ACADÉMIE DE PARIS

UNIVERSITÉ PIERRE ET MARIE CURIE (PARIS-VI)

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS-VI

SPÉCIALITÉ : **INFORMATIQUE**

SOUS LA DIRECTION DU :

Pr. ALAIN GREINER

PRÉSENTÉE PAR :

GILLES-ERIC DESCAMPS

POUR OBTENIR LE TITRE DE :

DOCTEUR DE L'UNIVERSITÉ PARIS-VI

SUJET DE LA THÈSE :

MÉTHODE DE DISTRIBUTION HIÉRARCHIQUE D'OUTILS DE VÉRIFICATION DE CIRCUITS INTÉGRÉS VLSI SUR UN RÉSEAU DE STATIONS DE TRAVAIL, APPLICATION À UN VÉRIFICATEUR DE RÈGLES DE DESSIN.

SOUTENUE LE 13 JUIN 1996 À 11H DEVANT LE JURY COMPOSÉ DE :

ALAIN GUYOT	RAPPORTEUR
PATRICE QUINTON	RAPPORTEUR
ISABELLE DEMEURE	EXAMINATRICE
CLAUDE GIRAULT	EXAMINATEUR
MICHEL ROBERT	EXAMINATEUR
ALAIN GREINER	EXAMINATEUR

Remerciements

Je remercie le Professeur Alain GREINER, directeur du D.E.A. Architecture des Systèmes Intégrés et Micro-Electronique pour ses encouragements constants et surtout la qualité de ses remarques. Je remercie aussi le directeur du MASI, Alain GREINER, pour m'avoir si souvent rappelé de me consacrer à la rédaction de ce mémoire.

Que les membres du Jury trouvent ici l'expression de ma profonde gratitude :

Les rapporteurs : Alain GUYOT, Professeur à l'Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble, et Patrice QUINTON, Professeur à l'Institut de Recherches en Informatique et Systèmes Aléatoires de Rennes, pour le soin qu'ils ont apporté à l'examen de ce document.

Les examinateurs : Isabelle DEMEURE, Maître de Conférences à l'Ecole Nationale Supérieure des Télécommunications, Claude GIRAULT, Professeur à l'Université Paris VI, Laboratoire MASI, et Michel ROBERT, Professeur au Laboratoire d'Informatique, de Robotique, et de Micro-Electronique de Montpellier, pour l'intérêt qu'ils ont porté à ce projet.

Je tiens à remercier tous les membres de l'équipe Architecture du Laboratoire MASI pour la bonne ambiance et la dynamique de travail, ainsi que pour leur patience et leur compréhension durant les phases d'expérimentation de ce mémoire. Plus particulièrement, je tiens à remercier Frédéric PÉTROT pour nous avoir permis de profiter d'une chaîne complête d'outils pour la CAO de circuits VLSI : Alliance, dont les sources sont disponibles et modifiables. Je remercie aussi Nicole FOUQUE, Nizar ABDALLAH, Olivier FLORENT, et Alexandre FENYÖ, Ingénieurs Systèmes du Laboratoire, qui ont su me décharger des soucis liés à l'utilisation d'un réseau. Enfin, je remercie Bertil FOLLIOT, Maître de Conférence à l'Université Denis Diderot, pour son intérêt soutenu pour mon travail ainsi que sa disponibilité.

Une pensée particulière à Mai-Trâm, ma femme, et Kim-Yên, mon fils, qui ont su me soutenir durant toutes ces années.

Enfin, je suis plus que reconnaissant envers mes parents qui m'ont donné le goût des études, et m'ont permis d'y réussir grâce à leur soutien tant moral que matériel.

À mes parents,

a

Résumé

Cette thèse introduit une méthode façilitant la parallélisation des algorithmes des outils de vérification de circuits VLSI.

Après avoir présenté une méthodologie de conception symbolique de circuits intégrés, nous retenons le Vérificateur de Règles de Dessin (VRD) comme archétype des outils de vérification. L'architecture parallèle support de notre méthode est celle d'un réseau de stations de travail assemblées en une machine parallèle virtuelle grâce au logiciel PVM. Notre méthode s'appuie sur la structure hiérarchique du circuit comme guide explicite du partitionnement pour la distribution des tâches. Afin de détacher les gains de performance obtenus par les différentes techniques, nous avons fait évoluer un VRD pour qu'il fonctionne en mode hiérarchique, puis en mode distribué. La robustesse de l'application n'est pas diminuée lors de la parallélisation grâce à un fonctionnement incrémental qui évite de recalculer les étapes intermédiaires. Les performances obtenues par cette méthode sont évaluées à l'aide d'un jeu de circuits qui ont tous été envoyés en fabrication. La taille de deux d'entre eux est proche du million de transistors.

a

Abstract

This thesis introduces a method to ease the parallelisation of the algorithms of VLSI circuits verification tools.

After presenting a methodology of symbolic conception of integrated circuits, we retain the Design Rule Checker (DRC) as the archetypal verification tool. The parallel architecture used for our method is a network of workstations tied together by the PVM software. Our method relies on the hierarchical structure of the circuit as an explicit guide of partitionnement for the tasks distribution. In order to clearly show the gains of each technique, we had a DRC evolved into a hierarchical one, then into a distributed one. Application robustness is not compromised, thanks to an incremental behaviour, that prevents from recalculating intermediate results. Performances obtained by this method are evaluated through a set of circuits that all have been sent to fabric. Size of two of them reaches the million of transistors.

a

1) Introduction générale

1.1) Contexte

Le temps écoulé entre la spécification fonctionnelle d'un *ASIC* et la disponibilité des masques permettant la fabrication est critique. La plupart des recherches actuelles se focalisent sur les outils de synthèse, qui permettent d'obtenir un schéma en portes à partir d'un comportement spécifié dans un langage de haut niveau. Ces outils de synthèse contribuent à réduire le temps de conception proprement dite d'un circuit. Toutefois, leur résultat doit être validé : Les outils d'abstraction fonctionnelle et de preuve formelle permettent la vérification fonctionnelle. Les analyseurs de performance vérifient les caractéristiques temporelles. Les vérificateurs de règles de dessin (VRD) permettent au concepteur de garantir que les masques qu'il a dessiné respectent les spécifications du fondeur. Les simulateurs de fautes assurent le calcul du taux de couverture des vecteurs de tests. Or, la complexité des circuits croissant plus vite que la puissance des stations de travail, le coût de ces vérifications devient prohibitif. (Les derniers processeurs de signaux (*DSP*) de chez Texas Instruments atteignent 20 millions de transistors).

A la différence des phases de conception, dont le coût est surtout un coût humain, le coût des phases de vérification n'est qu'un coût de machines.

1.2) Objectifs

On souhaite réduire la durée des phases de vérification en parallélisant les algorithmes, et en distribuant le traitement sur un réseau de stations de travail.

Pour atteindre cet objectif, trois conditions nous semblent nécessaires.

- la définition d'une méthode, s'appuyant éventuellement sur des composants logiciels préexistants.
- la démonstration de sa faisabilité pratique par le développement d'un outil logiciel représentatif de la classe d'applications visée.
- la vérification des gains de performance espérés par l'expérimentation de cet outil sur plusieurs circuits complexes.

1.3) Présentation du mémoire

Le chapître 2 est décomposé en deux parties. Comme certains de nos lecteurs ne sont pas familiers du domaine de la conception des circuits VLSI (*Very Large Scale Integration* : intégration à très grande échelle), nous exposons rapidement les différentes étapes de la construction d'un circuit VLSI en ne nous attardant que sur les phases de vérification. Après avoir justifié le choix du Vérificateur de Règles de Dessin (VRD) comme archétype des outils de vérifications, nous exposons quelques problèmes potentiels de l'usage de la hiérarchie comme guide du partitionnement.

Nous présentons dans le chapître 3 l'architecture de quelques Vérificateurs de Règles de Dessin, ainsi que quelques boites à outils logicielles destinées à faciliter la distribution de programmes en général sur un réseau de stations.

Nous souhaitons expérimenter une parallélisation des tâches de vérification s'appuyant sur la structure hiérarchique du circuit, définie par le chef d'un projet, lorsqu'il répartit les tâches entre plusieurs concepteurs afin de paralléliser les tâches de conception. La méthode générale de parallélisation et de distribution, qui peut être appliquée à différents outils de vérification, est présentée dans le chapître 4.

Pour démontrer le bien-fondé de notre méthode, nous avons fait évoluer un outil de vérification de règles de dessin pour qu'il fonctionne en mode hiérarchique puis en mode distribué. Les principes du VRD hiérarchique sont présentés dans le chapître 5.

Afin de bien détacher les gains de performances obtenus par les diverses techniques, nous décrivons l'implémentation des trois vérificateurs dans le chapître 6 :

- le VRD à plat
- le VRD hiérarchique
- le VRD distribué

Les circuits utilisés pour valider ces VRDs ainsi que les résultats expérimentaux sont présentés dans le chapitre 7.

a

2) Problématique

Ce chapitre a pour objet de préciser le cadre de notre étude. Notre travail vise des techniques de parallélisation de gros logiciels de CAO pour la conception VLSI (*Very Large Scale Integration* : intégration à très grande échelle). Nous commençons par présenter la méthode générale de conception d'un circuit VLSI, en ne nous attardant que sur ce qui concerne notre étude. Après avoir montré que les phases de vérifications des circuits sont les plus coûteuses en temps machine, nous retiendrons une application représentative des vérifications de circuits VLSI, de manière à démontrer tout au long de ce manuscrit la pertinence des techniques retenues.

2.1) Méthode de conception de circuits VLSI

La description d'un circuit intégré se fait à trois niveaux qui correspondent à différentes vues du même circuit :

- la vue comportementale : Elle sert de spécification et doit être simulable.
- la vue structurelle : assemblage hiérarchique de composants s'appuyant généralement sur des bibliothèques de cellules précaractérisées.
- la vue physique : description hiérarchique correspondant au dessin des masques de fabrication du silicium.

Ces trois vues correspondent à trois bases de données différentes nécessaires dans le processus de conception :

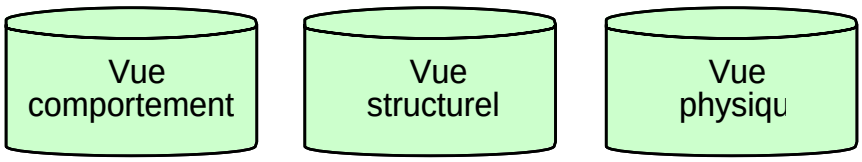

Figure F1. Les trois vues d'un circuit VLSI

La conception se fait séquentiellement. Afin de permettre une conception "zéro-défaut", chacune des phases de construction d'une vue est suivie par une phase de vérification (en terme de fonctionnalités) par rapport à la vue précédente : Les spécifications initiales du projet permettent de définir un jeu de stimuli pour valider la vue comportementale. Ces stimuli sont également utilisés pour valider la vue structurelle, qui, à son tour, permet de valider la vue physique. On génère par ailleurs un jeu de tests de fabrication qui sera remis au fondeur avec la vue physique du circuit afin de valider les circuits qui sortent de fabrication. Ces tests ont un but différent de ceux de conception, car ils visent à tester le maximum de composants du circuits en un minimum de temps.

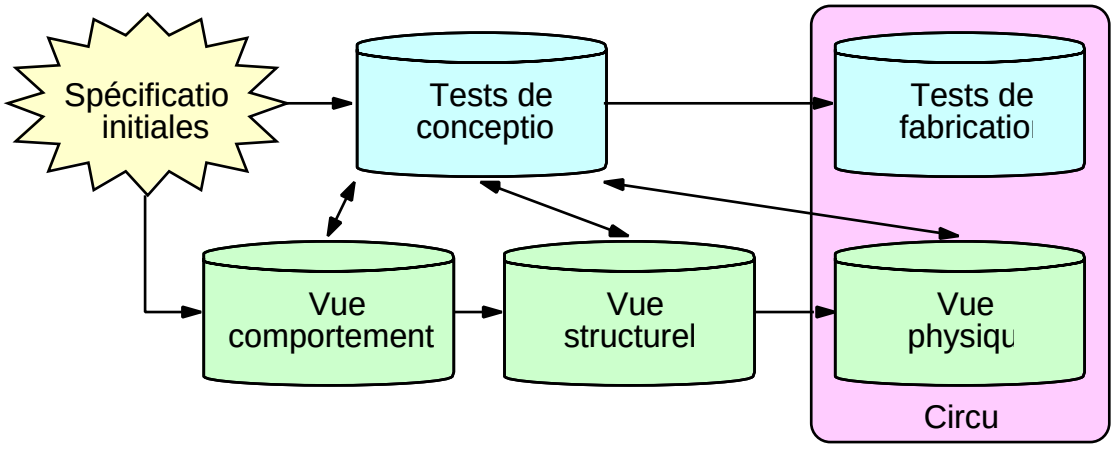

Figure F2. Conception d'un circuit VLSI

Les deux vues structurelle et physique possèdent généralement une organisation hiérarchique. A titre d'exemple, la vue physique du processeur des stations de travail RS/6000 d'IBM se divise en :

- circuits (c'est un processeur multi-chips)
- macro-cellules (caches, arithmétique, controle, ...)
- cellules (nand2, ...)
- transistors
- rectangles d'un niveau de métal du process cible (ALU1, ...)

La vue comportementale correspond à une description du circuit capable de fournir les bonnes réponses à des stimuli extérieurs. La vue structurelle est obtenue à l'issue d'une phase de conception descendante où l'on construit un schéma hiérarchique dont les feuilles sont des éléments de biblio-

thèques. La vue physique est générée au cours d'une phase de conception ascendante où l'on assemble progressivement le circuit en commençant par les feuilles.

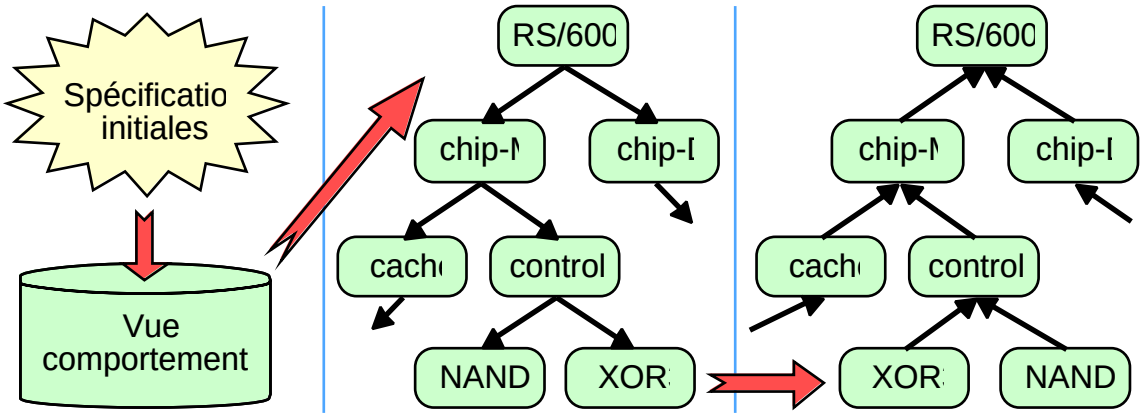

Figure F3. hiérarchie de conception d'un circuit VLSI

Les hiérarchies des vues structurelle et physique sont, en théorie, identiques. En pratique, certains outils de placement routage ont des contraintes qui nécessitent l'ajout ou la suppression de niveaux intermédiaires de hiérarchie.

La conception est influencée par divers facteurs :

- Performances : (en vitesse, en consommation, en fonctionnalité, ...)
- Taille du chip : (c'est aussi le coût du chip)
- Temps de conception : (Planning et coût de fabrication)
- Testabilité : (Planning et coût de fabrication)

La conception d'un circuit est un compromis permanent entre ces divers facteurs. Les outils d'aide à la conception de VLSI (CAO-VLSI) doivent réduire la complexité d'un circuit, accroitre la productivité des concepteurs, et garantir un produit qui marche. Un certain nombre de stratégies sont prises en comptes par ces outils :

2.1.1) Stratégie de conception

La stratégie utilisée pour maitriser la complexité dans les outils de CAO pour VLSI est la hiérarchie, l'exploitation de la régularité, et la recherche de la modularité.

2.1.1.1) Hiérarchie

L'usage de la hiérarchie implique la division d'un module en sous-modules. Cette opération est répétée sur chacun des sous-modules jusqu'à atteindre des composants existants dans des bibliothèques.

La hiérarchie est employée à plusieurs niveaux. Un concepteur l'utilise pour réduire son travail, tandis qu'un chef de projet l'utilise pour paralléliser les tâches entre les concepteurs.

Il existe plusieurs façons de traduire un comportement en un découpage hiérarchique. Le processus de conception fonctionne avec un découpage en boites, qui à leur tour, contiennent d'autres boites. Voici un exemple où des contraintes topologiques obligent à redéfinir la hiérarchie structurelle :

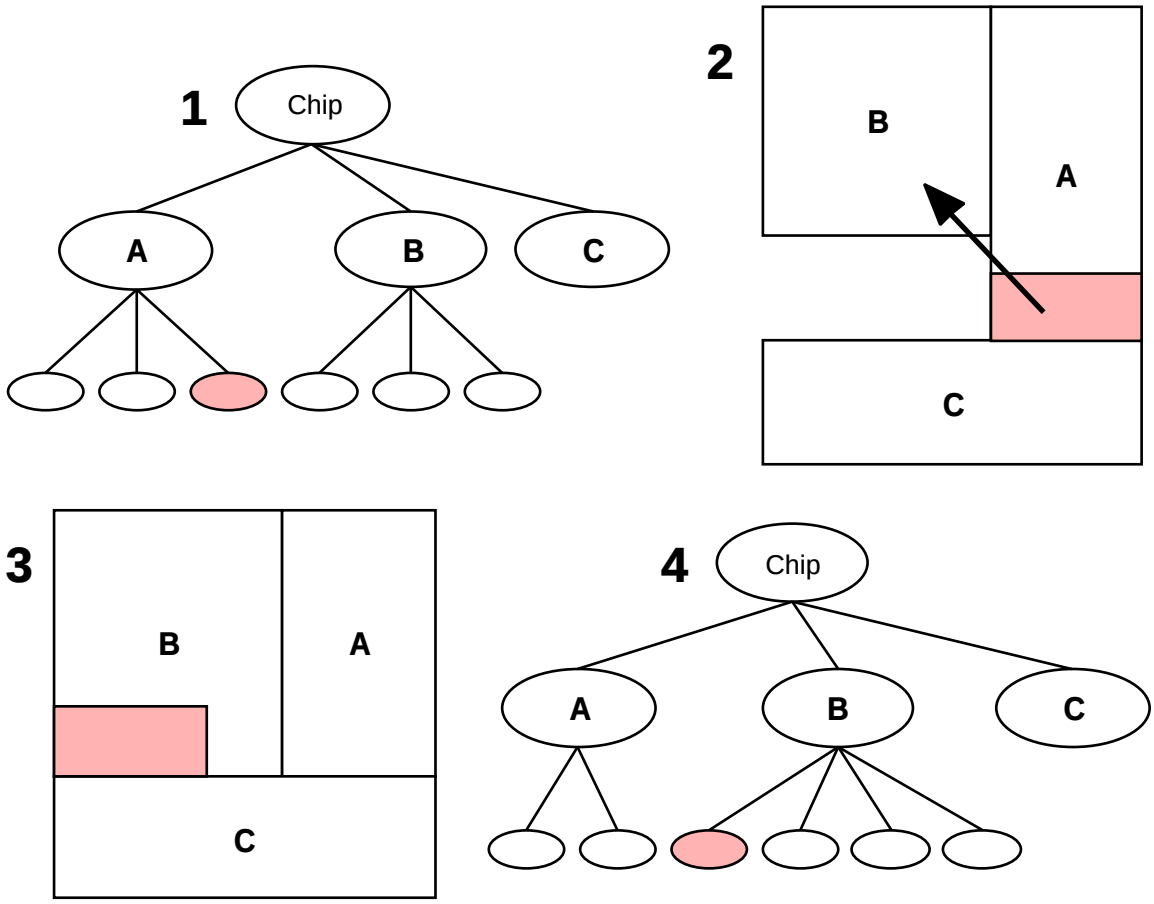

Figure F4. Influence de la hiérarchie physique sur la hiérarchie structurelle

2.1.1.2) Régularité

La hiérarchie ne suffit pas à elle seule à résoudre le problème de la complexité des données manipulées en VLSI. En gardant présent à l'esprit la notion de régularité, un concepteur essaie d'organiser la hiérarchie de façon à exploiter la répétition de fonctions identiques. A chaque niveau de la hiérarchie, la régularité peut exister. Au niveau des portes, une même porte logique est instanciée un grand nombre de fois. A des niveaux plus hauts, on peut construire des architectures qui utilisent des opérateurs optimisés répliqués (mémoires, opérateurs arithmétiques, etc). L'usage de la régularité facilite aussi bien la conception proprement dite que la vérification.

2.1.1.3) Modularité

La modularité rajoute à la hiérarchie et à la régularité la contrainte que les sous-modules définis aient des interfaces et des fonctionnalités “bien-formées”. Ces interfaces et les sémantiques associées doivent être définies d'une manière non-ambigüe. Chaque module peut ainsi faire l'objet d'une procédure de test indépendante.

D'autre part, l'intérieur du bloc doit aussi être “bien-formé”. A l'instar de la programmation structurée qui recommande l'usage de trois constructions de base : la concaténation, l'itération et la sélection conditionnelle. L'usage généralisé des *Hardware Description Language (HDL)* facilite la modularité des schémas.

Localité :

Une modularité réussie a comme conséquence des interfaces bien-connues pour chaque module. Ceci signifie que l'intérieur du module n'est d'aucune importance pour un élément du circuit qui est situé à l'extérieur. Dans le génie logiciel, ceci s'apparente aux fonctions de la librairie de base du langage C. Seule, l'interface des fonctions (fichiers `.h`) est disponible.

2.1.2) Outils de vérification

Ces outils sont extrêmement importants de par le coût de fabrication d'un circuit. On ne peut envoyer un circuit en fabrication sans être sûr de sa validité, d'où les vérifications.

La conception d'un circuit n'est pas un processus linéaire. C'est un processus itératif, fait de retours en arrière permanents. Des erreurs apparaissent à chaque première étape de vérification, provoquant ainsi un retour en arrière et des modifications dans la vue physique ou dans la vue structurelle, voire dans la spécification comportementale.

Le niveau d'abstraction, toujours plus grand, permis par les outils de conception, repousse un grand nombre de vérifications dans la phase finale avant envoi du circuit au fondeur. Par exemple, le concepteur ne contrôle que difficilement la notion de temps dans une description comportementale synthétisable. Il faut alors utiliser un analyseur temporel qui donnera la chaine longue du circuit et ses caractéristiques d'une manière précise.

2.1.3) Coût des vérifications

Les étapes de vérification représentent plus de la moitié du temps de développement d'un circuit complexe [5]. Voici à titre d'exemple la répartition des 27,5 mois cumulés nécessaires aux différentes phases de la conception du processeur 32-bit Toshiba TX1 de 460.000 transistors [35]:

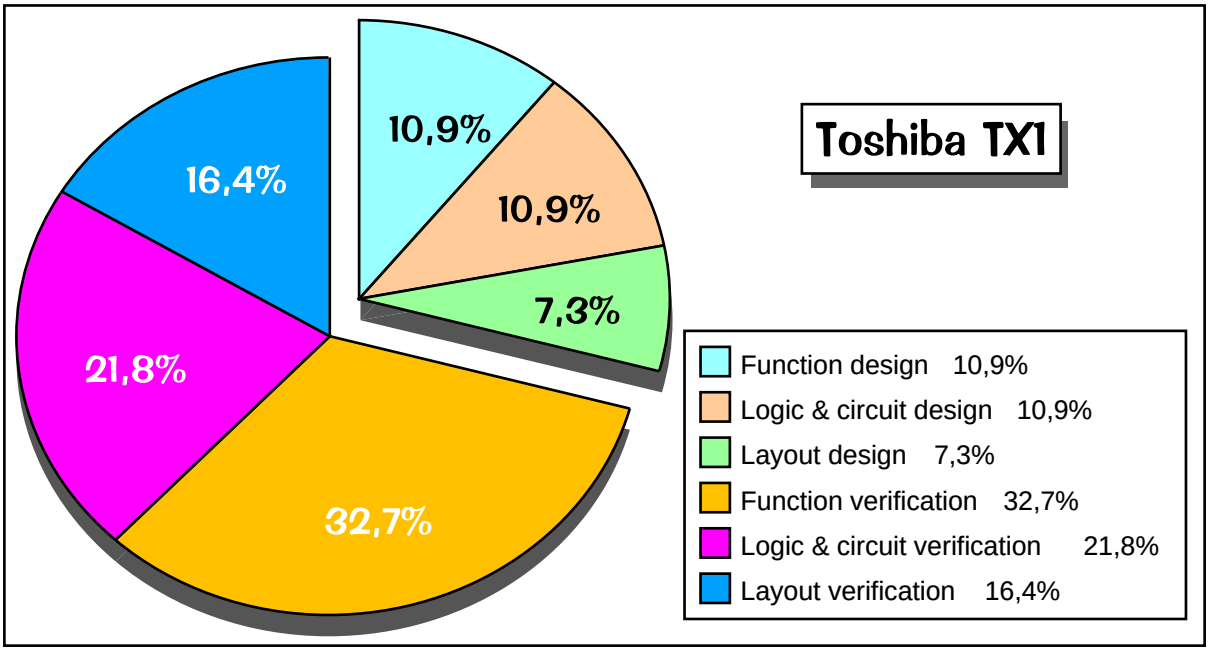

Figure F5. Ratio des différentes étapes de la conception du TX1

La partie éclatée du camembert regroupe les phases de conception. Le reste du camembert n'est que la durée des étapes de vérification. On peut remarquer que les phases de vérification représentent plus de **70%** du temps de conception du circuit.

M. Tabusse [28] indique aussi un ratio de 70% pour la vérification du modèle d'un ASIC, et de 30% pour son écriture.

Voici la répartition des 122 mois cumulés nécessaires aux différentes phases de la conception du processeur VLIW StaCS de 875.000 transistors :

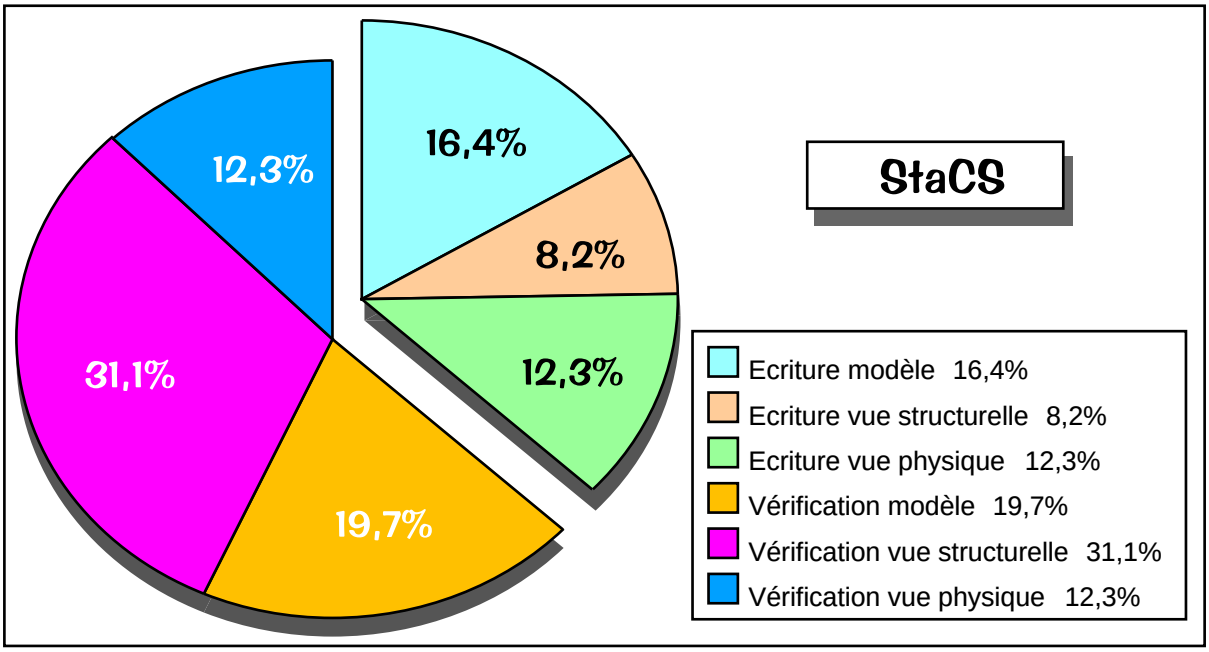

Figure F6. Ratio des différentes étapes de la conception du *StaCS*

Le processeur StaCS conçu au MASI a bénéficié d'une méthodologie de conception avancée, décrite dans [5]. Les phases de vérification représentent **63%** du temps de conception du circuit.

2.1.3.1) Vérifications plus nombreuses

Les phases de vérification sont complexes car elles manipulent simultanément plus de données. En effet, les vérifications finales portent sur le circuit complet après assemblage de tous les modules : Il est en effet nécessaire de vérifier les interactions entre les différents modules, que ce soit au niveau fonctionnel, temporel ou physique.

Il n'est pas exceptionnel que la phase finale de vérification des règles de dessin sur tout un grand circuit prenne jusqu'à huit jours (192h) de temps CPU. En effet, l'algorithme du VRD a une complexité en $O(n \log(n))$.

Le coût des phases de vérification est essentiellement un coût de ressources machines plutôt qu'un coût de ressources humaines.

