L : Slien) ;
000093         BEGIN  17
000094 000004 SYSTEMRESULT := NoError ;                                    14
000095 00000A IF VALIDE (M^)                                              165
000096 00000A    THEN
000097 000032       L.NoUse := Environment [Position].NoUse ;              151
000098 000052       L.OneUse := Environment [Position].OneUse ;            145
000099 000070       L.LinkTo := Environment [Position].LinkTo ;            151
000100 000090       L.Duplicable := Environment [Position].Duplicable ;    151
000101 0000B6    ELSE     30
000102 0000B6       SYSTEMRESULT := BadLink ;
000103 0000BE    END ;    30
000104         END GETLINK ;
```

J.2. ZEMIS

CONCAT

```
000025          PROCEDURE Concat (VAR x : ARRAY OF WORD) ;
000026         BEGIN  12
000027 000004 i := 0 ;    14
```

```
000028 00000A WHILE (i < HIGH (x)) DO                                    37
000029 00001A     Memoire [Haut] := x [i] ;                             122
000030 000046     INC (Haut) ;                                           16
000031 00004C     INC (i) ;                                              16
000032 000054     END ;   6
000033        END Concat ;                                               26
```

SSYSTSEND

```
000034        PROCEDURE Ssystsend (VAR Message : ARRAY OF WORD ;
000035                             ModuleDest : Smoduleid ;
000036                             ServiceDest : Sserviceid) : Smessageid ;
000037        BEGIN  90
000038 000018 INC (LastMessageId) ;                                      16
000039 00003E Concat (LastMessageId) ;                                  135
000040 00003E Tempo := 2                              (* Taille message   *)  123
000041 00003E          + 2                            (* Nbre de liens    *)
000042 00003E          + NbLien * (2 + TSIZE(Slien)) (* Liens effectifs  *)
000043 00003E          + 2 + HIGH (Message) ;         (* Message effectif *)
000044 00005A  Concat (Tempo) ;                                         135
000045 00007A FOR NL := 0 TO NbLien - 1 DO                               64
000046 00007A (*   Lien := ADR (Message [TabLien[NL]]) ; *)
000047 00009A    PSLien :=      Message [TabLien[NL]]  ;                111
000048 0000C6    GETLINK (Lien, ActLink) ;                             146
000049 0000E8    Concat (ActLink) ;                                    135
000050 000108    END ;    63-49
000051 00011C Concat (Message) ;                                       146
000052 000140 NbLien := 0 ;                                             14
000053 000146 RETURN LastMessageId ;                                    56
000054        END Ssystsend ;
```

BIBLIOGRAPHIE.

[Atamenia86] Atamenia A., Fouret D.
Compilateur modula2. Génération de code pour un microprocesseur Z8001,
Rapport de mémoire de D.E.A. Université de Lille 86.

[Bekkers84] Bekkers Y., Leroy H. *Un système de production d'assemebleurs: GAGE*, IRISA, Avril 1984

[Cheriton82] Cheriton D.R.,
«The Thoth System : Multi-process Structuring and Portability»,
North-Holland, 1982.

[Delattre79] Delattre E.
Contribution à la répartition d'un système à structure de domaines sur un réseau local de micro-ordinateurs faiblement couplés,
Thèse de Docteur-Ingénieur, Université de Lille, 1979.

[Delattre84] Delattre E., Geib J.M.
A distributed and protected system, which maintains modularity up to execution,
Communication à «Hardware supported implementation of concurrent languages in distributed systems», IFIP Working group 10.3, Bristol, Mars 1984.

[Delattre85a] Delattre E., Geib J.M., Place J.M.,
Two distributed hardware mechanisms for VLSI component sharing.
Communication à «Mini and micro computers and their applications», Gerona, Juin 1985.

[Delattre85b] Delattre E., Geib J.M., Mehaut J.F.,
The OMPHALE distributed system architecture: communication and concurrency issues,
Communication à «Parallel computing 85», Berlin 1985.