2.1.3.2) Vérifications plus complexes

Plus la complexité des VLSI est grande, plus il faut de temps pour vérifier. Pour la plupart des algorithmes de vérification, l'accroissement du temps nécessaire n'est pas linéaire avec l'accroissement du nombre de transistors. Afin de rester avec des temps de réponse raisonnables, il faut paralléliser.

Le temps de calcul n'est pas le seul problème. La complexité croissante provoque aussi une explosion de l'espace de travail (RAM) de la machine. La mémoire vive est une ressource coûteuse. Cet accroissement de l'espace de travail induit un coût caché : Une station de travail dont la mémoire est extensible à 512 Mo coûte beaucoup plus cher qu'une station dont la mémoire n'est extensible que jusqu'à 64 Mo à capacité de mémoire égale :

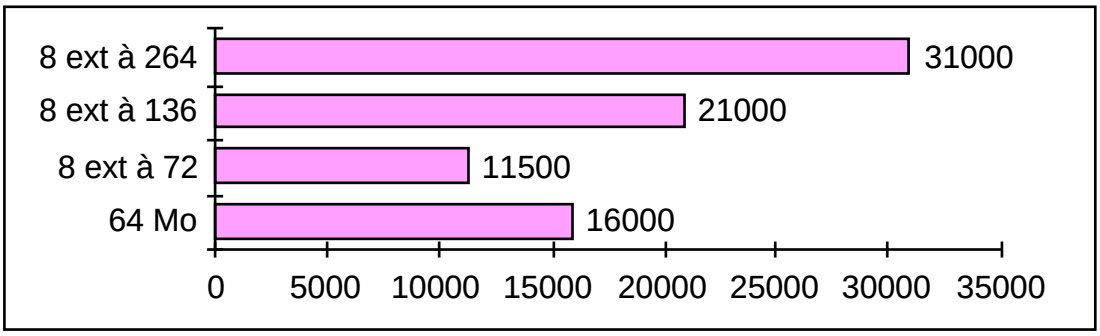

Figure F7. Coût d'une station en fonction de son extensibilité

L'axe des abscisses est le prix d'achat en Francs. Les trois valeurs supérieures de l'axe des ordonnées représentent des machines selon son extensibilité mémoire (1, 2, ou 4). La valeur d'une extension mémoire est aussi indiquée. Il est à noter que le prix de la mémoire n'est pas négligeable en regard du prix de la machine.

Les performances des machines ne suivent pas la courbe des prix :

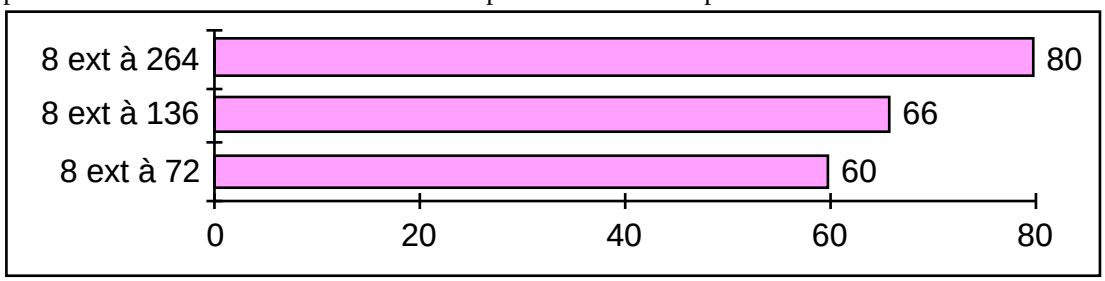

Figure F8. Performance des mêmes stations

L'axe des abscisses est la performance brute d'une machine. Les trois valeurs de l'axe des ordonnées représentent des machines selon son extensibilité mémoire (1, 2, ou 4).

Il est beaucoup plus rentable de répartir les calculs de vérification sur plusieurs machines plutôt que d'être obligé d'acheter une grosse machine, dont la capacité mémoire est sous-utilisée en dehors des tâches de vérification.

Les contraintes de disponibilité plaident aussi pour l'utilisation de plusieurs machines : la panne d'une machine ne provoque qu'un ralentissement du projet au lieu d'un arrêt total dans le cas d'une architecture centralisée.

2.1.3.3) Vérification incrémentale

A l'origine, les vérifications s'effectuaient uniquement sur le résultat de la conception : le fichier final prêt à envoyer au fondeur. L'accroissement de la complexité des VLSI fait qu'il n'est plus possible d'attendre la fin de conception pour vérifier le circuit.

Il est souhaitable de vérifier au fur et à mesure chacun des modules obtenus durant la phase de conception hiérarchique. Ceci permet de déterminer plus aisément quel outil (synthèse, routeur, ...) souffre d'un bug, sans avoir à attendre la phase finale d'assemblage du circuit pour apprendre que le générateur de ROM souffre d'un bug.

Toutes ces vérifications incrémentales durant la phase de conception réduisent le nombre d'erreurs détectées lors de la vérification finale, mais ne peuvent pas remplacer la vérification finale. Il reste nécessaire d'effectuer une vérification complète de tout le circuit avant l'envoi en fabrication.

2.1.4) Quelques vérifications

Ainsi que nous l'avons vu, les outils de vérification se caractérisent par la manipulation d'un grand nombre d'objets élémentaires dans une vue "à-plat" du circuit (non-hiérarchique).

2.1.4.1) La simulation de fautes

Un circuit est remis au fondeur accompagné d'une liste de vecteurs de test qui permettent de trier les circuits ne possédant pas de défauts de fabrication. Un vecteur de test est une configuration logique des entrées, ainsi qu'un résultat attendu en sortie. Lorsque le circuit est combinatoire (il ne contient que des portes logiques et aucun élément mémorisant), le résultat d'un vecteur de test ne dépend pas de l'application du vecteur précédent. Si le circuit est séquentiel (il y a au moins un élément mémorisant), les vecteurs de test forment un ensemble ordonné indissociable.

Il y a faute lorsque le résultat en sortie du circuit sur le banc de test diffère de celui attendu et décrit dans le vecteur de test. Une faute est une équipotentielle collée à 0 ou à 1, selon le modèle des collages. Si un circuit est composé de n équipotentielles, il y a donc $2n$ fautes associées.

La simulation de fautes a pour but principal de déterminer le taux de couverture fourni par un ensemble de vecteurs de test, c'est à dire l'»efficacité du tamis» permettant de rejeter les pièces défectueuses. Il s'agit du nombre de fautes détectées par l'ensemble de vecteurs de tests associé au circuit. La simulation de fautes fournit aussi un dictionnaire résultat : la liste des fautes détectées, ainsi que la liste des fautes non-détectées par le jeu de vecteurs de test fourni.

La principale difficulté de la simulation de fautes est le volume des données manipulées : un ensemble de *m* vecteurs, et un ensemble de *2n* fautes possibles. Ceci nécessite en principe *2n * m* simulations (*n* et *m* sont de l'ordre de 10^6).

Plusieurs algorithmes permettent de réduire cette complexité : la simulation déductive consiste à propager une liste de fautes à travers le circuit. La simulation parallèle profite de l'architecture de la machine support (taille du mot machine : 32bit) pour effectuer 32 simulations en parallèle dont 31 fautes.

La distribution sur un réseau de machines est une technique prometteuse pour la simulation de fautes.

2.1.4.2) L'analyse temporelle

L'analyse temporelle permet d'estimer le temps mis pour propager un signal d'un point du circuit à un autre. Son but principal est d'identifier le chemin le plus long et donc le temps de cycle du circuit.

L'analyse temporelle travaille sur un volume important de données : la totalité du circuit doit se trouver en mémoire car les chemins de propagation des signaux entre deux registres sont transversaux à la structuration modulaire du circuit : un signal qui part du coeur d'un bloc pour arriver dans le coeur d'un autre bloc peut éventuellement traverser toute la hiérarchie.

L'analyse temporelle génère aussi des données propres, en un nombre plus important encore, puisque pour *n* registres, il y a *(n * n-1)/2* chemins possibles au plus.

Le fonctionnement de l'analyseur temporel inclus dans la chaine de cao-vlsi support de notre étude est le suivant :

- Chargement de la netlist transistors du circuit.
- On reconstruit une *netlist* de portes.
- On calcule les temps de propagation associés à chaque porte.
- On construit ensuite la liste de tous les chemins possibles entre registres, que l'on écrit sur disque.

Les deux dernières opérations occupent 90% du temps d'exécution. Il est visible que l'analyse temporelle nécessite à la fois de la puissance CPU et de l'espace mémoire.

Une version hiérarchique de cet outil peut s'appuyer sur le découpage modulaire du circuit. Dans l'analyse hiérarchique, les informations temporelles caractérisant chaque module sont stockées dans des fichiers intermédiaires qui sont réutilisés lorsqu'on veut analyser les interactions entre modules.

On peut envisager un fonctionnement distribué pour l'analyseur hiérarchique, où chaque module de la hiérarchie serait traité sur une des machines. Ceci suppose, toutefois, que la configuration des machines est suffisante pour supporter le traitement final (analyse des interactions entre modules) qui nécessite d'avoir en mémoire la totalité de la hiérarchie.

2.1.4.3) Vérification de Règles de Dessin

La vérification de règles de dessin se base uniquement sur le dessin des masques remis au fabricant. Chaque masque correspond à un niveau dans le processus de fabrication (métal, polysilicium, ou isolant, ...) caractérisé par une couleur. Chaque niveau de masque est composé d'un ensemble de rectangles. Les règles de dessin sont un ensemble de conditions à respecter : Est-ce que tout rectangle rouge a bien une largeur minimum de 2 ? Est-ce que tout rectangle bleu est bien à une distance d'au moins 4 de tout rectangle vert ?

Tout vérificateur de règles de dessins contient un langage de description des règles.

La vérification finale des règles de dessin avant fabrication suppose généralement une mise à plat complète du circuit. Les circuits de l'ordre du million de transistors ont du mal à tenir à plat dans la mémoire centrale d'une station. Il n'est pas rare que sur un circuit de l'ordre du million de transistors, la vérification «à plat» nécessite une semaine de temps réel pour s'exécuter sur une station haut de gamme (SS10, 400 Mo RAM). Une solution se trouve dans les VRD hiérarchiques qui s'appuient sur la structure modulaire du circuit.

Cette approche modulaire se prête évidemment à la distribution.

2.1.4.4) Pourquoi retenons-nous le V.R.D. comme démonstrateur ?

Le vérificateur de règles de dessin a historiquement été le premier outil de vérification des VLSI.

Son fonctionnement est compréhensible (“le rectangle rouge a une largeur minimum de...”), mais la complexité de ses opérations élémentaires n'est pas négligeable (chaque rectangle doit respecter l'ensemble des règles).

Il est représentatif d'une large classe d'outils de vérification : la parallélisation et la distribution s'appuient explicitement sur la structuration hiérarchique du circuit. (C'est également le cas de l'analyseur temporel). Notre objectif est donc clair : **nous souhaitons montrer que le partitionnement exprimé par la hiérarchie modulaire du circuit, est utilisable pour paralléliser automatiquement et distribuer les vérifications de circuits VLSI**.

2.2) Emergence d'une nouvelle architecture

Du fait de la complexité croissante des circuits VLSI, la conception d'un gros circuit est toujours réalisée par une équipe de plusieurs ingénieurs travaillant de concert. Chacun de ces ingénieurs a sa station de travail généralement sous le controle du système d'exploitation Unix. Ces stations sont toutes reliées entre elles par un réseau de communication rapide. Afin de diminuer les coûts, le stockage des données est centralisé sur un serveur de fichiers. Ceci permet de protéger les données importantes d'une fausse manoeuvre par une protection physique ainsi que par une duplication régulière du travail en cours. L'architecture générale est la suivante :

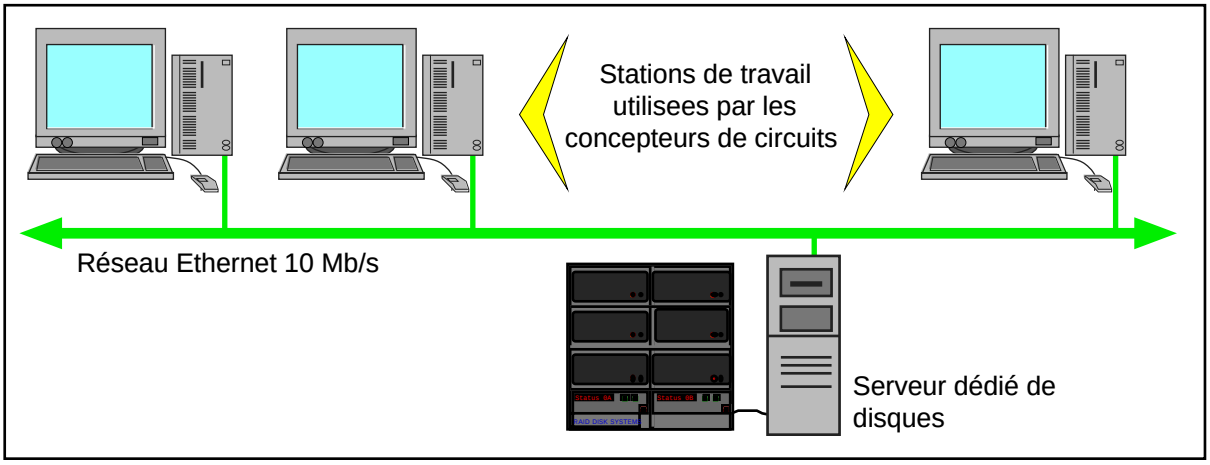

Figure F9. Architecture distribuée cible

Le système d'exploitation Unix, dont les premières versions remontent aux années 1970, sait exploiter optimalement tous les composants d'une station de travail, hormis le réseau. Seul, le partage de disques (NFS) est devenu un standard de fait.

Ce réseau de station de travail pris dans sa globalité représente un système capable d'exécuter une application distribuée. Par rapport à l'évolution des interfaces des disques (Normes SCSI, SCSI-2, fast SCSI, wide SCSI, SCSI-3, IDE, ATA, EIDE, IBM-SSA...) qui est constante et régulière, la vitesse d'Ethernet n'a pas changé. Ethernet est actuellement considéré comme "lent". Un réseau de stations est donc un système faiblement couplé.

L'architecture réseau recommandée et utilisée sur de nombreux sites est appelée "*dataless* " (sans données). Les disques locaux contiennent uniquement la zone de *swap* (mémoire virtuelle), le système d'exploitation Unix, et les applications les plus souvent utilisées. Toutes les données (le circuit en cours de construction) sont déportées sur le serveur de fichier pour des raisons de sécurité et d'accessibilité. Le protocole NFS se charge de mettre à dispostion ces données sur chaque station. Cet accès aux données se fait à travers le réseau Ethernet, et le charge d'une manière non-négligeable. Ethernet, dont le débit est divisé par le nombre de stations actives, n'offre pas une grande bande

passante pour les communications. Nous devrons garder à l'esprit que notre lien de communication est ténu, et veiller à l'utiliser avec parcimonie..

Par contre, ces stations de travail sont souvent sous-utilisées [87]. Nous nous proposons de réduire la durée de traitements extrêmement longs en profitant de cette puissance dormante. Nous cherchons à diviser un traitement séquentiel en plusieurs tâches capables de s'effectuer en parallèle en minimisant la communication entre tâches. Nous visons du parallèlisme à gros grain.

2.3) Problèmes potentiels

Bien que ces techniques d'accélération soient simples, beaucoup de problèmes peuvent potentiellement réduire le gain à néant s'ils ne sont pas en pris en compte à temps.

2.3.1) Volume des données manipulées

La vue physique d'un circuit comme StaCS (Voir stacs, page 77.) au format CIF est un fichier d'une vingtaine de Méga-octets. Le chargement de la hiérarchie complète en mémoire occupe déja une soixantaine de Méga-octets. C'est à dire toute la mémoire physique d'une station. Le même circuit «à plat» occuperait plus d'un Giga-octets. StaCS ne contient que 875.000 transistors. Nous souhaitons pouvoir traiter des circuits de plusieurs millions ou dizaine de millions de transistors.

Pour quantifier les circuits que notre VRD devra traiter, nous utiliserons les dénominations suivantes :

- **gros circuits** : il s'agit de circuits dont la vue physique ne peut tenir «à plat» dans la mémoire centrale d'une machine. Seule la vue physique en mode hiérarchique (l'ensemble des modèles sans que leurs instanciations ne soient expansées) tient en mémoire.
- **très gros circuits** : il s'agit de circuits dont même la vue physique hiérarchique ne peut tenir au complet en mémoire. Les méthodologies de conception hiérarchique actuelles, associées aux formats de fichiers non-monolithique (un fichier par modèle) autorisent une conception, où seuls un modèle et ses fils immédiats ont besoin d'être simultanément en mémoire centrale.

Comme nous souhaitons pouvoir traiter des très gros circuits, nous devrons garder à l'esprit ces contraintes — seuls, un modèle et ses fils immédiats tiennent en mémoire — lors de la définition de l'architecture de nos applications.

2.3.2) Fragilité

Leslie Lamport définit dans [53] les systèmes distribués de la façon suivante : «*You know that you have a distributed system when the crash of a computer, you have never heard of, stops you from getting any work done*» (Vous pouvez reconnaître un système distribué lorsque la panne d'un ordinateur, dont vous n'aviez jamais entendu parler, vous empêche de travailler).

La répartition des ressources augmente la probabilité de défaillance. Si l'on ne prend pas en compte le facteur de robustesse dans la conception de l'application, elle sera inutilisable. Il suffit d'une machine en panne pour que toute l'application soit arrêtée.

2.3.3) Fonctionnement incrémental

Nous pouvons espérer une accélération linéaire en distribuant les traitements élémentaires sur plusieurs machines. Il est possible de gagner encore plus si la méthode de vérification permet de réutiliser des résultats intermédiaires. La conception d'un circuit se fait selon un cycle similaire à celui de la conception d'un logiciel :

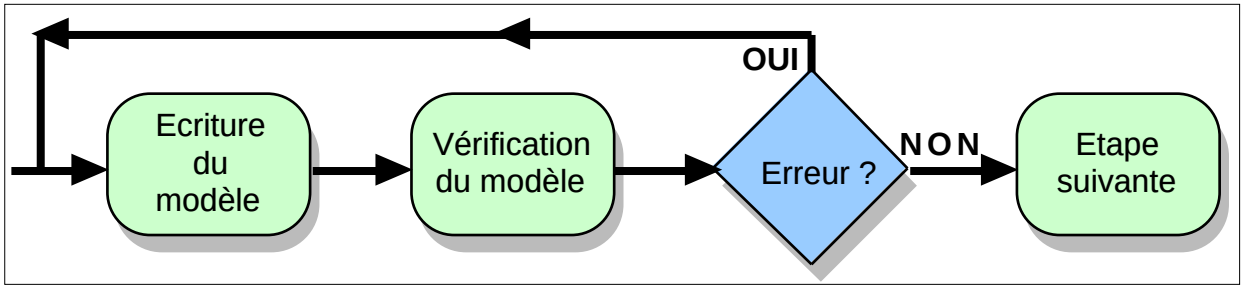

Figure F10. Cycle itératif de la conception

Le concepteur de logiciel, confronté à ce cycle corrections / re-vérifications, utilise l'outil `make` [88]. Cet outil lui permet de diviser ses sources en plusieurs modules, et `make` sait automatiquement reconstruire les modules qui ont été modifiés. Il nous faut offrir la même fonctionnalité.

Le fonctionnement incrémental d'un vérificateur facilite aussi le travail coopératif. Un concepteur peut fournir le résultat de la vérification d'une partie du circuit (la ROM) en libre-accès au reste de l'équipe, à travers un répertoire bibliothèque, même s'il n'a pas fini de valider cette ROM (analyse temporelle, vérification fonctionnelle, etc).

Pendant toute la phase de conception initiale du circuit, les concepteurs font beaucoup de vérifications "incrémentales". Plus le niveau de hiérarchie traité est élevé, plus le gain est important.

2.3.4) Complexification

De part la multiplicité des ressources utilisées, une application distribuée est intrinsèquement plus complexe qu'une application monolithique.

Ceci se ressent au niveau du développement d'applications. La possibilité de traitements simultanés impose de gérer les accès concurrents aux ressources critiques. D'autre part, la mise au point des applications est délicate. Le déverminage d'applications distribuées est un sujet de recherche en pleine expansion [49] [80] [82] [84].

L'utilisation d'une application distribuée est aussi difficile. Il faut pouvoir redéfinir facilement les stations à utiliser lors de la vérification.

2.3.5) Politique de distribution

Plusieurs options s'offrent à nous pour le placement d'une tâche, ce qui revient en pratique à choisir un processeur pour un *process* Unix. Lorsqu'une station, qui était inutilisée, redevient sollicitée pour une utilisation interactive, la tâche appartenant à l'application distribuée en cours d'exécution peut :

- être migrée vers une autre machine
- être suspendue
- continuer, privant ainsi l'utilisateur légitime d'une bonne partie de sa puissance de calcul.

Il faut que le système de distribution choisisse la machine la mieux adaptée pour une tâche. De nombreux critères peuvent être utilisés pour déterminer la meilleure machine :

- minimiser le temps d'exécution total de l'application
- choisir la machine ayant le plus de mémoire libre (capacité RAM)
- choisir la machine la plus rapide (performance CPU)
- choisir la machine la plus facilement accessible (coût du routage)

2.4) Conclusion

L'étude de la méthode de conception hiérarchique d'un circuit VLSI, nous a montré que les phases de vérification sont des phases critiques de la conception. Leur coût est essentiellement un coût de machine. Les réseaux de stations de travail représentent une nouvelle architecture dont nous pourrions profiter. Des caractéristiques de cette architecture découlent un certain nombre de problèmes

potentiels : Comment répartir le grand volume de données manipulées ? Comment développer une application de vérification efficace, robuste et agréable ?

Nous retiendrons le Vérificateur de Règles de Dessin (VRD) comme démonstrateur de notre méthode de distribution d'outils de vérification.

a

3) Etat de l'Art

Dans ce chapitre, nous ferons l'état de l'art de Vérificateurs de Règles de Dessin. Puis nous nous intéresserons aux librairies permettant de distribuer une application.

La notion de distribution de programmes est une notion relativement récente. Il a fallu la généralisation des réseaux au début des années 1980 pour que les premières publications scientifiques apparaissent. Ce n'est qu'une décennie plus tard, au début des années 90, que les concepts liés à la distribution sont clairement définis. Actuellement, de nombreuses sites sont physiquement connectés comme un système distribué, mais restent configurés dans une optique de relations maitre / esclave. Ce n'est que en 1994 que des systèmes d'exploitations de grande diffusion, permettant des relations d'égal à égal (*peer-to-peer*), comme Windows for Workgroups, ont été disponibles.

3.1) Les Vérificateurs de Règles de Dessin

Dans ce paragraphe, nous présentons quelques vérificateurs de règles de dessin.

3.1.1) Magic

Dès 1984, G. S. Taylor et J. K. Ousterhout ont présenté [27] un VRD à fonctionnement incrémental.

Leur VRD est intégré dans l'éditeur de *layout* de la chaine de cao-vlsi nommée *Magic*. Lorsque la vue physique est modifiée, il enregistre les zones qui ont été changées, et ne revérifie que celles-là. Le VRD fonctionne en continu en tâche d'arrière-plan, pendant l'attente de commandes du concepteur. Lorsqu'une erreur est trouvée, il l'illumine dans l'éditeur graphique

Les règles de dessin ont une forte localité. Il s'agit souvent d'écartements minimums à respecter. Il existe une certaine distance à partir de laquelle deux éléments ne peuvent plus interagir. Cette dis-

tance s'appelle *DRID* (*Design Rule Interaction Distance*) en anglais, et CTM (Contrainte Technologique Maximale) en français. Le halo d'une zone est constitué de la bordure externe d'une largeur de la distance d'interaction maximale (CTM) :

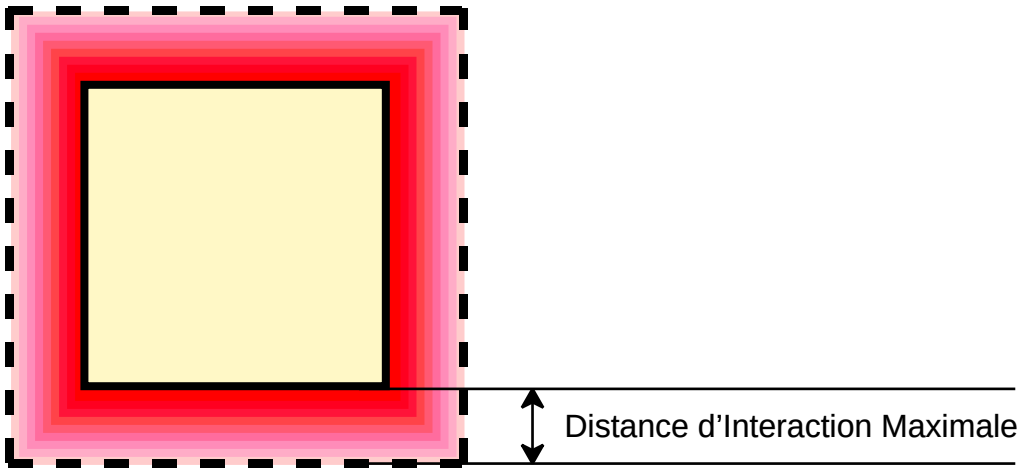

Figure F11. Le *halo* (externe) d'un bloc

Lorsqu'une zone est modifiée, magic ne revérifie que les interactions entre les objets situés dans cette zone et son halo.

Afin d'éviter d'imposer le fonctionnement permanent du VRD et de son interface graphique, l'information, qu'une certaine zone a besoin d'être revérifiée, est stockée dans la cellule. Ceci empêche l'usage de formats de fichier standards.

3.1.2) Cutter / Paster

Dès 1985, G. E. Bier et A. R. Pleskun ont présenté [23] un algorithme de VRD pour un multiprocesseur. Le premier programme Cutter charge la vue physique et crée des fichiers contenant des bandes verticales du circuit à plat. Ces bandes verticales sont vérifiés par un VRD à plat. Le deuxième programme — Paster — profite des informations de partitionnement fournies par Cutter pour corriger les fausses erreurs, et concaténer les messages d'erreur.

Ceci représente un partitionnement surfacique aveugle d'un circuit. Cette méthode présente deux défauts : La vérification ne peut commencer qu'après l'assemblage final du circuit : l'approche incrémentale n'est pas possible. D'autre part, la construction des bandes nécessite une mise à plat implicite du circuit complet, qui peut être difficile voire impossible pour un très gros circuit.

3.1.3) Corny

En 1989, N. Hedenstierna et K. O. Jeppson ont présenté [13] les *inverse layout tree* pour les vérifications de règles de dessin hiérarchiques. Avec ce concept, la vérification peut être effectuée sans

revérifier les zones de recouvrement des blocs.

Pour chaque bloc terminal (feuille) de l'arbre hiérarchique représentant le circuit, on cherche à déterminer l'ensemble de ses environnements possibles. (Il y a autant d'environnements que d'instances du bloc considéré). L'union de ces environnements constitue le «*halo*». Enfin, il suffit de vérifier chaque bloc avec son environnement pour être dispensé d'une vérification sur toutes les instances d'un bloc.

L'algorithme du *halo* vérifie chaque modèle successivement. Les primitives externes (rectangles) qui interagissent identiquement lors de plusieurs instanciations ne sont prises en compte qu'une seule fois.

Une description imagée de cet algorithme est la suivante : Prenons une impression du circuit complet sur papier transparent. Découpons chaque instance d'une cellule avec son voisinage (le *halo*) d'une distance inférieure à la CTM. Empilons chacune de ces découpes et alignons-les. L'ensemble des différents voisinages du modèle apparait comme un seul voisinage. Les interactions répétées ne sont vues que comme une seule interaction. La vérification de cet objet fait que les interactions identiques ne sont vérifiées qu'une seule fois.

En 1992, les mêmes auteurs ont présenté [14] une version parallèle et hiérarchique de leur algorithme. Mais dans le pire cas, la construction du *halo* d'un seul bloc peut nécessiter l'usage de tous les blocs de la hiérarchie. Sur un seul processeur, le programme hiérarchique est 20% plus lent que la version séquentielle.

En fait, une telle approche n'est intéressante que pour la vérification de structures régulières. Ce genre d'approche ne permet pas un fonctionnement incrémental.

3.1.4) Pace

En 1988, K. P. Belkhale et P. Banerjee ont présenté [16] PACE : un extracteur parallèle. A la différence d'un vérificateur de règles, un extracteur fournit à partir du *layout* final d'un circuit une *netlist* avec résistances et capacités. Mais la problèmatique est très voisine. Cet extracteur est conçu pour l'Intel Hypercube [42]. Il s'agit d'un partitionnement en surface. Le circuit est divisé en autant de zones rectangulaires qu'il y a de processeurs disponibles.

En 1989 dans PACE2 [17], le partitionnement est toujours un découpage en surface, mais les zones résultant de la partition ne sont plus de surfaces égales. Pour un circuit non-homogène, cela conduisait à une mauvaise répartition de charge. Le partitionnement de PACE2 repose sur le nombre de rectangles. Chaque case contenant le même nombre de rectangles.

En 1990 dans [18], K. P. Belkhale et P. Banerjee ont présenté une version hiérarchique de leur extracteur adaptée aux circuits possédant eux-mêmes une structure hiérarchiques. Deux algorithmes

sont utilisés pour paralléliser :

- Chaque bloc de la hiérarchie est assigné à un processeur. Ils démarrent selon leur nombre de rectangles (algorithme LPT : *largest processing time first*). L'inconvénient de cet algorithme est qu'un bloc très gros peut constituer un goulet d'étranglement pour toute l'application.
- Le traitement de chaque bloc est lui-même parallélisé selon l'algorithme de PACE2. La répartition des sous-tâches est controlée par l'algorithme PITS (*partitionable independent task scheduling*)

3.1.5) ProperCAD

En 1994, dans «*Parallel algorithms for VLSI Computer-Aided Design*» [2], P. Banerjee présente le projet ProperCAD (*Portable object-oriented parallel environment for CAD*) qui vise à construire une chaine de CAO pour VLSI utilisant des algorithmes parallèles pour machines multiprocesseurs à mémoire partagée.

P. Banerjee y recense aussi une approche originale pour la vérification de règles de dessin : la décomposition fonctionnelle d'un circuit. Ce ne sont pas les rectangles qui sont répartis, mais les règles de dessin. Chaque processeur vérifie une règle de dessin sur l'ensemble des rectangles.

Cette méthode ne peut être utilisée sur de gros circuits car la totalité du circuit doit être chargée sur chaque machine.

En 1993, dans «Task scheduling for exploiting parallelism and hierarchy in VLSI CAD algorithms» [19], P. Banerjee indique que le facteur déterminant de l'efficacité des techniques hiérarchiques combinées avec l'approche parallèle est l'ordonnancement des tâches résultantes.

3.1.6) Versatil et DRuC

Cette thèse fait suite au travail [7] de J. Medou Zengue Ze sur la vérification des règles de dessin.

Dans sa thèse, J. Medou Zengue Ze décrit un langage formel de description de règles de dessin, implémenté dans un outil prototype de vérification : Versatil. Il propose par ailleurs une méthode de vérification hiérarchique, en introduisant le concept de «couronne». Cependant cette méthode n'a pas été démontrée expérimentalement. En effet, la mise en oeuvre effective de cette approche hiérarchique nous a amené à définir plusieurs mécanismes originaux non décrits dans la thèse de J. Médou Zengue Ze (Vias virtuels, génération automatique des règles hiérarchiques, etc).

Le halo représente les environnements possibles d'un bloc et ne contient que des rectangles extérieurs au bloc.

La couronne est une notion totalement indépendante de l'environnement du bloc. C'est une zone

«interne» qui ne contient que des rectangles appartenant au bloc :

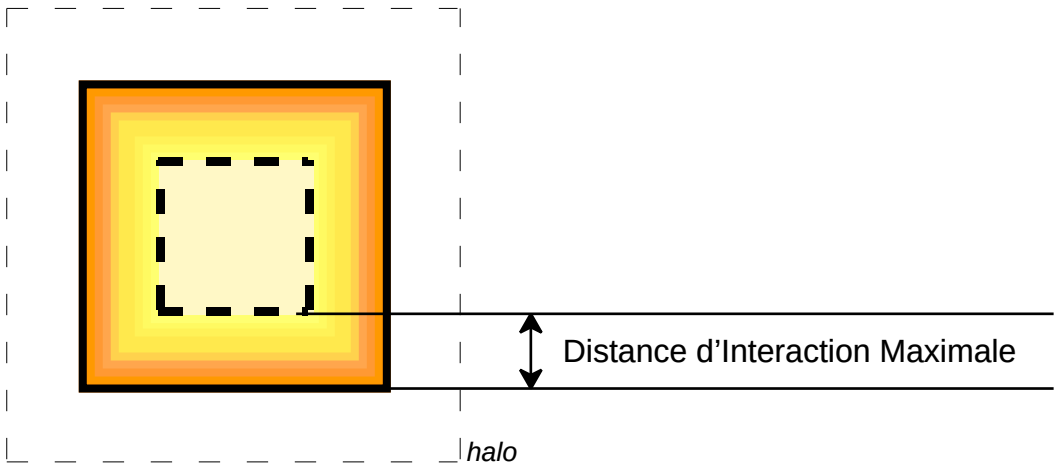

Figure F12. La couronne (interne) d'un bloc

Le concept de couronne se prête mieux au partitionnement strict imposé par l'architecture d'un réseau de stations.

Seul le format CIF etait traité par le prototype *Versatil* de J. Medou Zengue Ze. La recherche de performances — Intel sort un nouveau processeur complexe tous les dix-huit mois. — font que le développement de la technologie et le travail des concepteurs a lieu en parallèle. Le concepteur peut donc être certain que son circuit connaitra au moins un changement majeur de technologie pendant la conception même.

Pour cette raison la chaine de CAO-VLSI Alliance dans laquelle s'inscrit notre travail utilise la technologie du «layout symbolique».

Pour permettre la prise en compte de cette technique, un nouveau vérificateur de règles de dessin a été développé par P. Renaud dans le cadre de son stage de D.E.A., et intégré dans la chaine Alliance : DRuC.

Comme Versatil, DRuC est un vérificateur à plat. Il utilise la structure RDS décrite dans [39] avec fenétrage pour bénéficier des propriétés de voisinage. Cette structure offre de bonnes performances lorsque la mémoire est disponible, (complexité en $n\sqrt{n}$) mais elle peut aussi être optimisée pour préserver la mémoire en échange d'un ralentissement de la procédure de recherche.

DRuC est capable de travailler sur les formats de fichiers suivants : .ap (Alliance), .cp (Compass), CIF et GDS-II.

Sur les machines utilisées (28 Mips, 64 Mo RAM), DRuC est limité à des circuits d'une taille de 50 Ktransistors.

3.2) Les boites à outils de distribution

Nous présentons dans ce paragraphe différents outils permettant la programmation d'applications parallèles.

3.2.1) rsh (Remote SHell)

Rsh signifie "interpréteur de commandes distant". Le but de rsh (et de ses programmes frères tels que rcp, rexec, rdist, rdump et rlogin) est de permettre à un processus sur une machine d'exécuter un programme sur une autre machine.

Avantages

Rsh est extrémement répandu et constitue un standard de fait. Chaque implémentation du protocole TCP/IP le livre en standard

Inconvénients

Rsh n'est concu que pour permettre l'exécution d'un programme sur une machine distante. Il ne fournit aucun moyen d'échange des données. Toutes les informations de topologie du réseau doivent être codées dans l'application.

3.2.2) rpc (Remote Procedure Call)

Rpc [45] signifie "appel de procédure à distance". A l'instar d'un appel de procédure, il permet à un processus de transférer le controle d'une partie de lui-même à un autre processus situé sur une machine différente, avec retour à l'appelant.

Le terme RPC est un terme générique. De nombreuses implémentations en existent. Xerox, Apollo en ont été les précurseurs. Le standard de fait est l'implémentation de SUN dans ONC (Open Network Computing).

Cette implémentation comprend deux parties distinctes :

- RPC: la spécification des appels distants
- XDR (eXternal Data Representation) : le standard d'échange de données entre architectures hétérogènes.

Avantages

ONC/RPC est répandu. Il sert de base au système de partage de fichiers SUN / NFS. Il assure aussi une transparence des données à travers des architectures hétérogènes. L'application n'a pas à se préoccuper de l'ordre des octets sur la machine (*big endian / little endian*).

Inconvénients

RPC est un protocole de base. Son interface est trop complexe et de trop bas niveau pour le programmeur d'applications.

3.2.3) DCE (OSF / Distributed Computing Environment)

DCE est concu par l'"Open Software Foundation" un consortium de constructeur de machines Unix. Il s'agit de résoudre les problèmes de distribution d'applications, de la même manière que OSF/Motif résoud les problèmes d'interface utilisateur graphique.

Avantages

DCE est un produit ambitieux qui tente de répondre à tous les besoins. Le slogan de DCE, identique à celui du constructeur SUN, est "Le réseau est l'ordinateur". C'est aussi ce que nous souhaitons fournir au concepteur d'applications de CAO pour VLSI.

DCE fournit, entre autres, les services suivants :

- partitionnement : à travers le concept de cellule, les ressources présentes sur le réseau sont partitionnées de manière à autoriser des réseaux hiérarchiques complexes
- RPC : DCE a son propre appel de procédures à distance.
- parallélisation : bien que DCE soit déja un environnement distribué, il fournit des moyens pour paralléliser cette distribution avec le concept de *threads*. Ceci permet à un serveur de traiter en parallèle plusieurs requètes simultanément.
- sécurité : de nombreux contrôles d'accès sont effectués.
- répertoire : diverses informations peuvent être mises en commun par ce service.
- gestion du temps : permet une synchronisation effective des processus.
- partage de fichiers : DCE/DFS (*Distributed File Service*) est similaire à NFS.

Inconvénients

DCE est très complexe : il existe des livres pour utiliser DCE [51] [52], et d'autres pour **comprendre** DCE [50].

D'autre part DCE doit être intégré au système d'exploitation de la machine. Ceci n'est pas le cas de la plupart des stations actuellement utilisées.

DCE ne gère pas le placement des tâches. Il est du ressort de l'application ou de l'utilisateur de déterminer les machines utilisées.

3.2.4) pvm (Parallel Virtual Machine)

Pvm est issu d'un programme de recherches sur les sciences mathématiques appliquées démarré en 1989 aux Etats-Unis (OakRidge, Knoxville, Pittsburg, Atlanta). Son but est de rassembler un ensemble défini par l'utilisateur d'ordinateur séquentiels, parallèles, ou vectoriels, en un seul ordinateur à mémoire distribuée utilisant le paradigme de communication du passage de messages (*send / receive*).

Ce logiciel est structuré sous la forme de :

- un démon par machine controlant le bon déroulement de l'application
- une librairie de programmation a intégrer dans l'application
- une "console" : qui sert à lancer l'application et à visualiser son déroulement.
- divers autres programmes utilitaires.

PVM est transparent : il évite toute différence (au niveau du programme applicatif) entre l'exécution locale et l'exécution distante d'une tâche.

Avantages

PVM est un standard de fait : De nombreux constructeurs de machines parallèles (Cray, Convex, Dec, IBM, etc) livrent les librairies pvm en standard avec leur système d'exploitation.

Pvm supporte des réseaux hétérogènes.

Le demon pvm fonctionne en mode utilisateur.

Pvm reconnait et gère plus d'une trentaine d'architectures différentes allant du PC sous Linux au multi-processeur massif Paragon en passant par le Cray Y-MP.

Pvm gère les architectures multi-processeurs et les intégre dans son modèle. Il utilise des appels natifs pour communiquer entre chacun des processeurs. Les stations de travail multi-processeurs de SUN et SGI sont supportées.

Inconvénients

Pvm est issu du monde du calcul scientifique. Il est adapté à des programmes en Fortran manipulant des matrices de nombres. Il ne fournit aucune optimisation pour les entrée/sorties disques volumineuses en VLSI.

La tolérance de panne est du ressort de l'application.

Pvm n'intégre actuellement aucune méthode pour le choix de la meilleure machine pour une tâche donnée. Or, le choix de la machine optimale est le critère de performance principal d'une application distribuée selon [89]. Lorsqu'une tâche est démarrée, sans que l'application précise la machine dé-

sirée, la machine est choisie selon l'algorithme *round-robin*. Toutefois la documentation de *pvm* mentionne que des algorithmes meilleurs seront fournis dans les versions futures. La version actuelle intégre des points d'entrée pour accrocher son propre allocateur.

3.2.5) gatos

Gatos est aussi issu d'une recherche démarrée en 1989. Gatos a été conçu au Laboratoire MASI dans l'équipe "Systèmes Distribués". Le but de Gatos est similaire à celui de pvm : utiliser au mieux la capacité de traitement d'un système réparti, principalement les processeurs, mais également les mémoires, les disques et les lignes de communication.

A la différence de PVM, Gatos n'est pas un produit. C'est un support de recherches en informatique distribuée. Nous nous sommes basé sur la version disponible en décembre 1993. Depuis, Gatos a évolué pour, par exemple, apporter la tolérance aux pannes à des applications sous le nom de Gatostar. D'autres développements sont en cours.

Gatos vise les objectifs suivants :

- gérer la charge des machine (prend en compte l'utilisateur interactif)
- être multi-applications : plusieurs applications distribuées peuvent s'exécuter concurrement sous le contrôle du même superviseur.
- être multi-réseaux : gatos est adapté aux WANs (*Wide Area Network*) et intégre le coût des communications dans ses algorithmes de placement.
- offrir plusieurs algorithmes de placement avec un langage de contraintes.
- fournir un système de statistiques variées (mémoire, cpu, communications, ...) qui sont générées à la demande et peuvent être prise en compte pour le placement lors d'une prochaine exécution.
- offrir la migration de tâches en cas de surcharge d'une machine (utilisateur interactif revenant sur sa console).
- offrir un mécanisme de tolérance de pannes avec reconfiguration de la machine en cas de panne, et relancement des tâches sur d'autres processeurs.

Avantages

Gatos est flexible : il dispose de plusieurs algorithmes pour calculer le meilleur placement. L'utilisateur peut indiquer des criteres variés comme architecture, charge, espace mémoire ou disque.

Gatos gére la localisation des fichiers : il est capable de déplacer un fichier d'une machine à une autre selon les besoins de l'application.

Gatos calcule le cout des communications dans le cas d'une utilisation sur un réseau longues dis-

tances (*WAN*).

Gatos est sensible : il dispose d'un serveur dédié aux mesures qui permet de connaitre l'état dynamique d'une machine (inoccupée, chargée, surchargée,...) d'une manière bien plus efficace que par les fonctions du système d'exploitation. Ce serveur anticipe la charge future.

Gatos est performant : son cout de calcul du placement est trés faible pour un resultat de qualité.

Gatos dispose d'outils d'analyse : il peut fournir le graphe d'exécution de l'application. Il peut réutiliser ce graphe pour améliorer le placement à la prochaine exécution. L'utilisateur peut aussi décrire dans un langage toutes les contraintes d'exécution qu'il souhaite. Un exemple de traducteur automatique de fichiers Makefile vers le langage de Gatos est fourni. Il permet de distribuer automatiquement l'exécution de la commande make.

La version gatostar permet la tolérance de pannes et la migration de processus en cours d'execution.

Inconvénients

Gatos est encore un prototype de recherche. Ce n'est pas un produit comme PVM. Il est moins bien documenté.

PVM est une librairie de programmation au niveau application, tandis que gatos est beaucoup plus proche du système (démon type rstatd). il ne fonctionne qu'en démon intégré au système d'application.

Gatos ne connait que le protocole TCP/IP dans sa version actuelle. Bien que les points d'entrée soient prévus, il n'utilise pas de protocole propriétaire pour la communication sur des machines multi-processeurs.

Gatos n'est pas un petit logiciel : la distribution que nous avons utilisé représente plus de 25000 lignes de programme, sans documentation utilisateur.

3.3) Conclusion

Nous avons vu qu'il est possible de distribuer un VRD sur un réseau de stations selon un partitionnement guidé par la hiérarchie existante dans le circuit. Le *halo* utilisé dans plusieurs VRD hiérarchique est la périphérie externe d'un modèle. Le *halo* est construit à partir de l'environnement du modèle. Ce concept ne permet pas un fonctionnement incrémental, puisqu'il faut connaitre l'environnement du bloc pour en construire son *halo*. Pour échanger les interactions entre les différents blocs, nous retenons le concept de **couronne**. C'est aussi ce concept qui permet un fonctionnement incrémental, donc robuste. Le vérificateur «à plat» DRuC, issu de la chaine Alliance, nous servira de support pour appliquer notre technique de distribution hiérarchique incrémentale à un outil de vérification pré-existant.

Les principales caractéristiques des bibliothèques de fonctions d'aide à la distribution de programmes sont les suivantes :

	RSH	RPC	DCE	PVM	GATOS
exécution à distance	Oui	Oui	Oui	Oui	Oui
échange de données	Non	Oui	Oui	Oui	Oui
topologie dynamique	Non	Non	Oui	Oui	Oui
placement multi-critères	Non	Non	Non	Non	Oui
tolérance aux pannes / migration	Non	Non	Non	Non	Oui
simple a apprendre / utiliser	Oui	Non	Non	Oui	Non
répandu	Oui	Oui	Non	Oui	Non
standard de fait	Non	Non	Non	Oui	Non

Tableau T1. Récapitulatif des librairies d'aide à la distribution d'applications.

De part sa très large diffusion, et sa robustesse, nous retiendrons PVM comme support pour la distribution sur un réseau de stations de travail. Toutefois, PVM n'offre qu'un service de base. Cette bibliothèque ne traite aucun des problèmes classiques de la distribution tels que ordonnancement, placement, migration.

a

4) Principe de la méthode

Nous cherchons ici à définir une méthode **générale** de parallélisation et de distribution qui puisse s'appliquer à différents types de vérification de circuits VLSI. Le cas particulier du VRD distribué est décrit au chapitre 5.

4.1) Objectifs visés

Dans ce paragraphe, nous rappelons nos principaux objectifs.

4.1.1) Augmentation des capacités

En février 1995, l'éditorial d'Electronic Design [36] posait le problème du défi des nouveaux circuits sous-microniques, totalisant de deux à cinq millions de portes. Les programmes de vérification «à plat» ne peuvent plus traiter autant de données.

Notre méthode vise la vérification des gros circuits (ceux qui ne peuvent pas être chargés à plat dans la mémoire d'un processeur) mais aussi des très gros (ceux qui ne peuvent pas être chargés dans la mémoire d'un processeur même en mode hiérarchique : tous les modèles instanciés dans la hiérarchie ne tiennent pas simultanément en mémoire).

L'objectif principal de notre méthode n'est pas l'augmentation de vitesse, mais l'augmentation de capacité. Des circuits impossibles à traiter actuellement en raison de leur taille, doivent pouvoir être vérifiés grâce à notre méthode.

4.1.2) Réduction des temps de calcul

Dans la course que se livrent les concepteurs et fabricants de systèmes intégrés, la durée de conception est très importante. Les retards pour la sortie d'un produit sur le marché coûtent très cher. De plus, la durée de vie d'un circuit est de plus en plus courte. Les contraintes de temps de mise sur le marché de plus en plus fortes imposent une diminution des temps de vérification.

Nous visons une accélération d'un ou deux ordres de grandeur : une vérification nécessitant actuellement une semaine de temps CPU doit pouvoir être effectuée en quelques heures.

4.1.3) Possibilité de vérification incrémentale

Nous avons vu que si la conception d'un gros circuit suit généralement une approche descendante, sa réalisation physique est un processus ascendant, où les différents blocs sont construits séparément avant d'être assemblés.
Il est donc souhaitable que la méthode permette une vérification **locale** de chaque bloc constitutif, et que les résultats de ces vérifications partielles puissent être **réutilisés** lors de la vérification finale.

4.1.4) Facilité d'utilisation

Il est essentiel de proposer une méthode qui soit facilement appréhensible par l'utilisateur final. Le concepteur d'outils de vérification n'est pas toujours un informaticien d'origine. Il faut lui faciliter la tâche.

Comme nous visons une augmentation de capacité ainsi qu'un fonctionnement incrémental de nos opérations de vérifications, nous retiendrons le partitionnement guidé par la hiérarchie. La hiérarchie pilote, non seulement le partitionnement du circuit, mais aussi le placement / ordonnancement des tâches.

C'est le concepteur de circuit qui construira la machine parallèle virtuelle : C'est lui qui définit quelles stations seront utilisées lors d'une exécution de l'outil.

Le paramétrage d'une exécution pour tirer profit au mieux d'une configuration donnée (directives de placement / ordonnancement) doit rester aussi simple que possible.

4.1.5) Portabilité

Pour diffuser la méthode et les outils qui en résultent, plus particulièrement dans le cadre d'applications industrielles, nous devons prendre en compte les problèmes liés à la portabilité :

- Les applications de vérifications doivent accepter en entrée, et générer en sortie des formats de fichiers standardisés
- Nous visons la réutilisation de composants logiciels existants. Le choix de ces composants devra permettre l'utilisation de différentes plate-formes matérielles.
- Ceci est d'autant plus important que nous souhaitons que nos applications de vérifications puissent être distribuées sur des réseaux de machines de faible puissance.

4.2) Partitionnement hiérarchique

Nous souhaitons utiliser l'architecture distribuée d'un réseau de stations de travail pour augmenter les capacités et les performances des outils de vérification de circuits VLSI.

Distribuer, c'est découper. Il nous faut un découpage pour répartir les traitements sur les différentes stations. L'objectif d'augmentation des capacités nous impose un partitionnement des données et non des traitements.

Or, le partitionnement d'un projet complexe est guidé par la recherche de :

- la modularité. Le coût de développement d'un projet complexe se mesure en homme*an. Mais il est aussi soumis à des contraintes de temps et de développement. Afin de réduire le facteur temps, un industriel pressé cherchera à paralléliser la conception. Pour que la parallèlisation soit efficace, il faut minimiser les interactions entre les différentes tâches. Lors de la rédaction des spécifications, ce critère est pris en compte afin d'éviter de payer des employés à attendre le résultat du travail d'une autre équipe.
- la régularité. Dans un projet complexe, un certain nombre de briques de base sont réutilisées dans différents blocs. Il est nécessaire de bien identifier tous les motifs répétés dans l'analyse du projet, afin d'éviter de faire faire plusieurs fois le même travail, et de laisser les machines se charger de la duplication.

C'est la qualité du partitionnement lors de la phase de spécification qui fait la qualité globale du projet. La qualité des interactions entre les équipes en dépend. **Lorsque le projet arrive en phase de vérification finale, la qualité du partitionnement a déja été validée**.

Notre méthode de vérification, au lieu de redéfinir un découpage topologique arbitraire, s'appuie sur la structure hiérarchique du circuit définie dans la phase de conception descendante.

Dans un circuit réel :

- Le nombre de niveaux de hiérarchie est faible, mais non-négligeable (une dizaine).
- Il n'y a généralement pas de recouvrements entre les blocs à un même niveau hiérarchique.

- La structuration est souvent utilisée pour exploiter la régularité à travers la répétition.

Nous proposons de retenir comme guide du partitionnement la hiérarchie existante.

Soit le circuit suivant :

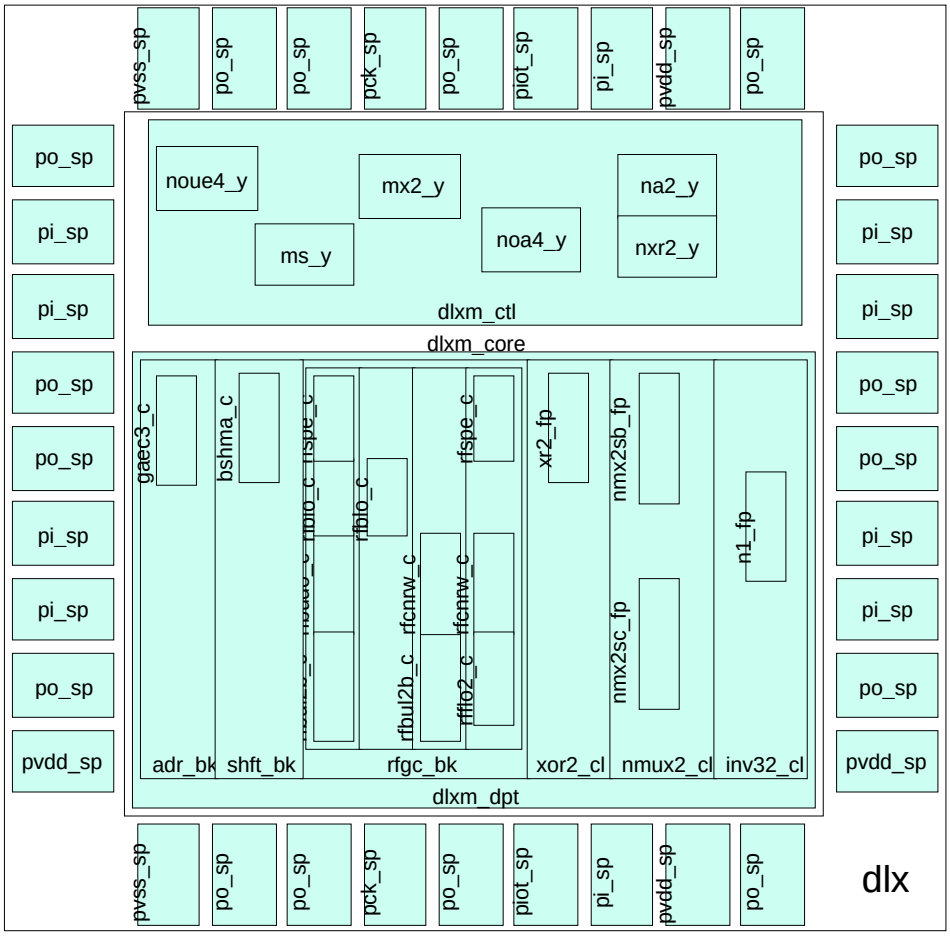

Figure F13. Vue à plat d'un circuit

En utilisant la hiérarchie comme guide du partitionnement, nous bénéficions des principales qualités de cette hiérarchie pour notre découpage :

- Modularité : les interactions entre ces blocs sont moins nombreuses que dans le cadre d'un découpage surfacique aveugle.
- Régularité : Si le circuit contient des éléments répétés, nous ne les vérifions qu'une seule fois.

La représentation hiérarchique du circuit DLX est la suivante : Il s'agit d'un graphe orienté acyclique (DAG) dont les noeuds sont les modèles instanciés et dont les arêtes expriment les dépendances d'instanciation :

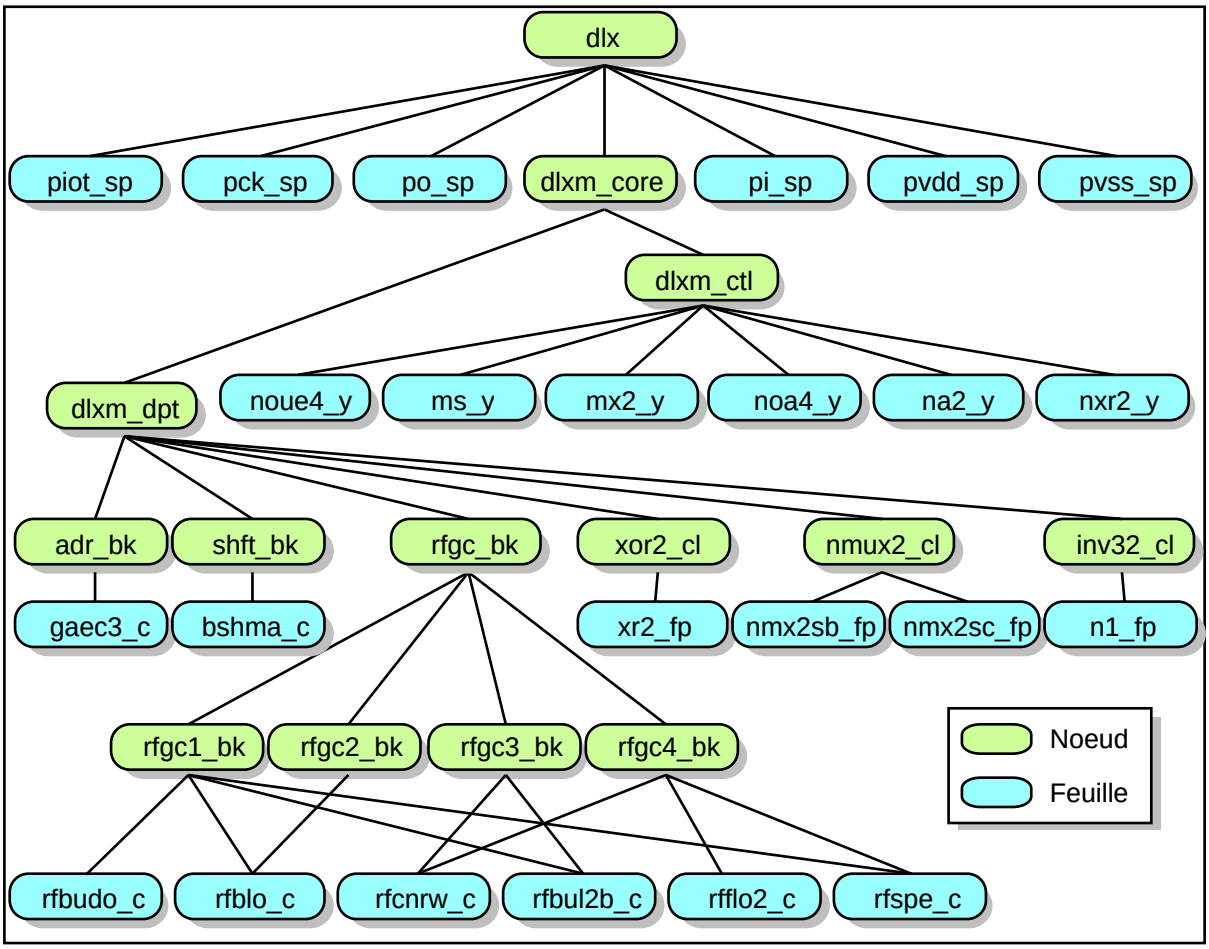

Figure F14. Vue hiérarchique d'un circuit

Notre unité de partitionnement est n'importe quel noeud ou feuille de l'arbre hiérarchique. Le traitement à appliquer diffère selon qu'il s'agit d'un noeud (non-terminal) ou d'une feuille (terminale) de ce graphe.