[Delattre88a] Delattre E., Geib J.M., Mehaut J.F., Place J.M., *Deux impllantations de l'architecture de système réparti OMPHALE,* Communication à «IASTED International Symposium Applied Informatics», Grindelwald, Février 1988.

[Delattre89a] Delattre E., *Un bilan du projet OMPHALE. Le développement d'un noyau de système réparti.* Mémoire d'Habilitation à diriger des recherches, Université de Lille, 1989.

[Delattre89b] Delattre E, Geib J.M., *OMPHALE, an active Object-based Distributed Operating System,* Communication à IFIP WG 10.3 Working conference on decentralized Systems, Lyon, Décembre 1989.

[Geib86] Geib J.M., *OMPHALE: système réparti orienté objet,* Communication aux 3èmes journées d'étude Langages Orientés Objets, Paris, Janvier 1986.

[Geib89] Geib J.M., *l'architecture du système réparti OMPHALE.* Thèse de doctorat, Université de Lille, 1989.

[Goldberg83] Goldberg A, Robson D. «*SMALLTALK-80. The language and its implementation*», Addison-Wesley, 1983.

[Lampson80] Lampson B.W., Redell D.D. *Experiences with processes and monitors in Mesa* in «Communication of the ACM», Vol. 23, N° 2, February 1980, Pages 105-117.

[Lebail86] Lebail F., *Réalisation d'un noyau minimal de multiprogrammation destiné au Site0 d'OMPHALE,* Rapport de mémoire de D.E.A., Université de Lille, 1986.

[Levy84] Levy H.M. «Capability-Based Computer Systems», Digital Equipment Corporation, 1984.

[Méhaut89] Mehaut J.F. *Une implantation de l'architecture de système réparti OMPHALE,* Thèse de doctorat, Université de Lille, 1989.

[Meyer88] Meyer B., «Obect-oriented Software Construction». Prentice-Hall, 1988.

[Organick83] Organick E.I.
A programmer's view of the Intel 432 system.
Mac Graw-Hill 1983.

[Place88] Place J.M., *Préprocesseur omphale: Guide d'utilisation*, Laboratoire d'Informatique Fondamentale de Lille, Note interne, 1988.

[Pujolle] Pujolle R.A.I.R.O. vol 13 N° 2 1979

[Stroustrup81] Stroustrup B. *'Long Return' : A Technique for Improving the Efficiency of Inter-module Communication in «Software - Practice and experience»*, Vol. 11, 1981.

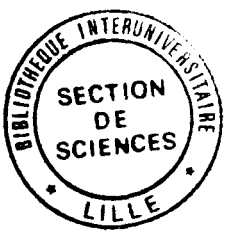
[Tanenbaum87] Tannenbaum A.S. «Operating Systems. Design and implementation», Prentice-Hall, 1987.

[Wirth84] Wirth N. «Programmer en MODULA-2», Presses Polytechniques Romandes, 1984.

[Yonezawa87] Yonezawa A., Yokotte M. (Editors)
Object oriented concurrent programminbg.
The MIT Press, 1987.

[Zilog81] Zilog CORP. «Z8000 CPU -Technical Manual», 1980.

[Zilog82] Zilog CORP. «Z8010 - Z8000 Z-MMU Memory Management Unit - Product specification», 1981.



Ce travail été réalisé dans le cadre du projet de recherche OMPHALE . Ce projet présente deu

- OMPHALE est une architecture répartie. L'élément de base d'un logiciel destiné a êt sous OMPHALE est le module. Un module est à la fois un processus séquentiel, un objet au sens du modèle objet. Lors de l'exécution, les modules communiquent e l'aide de messages. Nous avons défini un sur-ensemble de MODULA2 qui la programmation sous OMPHALE .
- D'autre-part, nous avons développé une architecture matérielle bi-processeur qui supporte de manière efficace de nombreux changements de contexte, permettant ainsi l'exécution de programmes hautement modulaires. Cette organisation offre également un espace mémoire protégé (adressage segmenté) au logiciel d'application. Un prototype baptisé Site zéro a été réalisé.

L'évaluation des performances de l'organisation matérielle a été faite selon deux méthodes complémentaires:

- Une simulation a permis de chiffrer le gain de performances en fonction des paramètres du matériel mis en œuvre. Nous décrivons la constitution du programme de simulation. Les résultats obtenus sont présentés et analysés.
- Nous avons défini une implantation d'OMPHALE sur le site zéro que nous avons partiellement réalisé en MODULA2, puis imaginé une application typique (de gestion de pile d'entiers). Nous avons alors pu chiffrer précisement les durées d'exécution du système d'exploitation et de l'application. Enfin, les résultats fournis sont confrontés avec ceux de la simulation.