4.3) Traitement d'une feuille (terminale)

Les vérificateurs de circuits intégrés VLSI ont un fonctionnement de type *batch* :

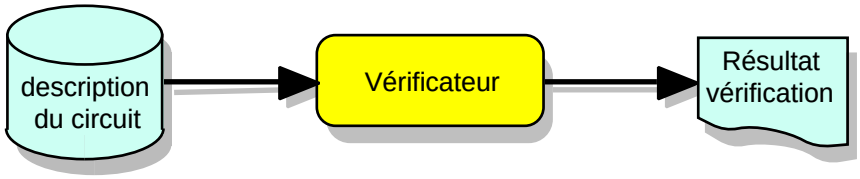

Figure F15. Fonctionnement d'un vérificateur

Il prennent en entrée la vue du circuit et produisent un résultat.

En mode hiérarchique, le vérificateur de feuille vérifie à plat tout ce qui peut être vérifié à son niveau. Il ne raisonne que sur les éléments de la granularité la plus fine que l'on cherche à traiter (rectangles pour un VRD, transistors pour un abstracteur fonctionnel, portes pour un analyseur de performances). L'information qu'il manipule est non-structurée vis-à-vis de la hiérarchie.

En plus, il doit générer une représentation simplifiée du bloc que l'on est en train de traiter :

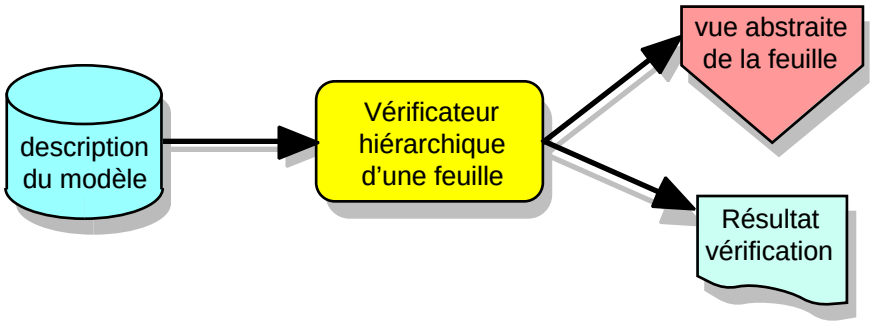

Figure F16. Fonctionnement d'un vérificateur de feuilles

Cette «vue abstraite» sera utilisée au niveau hiérarchique *n+1*. Il s'agit d'une description réduite aux éléments nécessaires pour vérifier les interactions entre blocs au niveau hiérarchique suivant. L'information contenue dans la vue abstraite est évidemment différente de l'information contenue dans le modèle initial.

Le traitement des feuilles peut être résumé par le pseudo-code suivant :

```
traiter_feuille (bloc), c'est :
       charger_à_plat (bloc)
       vérifier_feuille (bloc)
       générer_abstraction (bloc
fin
```

Figure F17. Pseudo-code d'un vérificateur de feuilles

«`Générer_abstraction`» signifie «fournir la vue abstraite qui va pouvoir être réutilisée au niveau *n+1*».

4.4) Traitement d'un noeud (non-terminal)

Un noeud est un élément non-terminal de l'arbre hiérarchique. Par rapport à un vérificateur de feuilles, un vérificateur de noeud doit aussi prendre en compte les résultats de l'étape précédente. Il utilise la vue abstraite de chacun des modèles qu'il instancie. (Par exemple, la vérification du noeud *dlxm_core* [voir Figure 14, "Vue hiérarchique d'un circuit", page 34] utilise les vues abstraites de *dlxm_ctl* et *dlxm_dpt*):

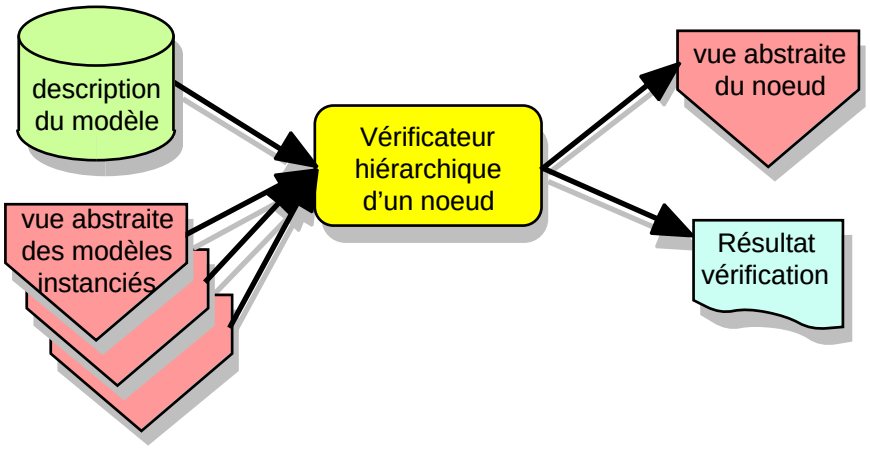

Figure F18. Fonctionnement d'un vérificateur de noeuds

Le processus est réentrant : la vue abstraite d'un noeud contient les mêmes informations que la vue abstraite d'une feuille.

Le traitement des noeuds peut être résumé par le pseudo-code suivant :

```
traiter_noeud (bloc), c'est :
       charger (bloc)
       pour tous les fils (bloc)
             charger_abstraction (fils)
       finpour
       vérifier_noeud (bloc)
       générer_abstraction (bloc)
fin
```

Figure F19. Pseudo-code d'un vérificateur de noeuds

Par rapport à la vérification d'une feuille (`vérifier_feuille`), la vérification d'un noeud (`vérifier_noeud`) doit en plus vérifier les interactions entre blocs.

4.5) Traitement hiérarchique récursif

Pour vérifier tout le circuit, il suffit de parcourir récursivement l'arbre hiérarchique :

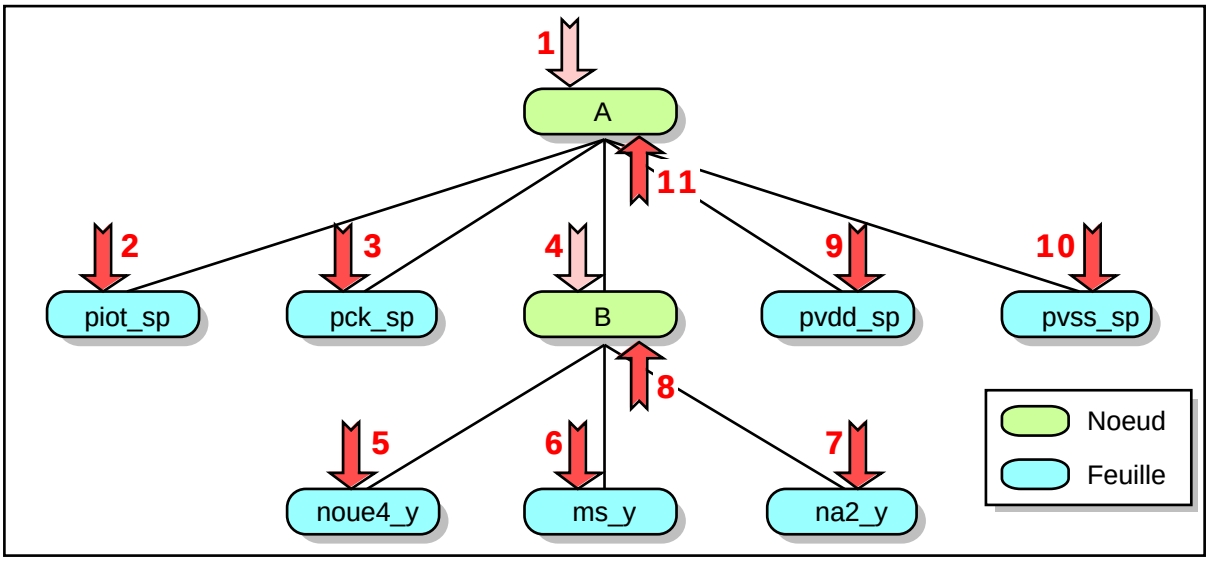

Figure F20. Parcours récursif séquentiel d'un arbre hiérarchique

selon le pseudo-code récursif suivant :

```
traiter_bloc (bloc), c'est :
        si (terminal (bloc))
                charger_à_plat (bloc)
                verifier_feuille (bloc)
        sinon
                charger (bloc)
                pour tous les fils (bloc)
                        si (pas_abstraction (fils)
                                traiter_bloc (fils)
                        finsi
                        charger_abstraction (fils)
                finpour
                vérifier_noeud (bloc)
        finsi
        générer_abstraction (bloc)
fin
```

Figure F21. Pseudo-code d'un vérificateur hiérarchique récursif

Ce pseudo-code récursif présente l'inconvénient d'empiler les chargements de bloc puisqu'il faut vérifier les feuilles pour pouvoir vérifier les noeuds intermédiaires. Ceci conduit à avoir la totalité de la hiérarchie en mémoire. Ceci est incompatible avec notre objectif de traitement des très gros circuits.

4.6) Traitement hiérarchique avec échec

Pour pouvoir traiter les très gros circuits, il faut limiter l'encombrement mémoire. Le chargement d'un bloc que l'on ne peut pas vérifier immédiatement est un échec puisque les abstractions de tous ses fils ne sont pas disponibles. Il faut alors libérer la mémoire et décharger le bloc pour pouvoir vérifier chacun de ses fils.

Ce comportement est celui d'un algorithme de type *Branch-and-Bound* [90].

Ceci peut être résumé par le pseudo-code suivant :

```
traiter_bloc (bloc), c'est :
       si (terminal (bloc))
             charger_à_plat (bloc)
             verifier_feuille (bloc)
       sinon
             charger (bloc)
             pour tous les fils (bloc)
                   si (pas_abstraction (fils))
                         mémoriser (fils_manquant
                   finsi
             finpour
             si (abstraction_manquante)
                   décharger (bloc)
                   renvoyer (liste_fils_manquant)
             finsi
             pour tous les fils (bloc)
                   charger_abstraction (fils)
             finpour
             vérifier_noeud (bloc)
       finsi
       générer_abstraction (bloc)
       renvoyer (liste_vide)
fin

traiter_circuit (bloc) c'est :
       manquants := traiter_bloc (bloc)
       si (manquants)
             pour tous les (manquants)
                   traiter_circuit (manquant)
             finpour
             traiter_circuit (bloc)
       finsi
fin
```

Figure F22. Pseudo-code d'un vérificateur hiérarchique à echecs

On peut noter que seule la fonction `traiter_circuit` est récursive.

Sous Unix, l'architecture du sous-système gérant la mémoire est basée sur l'unique appel système `brk`. Même si un programmeur utilise l'instruction `free` pour désallouer les zones mémoires acquises par `malloc`, le *process* Unix garde la mémoire réclamée. Un programme récursif complexe con-

duit souvent à des «fuites» mémoires : la quantité de mémoire utilisée ne cesse de croître. Afin de contourner ce problème, et surtout, d'être proche de l'architecture de la version distribuée, nous recommandons une architecture à deux *process* Unix :

- La fonction traite_bloc devient une tâche esclave. Si elle ne dispose pas de toutes les abstractions dont elle a besoin, elle meurt — libérant ainsi la mémoire —, et renvoie la liste des fils, dont l'abstraction manque, par un fichier temporaire.
- La fonction traite_circuit devient un programme maître, dont le seul but est de lancer récursivement la tâche esclave. Son pseudo-code pourrait être :

```
traiter_circuit (bloc) c'est :
       appeler (traiter_bloc (bloc))
       si (existe_fichier (manquants))
             lire_liste (manquants)
             pour tous les (manquants)
                   traiter_circuit (manquant)
             finpour
             traiter_circuit (bloc)
       finsi
fin
```

Figure F23. Pseudo-code d'un vérificateur maître

La fonction «appeler» est un appel système d'Unix. Il s'agit de la fonction `system`. Son usage pour l'appel d'un programme exécutable est aussi simple que l'appel d'une fonction dans un programme. Elle exécute une commande en appelant «`/bin/sh -c commande`» et ne rend le contrôle au programme appelant que lorsque la commande est finie.

Cette architecture permet de n'avoir de traitement de vérifications que dans le programme esclave. Ceci permet de découpler les problèmes liés à la hiérarchie des problèmes de vérification.

Le programme maître doit récupérer la liste des blocs qu'il doit traiter avant de relancer le traitement du premier bloc. Le programme esclave doit, quant à lui, récupérer les abstractions qu'une précédente exécution a calculée.

4.7) Communication par fichiers

Pour renvoyer la liste des modèles dont l'abstraction est manquante, le programme esclave l'écrit dans un fichier temporaire (*nom_du_bloc*`.missing`) dont le programme maître teste l'existence au retour de l'appel à l'esclave. Si ce fichier n'existe pas, alors c'est que l'esclave a pu faire tout le traitement. La vérification est terminée.

Pour que le programme esclave (traiter_bloc) puisse utiliser les abstractions dont il a besoin, nous avons choisi de stocker ces abstractions sur disque. Ceci permet :

- Une robustesse de l'application : les calculs intermédiaires sont sauvegardés.
- Un fonctionnement incrémental : seuls les blocs qui ont été modifiés sont re-vérifiés.

L'utilisation de fichiers textuels facilite la mise au point de l'application, ainsi que son usage par un concepteur de circuits en permettant une certaine lisibilité des fichiers.

L'emploi de fichiers binaires autorise de meilleures performances par leur compacité.

Afin de profiter des avantages des deux types de fichiers, nous recommandons l'usage d'une bibliothèque de fonctions en C de compression en ligne. Ceci permet d'utiliser des fichiers textes, en bénéficiant d'une compacité meilleure que les fichiers binaires. La réduction de taille est un facteur important car ces fichiers sont échangés par NFS, qui est un protocole très lent en écriture.

Par exemple, le fichier hiérarchique descriptif du processeur StaCS au format CIF occupe 22 Mo sur disque. Ce même fichier réduit par compression sans perte de données n'occupe plus que 4 Mo sur disque. La compression nous offre un gain de 80%. Ce gain ne vise pas à limiter l'occupation disque. En mars 95, 8000F permettent d'acheter 32 Mo de RAM ou 4 Gigas de disque. L'espace disque n'est pas une ressource critique. En revanche, le gain offert par la compression nous permet une augmentation des performances. La lecture de ce fichier de 22Mo disponible sur un serveur NFS de la dernière génération, à travers un réseau Ethernet prend 73 secondes. Lorsqu'il est réduit par compression à 4 Mo, sa lecture ne prend que 13 secondes. La différence entre les deux (73 - 13) nous laisse une minute de temps CPU pour décompacter le fichier. Or, il ne faut pas autant de temps pour la décompaction. D'un point de vue du temps écoulé, à cause de la lenteur de NFS, il est plus rapide de lire un fichier compressé et de le décompacter au vol, que de lire ce fichier non-compressé.

Nous avons fait des mesures sur la décompression d'un fichier d'un disque réseau NFS au disque local d'une station. La commande "`gzip -dc source > dest`" est beaucoup plus rapide qu'une simple copie. Bien que la compression soit plus coûteuse en temps processeur que la décompression, l'inverse est aussi vrai. Le transfert vers un disque réseau par la commande "`cat source | gzip > dest`" est plus rapide qu'une simple copie par la commande "`cp source dest`".

(Remarque nous avions construit une librairie de compression en ligne basée sur le programme standard compress qui offre un taux de compression moyen de 2 à 1. Notre fichier exemple était réduit à environ 6Mo, soit un gain de 75%. Ce problème d'accès aux disques partagés par le réseau ne doit pas être caractéristique que des applications de CAO pour VLSI puisque, en mai 95, nous avons eu accès à une version bêta d'une librairie similaire conçue par Jean-Loup Gaillyet Mark Adler [61]. Leur librairie est plus complète, et est dérivée du programme plus performant gzip qui offre un taux de compression moyen de 3 à 1. Nous avons intégré cette librairie à la place de la notre.)

Les tâches esclaves ne communiquent pas directement entre elles. Elles passent par l'intermédiaire de fichiers. Ceci permet une indépendance entre tâches et simplifie la programmation.

4.8) Parallélisation et distribution

Il semble aisé de paralléliser notre vérificateur hiérarchique. Il suffit de remplacer l'appel de la fonction `system` d'Unix dans le programme maître (Voir Pseudo-code d'un vérificateur maître, page 39.), par un appel à la fonction `pvm_spawn` de la librairie de distribution. Le parcours récursif de l'arbre hiérarchique s'effectue en parallèle :

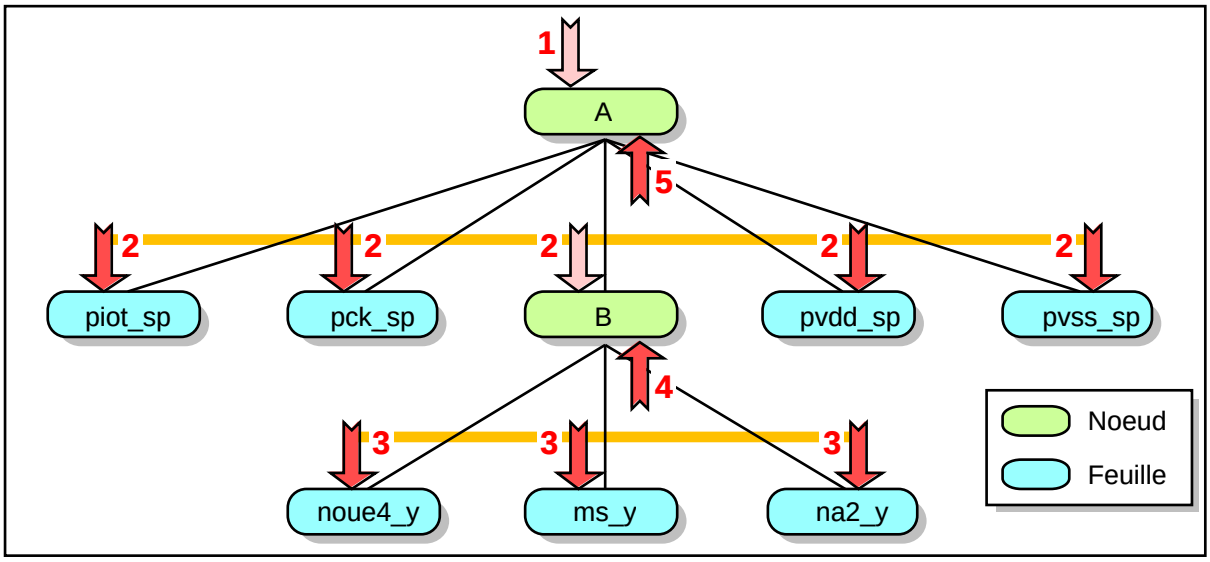

Figure F24. Parcours récursif parallèle d'un arbre hiérarchique

Mais, cette fonction `pvm_spawn` est asynchrone : elle rend la main immédiatement et non à la fin du programme lancé. Ceci permet de paralléliser les opérations, puisque nous pouvons lancer plusieurs programmes esclaves successivement. Il faut attendre la fin des programmes lancés pour agir en réaction.

Le programme esclave peut se terminer de deux manières :

- FINI : la vérification s'est bien passée.
- MANQUE : la vérification n'a pu être effectuée par suite du manque d'une ou plusieurs abstractions des fils.

Le pseudo-code du programme maître devient :

```
traiter_circuit (bloc) c'est :
      lancer (traite_bloc (bloc))
      attendus := 1
      repeter
            attendre (message)        bloquant
            si (message = MANQUE)
                  lire (blocs_besoin)
                  memoriser (bloc_en_manque, blocs_besoin)
                  pour tous (blocs_besoin)
                        lancer (traite_bloc (bloc_besoin))
                        attendu += 1
                  finpour
            finsi
            si (message = FINI)
                  attendus -= 1
                  pour tous (blocs_en_manque)
                        retirer (bloc_fini) des (blocs_besoin)
                        si (bloc_fini = dernier_bloc_besoin)
                              lancer (traite_bloc (bloc_en_manque))
                        finsi
                  finpour
            finsi
      tantque (attendus > 0)
fin
```

Figure F25. Pseudo-code d'un vérificateur maître parallèle

La fonction `traiter_circuit` n'est plus récursive.

Si la vérification du bloc en cours ne peut avoir lieu par manque d'abstractions, alors, le programme maître mémorise que le bloc en cours a besoin de certains de ses fils. Puis il lance successivement en parallèle la vérification de ces fils. Si la vérification du bloc en cours s'est bien passée, alors, il faut relancer les blocs-pères qui étaient en attente de l'abstraction de ce bloc.

4.9) Synchronisations

Les traitements des blocs ont lieu en parallèle. Ceci ne pose pas de problème pour le chargement simultané d'une même abstraction par plusieurs tâches esclaves. Mais la génération de l'abstraction doit être protégée vis-à-vis des tâches utilisatrices.

Pour cela, un mécanisme de synchronisation, soit à base de messages échangés avec le programme maître, soit à base de verrouillage de fichiers, doit être utilisé.

4.10) Réglage de granularité

L'abstraction est une vue réduite d'un bloc. Pour que la méthode soit efficace, il faut que l'abstraction entraine une réduction significative de la quantité d'informations manipulée. Pour cela, il faut que les feuilles de la hiérarchie ne soient pas «trop petites». Les feuilles de l'arbre hiérarchique sont a priori des cellules de bibliothèques standards. Il faudrait pouvoir considérer comme feuilles des blocs plus gros :

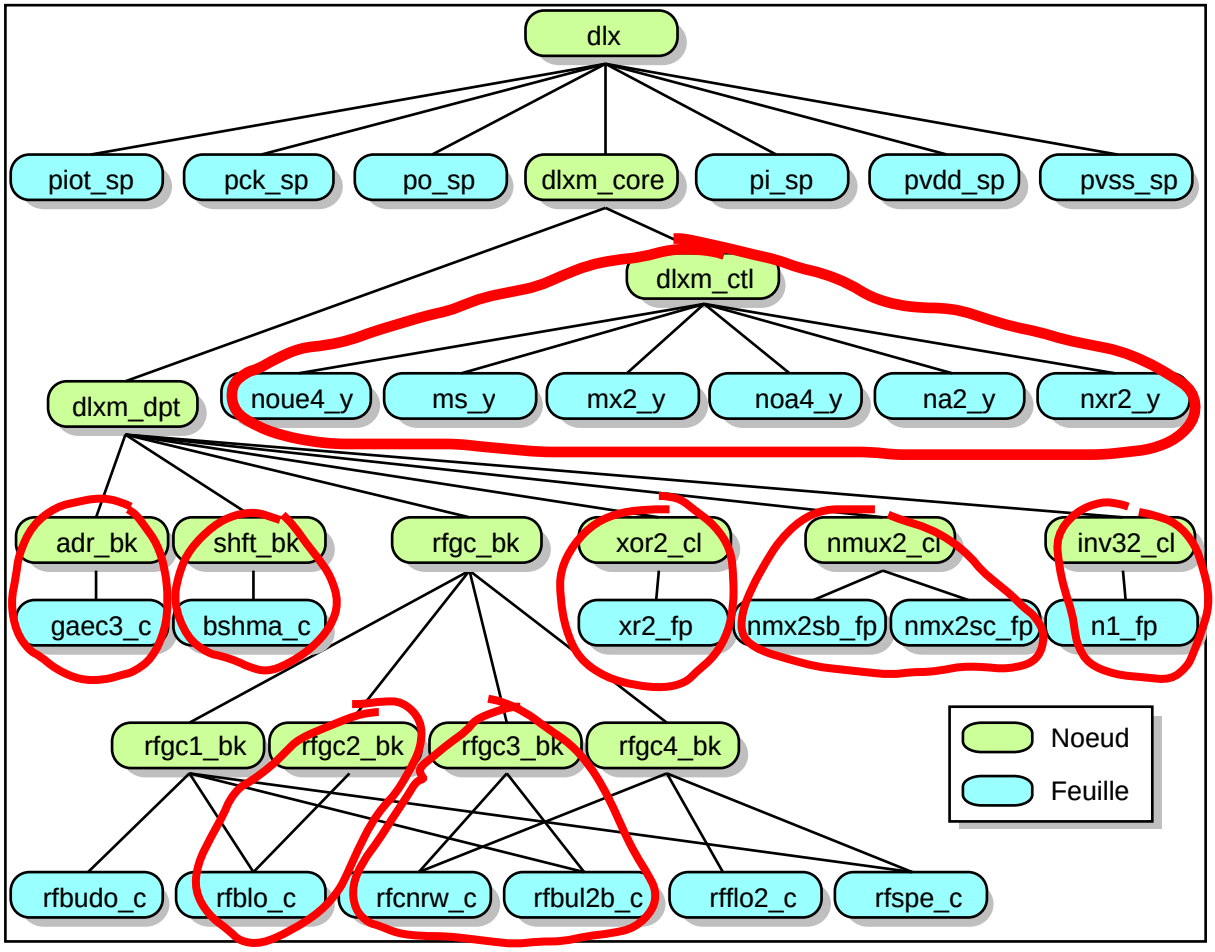

Figure F26. Granularité explicite d'un arbre

Le concepteur de circuit, qui utilise le vérificateur distribué, doit avoir la possibilité de fournir une liste de blocs à considérer comme des éléments terminaux. Ces blocs seront mis «à plat» et traités comme des feuilles.

4.11) Granularité maximum

Le plus gros bloc à plat, que nos applications de vérifications puissent traiter, est déterminé par la taille mémoire des stations de travail utilisées. Une station de travail sous Unix dispose de deux types de mémoire :

La mémoire **physique** de la station qui se compte en dizaines de Méga-octets. Le temps d'accès a peu évolué depuis une demi-décennie. Il s'agit toujours de circuits d'une vitesse de 60-80ns de temps d'accès. La capacité a, par contre, augmenté (Vitesse ± 17%, capacité 50% / an).

La mémoire **virtuelle** de la station : Unix offre un mécanisme d'extension de l'espace adressable en utilisant le disque. Ceci permet de doubler à quadrupler la mémoire réelle. Ce mécanisme est souvent utilisé par les applications de VLSI qui sont extrêmement gourmandes en mémoire.

Afin de déterminer quelle taille mémoire nous devons retenir pour déterminer la granularité maximum d'une tâche, nous avons mesuré la bande passante de trois stations. Nous avons effectué une requête consistant à demander l'allocation (`malloc`) et l'initialisation (`memset`) de *N* zones mémoires de 1 Méga-octets en faisant varier *N*. Nous avons obtenu le graphique suivant :

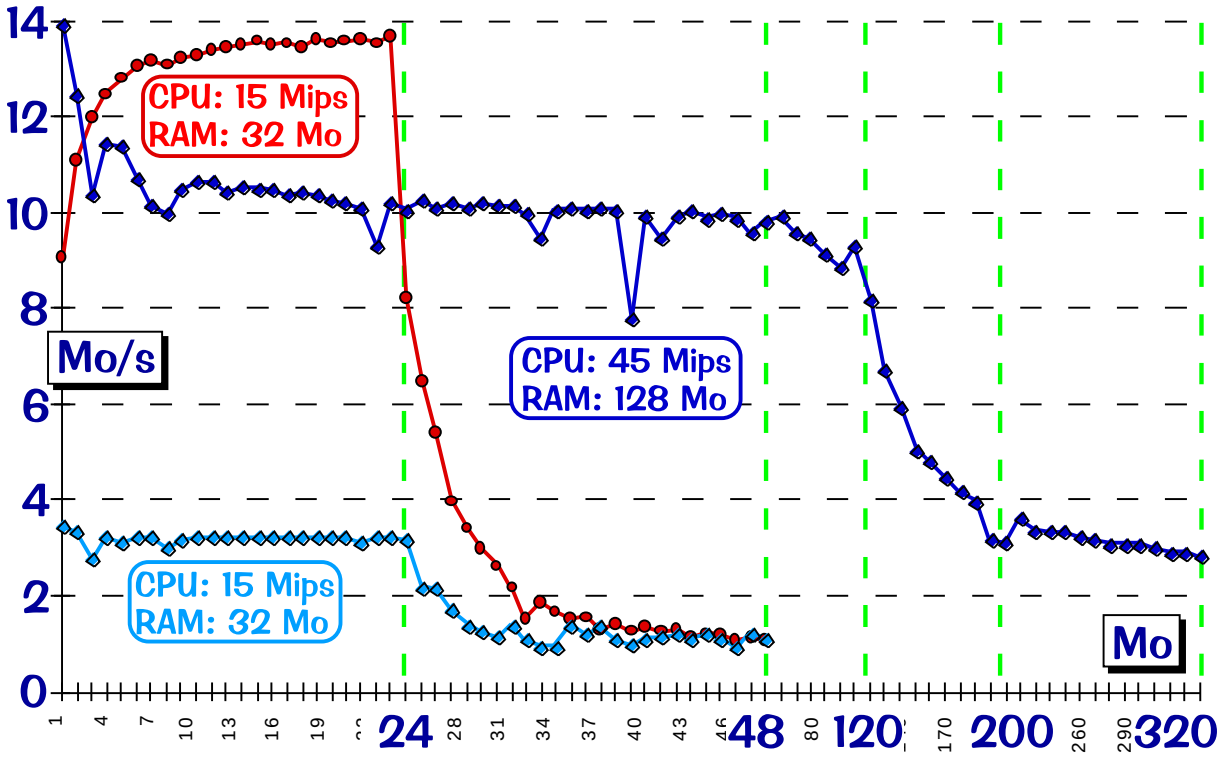

Figure F27. Mesures de bande passante de la mémoire

L'axe des abscisses représente la quantité d'octets utilisés à chaque essai. L'axe des ordonnées représente le débit obtenu. Afin de faciliter la lisibilité globale du schéma, l'axe des abscisses subit un changement d'échelle : l'échelle correspond aux unités jusqu'à la valeur 48, puis représente les dizaines d'unités ensuite.

Nous avons effectué des mesures sur trois machines :

La première série de mesures visait à étalonner les performances. Nous avons utilisé une machine ancienne (Sun Sparc IPC sous SunOS 4.1.1) de 15 Mips de puissance et dotée de 32 Mo de mémoire centrale (courbe losanges bleu clair). La courbe est une forme à deux paliers. Cette machine pour toute demande inférieure à 24 Mo offre une bande passante de 3 Mo/s. Si la demande est supérieure, sa vitesse chute à 1 Mo/s.

Pour la deuxième série de mesures, nous avons choisi une machine du même constructeur, sous le même système d'exploitation, mais d'une génération beaucoup plus récente (Sun Sparc 10/30). Elle offre 45 Mips et est dotée de 128 Mo de mémoire centrale (courbe losanges bleu foncé). La courbe a aussi une forme à deux paliers. Pour toute demande inférieure à 120 Mo, cette machine offre une bande passante de 10 Mo/s, sinon elle chute à 3 Mo/s.

Remarquons que 120=128-8, et 24=32-8. Ces huit Mo constants peuvent être assimilés à l'occupation en mémoire du noyau du système et des parties résidentes de l'interface graphique. Nous pouvons donc interpréter cette courbe de la façon suivante : Dès que la demande dépasse la mémoire centrale, la vitesse d'accès à la mémoire chute à la vitesse d'accès au disque.

Nous avons mesuré une troisième machine : un 486DX-33 sous Linux 1.0.9. L'Unix des deux premières machines considère l'espace de swap comme étant l'espace adressable total. L'Unix de la troisième machine considère l'espace de swap, comme une extension de la mémoire vive. A la différence des deux premières, il n'y accède que lorsqu'il n'a plus assez de mémoire libre. Les mesures (courbe cercles rouge) montrent l'aspect encore plus prononcé : Pour toute demande contenue dans la mémoire centrale, la machine offre une excellente vitesse d'accès de presque 14 Mo/s. Cette vitesse est intéressante en regard des performances brutes du processeur : 15 Mips. Dès que la demande dépasse la mémoire centrale, la vitesse d'accès chute dramatiquement pour atteindre le palier de 1Mo/s qui correspond à la vitesse du sous-ensemble disque.

Afin de garantir une performance maximale, toutes les tâches résultant du partitionnement de l'application devront avoir une taille compatible avec la mémoire physique du processeur exécutant la tâche. L'utilisation du système de swap est à réserver à Unix pour évacuer de la mémoire physique les processus inactifs. Si l'on utilise le swap pour un besoin d'augmentation de capacités, qui ne peut être satisfait par la distribution, il faut s'attendre à une forte dégradation des performances.

4.12) Ordonnancement dirigé par l'utilisateur

Prenons comme exemple un ensemble de cinq blocs dont le traitement prend 2, 1, 4, 3 et 10 heures. Le traitement total de ces cinq blocs sur un bi-processeur peut être:

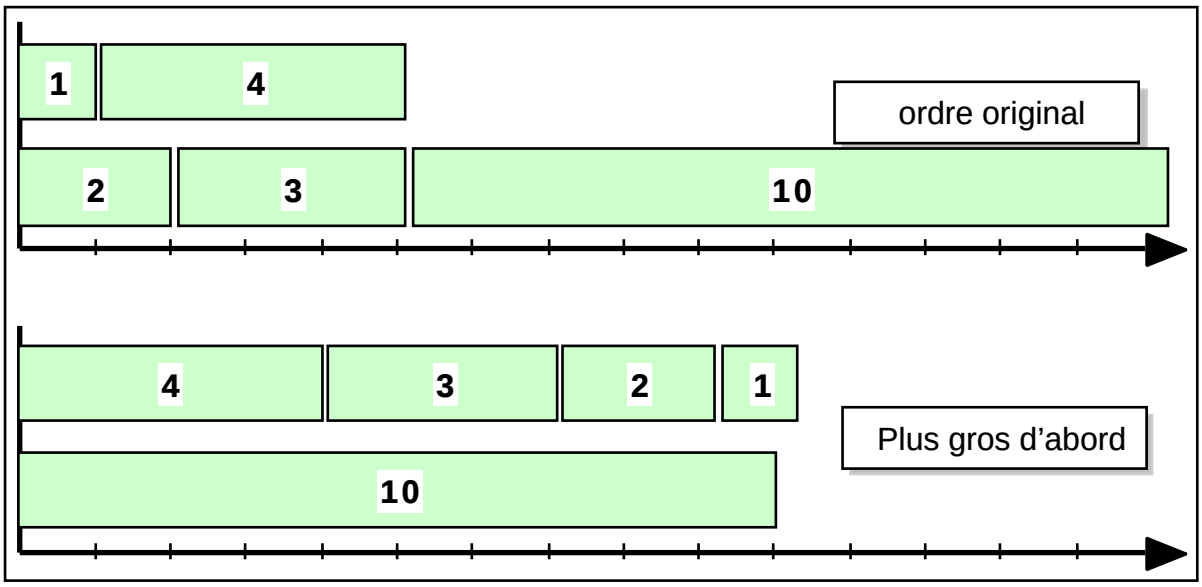

Figure F28. Ordonnancement de cinq traitements

L'axe des abscisses représente le temps. Les deux processeurs disponibles sont en ordonnées.

Selon l'algorithme d'ordonnancement retenu, la durée du traitement est de 15 ou de 10 heures.

Il est donc essentiel d'ordonnancer les requêtes lors du lancement en parallèle des tâches esclaves. Il faut disposer d'un critère permettant de prédire leur durée. Aussi bien la tâche maître, que la tâche esclave ne peuvent déterminer la durée du traitement simplement à partir du nom du modèle. Il faut donc que l'utilisateur du programme de vérification assiste le programme en lui fournissant comme directive une liste ordonnée de modèles.

L'algorithme purement récursif est non-optimal : tous les noeuds de l'arbre subissent un double chargement — une fois pour déterminer la liste de leur fils, une autre lors de leur traitement —.

L'arbre de la hiérarchie du circuit peut être considéré comme un graphe de dépendance. Cela résout partiellement le problème d'ordonnancement : en appliquant les contraintes du graphe de dépendance à la liste ordonnée des modèles, nous obtenons un ordonnancement partiel.

L'utilisateur du vérificateur est aussi le concepteur du circuit. Il est donc à même de fournir une liste ordonnée des modèles avec leurs dépendances : Il lui est facile de savoir quels blocs sont utilisés par d'autres blocs. Une syntaxe simple est de mettre sur une ligne le nom du bloc suivi des modèles qu'il utilise. Ceci nous fournit une liste des noms des modèles. Le concepteur peut aussi trier cette liste par ordre de taille relatif.

Voici un exemple de représentation textuelle de l'arbre hiérarchique suivant :.

```
noue4_y
ms_y
na2_y
B               noue4_y ms_y na2_y
piot_sp
pck_sp
pvdd_sp
pvss_sp
A               B piot_sp pck_sp pvdd_sp pvss_sp
```

Figure F29. Représentation textuelle d'un arbre hiérarchique

Avec de telles directives, le traitement peut alors commencer en parallèle de la manière suivante :

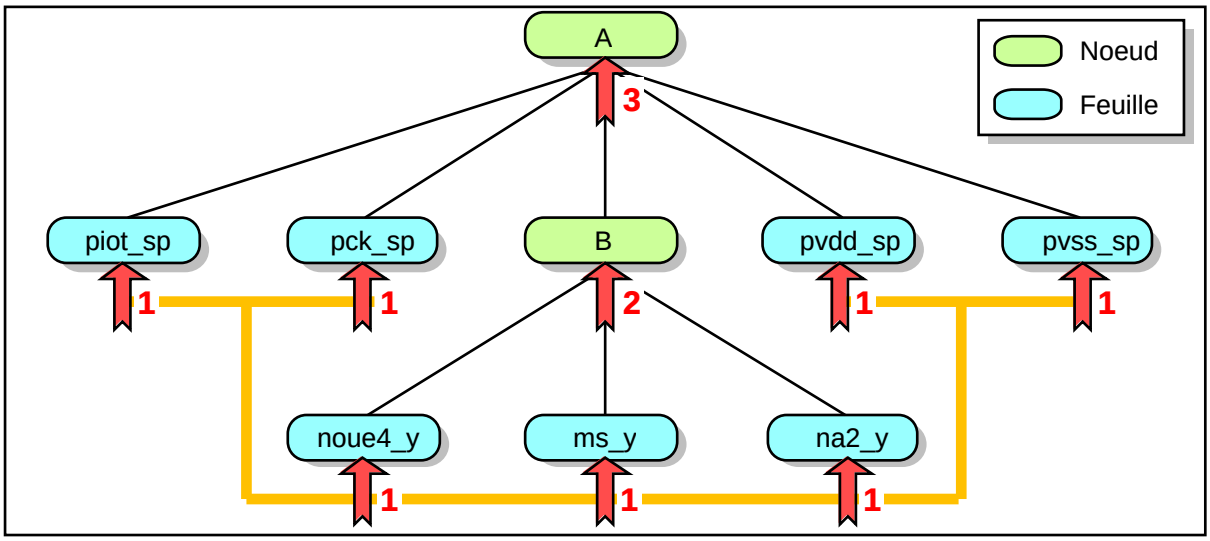

Figure F30. Parcours parallèle d'un arbre hiérarchique avec directives

Ce fichier de directives décrit en fait deux types d'information :

- Il décrit explicitement le graphe de dépendance entre les modèles du circuit : Pour chaque modèle, on décrit en effet la liste des modèles qu'il instancie. Cette première information permet d'éviter les essais infructueux (C'est à dire le lancement d'une tâche de vérification d'un modèle non-terminal, alors que toutes les abstractions filles ne sont pas encore disponibles).
- Par ailleurs, l'ordre dans lequel les modèles apparaissent dans le fichier est significatif : si deux modèles ne sont pas liés par une relation de dépendance, on lancera d'abord la vérification du premier dans la liste. En ordonnant la liste des modèles par complexité décroissante, il est possible de lancer «au début» les tâches les plus longues, ce qui est idéal du point de vue d'une exécution parallèle.

4.13) Anticipation des traitements

Certains algorithmes de vérification — comme le vérificateur de règles de dessin — n'ont pas besoin que la vérification soit terminée pour générer la vue abstraite (la vue abstraite est la couronne dans le cas du VRD). Il est donc possible de générer l'abstraction du bloc dès que celui-ci est chargé. En effet, les blocs qui ont besoin de cette abstraction sont en attente de sa génération. Cette anticipation de la mise à disposition de l'abstraction permet un parallélisme plus élevé des traitements. Le programme esclave doit signaler que l'abstraction est prête :

```
traiter_bloc (bloc), c'est :
      si (terminal (bloc))
            charger_à_plat (bloc)
            générer_abstraction (bloc)
            envoyer (PRET, bloc)
            verifier_feuille (bloc)
      sinon
            charger (bloc)
            pour tous les fils (bloc)
                  si (pas_abstraction (fils))
                        mémoriser (fils_manquant)
                  finsi
            finpour
            si (abstraction_manquante)
                  décharger (bloc)
                  envoyer (MANQUE, bloc, liste_fils_manquant)
                  terminer
            finsi
            pour tous les fils (bloc)
                  charger_abstraction (fils)
            finpour
            générer_abstraction (bloc)
            envoyer (PRET, bloc)
            vérifier_noeud (bloc)
      finsi
      envoyer (FINI, bloc)
fin
```

Figure F31. Vérificateur esclave avec anticipation

Le programme maître de la vérification doit prendre en compte ce nouveau message :

```
traiter_circuit (bloc) c'est :
      lancer (traite_bloc (bloc))
      attendus := 1
      repeter
            attendre (message)        bloquant
            si (message = MANQUE)
                  lire (blocs_besoin)
                  memoriser (bloc_en_manque, blocs_besoin)
                  pour tous (blocs_besoin)
                        lancer (traite_bloc (bloc_besoin))
                        attendu += 1
                  finpour
            finsi
            si (message = PRET)
                  pour tous (blocs_en_manque)
                        retirer (bloc_pret) des (blocs_besoin)
                        si (bloc_pret = dernier_bloc_besoin)
                              lancer (traite_bloc (bloc_en_manque))
                        finsi
                  finpour
            finsi
            si (message = FINI)
                  attendus -= 1
            finsi
      tantque (attendus > 0)
fin
```

Figure F32. Vérificateur maître avec anticipation

Ceci permet de déséquentialiser la phase finale du traitement. Le traitement du bloc racine peut démarrer parallèlement à la vérification de ses blocs fils.

4.14) Régulation de charge

Lorsque toutes ces tâches démarrent simultanément, elles risquent de saturer la machine virtuelle (l'ensemble des processeurs). Dans le meilleur cas, toutes ces tâches entrent en compétition pour la mémoire physique et déciment la performance brute du processeur. Dans le pire cas, la capacité mémoire est dépassée, et l'application ne peut parvenir à son terme.

Afin de prendre en compte ce problème, l'application de vérification de VLSI doit limiter la charge qu'elle impose à la machine virtuelle.

Une heuristique, simple mais efficace, est d'affecter une seule tâche par processeur, et de mettre en attente le reste des tâches.

Il est possible de suspendre les requêtes d'exécution jusqu'à ce qu'un processeur se libère, car les tâches esclaves ne communiquent pas directement entre elles.

4.15) Placement dynamique et migration

Il est possible de compléter une librairie de distribution de programmes par un gestionnaire de ressources adapté aux applications de vérifications de VLSI.

Sur des stations homogènes et dédiées à la vérification, le seul critère à prendre en compte est la taille de mémoire physique de la station. La somme des tailles des tâches sur une même station doit être inférieure à la taille de la mémoire physique de cette station. Le multiplexage des phases de calcul avec celles d'entrée-sortie d'Unix apportera alors un léger gain de performances, si le nombre de tâches simultanées reste limité.

Dans le cadre d'un réseau de machines hétérogènes, un système de placement dynamique, selon les caractéristiques prévues de la tâche (taille, durée) doit être offert par le système de distribution.

Sur une architecture non-dédiée (avec d'autres utilisateurs), le système de placement dynamique doit être complété par un système de migration dynamique en cas de surcharge soudaine d'un processeur.

4.16) Conclusion

Dans ce chapitre, nous avons présenté les différentes techniques qui nous permettent de faire de l'approche «distribution guidée par la hiérarchie» une méthode à la fois souple et efficace.

Au prix de la définition explicite par le programmeur d'un concept : — l'abstraction d'un bloc qui dépend évidemment du type de vérification à effectuer —, notre méthode offre une réponse à l'accroissement de taille toujours plus importants des circuits VLSI : la parallélisation par distribution de tâches sur un réseau de stations.

Cette distribution reste robuste grâce aux techniques de fonctionnement incrémental employées.

Pour valider notre méthode, et servir d'exemple aux concepteurs d'outils de vérification de VLSI, nous avons appliqué notre procédé sur un outil existant de vérification à plat : le vérificateur de règles de dessin *DRuC*.

a

5) Application au VRD

Dans ce chapitre, nous présentons l'application de la méthode d'accélération à un programme particulier, représentatif de la classe des outils de vérification de circuits VLSI : le vérificateur de règles de dessin (VRD). Nous commencons par expliciter les concepts introduits par la hiérarchisation des circuits. Nous détaillerons le fonctionnement de notre outil en faisant ressortir les diverses techniques appliquées (aspects hiérarchique, distribué, incrémental, et réentrant).

5.1) Concepts introduits

La vérification hiérarchique introduit quelques concepts dans l'ensemble de la chaine de cao-vlsi Alliance qui nous sert de support. Certains sont très généraux (modèles, instances, boite d'aboutement). D'autres sont plus spécifiques à la vérification hiérarchique (transparences, vias virtuels).

5.1.1) Modèles et instances

L'introduction de la hiérarchie dans les dessins peut se résumer par : un dessin n'est plus seulement fait de rectangles, mais il peut aussi contenir des références à d'autres dessins. Voici quelques définitions :

- modèle : il s'agit de l'unique description d'un bloc.
- instance : il s'agit de la référence à un bloc.

5.1.2) Connecteurs

Un connecteur est composé d'un rectangle généralement situé en périphérie du bloc auquel on affecte une étiquette particulière. Les connecteurs définissent l'interface du bloc. Il servent à indiquer à un programme de niveau hiérarchique immédiatement supérieur (un routeur) où il peut venir se connecter.

5.1.3) Vias

Le via est un objet qui permet d'effectuer une connexion entre deux niveaux de métal. Dans les vues physiques, le via est réalisé par un rectangle de métal de niveau inférieur, un rectangle de métal de niveau supérieur, et un rectangle représentant le trou de communication entre les deux niveaux :

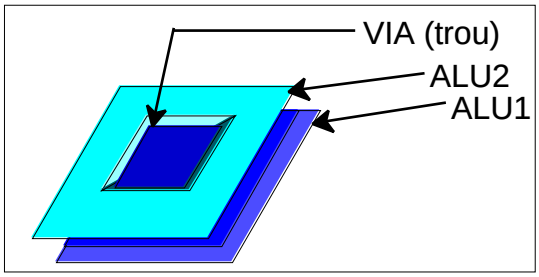

Figure F33. Un «via»

Par abus de langage, le nom de via est utilisé pour désigner aussi bien l'objet complexe macro-généré en trois rectangles de métal et trou, que le niveau correspondant au trou lui-même.

5.1.4) Boite d'aboutement

Afin de simplifier les rafraichissements d'écrans dans un éditeur de layout, et de réduire la complexité des informations manipulées par les outils de placement / routage, une instance n'est affichée ou manipulée que sous une forme simplifiée : une boite (à l'instar de la boite noire des schémas logiques).

Cette boite aurait pu être l'enveloppe (*boundingbox*) constitué du plus petit rectangle englobant tous les éléments du modèle. L'expérience acquise au cours de dix ans d'utilisation de la chaine de cao-vlsi supportant notre démonstrateur, ainsi que la volonté de laisser le concepteur de circuits le plus libre possible, ont conduit à l'adoption d'une nouvelle boite dite “boite d'aboutement” (*abut-*

mentbox) qui est à la base de l'approche hiérarchique. Deux types d'assemblage hiérarchique sont utilisés :

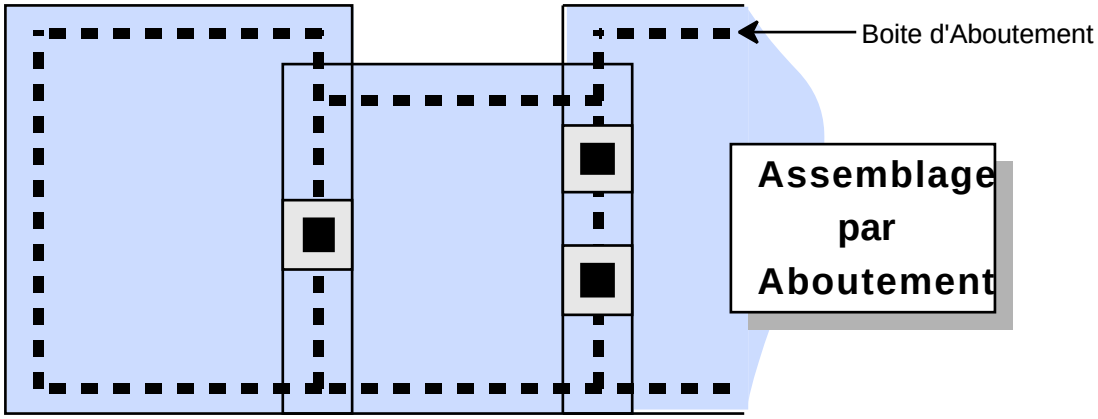

Figure F34. Assemblage par aboutement

- L'assemblage par aboutement : les blocs se chevauchent sur une zone de la largeur d'un via. Leur recouvrement est limité par l'interdiction de recouvrement de deux boites d'aboutement situées au même niveau hiérarchique. Chaque bloc est pénétré par un rectangle dont la longueur est en général un demi-via. Les blocs peuvent partager un via qui est alors dupliqué dans chacun des modèles.

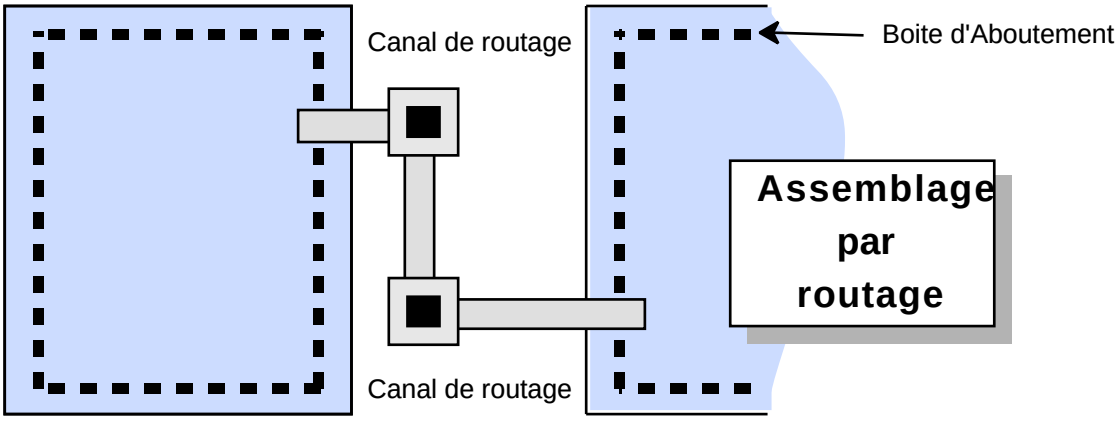

Figure F35. Assemblage par routage

- L'assemblage par routage : les blocs ne se chevauchent pas. Les fils contenus dans les canaux de routage pénètrent la boite d'aboutement des autres blocs d'un demi connecteur au plus.

L'utilisation de boites d'aboutement rectangulaires assure la simplicité des modèles d'assemblage pour le placement des instances. La hiérarchie introduit la notion de niveaux hiérarchiques. Seuls, certains des éléments, qui composent un bloc de niveau *n*, seront utiles au niveau *n+1*. Dans notre contexte, c'est la boite d'aboutement qui définira les règles de recouvrement entre deux instances ou entre une instance et un fil de routage.

La boite d'aboutement est destinée à spécifier la contrainte de recouvrement entre deux blocs qui est :

- Deux blocs de même niveau hiérarchique ne peuvent pas superposer leur boite d'aboutement.

Remarque: Il est possible d'utiliser l'approche hiérarchique même si la boite d'aboutement n'est pas définie. Prenons le cas de l'utilisation d'un format CIF (*Caltech Intermediate Form*) : Il n'est pas obligatoire que chaque modèle soit doté d'une boite d'aboutement. Dans ce cas, le vérificateur de règles de dessin dotera automatiquement chaque modèle d'une telle boite si nécessaire. Elle sera égale à l'enveloppe réduite d'une constante *mu* spécifiée par le concepteur. Cette boite calculée ne sera toutefois pas optimale par rapport à une boite d'aboutement définie explicitement par le concepteur.

5.1.5) Constante de pénétration

Les blocs sont considérés par la méthodologie comme essentiellement opaques. Nous avons pu remarquer que les dessins conçus par aboutement étaient généralement bornés par une pénétration d'une profondeur inférieure ou égale à un demi-via. Les dessins provenant d'une phase de routage sont, quant à eux, pénétrés d'au plus un demi-connecteur en général. Les circuits provenant d'un compilateur de chemin de données (*datapath compiler*) partagent les rails d'alimentation entre les cellules précaractérisées. La profondeur de pénétration dans ce cas est la largeur du rail d'alimentation partagé.

Bien que l'on sente intuitivement que la distance de pénétration des rectangles étrangers dans la boite d'aboutement est faible, il est impossible de la quantifier de façon générale. Sa valeur dépend de la stratégie d'assemblage choisie et de la bibliothèque de cellules utilisée. Il faut pouvoir modifier facilement ce paramêtre. C'est le concepteur de circuits qui sait si son fichier provient d'un routage ou d'un aboutement. Nous nous remettons au concepteur de circuit en lui laissant la possibilité de spécifier optionnellement la valeur de la pénétration maximum autorisée.

La méthode hiérarchique décrite ici n'impose pas une technique particulière d'assemblage des blocs. Dans 90% des cas, l'assemblage hiérarchique sera obtenu par un routeur, et nous pouvons donc fixer une valeur par défaut qui soit minimale.

Nous pouvons completer les contraintes de recouvrement entre deux blocs énonçées au paragraphe précédent :

- La boite d'aboutement réduite (BAR) est le rectangle obtenu en réduisant la boite d'aboutement d'une distance égale à la distance de pénétration maximale : Un rectangle étranger au bloc ne peut pas pénétrer la BAR. (Nous appelons rectangle étranger à un bloc tout rectangle appartenant à un des fréres ou au père dans la hiérarchie).

5.1.6) Transparences

La notion de boîte d'aboutement nous permet de spécifier des blocs essentiellement opaques. Le concepteur peut toutefois indiquer de manière explicite à un routeur qu'il peut faire passer un fil à travers un bloc à tel endroit bien précis.

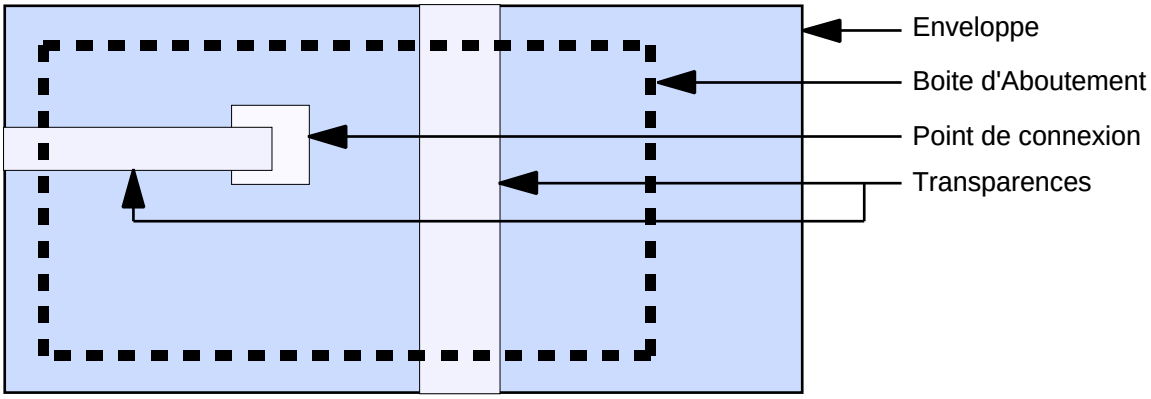

Figure F36. Transparences

La transparence permet à un routeur d'introduire un fil d'un certain niveau de métal. La transparence représente, lors de l'instanciation d'un bloc, la possibilité de violation de ses frontières définies par sa boite d'aboutement réduite (BAR). La notion de transparence permet une vérification hiérarchique, c'est à dire sans avoir à considérer le contenu explicite du bloc. Les contraintes de recouvrement entre deux blocs deviennent donc :

- Deux blocs de même niveau hiérarchique ne peuvent pas superposer leur boite d'aboutement.
- Un rectangle étranger à un bloc ne peut pénétrer la boite d'aboutement réduite (BAR) qu'à l'intérieur d'un rectangle de transparence du même métal.

Les transparences sont des objets virtuels : Ces objets ne correspondent pas à des niveaux physiques et sont ignorés lors de la fabrication.

Comme l'extracteur d'équipotentielles ne voit pas les transparences, il ne peut déterminer les cas de court-circuits potentiels. Il est du ressort du VRD hiérarchique de procéder à des verifications "connectiques" en plus des vérifications de règles de dessin :

- Afin d'éviter les court-circuits, les transparences ne doivent pas être en contact avec un rectangle du métal qu'elles représentent.

Les niveaux de métal soumis à transparence sont appelés les "T-Niveaux".

La vérification des règles de dessin comporte une suite de mesures. Chaque mesure de distance peut être inférieure, supérieure, où égale à une valeur. La définition de la transparence que nous avons choisie — "une zone dans laquelle on peut passer n'importe quel rectangle de même métal" — em-

pêche de vérifier des règles de “distance maximale” pour les niveaux de métal soumis à transparence : “distance entre ALU et DIFF doit être inférieure à 3”. Ce cas est académique et ne se rencontre pas avec les technologies actuelles.

Ainsi que cela est précisé dans la Figure 36, “Transparences”, page 55, une transparence sert à deux besoins : passer impunément à travers un bloc sans interagir avec ce bloc, mais aussi à se connecter dans un bloc. Nous avons besoin d'un objet pour établir la connection.

5.1.7) Connecteurs virtuels

Un connecteur virtuel correspond à une autorisation donnée au routeur de poser un rectangle susceptible de créer une connexion avec l'intérieur du bloc.

Un connecteur virtuel est décrit par un nouveau dédoublement des niveaux de métal soumis à transparence : les “V-Niveaux”. A l'instar des “T-Niveaux”, ils respectent les mêmes règles de dessin que les rectangles de métal dont ils sont issus. A la différence des transparences, il peuvent recouvrir n'importe quel rectangle de métal ou de transparence du métal dont ils sont issus. Par exemple, au niveau de métal ALU1, on associe les niveaux TALU1 et VALU1.

Une Transparence sert à traverser un circuit sans s'y connecter, tandis qu'un connecteur Virtuel permet de traverser un circuit pour s'y connecter. Un rectangle de Transparence ne doit pas provoquer de court-circuit, tandis qu'un connecteur virtuel permettra une connexion. Les connecteurs virtuels sont très utilisés dans le compilateur de chemin de données pour indiquer plusieurs points de connexion possibles. Le routeur choisira le meilleur. Dans les cellules du compilateur de chemin de données, les connecteurs virtuels ne peuvent pas être tous simultanément utilisés sans provoquer de courts-circuits. Le modèle des connecteurs virtuels supporte évidemment l'extraction hiérarchique : alors que l'extracteur d'équipotentielles ne “voit” pas les T-niveaux, il traite les V-niveaux comme des rectangles physiques.

Un via virtuel correspond à une autorisation donnée à un programme de niveau hiérarchique immédiatement supérieur de poser un via physique à l'emplacement spécifié par le via virtuel. Le via virtuel sera physiquement réalisé par l'usage de trois “V-Niveaux”.

Dans une même cellule, il peut exister plusieurs vias virtuels électriquement équivalents. Grace à ce concept, on renverse le sens de propagation des contraintes d'assemblage : c'est le routeur de niveau *n+1* qui décide où effectuer la connection.

Ce mécanisme est largement utilisé par le compilateur de chemin de données [9] de la chaine Alliance.

Les “V-niveaux” sont des concepts transitoires. Ils n'existent que dans le niveau de hiérarchie en cours de traitement. Lors du passage au niveau de hiérarchie supérieur, ils sont traduits en “T-niveaux” de manière à simplifier les règles hiérarchiques sur la boite d'aboutement.

5.2) Traitement Hiérarchique

Dans ce paragraphe, nous présentons l'aspect hiérarchique dans le traitement que doit effectuer le Vérificateur de Règles de Dessin.

5.2.1) Architecture simplifiée du vérificateur

Notre vérificateur classe les blocs en deux types :

- les blocs qui seront considérés comme des “feuilles” de la structure et seront traités à plat par le VRD.
- les blocs qui seront traités de façon hiérarchique.

Si aucune indication ne lui est fournie par l'utilisateur, le VRD effectue un parcours de l'arbre hiérarchique de manière à traiter d'abord les blocs feuilles d'une manière monolithique. Puis il utilise les résultats obtenus pour traiter l'étape hiérarchique immédiatement supérieure.

Le programme de vérification, lorsqu'il vérifie un bloc, n'a pas connaissance de tout le contenu des sous-blocs. Comme les pénétrations et les recouvrements entre sous-blocs sont limités, une erreur de dessin ne peut intervenir qu'avec un des rectangles situé sur la périphérie d'un sous-bloc. l'»ensemble des rectangles situés sur la périphérie du bloc» constitue la «couronne».

La couronne est l'ensemble des données qui seront suffisantes pour la vérification de niveau hiérarchique supérieur.

5.2.2) Traitement au niveau hiérarchique zéro

Notre VRD fonctionne selon le schéma suivant :

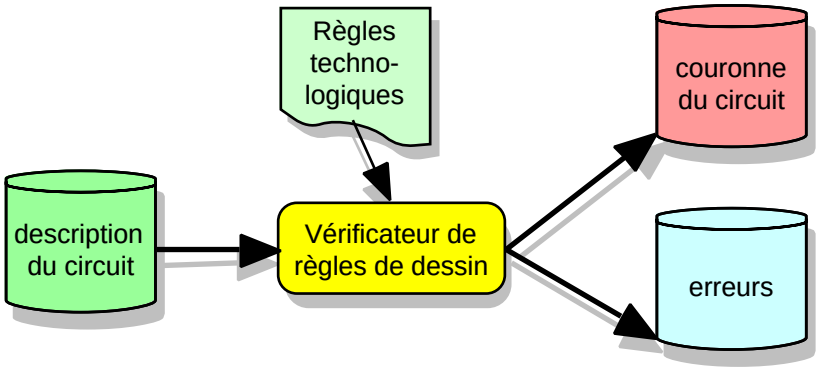

Figure F37. Fonctionnement du VRD

Deux résultats sont fournis par le vérificateur :

- Une base de données contenant les violations de règle de dessin relevées dans le bloc traité. Cette base de données est destinée à être consultée par l'utilisateur, et à guider l'éditeur de layout dans son positionnement sur les rectangles en erreur.
- la réduction du bloc traité aux informations que le vérificateur de niveau supérieur doit connaitre : la couronne.

5.2.2.1) Détection des erreurs

Dans le cadre du traitement à plat d'un bloc terminal (c'est-à-dire ne contenant pas d'instances), nous utilisons le fonctionnement suivant pour un vérificateur de régles de dessins : Le vérificateur ne travaille que sur un ensemble de rectangles. La liste des contraintes technologiques à respecter fournie par le fondeur peut s'exprimer en termes de règles de dessin de deux types : les règles à un corps qui ne s'appliquent que sur un seul rectangle (largeur > 2), et des règles s'applicant à deux rectangles (distance > 4). Il est paramétré par un ensemble exhaustif de régles à un ou deux corps suffisant à assurer la validité d'un dessin. Ces régles sont constituées d'une vérification élémentaire (largeur<5, distance>32, ...) suivie du ou des deux niveaux auquels cette vérification élémentaire s'applique ([ALU1], ou [DIFN,ALU1]). Ces vérifications élémentaires sont asymétriques et permettent de garantir qu'une même vérification ne sera pas appliquée deux fois sur une paire de rectangles.

Le niveau hiérarchique zéro est différent d'un traitement non hiérarchique dans le sens ou les concepts introduits par la hiérarchie doivent être traités spécialement :

- On vérifie l'existence de la boite d'aboutement.

- Afin de traiter les transparences, nous devons modifier le jeu de règles standard. Pour tous les niveaux de métals qui sont sujets à transparence, des règles de vérification du niveau de transparence associé sont générées. Le VRD considère les transparences comme des objets physiques à vérifier :

Les règles à un corps sont expansées de la manière suivante, lorsqu'elles s'appliquent sur un niveau de métal sujet à transparence : une nouvelle règle comportant les mêmes vérifications élémentaires mais s'applicant au niveau de transparence de ce métal est générée.

Les règles à deux corps se subdivisent en trois classes :

- Soit un seul des niveaux est sujet à transparence, alors une nouvelle règle contenant les mêmes vérifications mais avec le niveau remplaçé par sa transparence est générée.
- Soit les deux niveaux sont sujets à transparence, et il s'agit du même niveau, alors deux nouvelles règles sont générées, l'une avec le remplacement d'un seul niveau par sa transparence, l'autre avec le remplacement des deux niveaux par leur transparence.
- Soit les deux niveaux sont différents et sujets à transparence, alors trois règles sont générées avec le remplacement successif des niveaux par leur transparence, puis avec les deux transparences.

Exemple: Il existe des niveaux T_ALU1 et T_ALU2 mais pas T_DIFN et T_DIFP

R(DIFN) et R(DIFN, DIFP) ne donnent rien.

R(ALU1) donne R(T_ALU1), et R(DIFN,ALU1) donne R(DIFN,T_ALU1)

R(ALU1,ALU1) donne R(ALU1,T_ALU1) et R(T_ALU1,T_ALU1)

R(ALU1,ALU2) donne R(ALU1,T_ALU2), R(T_ALU1,ALU2), et R(T_ALU1,T_ALU2).

Enfin une règle interdisant le chevauchement d'un rectangle de transparence sur un rectangle de son métal est ajoutée.

- Les "V-niveaux" des connecteurs virtuels donnent lieu aux mêmes expansions de règles de dessin que les transparences à l'exception de la règle qui interdit le chevauchement. Le nombre de règles générées est plus important que pour les transparences puisque les "V-niveaux" interagissent avec le métal dont ils sont issus, mais aussi avec la transparence associée à ce métal.

5.2.2.2) Génération de la couronne

Puisque les blocs ne se penétrent pas (ou peu), il est raisonnable de penser que seuls les rectangles situés en périphérie sont nécessaires pour la vérification au niveau hiérarchique supérieur.

Il ne faut pas confondre le *halo* [14] de l'instance, et la couronne du modèle. Le *halo* est externe au bloc, tandis que la couronne est interne au bloc. Voici comment construire la couronne.

L'ensemble des règles à vérifier qui sont décrites par le technologue nous permet de définir la C.T.M.(Niv) la Contrainte Technologique Maximale pour un niveau donné. Il s'agit de la distance à partir de laquelle plus aucun rectangle ne peut causer d'erreur avec un rectangle de ce niveau. Au-dela de cette distance, un rectangle est "trop loin" pour pouvoir créer une erreur. Cette distance s'appelle *DRID (Design Rule Interaction Distance)* en anglais.

La couronne de chaque niveau est composée de tous les rectangles qui ne sont pas inclus dans le rectangle défini par la boite d'aboutement réduite (BAR) moins une épaisseur CTM(Niv).

- La couronne du bloc est obtenue par la réunion des couronnes de chaque niveau.
- La couronne contient aussi tous les rectangles de transparence du bloc.
- Les rectangles de V-niveau sont remplacés par leur équivalent en T-niveau (transparences), et sont ajoutés à la couronne après modification, de manière à simplifier les règles hiérarchiques.

Voici la couronne d'un modèle sans transparences :

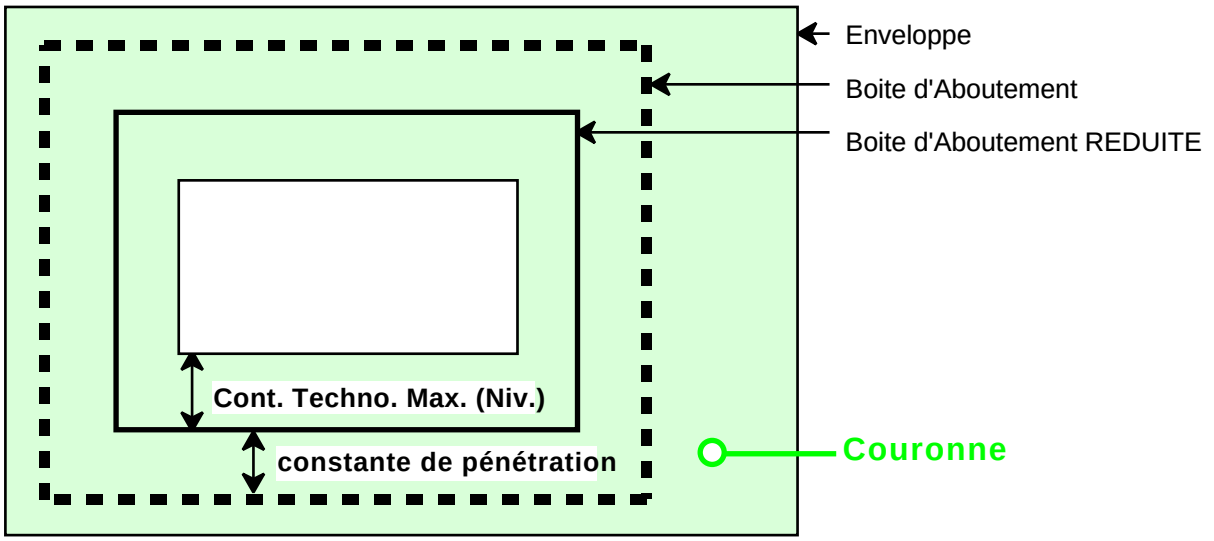

Figure F38. Couronne d'un modèle

La couronne ainsi obtenue est associée au modèle et non à l'instance.

Toutefois, notre définition de la couronne est dépendante de la *constante de pénétration* qui est un paramêtre controlé par l'utilisateur. Le VRD hiérarchique doit vérifier, lorsqu'il utilise une couronne à la place d'un modèle, que sa constante de pénétration actuelle est inférieure ou égale à la constante de pénétration qui a permis d'obtenir la couronne. Sinon la couronne doit être recalculée.

5.2.3) Traitement au niveau hiérarchique *n*

Nous sommes maintenant dans le cas où notre bloc contient une ou plusieurs instances, qu'il faut remplacer par la couronne telle qu'elle a été définie au paragraphe précédent.

5.2.3.1) Détection des erreurs

Sur les rectangles propres au bloc père, toutes les vérification énumérées lors de la vérification des erreurs du niveau zéro sont effectuées.

Elles sont complétées par des traitements qui concernent l'ensemble des rectangles avec les optimisations suivantes :

- Les rectangles des couronnes ayant déja été vérifiés, aucune règle à un corps n'est appliquée.
- Aucune règle à deux corps n'est appliquée si les deux rectangles appartiennent à la même couronne.
- les boites d'aboutement issues des couronnes sont traitées d'une manière particulière : 1) on vérifie qu'elles ne se recouvrent pas. 2) Une boite d'aboutement réduite ne peut être pénétrée par un rectangle que si sa pénétration est incluse dans une transparence.

5.2.3.2) Génération de la couronne

La génération de la couronne s'applique à l'ensemble des rectangles (bloc père + couronnes des instances) moins les transparences du niveau hiérarchique inférieur. En effet une transparence est une indication donnée par le niveau *n* au niveau *n+1*. Elle n'a pas à être "remontée" au niveau supérieur *n+2*. Une transparence permet un passage à travers le bloc *n*, lequel passage n'est pas forçément étendu à la traversée du bloc *n+1*. Il n'y a pas de routage hiérarchique. Un routeur ne peut accéder au niveau hiérarchique inférieur pour y insérer ses éléments de routage.

Un bloc B dont certaines instances n'ont pas de couronnes ne peut être traité directement par le VRD. Celui-ci va parcourir récursivement l'arbre des instances afin de calculer toutes les couronnes dont il a besoin pour traiter le bloc B.

5.2.4) Jusqu'où aller ?

Il apparait d'une manière intuitive que la couronne d'une porte élémentaire risque d'être peu différente de la porte elle-même. Les dimensions d'une porte élémentaire sont en effet de même ordre que l'épaisseur de la couronne. Jusqu'où le partitionnement hiérarchique doit-il aller ? Il parait important de choisir un partitionnement qui dépende de la capacité mémoire et de la vitesse du processeur des stations de travail utilisées. Actuellement pour une SparcStation 2 dotée de 64 Mo de

RAM, on peut traiter à plat tout bloc d'une taille inférieure à cinquante mille transistors (un million de rectangles). Il faut en effet 52 octets pour décrire un rectangle : (prochain rectangle dans la liste, coordonnées X, Y, DX, DY, nom, flags, fenétrage, équipotentielle, champ utilisateur), Le vérificateur met à plat ce bloc avant de procéder à la vérification des règles : toutes les instances sont remplacées par une copie compléte de leur modèle.

Il n'est pas possible de coder en dur dans le programme une telle constante qui dépend de la machine. Aussi nous ferons encore confiance à l'utilisateur. C'est lui qui réglera la profondeur de la hiérarchie par le biais d'un fichier de directives :

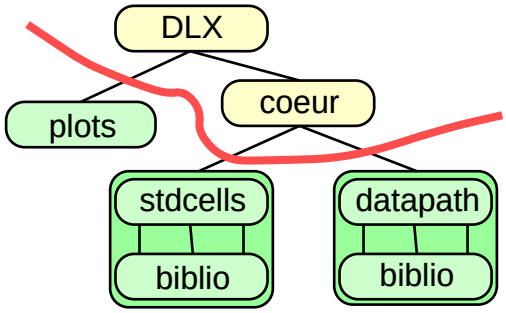

Figure F39. Description explicite de la profondeur hiérarchique souhaitée

Tous les modèles, dont le nom est dans la description explicite, sont mis à plat avant traitement. Ceci permet au concepteur de régler la granularité des traitements élémentaires en fonction de ses ressources disponibles : (Sparc-1 de 15 Mips et 16 Mo RAM, ou Sparc-10 de 75 Mips et 400 Mo RAM)

5.3) Optimisations

Maintenant que nous disposons d'un VRD hiérarchique, nous présentons dans ce paragraphe l'application des techniques de distribution précédement évoquées.

5.3.1) Accès exclusif

Le VRD distribué ne présente qu'une seule section délicate :

Un même modèle peut être présent à différents niveaux hiérarchique. Comme chaque niveau hiérarchique procède à des lancements de tâche sans aucune concertation avec les niveaux hiérarchique supérieurs, il est possible que deux tâches lancent simultanément un VRD sur le même modèle. Unix gère très bien deux lectures simultanées d'un même fichier, mais pas du tout deux écritures simultanées du même fichier. Il faut empécher la génération multiple de la même couronne.

On utilise des fonctions de verrouillage de fichiers. Avant tout accès au fichier de la couronne, que ce soit en lecture ou en écriture, un verrou d'exclusivité est posé. A la fin de l'opération d'écriture, le verrou est levé. La fin de l'opération de lecture ne signifie pas forcément la levée du verrou. La

lecture de la couronne peut très bien indiquer que cette couronne ne convient pas (Par exemple : distance de pénétration trop faible). Dans ce cas, il convient que l'application ne lêve pas le verrou tant que la nouvelle couronne n'est pas créée.

Il est délicat pour un programmeur non-spécialiste en programmation parallèle de savoir où est-ce qu'il doit insérer la pose / dépose du verrou dans son code. Afin de faciliter cette mise au point, nous utilisons des verrous conseillés (*advisory locking*) par opposition aux verrous impératifs (*mandatory locking*). Ceci signifie qu'il est aisé de les faire sauter. Un simple `rm` du fichier `.lock` suffit. D'autre part, le système de verrouillage utilisé permet, par un simple `ls`, de connaitre les `process` en attente du verrou.

5.3.2) Approche incrémentale

Nous avons vu que du point de vue global de l'application, il était préférable que les tâches échangent leur données par des fichiers sur disques. Ceci permet en plus une robustesse de l'application.

Ces fichiers intermédiaires sont-ils temporaires ? La couronne est une entité qui ne dépend que de la distance de pénétration , et de la technologie utilisée. Ces deux paramêtres ne changent pas fréquement.

Le fonctionnement d'un vérificateur de règles de dessin est similaire à celui de l'analyseur syntaxique d'un langage de programmation. Il fournit une liste d'erreurs dont l'utilisateur ne corrigera que les premières, puis relancera l'analyse car les erreurs suivantes peuvent être induites par la première. L'utilisateur sera pénalisé si pour détecter des erreurs provenant de rectangles appartenants au niveau hiérarchique en cours, le VRD doit d'abord vérifier tous les blocs de tous les niveaux hiérarchiques inférieurs.

Les couronnes sont donc des objets persistants (à l'instar des fichiers objets lors d'une compilation). L'utilisateur peut indiquer un paramêtre au lancement de la vérification qu'il désire que toutes les courones soient reconstruites pour être sûr de sa vérification.

Une couronne n'est considéré valide que si :

- l'usage des couronnes pré-existantes est autorisé par l'utilisateur,
- le fichier décrivant la couronne est antérieur à celui du bloc en cours de traitement,
- la technologie utilisée est la même,
- la distance de pénétration utilisée lors de la génération de la couronne est supérieure ou égale à celle définie pour le traitement en cours.

Si et seulement si toutes ces conditions sont satisfaites, alors le VRD ne lancera pas de tâches pour calculer la couronne, mais réutilisera celle présente sur disque.

5.3.3) Génération anticipée de la couronne

La notion de couronne pour un vérificateur de règles de dessin est-elle identique à celle de fichier objet pour un compilateur ? En d'autres termes : si le VRD détecte des erreurs dans le bloc, doit-il procéder à la génération de la couronne ? Si nous suivons la métaphore de l'atelier de génie logiciel qui nous a guidée jusqu'à présent, nous repondrions non.

Or, durant la conception d'un circuit répartie sur plusieurs concepteurs, la conception du plan de masse peut être démarrée sans que les divers éléments constitutifs du circuit ne soient finalisés. La mise à disposition d'une couronne par un groupe de travail, permet à l'autre groupe d'avancer, même si il est certain que cette couronne provient d'un bloc incorrect.

Il n'est donc pas indispensable que le circuit soit juste pour générer une couronne.

Quand est-ce que cette couronne doit elle être générée ? Nous sommes dans une optique distribuée, où la tâche de niveau hiérarchique *n+1* vient de lancer des tâches, et est en attente de la fabrication des couronnes par ces tâches-filles. Il est donc préférable que la génération de la couronne ait lieu le plus tôt possible, avant même le début des vérifications.

Remarque : dans le cas d'une application distribuée, ceci provoque une désynchronisation complète des tâches. Il est possible pour une tâche-mère de finir avant que toutes ses filles soient finies. Un effet secondaire de cela est que toutes les tâches sont lancées presque simultanément. Comme la machine parallèle virtuelle n'est pas capable de supporter toutes les tâches simultanément, un mécanisme de régulation de charge doit suspendre les requètes d'exécution jusqu'à ce que la charge redevienne acceptable.

5.4) Conclusion

Nous avons vu que le principal coût de la distribution est la définition de la méthode hiérarchique, et surtout de l'abstraction retenue pour «réduire» le bloc à ses interactions avec le niveau hiérarchique supérieur. Nous avons défini précisément les concepts et les mécanismes de génération des couronnes.

Cette technique de parallélisation qui s'appuie explicitement sur la hiérarchie structurelle des circuits ne se limite pas à la vérification de règles de dessin. Cette méthode peut être aisément généralisée à d'autres outils de vérification.

Des versions hiérarchiques et distribués d'un abstracteur fonctionnel et d'un analyseur de timing sont en cours de développement.

a

6) Architecture logicielle

Pour pouvoir expérimenter la méthode proposée, nous avons développé une maquette logicielle. Ce prototype regroupe plusieurs programmes : le vérificateur de règles de dessin à plat original, la version hiérarchique, et la version distribuée.

Nous nous intéressons dans ce chapitre à l'organisation logicielle des vérificateurs de règles de dessin. Une approche ascendante est suivie lors de cette présentation : présentation du VRD original, puis des améliorations successives.

6.1) Architecture originelle

Le vérificateur de règles de dessin de la chaine Alliance représente 45.000 lignes de code C réparties en 135 fichiers sources. Il a été réalisé par P. RENAUD dans le cadre de son stage de D.E.A.. Son architecture est la suivante :

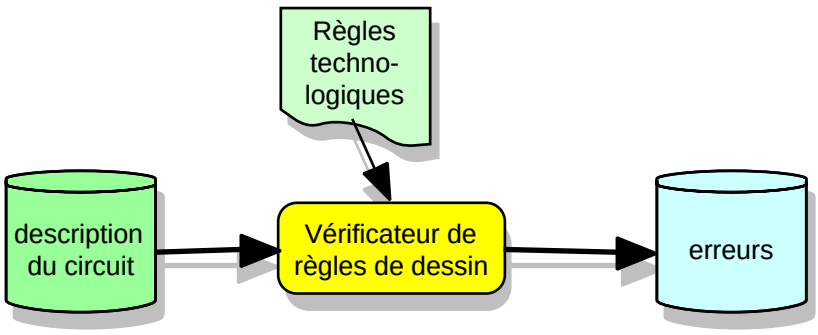

Figure F40. Architecture du VRD "à plat" originel

Il s'agit essentiellement d'un programme *batch* qui prend un fichier de données en entrée, le traite suivant une liste de paramêtres, et fournit un fichier résultat en sortie.

Une technique de fenêtrage permet d'exploiter la localité intrinsèque des règles de dessins [7].

6.1.1) Décomposition en étapes

Ce VRD effectue les étapes suivantes pour son traitement :

- Chargement du bloc racine et de tous les blocs utilisés.
- Mise à plat récursive du modèle racine.
- Chargement des règles de dessin technologiques
- Construction du fenêtrage
- Construction de la liste des équipotentielles
- Unification des rectangles
- Reconstruction du fenêtrage (Il est invalidé par l'unification)
- Mise à jour de la liste des équipotentielles
- Vérification de chaque règle sur chaque rectangle
- Sauvegarde des erreurs

L'unification des rectangles est effectuée avant les vérifications de manière à éliminer les masques redondants, et de corriger les fausses erreurs. (Voir Chapitre IV.4.2.4 dans [7]).

6.2) Architecture hiérarchique séquentielle

Notre VRD hiérarchique traite les blocs différemment selon qu'il s'agisse d'un bloc terminal ou d'un bloc hiérarchique.

6.2.1) Traitement à plat

Le traitement des blocs à plat est similaire à celui du VRD originel :

- Chargement du bloc racine et de tous les blocs utilisés.
- Mise à plat récursive du modèle racine.
- Chargement des règles de dessin technologiques
- **Génération des règles de dessin hiérarchiques**
- Construction du fenêtrage
- Construction de la liste des équipotentielles
- Unification des rectangles

- Reconstruction du fenêtrage (Il est invalidé par l'unification)
- **Construction et sauvegarde de la couronne de ce bloc**
- Mise à jour de la liste des équipotentielles
- **Vérification de chaque règle hiérarchique**
- Vérification de chaque règle technologique sur chaque rectangle
- Sauvegarde des erreurs

La génération des règles hiérarchiques correspond à la macro-génération de règles de dessin adaptées à la méthode hiérarchique ainsi que cela a été décrit plus haut (Voir Détection des erreurs, page 58.).

Afin de maximiser le parallélisme potentiel, la couronne est générée dès que possible.

6.2.2) Traitement hiérarchique

Le traitement d'un bloc hiérarchique ne peut se faire que lorsque toutes ses couronnes sont disponibles :

- Chargement du bloc racine et de toutes les couronnes utilisés.
- Chargement des règles de dessin technologiques
- **Génération des règles de dessin hiérarchiques**
- Construction du fenêtrage
- Construction de la liste des équipotentielles
- Unification des rectangles
- **Mise à plat des couronnes (Chaque nouveau rectangle garde toutefois la trace de la couronne dont il provient)**
- Reconstruction du fenêtrage (Il est invalidé par l'unification)
- **Construction et sauvegarde de la couronne de ce bloc**
- Mise à jour de la liste des équipotentielles
- **Vérification de chaque règle hiérarchique**
- Vérification de chaque règle technologique sur chaque rectangle
- Sauvegarde des erreurs

6.2.2.1) Optimisation pour formats de fichier monolithiques

Un surcout de cette architecture est que chaque modèle dont les couronnes ne sont pas disponibles est chargé deux fois : Une pour connaitre la liste des modèles dont il a besoin, l'autre pour le vérifier.

Lorsque le circuit est décrit dans un format de fichier monolithique — comme CIF (*Caltech Intermediate Form*) —, c'est toute la hiérarchie qui est rechargée. Afin de minimiser cela, la première étape partitionne le fichier original : Chaque fichier généré contient un seul modèle avec des références aux blocs instanciés.

6.2.3) Vue globale

L'architecture générale du VRD est celle décrite à la Section 4.6, "Traitement hiérarchique avec échec", page 38.

Le DRuC de P. Renaud est devenu un programme esclave (`epictete`), qui effectue la vérification du modèle après en avoir généré une couronne. Si toutes les couronnes nécessaires ne sont pas présentes, il ne fait que les signaler.

L'appel de l'esclave par le maître (`hsdruc`) se fait à l'aide de la simple fonction `system` du langage C. Le signalement des couronnes manquantes par l'esclave est obtenu en les écrivant dans un fichier texte dont le maître teste l'existence.

Le programme maître est un programme nouveau tandis que l'esclave n'est que le VRD «à plat» original légérement modifié pour sortir en erreur si toutes les couronnes ne sont pas présentes :

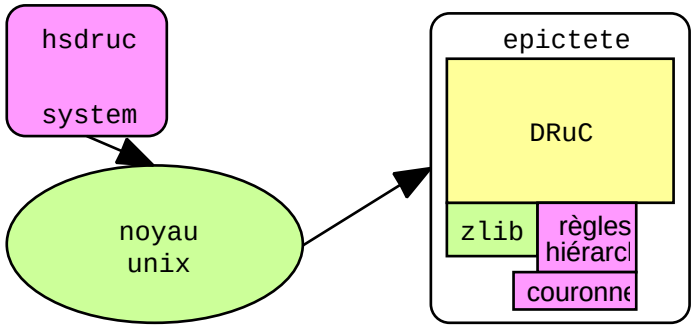

Figure F41. Vue globale du vérificateur séquentiel

6.2.3.1) Dépassement de capacité

Le vérificateur original «à plat» retourne brutalement au système, après avoir affiché un message d'erreur, lorsqu'il manque de mémoire.

Le vérificateur hiérarchique esclave a le même comportement.

Le programme maître détecte cette sortie brutale, par l'absence d'un fichier spécial, et s'interrompt aussi en affichant un message d'erreur précisant quel modèle a provoqué un dépassement de capacité, et recommandant de le diviser hiérarchiquement.

6.2.4) Coût de l'évolution

L'évolution de cette application en application hiérarchique est minime :

- Ajout d'un module pour macro-générer des règles hiérarchiques à partir des règles de dessin.
- Ajout d'un module pour le traitement des couronnes.
- Modification du coeur du VRD.
- Ajout d'un module pour le programme maitre (`hsdruc`).

Ceci représente 4 fichiers d'un total de 2000 lignes, soit 5 % du code.

6.3) Architecture distribuée

Nous avons dérivé notre vérificateur distribué du vérificateur hiérarchique séquentiel présenté au paragraphe précédent.

L'architecture générale retenue est celle décrite Section 4.8, “Parallélisation et distribution”, page 41.

Le programme maitre séquentiel (`hsdruc`) est remplacé par un nouveau programme maitre (`hpdruc`) qui distribue les tâches sur les processeurs.

Le programme esclave reste le même. Il est complété par des envois de messages ainsi qu'une protection d'accès exclusif pour la génération de la couronne (Voir Accès exclusif, page 62.).

6.3.1) Messages

Les messages s'échangent selon un modèle de communication en étoile — maître ↔ esclaves — sans communications entre les esclaves.

Les tâches esclaves communiquent avec la tâche maitre pour indiquer leur état :

- Manque des couronnes : Toutes les couronnes nécessaires à la vérification du modèle en cours ne sont pas disponibles. La tâche courante se termine en indiquant au maître de quelles couronnes, elle a besoin.
- Couronne prête : Durant la vérification du modèle, la couronne de ce modèle vient d'être

générée.

- Fini : La vérification du modèle en cours est finie

La tâche maître utilise le message "manque des couronnes" pour mémoriser le modèle à relancer, et démarre immédiatement en parallèle les traitements nécessaires à l'obtention des couronnes.

La tâche maître utilise le message "couronne prête" pour relancer les modèles qui étaient en attente de cette couronne. Les listes de dépendances des modèles, qui contiennent cette couronne, sont mises à jour.

6.3.2) Mécanisme de régulation de charge

La librairie de distribution utilisée — PVM — offre des fonctions pour configurer la machine virtuelle (ajout/retrait de processeurs), des fonctions pour gérer les tâches (lancer/arréter), des fonctions d'échanges de message entre tâches. Toutefois, l'algorithme de lancement des tâches est simpliste : Il ne tient pas compte de la charge des machines.

La charge d'une machines peut être représentée de la manière suivante :

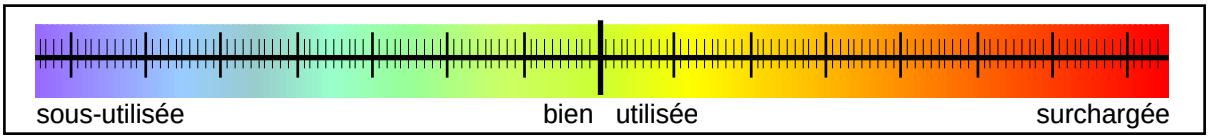

Figure F42. Axe de la charge d'une machine

Dès que la charge d'une machine dépasse le seuil "bien utilisée", elle devient surchargée. Lorsqu'elle est surchargée, elle ne peut plus répondre rapidement aux requètes visant à diminuer sa charge. Il faut donc **prévenir les surcharges** par un mécanisme de régulation de charge.

Les tâches de notre application consomment deux ressources : de la puissance CPU, et de la mémoire. Sans directives, le VRD ne peut à partir du nom d'un modèle déterminer la quantité de mémoire nécessaire pour le traiter. Comme nous visons une granularité forte (chaque tâche occupe une grosse partie de la mémoire disponible), et que notre application fait essentiellement du calcul (et non des entrées/sorties), nous nous limitons à une tâche par processeur.

Les réquètes d'exécution sont satisfaites s'il reste une machine libre, et sinon elles sont mises en attente par la tâche maître de l'application (`hpdruc`).

6.3.3) Vue globale

Le programme maitre (hpdruc) pilote les vérificateurs esclaves (epictete) de la manière suivante :

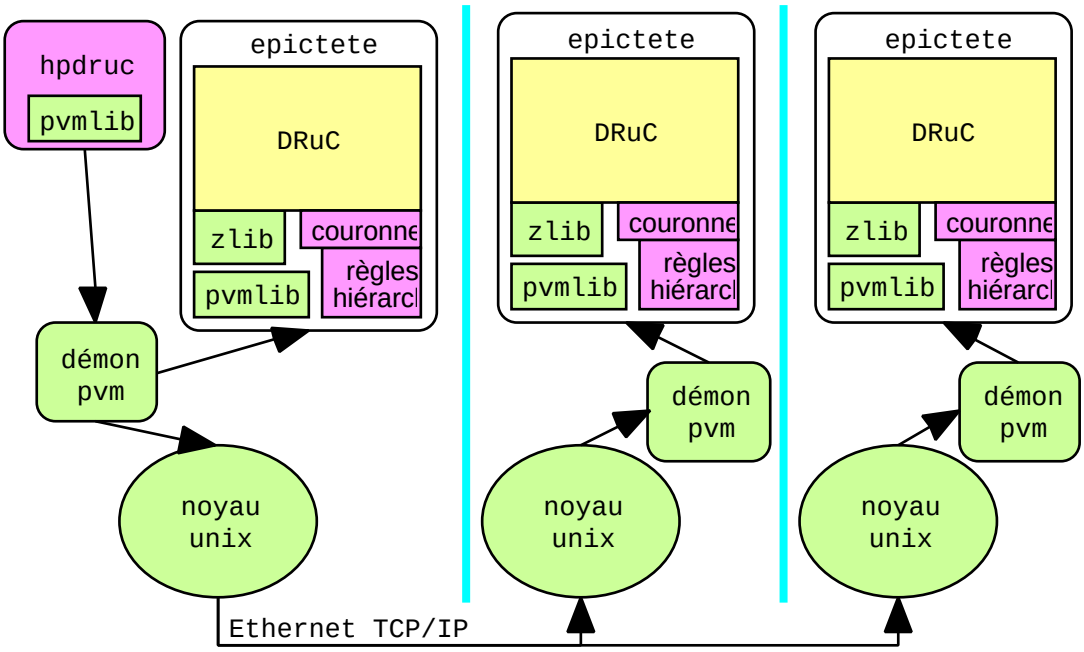

Figure F43. Vue globale du VRD distribué

PVM sert d'interface de la machine virtuelle parallèle. Le programmeur d'application ne connait pas la configuration exacte qui sera utilisée. C'est l'utilisateur qui la définit au démarrage de PVM.

6.3.3.1) Dépassement de capacité

Le vérificateur hiérarchique esclave distribué retourne brutalement au système, après avoir affiché un message d'erreur, lorsqu'il manque de mémoire.

La tâche maitre utilise un mécanisme de *time-out*. Si une tâche n'a pas envoyé de messages depuis plus de deux heures (c'est un paramêtre utilisateur), elle est considérée comme terminée brutalement — soit par manque de mémoire, — soit par un *kill*, ou une panne.

Si l'esclave a pu générer sa couronne, le maître continue normalement et attend la fin de l'application pour afficher un message d'erreur. Sinon le programme maître attend la fin des tâches en cours d'exécution pour stopper l'application.

La quasi-totalité des erreurs-mémoire intervient avant la génération de la couronne.

Dans l'implémentation actuelle, le *time-out* est vérifié à chaque reception de message. Si la tâche brutalement terminée était la dernière, le programme se bloque. (Il est possible de régler ce problème en utilisant un mécanisme de réveil périodique).

6.3.4) Ordonnancement des requètes

Tous les modèles traités ne nécessitent pas le même temps de traitement. Pour tirer profit au mieux du parallélisme, il faut que les tâches les plus longues commencent en premier. Nous avons besoin d'un ordonnancement de tâches.

Nous avons rajouté dans notre application la prise en compte d'un fichier de directives qui permet de controler le fonctionnement du mini-*scheduler* intégré. Ce fichier de directives est optionnel, mais reste obligatoire pour tirer le maximum d'une configuration donnée. Deux types d'usage peuvent être précisés :

- Le concepteur sait qu'il y a un gros bloc, et précise qu'il faut commencer par traiter celui-la.
- Le concepteur connait son circuit et souhaite éviter la phase inutile de chargement de modèles dont les couronnes ne sont pas disponibles. Il fournit alors un fichier de directives détaillé tel qu'il est décrit dans la Section 4.12, page 46.

Comme il s'agit d'un fichier utilisateur, notre VRD effectue des vérifications syntaxiques et sémantiques sur le contenu de ce fichier. Une erreur dans ce fichier ne provoquera au pire qu'un ralentissement de la vérification :

- S'il manque un modèle, le mécanisme de traitement avec echec lancera le traitement de ce modèle.
- S'il y a un modèle en trop, (et qu'il s'agit d'un modèle valide), le VRD traitera ce modèle.
- S'il manque une dépendance, le mécanisme de traitement avec echec provoquera le traitement de cette dependance.
- S'il y a une dépendance en trop, une contrainte syntaxique permet d'éviter que le VRD ne se bloque : toute dépendance doit avoir été mentionnée auparavant comme modèle à traiter.

Des directives complémentaires permettent de préciser qu'un modèle doit être traité : — seul, — sur un certain type de machine, — sur une certaine machine.

6.3.5) Robustesse

Lorsque la tâche maître est interrompue, la permanence des résultats intermédiaires sous forme de fichiers, permet de ne perdre que l'étape en cours. Notre application offre une certaine robustesse vis-à-vis des pannes.

La librairie PVM support de la distribution n'est pas tolérante aux pannes. La librairie GatoStar offre cette tolérance aux pannes aux applications qui l'utilisent. Elle intègre cela avec le mécanisme de migration automatique.

6.3.6) Coût de l'évolution

L'évolution de notre application hiérarchique séquentielle en application distribuée a nécessité le travail suivant :

- Compréhension de la librairie PVM : 110 fichiers totalisant 35.000 lignes de C (avec une bonne documentation comprenant un tutoriel facilitant la prise en main).
- Modification du coeur du VRD pour les messages et verrouillages.
- Ajout d'un module pour le programme maitre en distribué (`hpdruc`).

Ceci représente 2 fichiers d'un total de 1500 lignes, soit 2 % du code.

6.4) Conclusion

Nous avons vu dans ce chapitre que l'évolution d'une application hiérarchique incrémentale en application distribuée a un coût minime.

Le coût principal est la version hiérarchique incrémentale de l'application.

a

7) Résultats expérimentaux

Nous présentons dans ce chapitre des mesures de performance de nos prototypes sur un jeu de circuits de tests. Nous commençons par définir le cadre de nos mesures. Puis, nous présentons les circuits qui constituent notre jeu d'essais. Enfin, les mesures obtenues sont présentées et commentées.

7.1) Cadre des mesures

Dans ce paragraphe, nous présentons la plate-forme matérielle sur laquelle nous avons effectué nos mesures.

Un réseau peut être constitué de stations de travail homogènes ou hétérogènes. L'hétérogénéité des matériels porte sur plusieurs critères : Les stations diffèrent selon leur processeur, leur capacité mémoire ou disque, leur contrôleur réseau, leur charge résiduelle, etc.

Notre problème étant d'évaluer l'efficacité du partionnement hiérarchique, nous avons retenu une architecture de stations homogènes selon tous les critères.

7.1.1) Machine de référence

Nous avons choisi comme station de travail de référence une SPARCstation 2 dotée de 64 Mo de mémoire centrale, de 424 Mo de disque, de l'interface Ethernet 10 Mb/s, et configurée avec 160 Mo d'espace adressable total (*swap*). Elle offre une puissance de calcul de 28 Mips (24 SPECint92). Il y a quatre ans, cette machine était la référence pour la CAO VLSI. Nous avons retenu cette station car notre laboratoire dispose d'un nombre important de ces machines identiquement configurées. Les résultats obtenus peuvent être extrapolés à la génération suivante (SPARCstation 10 d'environ 75 Mips).

Nous avons compilé nos protoypes avec gcc 2.6.3, car ce compilateur ANSI permet une meilleure qualité du logiciel, et offre des fonctions de statistiques sur la mémoire utilisée (`mstats` ou `mallinfo`).

Nous mesurerons la quantité de mémoire nécessaire pour effectuer la vérification ainsi que le temps écoulé. La quantité de mémoire nécessaire est le facteur limitant dans la plupart des cas.

Nous avons utilisé le système d'exploitation SunOS 4.1.1rb sur chaque machine avec les démons de base du système. Les mesures ont été effectuées sur des machines en état de marche normal. C'est-à-dire avec tous les démons nécessaires à un fonctionnement multi-utilisateurs.

7.1.2) De la mesure du temps

Le système Unix offre une commande standard pour la mesure du temps : `time`. La duplication d'un fichier d'une vingtaine de Méga-octets sur différents disques nous donne les resultats suivants :

	temps utilisateur	temps système	temps écoulé	% CPU
disque local	0,02 s	6,14 s	40,2 s	15 %
lecture NFS	0,02 s	8,71 s	36,2 s	24 %
écriture NFS	0,02 s	8,59 s	217 s	4 %

Tableau T2. Copies d'un fichier d'une vingtaine de Méga-octets

Nous remarquons que nos trois copies ont des temps CPU (temps utilisateur + temps système) presque équivalents (6,2s ; 8,7s ; 8,6s), tandis que les temps écoulés diffèrent fortement.

L'écriture de donnée par NFS est très lente (requêtes synchrones). Il convient de minimiser la taille des données écrites sur un disque réseau.

La lecture de données par NFS est semblable à la lecture de données sur un disque local. On peut donc tenter d'utiliser le partage de disques par NFS dans le cadre d'une application distribuée.

Les routines de mesures du temps CPU ne renvoient que le temps processeur, qui n'est pas une information pertinente sur la performance du programme. Nous utiliserons le temps écoulé (*elapsed time*) comme mesure de la performance de nos prototypes. Afin de garantir la validité des mesures, nous nous assurerons l'exclusivité des stations utilisées, du réseau Ethernet, ainsi que du serveur de fichiers NFS, à chaque phase de mesure.

7.1.3) Jeu de circuits de tests

Nous n'avons pas construit de circuit de test adapté au fonctionnement de notre vérificateur. Nous avons utilisé des circuits réels, qui ont **tous été envoyés en fabrication**. Nous avons retenu quatre circuits de taille croissante, dont deux très gros (875 000 et 650 000 transistors). Notre jeu de

circuits de tests est restreint mais réaliste. Nous ne voulions pas de circuits construits spécialement pour notre VRD, ni de circuits qui n'aient pas été validés par l'expérience.

7.1.3.1) amd

Le plus petit des circuits est l'amd2901 [10]. Sa complexité est d'environ 3.000 transistors (379.000 rectangles «à plat»). L'ensemble des 42 fichiers nécessaires à la description hiérarchique de la vue physique a une taille de 698 kilo-octets. Celle du fichier CIF pour le fondeur est de 920 kilo-octets.

Il est surtout destiné à faciliter la mise au point de nos prototypes logiciels. Sa taille est trop petite pour que ses résultats soient pris en compte.

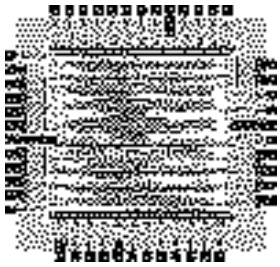

Figure F44. *Layout* du circuit *amd*

7.1.3.2) dlx

Le circuit moyen est le dlx-m [11] : la version micro-programmée du microprocesseur 32-bit dlx décrit dans Patterson & Hennessy [4]. Sa complexité est d'environ 30.000 transistors (1.767.000 rectangles «à plat»). L'ensemble des 193 fichiers nécessaires à la description hiérarchique de la vue physique a une taille de 4 méga-octets. Celle du fichier CIF pour le fondeur est de 5,1 méga-octets.

Les chiffres obtenus avec le dlx sont à prendre avec précautions, car il n'est pas représentatif de la classe des circuits que nous visons : le million de transistors.

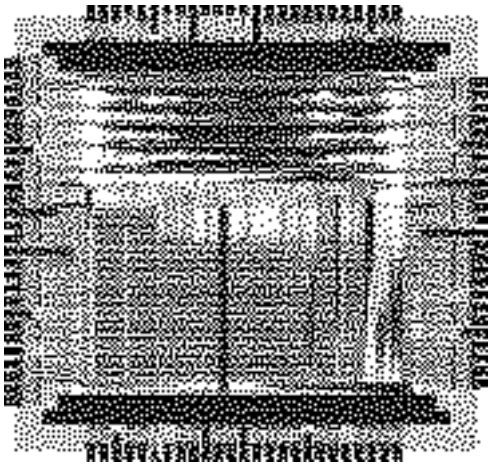

Figure F45. *Layout* du circuit *dlx*

7.1.3.3) stacs

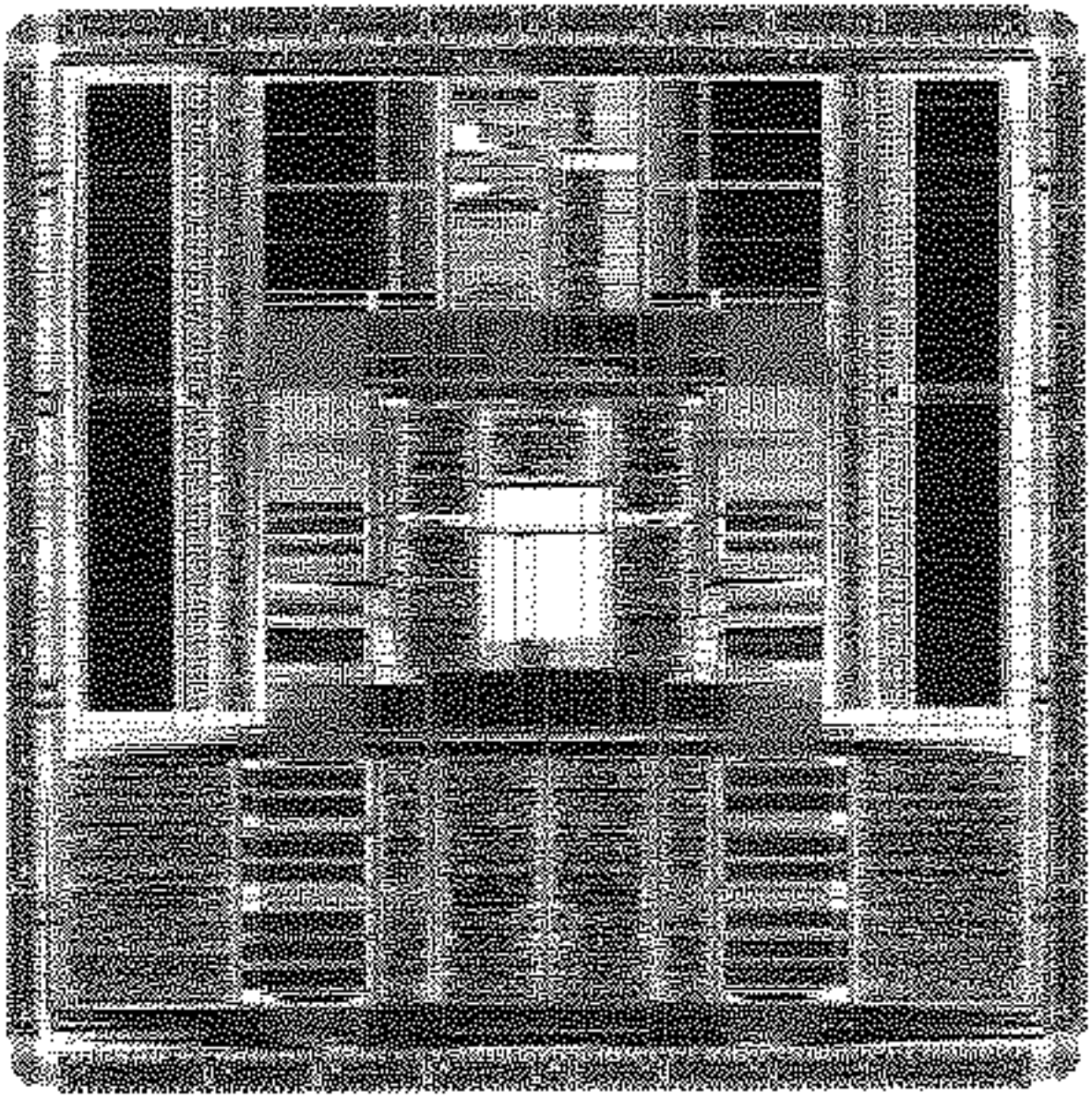

Figure F46. *Layout* du circuit *StaCS*

Le plus gros circuit est StaCS : un processeur VLIW [5]. Sa complexité est d'environ 875.000 transistors (16.520.000 rectangles «à plat»). L'ensemble des 309 fichiers nécessaires à la description hiérarchique de la vue physique a une taille de 17,5 méga-octets. Celle du fichier CIF pour le fondeur est de 21,3 méga-octets.

7.1.3.4) rapid16

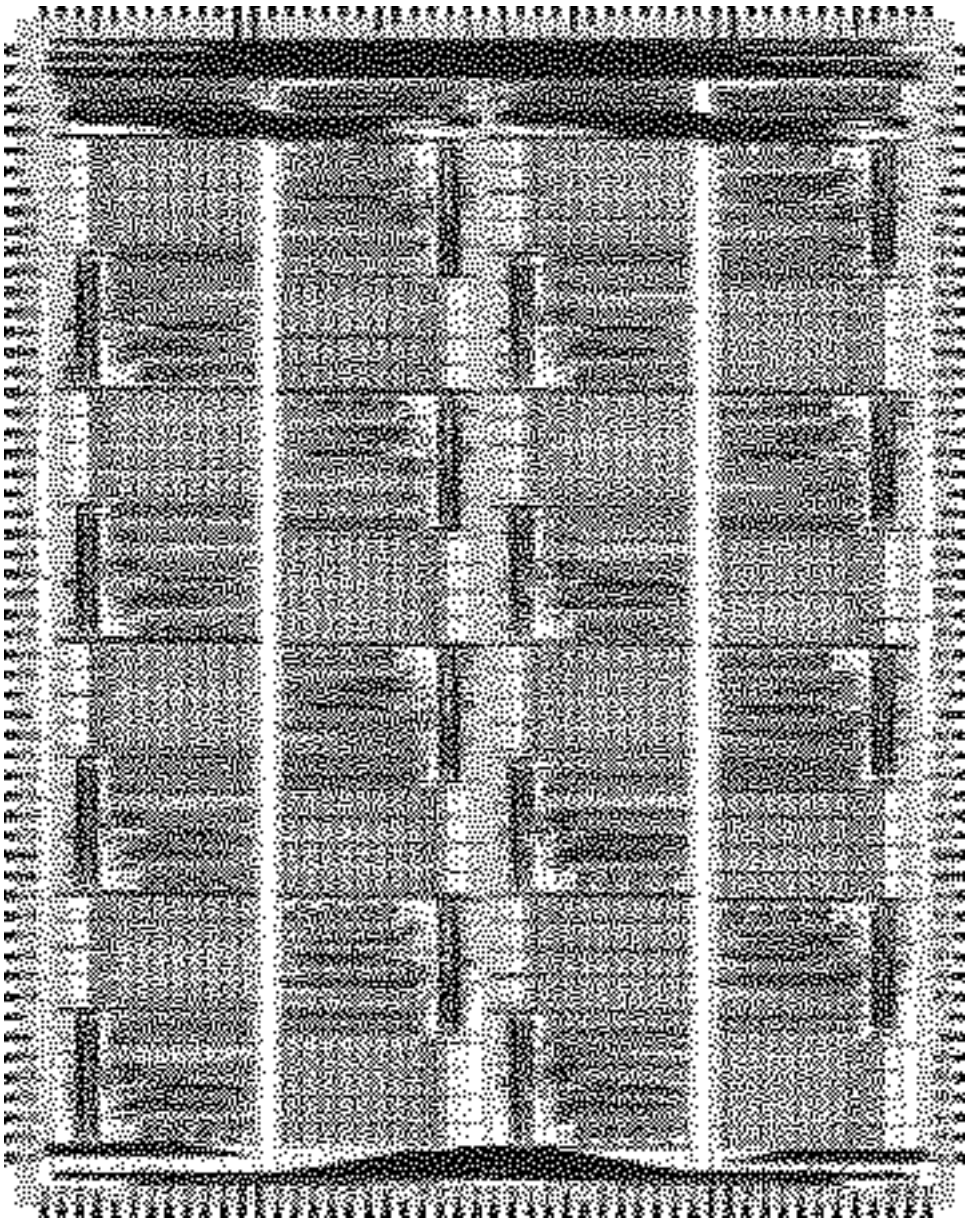

Figure F47. *Layout* du circuit *rapid16*

Nous avons aussi traité le circuit rapid16 qui contient 16 processeurs élémentaires identiques. La complexité de rapid16 à plat est de l'ordre de celle de StaCS, tandis que celle de son processeur élémentaire est de l'ordre de celle du circuit dlx, soit 650.000 transistors (18.550.000 rectangles «à plat»). L'ensemble des 115 fichiers nécessaires à la description hiérarchique de la vue physique a une taille de 4,4 méga-octets. Celle du fichier CIF pour le fondeur est de 4,8 méga-octets.

7.2) Mesures concernant le VRD original (à plat)

Nous avons utilisé comme base pour notre étude le vérificateur de règles de dessin de la version 3.0 de la chaine Alliance : *druc* [12].

Sur notre jeu d'essai, les résultats de ce VRD sont :

	amd	dlx	rapid16	stacs
temps nécessaire	7mn 8s	150mn	impossible *out of swap space* après 4mn 16s	impossible *out of swap space* après 10mn 56s
mémoire nécessaire	26 Mo	119 Mo	impossible *out of swap space* (swap : 160 Mo)	impossible *out of swap space* (swap : 160 Mo)
taille hiérarchique	3,5 Mo	15 Mo	12 Mo	55 Mo
taille à plat	20 Mo	91 Mo	*out of swap space*	*out of swap space*

Tableau T3. Résultats VRD à plat

Le «temps nécessaire» représente le temps nécessaire pour effectuer une vérification de règles de dessin sur le circuit préalablement mis à plat. La «mémoire nécessaire» représente l'espace mémoire occupé par le circuit à plat ainsi que les structures temporaires nécessaires au VRD. La «taille hiérarchique» représente l'espace mémoire occupé par le circuit chargé en conservant la structure hiérarchique. La «taille à plat» représente l'espace mémoire occupé par le circuit chargé à plat (sans instances ni modèles). La valeur obtenue par la formule : (mémoire nécessaire) - (taille hierarchique) - (taille à plat) donne une idée de l'espace mémoire occupé par les structures propres au VRD.

Nous avons essayé ces circuits sur la génération suivante de machines : une Sparc 10 de 73 SPECint92 (au lieu de 24), 410 Mo de RAM (au lieu de 64), 700 Mo de swap (au lieu de 160). :

	amd	dlx	rapid16	stacs
temps nécessaire	2mn 58s	16mn 3s	impossible *out of swap space* après 13mn 22s	impossible *out of swap space* après 19mn 20s

Tableau T4. Résultats VRD à plat sur SS10

Bien que cette machine soit dotée d'un espace de travail confortable : **700 Mo** de ***swap***, elle est incapable de traiter à plat les gros circuits.

Nous avons profité d'une période de vacances pour pousser le swap de cette machine à 1,75 Go, et relancer la vérification de StaCS à plat. Nous avons obtenu les mesures suivantes :

- En 55 Mo et 3 mn, la hiérarchie des fichiers est chargée.
- Il faut 991 Mo et 26 mn de plus pour mettre à plat le circuit.

- il faut encore 303 Mo pour préparer le circuit. Cette préparation prend 126 heures.
- Seuls 31 Mo sont consommés par la phase de vérification qui prend 152 heures.

Au total, il aura fallu 1380 Mo de mémoire et 12 jours de calcul. Pendant cette période, l'occupation du processeur était élevée, ce qui indique que la taille mémoire physique (416 Mo) était suffisante vis-à-vis de l'espace de swap (1,75 Go). Ces résultats confirment l'impossibilité d'un traitement à plat sur une station de milieu de gamme.

7.3) Mesures concernant le VRD hiérarchique séquentiel

A partir du VRD à plat dont les performances ont été présentées dans le paragraphe précédent, nous avons construit une version hiérarchique séquentielle de ce DRC. Hiérarchique signifie que nous avons modifié les traitements effectués par le VRD selon la méthode présenté au chapitre 5. Séquentiel signifie que chaque modèle est traité successivement sur la même machine (pas de distribution). Nous l'avons appelé «`hsdruc`».

7.3.1) Traitement à plat

La granularité du traitement hiérarchique à effectuer est un paramètre réglable par l'utilisateur. Si le seul niveau hiérarchique déclaré est le circuit lui-même, notre VRD (hsdruc) ne diffère pas du VRD à plat.

	amd	dlx	rapid16	stacs
temps nécessaire	7mn 18s	151mn	impossible *out of swap space* après 4mn 20s	impossible *out of swap space* après 10mn 47s

Tableau T5. Résultat VRD Hiérarchique et Séquentiel sur circuits à plat

Les règles de dessin hiérarchiques ne s'appliquent que sur un nombre très restreint de rectangles. Le surcoût de leur traitement est négligeable.

7.3.2) Granularité «grain fin»

Si l'utilisateur ne précise aucune granularité explicitement, hsdruc utilise la plus fine granularité,

c'est à dire que les cellules traitées à plat sont les cellules des bibliothèques de base :

	amd	dlx	rapid16	stacs
temps nécessaire	6mn 21s	45mn 13s	37mn 6s	172mn
mémoire nécessaire	6 Mo	21 Mo	35 Mo	43 Mo
nombre de modèles traités à plat	30	166	101	270
nombre de modèles traités en hiérarchique	12	46	33	43
nombre de tâches lancées	54	258	167	356

Tableau T6. Résultats VRD Hiérarchique et Séquentiel à granularité fine

Le «temps nécessaire» représente le temps nécessaire pour effectuer une vérification de règles de dessin sur le circuit complet, sans traitement préalable. La «mémoire nécessaire» représente l'espace mémoire occupé lors de la vérification du plus gros modèle. Le «nombre de modèles traités à plat» correspond au nombre de feuilles de la hiérarchie sur lesquelles une vérification «à plat» est effectuée. Le «nombre de modèles traités en hiérarchique» correspond aux autres noeuds de la hiérarchie vérifiés avec les couronnes de leurs instances. Le «nombre de tâches lancées» correspond au nombre total de tâches lancées par la tâche maître du VRD hsdruc.

Sur le dlx, nous voyons que l'utilisation massive de la hiérarchie nous donne un gain de **70%** en vitesse et de **82%** en espace mémoire nécessaire.

Par ailleurs, l'approche hiérarchique rend possible la vérification des deux gros circuits.

La taille des tâches lancées se répartit de la manière suivante :

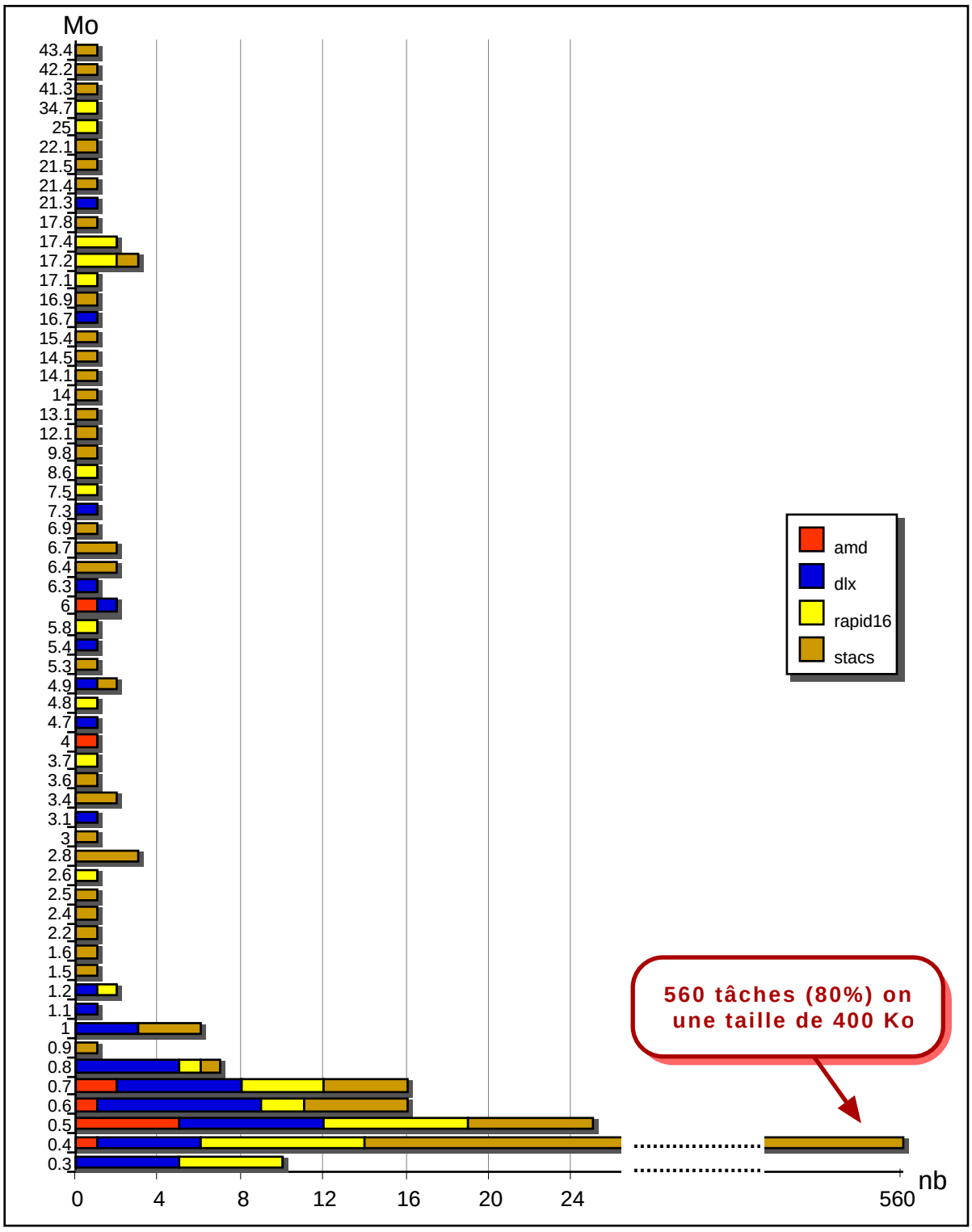

Figure F48. Taille des tâches pour un partitionnement grain fin

L'axe des ordonnées représente la mémoire nécessaire au VRD (en Mo) pour traiter chaque modèle. L'axe des abscisses représente le nombre de tâches. Nous pouvons remarquer que, dans cette configuration, 80% des tâches ont une taille de 400 Ko. Ceci correspond à la taille d'une cellule de base qui est une porte logique.

La vérification d'une telle cellule de 400 Ko, qui prend environ 5s, se divise en :

- 4% (0,2s) pour lancer l'éxécutable, et libérer la mémoire allouée
- 20% (1,0s) pour charger les rêgles technologiques
- 6% (0,3s) pour charger le modèle
- 18% (0,9s) pour unifier le modèle
- 2% (0,1s) pour créer la couronne
- 50% (2,6s) pour vérifier les règles de dessin

Le traitement de la plus grosse cellule, qui prend environ 960s et 43,5 Mo, se divise en :

- 0,1% (1,5s) pour lancer l'éxécutable, et libérer la mémoire allouée
- 0,1% (1s) pour charger les rêgles technologiques
- 20% (194s) pour charger le modèle et ses couronnes
- 23% (221s) pour unifier le modèle
- 2% (19s) pour créer la couronne
- 54% (523s) pour vérifier les règles de dessin

Les trois premières étapes et la cinquième correspondent à des phases de communication entre tâches : charger un modèle utilise le réseau NFS. Les quatrième et sixième étapes sont des phases de calcul.

Les deux premières étapes correspondent à une surcharge de partitionnement : recharger l'éxécutable ainsi que ses paramêtres globaux à chaque modèle. Le grand nombre de tâches de 400 Ko implique que le quart du temps est consommé par cette surcharge de partitionnement. Pour réduire ce surcoût, il faut augmenter la taille des tâches.

7.3.3) Granularité «gros grain»

Nous avons essayé un partitionnement minimisant le nombre de blocs traités à plat. Il s'agit donc de traiter à plat les blocs les plus gros possible. Nous avons dû utiliser le partitionnement en bloc de la RAM du processeur *StaCS*, tandis que nous avons dû descendre jusqu'au processeur élémentaire pour le circuit *rapid16* de part sa structure extrêmement répétitive.

	amd	dlx	rapid16	stacs
temps nécessaire	4mn 45s	28mn 27s	35mn 49s	266mn
mémoire nécessaire	8 Mo	54 Mo	42 Mo	80 Mo

Tableau T7. Résultats VRD Hiérarchique et Séquentiel à granularité forte

	amd	dlx	rapid16	stacs
nombre de modèles traités à plat	11	13	20	17
nombre de modèles traités en hiérarchique	1	2	7	2
nombre de tâches lancées	13	17	34	21

Tableau T7. Résultats VRD Hiérarchique et Séquentiel à granularité forte

Le «temps nécessaire» représente le temps nécessaire pour effectuer une vérification de règles de dessin sur le circuit complet, sans traitement préalable. La «mémoire nécessaire» représente l'espace mémoire occupé lors de la vérification du plus gros modèle. Le «nombre de modèles traités à plat» correspond au nombre de feuilles de la hiérarchie sur lesquelles une vérification «à plat» est effectuée. Le «nombre de modèles traités en hiérarchique» correspond aux autres noeuds de la hiérarchie vérifiés avec les couronnes de leurs instances. Le «nombre de tâches lancées» correspond au nombre total de tâches lancées par la tâche maître du VRD hsdruc.

Le nombre de modèles vérifiés à plat est grandement réduit : 61 au lieu de 567.

Nous pouvons remarquer qu'aussi bien en granularité «grain fin» qu'en granularité «gros grain», le temps de calcul pour le circuit *rapid16*, est du même ordre que pour le circuit *dlx*. La forte répétition de son «processeur élémentaire» est mise à profit par le mode hiérarchique.

Cette augmentation de la granularité nous fait gagner en temps pour tous les circuits sauf StaCS, en échange d'une consommation mémoire plus importante.

La taille des tâches lancées se répartit de la manière suivante :

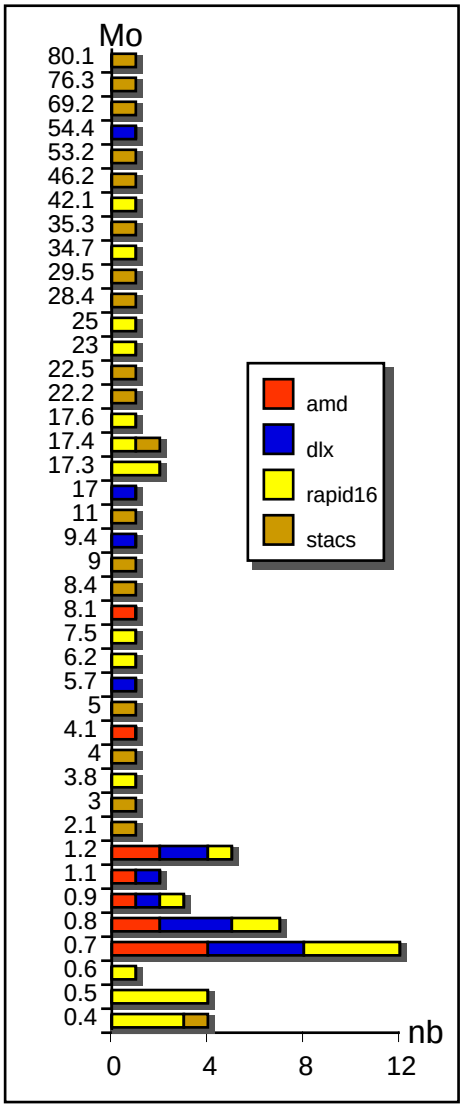

Figure F49. Taille des tâches pour un partitionnement gros grain

L'axe des ordonnées représente la mémoire nécessaire au VRD (en Mo) pour traiter chaque modèle. L'axe des abscisses de l'histogramme représente le nombre de tâches.

Le temps de traitement des tâches se répartit de la manière suivante :

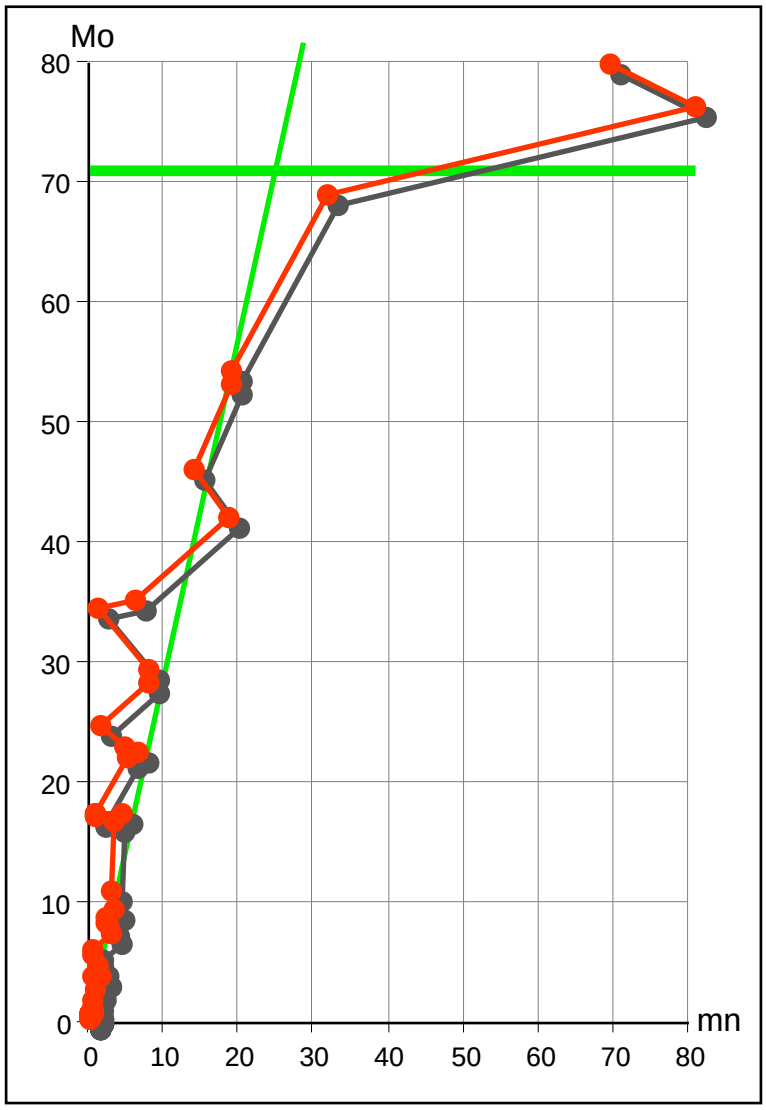

Figure F50. Temps de traitement des tâches pour un partitionnement gros grain

L'axe des ordonnées représente la mémoire nécessaire au VRD (en Mo) pour traiter chaque modèle. L'axe des abscisses de la courbe cartésienne représente le temps nécessaire par modèle en minutes.

De cette courbe, on peut remarquer que

- il n'y a pas de lien entre la mémoire utilisée et le temps de calcul.
- On peut remarquer un plafond vers 70 Mo à partir duquel le temps croît beaucoup. Ceci est dû à un net dépassement de la taille de mémoire physique : la machine est alors pénalisée par la mémoire virtuelle. Elle *swappe* trop.

L'augmentation du temps de traitement du circuit *StaCS* est principalement dûe à sa consommation mémoire : à chaque dépassement de la mémoire physique, la machine gaspille ses ressources à swapper. Les trois plus gros modèles traités dans *StaCS* dépassent chacun la capacité de la mémoire physique. Le temps de leur traitement représente **85%** du temps total. Il faut réduire la granu-

larité.

7.3.4) Granularité intermédiaire

Après plusieurs essais, nous avons obtenu les meilleurs résultats pour une granularité intermédiaire : La description explicite contient essentiellement des noeuds de premier niveau de la hiérarchie. Il s'agit de noeuds dont les instances sont des feuilles de la hiérarchie (cellules des bibliothèques de base).

Voici les résultats obtenus :

	amd	dlx	rapid16	stacs
temps nécessaire	4mn 45s	26mn 12s	24mn 32s	163mn 42s
mémoire nécessaire	8 Mo	21 Mo	35 Mo	53 Mo
nombre de modèles traités à plat	11	52	37	46
nombre de modèles traités en hiérarchique	1	4	6	5
nombre de tâches lancées	13	60	49	56

Tableau T8. Résultats VRD Hiérarchique et Séquentiel à granularité retenue

Le «temps nécessaire» représente le temps nécessaire pour effectuer une vérification de règles de dessin sur le circuit complet, sans traitement préalable. La «mémoire nécessaire» représente l'espace mémoire occupé lors de la vérification du plus gros modèle. Le «nombre de modèles traités à plat» correspond au nombre de feuilles de la hiérarchie sur lesquelles une vérification «à plat» est effectuée. Le «nombre de modèles traités en hiérarchique» correspond aux autres noeuds de la hiérarchie vérifiés avec les couronnes de leurs instances. Le «nombre de tâches lancées» correspond au nombre total de tâches lancées par la tâche maître du VRD hsdruc.

Le nombre total de modèles traités à plat est de 146.

Bien que la quantité de mémoire nécessaire soit équivalente entre la granularité «grain fin» et la granularité intermédiaire, le temps obtenu est nettement meilleur pour la granularité intermédiaire.

Selon la granularité, la taille et le temps moyen d'une tâche sont :

	granularité fine	granularité intermédiaire	granularité forte
taille moyenne	1,4 Mo	5,3 Mo	12 Mo
temps moyen	21s	1mn 19s	4mn 32s

Tableau T9. Taille et temps moyen du VRD d'un modèle

La «taille moyenne» représente l'espace mémoire occupé en moyenne lors de la vérification de tous les modèles. Le «temps moyen» représente le temps moyen nécessaire pour vérifier un modèle.

La taille des tâches lancées se répartit de la manière suivante :

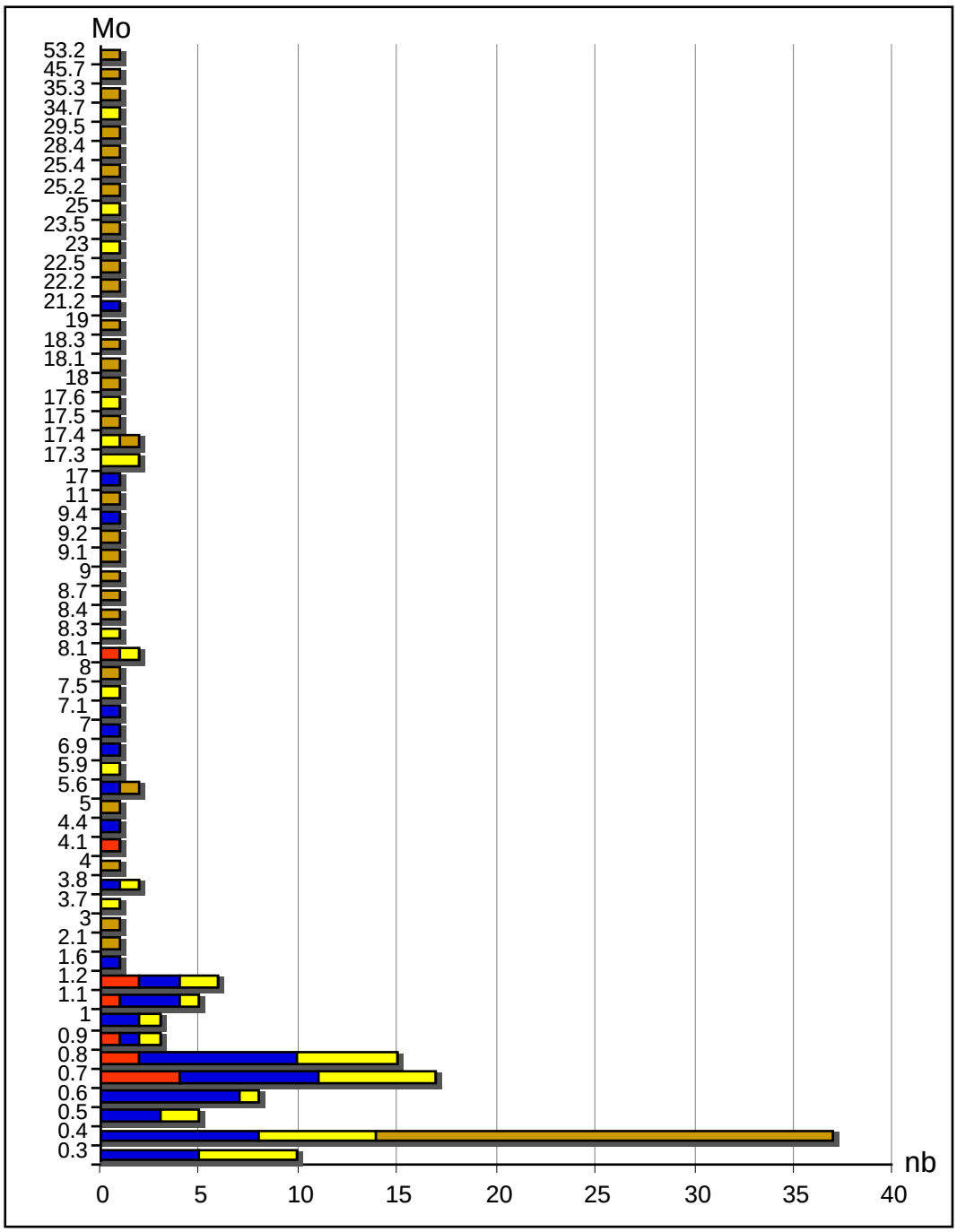

Figure F51. Taille des tâches pour le partitionnement intermédiaire

L'axe des ordonnées représente la mémoire nécessaire au VRD (en Mo) pour traiter chaque modèle. L'axe des abscisses de l'histogramme représente le nombre de tâches.

Le temps nécessaire par tâche se répartit de la manière suivante :

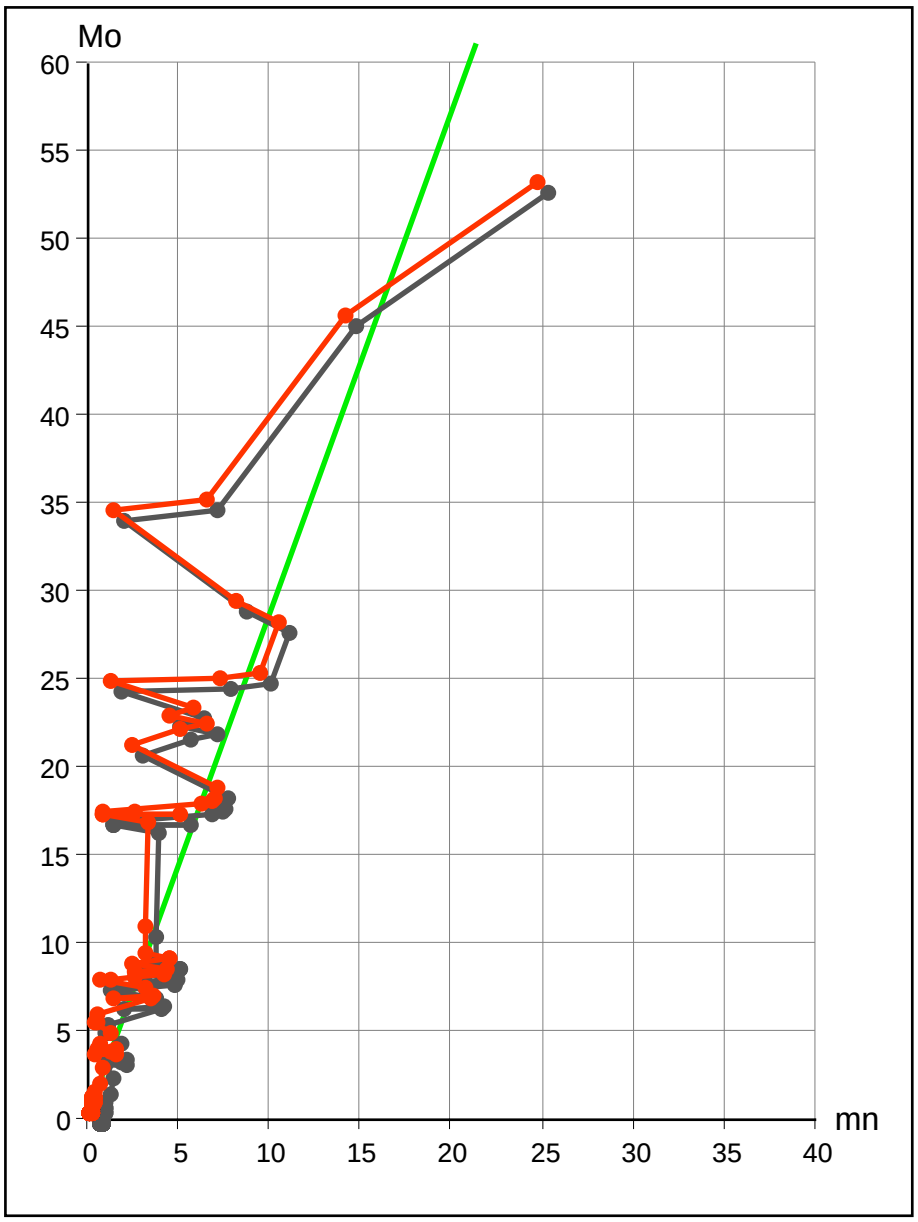

Figure F52. Temps de traitement des tâches pour le partitionnement intermédiaire

L'axe des ordonnées représente la mémoire nécessaire au VRD (en Mo) pour traiter chaque modèle. L'axe des abscisses de la courbe cartésienne représente le temps nécessaire par modèle en minutes.

Nous pouvons aisément qualifier la granularité intermédiaire. Les machines utilisées laissent environ 56 Mo de libre pour un programme utilisateur. Nous constatons que le maximum de performance est obtenu pour un traitement où la quantité maximale de mémoire disponible n'est jamais dépassée.

7.3.5) Gain incrémental

Les mesures présentées jusqu'à présent ont été obtenues dans le cadre d'un fonctionnement non-incrémental. Le VRD effectuait une vérification complète du circuit. Dans le cadre d'une vérification itérative — séquences vérification, correction —, le vérificateur n'aurait à traiter qu'un seul modèle grace à la permanence des résultats intermédiaires.

Le temps de traitement du modèle racine, lorsque toutes les couronnes filles sont disponibles sur disque, est le suivant :

	amd	dlx	rapid16	stacs
temps circuit total	4mn 45s	26mn 12s	24mn 32s	163mn 42s
mémoire circuit total	8 Mo	21 Mo	35 Mo	53 Mo
temps modèle racine	28s	3mn 14s	1mn 7s	14mn 12s
mémoire modèle racine	4,1 Mo	17 Mo	25 Mo	46 Mo
gain temps	**90 %**	**88 %**	**95 %**	**91%**

Tableau T10. Résultats VRD hiérarchique séquentiel avec couronnes disponibles

le «temps circuit total» et la «mémoire circuit total» sont des rappels des valeurs obtenues lors de la vérification du circuit total. Le «temps modèle racine» et la «mémoire modèle racine» sont les valeurs obtenues lors du traitement du modèle racine uniquement (lorsque toutes les couronnes des modèles instanciés sont disponibles). Le «gain temps» est le gain en temps obtenu par le fonctionnement incrémental. Les modèles supérieurs des circuits ayant la taille totale du circuit, il n'y a pas un gain aussi significatif sur la quantité de mémoire nécessaire pour traiter le circuit.

7.3.6) Mesures de la génération suivante de matériels

Nous avons essayé le VRD hiérarchique séquentiel «`hsdruc`» sur la génération suivante de machines : une Sparc 10 de 73 SPECint92 (au lieu de 24), 410 Mo de RAM (au lieu de 64), 700 Mo de swap (au lieu de 160). :

	amd	dlx	rapid16	stacs
temps nécessaire	1mn 37s	9mn 43s	9mn 16s	60mn 7s

Tableau T11. Résultats VRD hiérarchique séquentiel sur SS10

Les besoins en mémoire sont les mêmes sur une SS10 que sur une SS2 (ce ne serait pas le cas sur

une machine DEC-Alpha où chaque pointeur occupe 8 octets au lieu de 4).

Cette machine a réduit le temps de traitement de StaCS de 163mn. à 60mn., soit un rapport de 2,8 ce qui est comparable avec le rapport des puissance brutes (73 à 24 = 3). La lecture des 22 Mo de données qui passe par le même serveur NFS sur Ethernet ne semble pas être un facteur de ralentissement. Ceci est de bon augure pour une distribution de l'application :

7.4) Mesures concernant le VRD hiérarchique distribué

A partir du VRD hiérarchique séquentiel nous avons construit un VRD hiérarchique distribué.

Nous avons gardé le même programme esclave (`epictete`) en rajoutant des envois de message à au programme maître (`hpdruc`), ainsi que des verrouillages d'accès exclusif aux fichiers en écriture.

Le programme maître (`hpdruc`) a été réécrit en profitant de la librairie de distribution PVM. C'est ce programme qui gère le placement des tâches esclaves. Il joue aussi le rôle de régulateur de charge de manière à éviter l'écroulement des machines. C'est lui qui prend en compte les directives de placement et d'ordonnancement fournies par l'utilisateur.

7.4.1) VRD distribué à placement dynamique

Au démarrage, le programme maître (`hpdruc`) demande à la librairie de distribution (PVM) quelle est la configuration de la machine parallèle virtuelle sur laquelle il est lancé (nombre de processeurs). Il lance le premier esclave, sur le bloc qui lui a été indiqué, en respectant les contraintes de placement optionnelles.

Lorsqu'un esclave (`epictete`) signale qu'il a besoin que soient traités un ou plusieurs sous-blocs, le maître (`hpdruc`) mémorise la liste des calculs demandés, et lance les esclaves en fonction de la charge des processeurs, ainsi que des contraintes de placement et d'ordonnancement optionnelles. La charge des machines est un indicateur binaire. Les requêtes, qui n'ont pu être satisfaites suite à la charge de la machine parallèle virtuelle, sont mise en attente. A chaque échange de message, la pile des attentes est parcourue, et une nouvelle tâche est lancée si la machine s'est désaturée.

Lorsque toutes les requètes d'un esclave sont satisfaites, le maître relance alors l'esclave, qui dispose de toutes les couronnes dont il a besoin.

Grâce à la console de PVM, nous avons assemblé huit SS2 en une seule machine parallèle virtuelle dotée de huit processeurs ayant chacun 64 Mo de RAM et 160 Mo de *swap*. Ceci nous fait une ma-

chine dotée virtuellement de 192 Mips et de 512 Mo de RAM. Nous avons lancé nos vérifications sur cette machine. Puis nous avons fait varier le nombre de processeurs de cette machine virtuelle. Nous nous sommes assurés de l'exclusivité des machines vis-à-vis des autres utilisateurs durant toutes les éxécutions.

Pour chaque valeur indiquée, plusieurs mesures ont été relevées. La meilleure a été conservée.

Le partitionnement utilisé pour définir les blocs devant être traités à plat est le partitionnement explicite de granularité intermédiaire.

Les résultats sur une machine virtuelle allant de 1 à 8 processeurs sont les suivants :

	amd	dlx	rapid16	stacs
séquentiel	4mn 45s	26mn 12s	24mn 32s	163mn 42s
1	4mn 21s	25mn 6s	29mn 59s	166mn 15s
2	3mn 14s	14mn 42s	15mn 49s	91mn 40s
3	3mn 2s	11mn 49s	11mn 45s	65mn 57s
4	3mn 9s	9mn 43s	8mn 58s	54mn 8s
5	3mn 14s	8mn 38s	9mn 59s	46mn 42s
6	3mn 6s	8mn 29s	7mn 11s	42mn 40s
7	3mn 2s	8mn 8s	6mn 27s	38mn 51s
8	3mn 8s	7mn 55s	6mn 46s	36mn6s
plus grossse tâche	2mn 49s	4mn	4mn 46s	23mn

Tableau T12. Temps du VRD hiérarchique distribué

La ligne «séquentiel» est un rappel de la meilleure valeur obtenue par le VRD hiérarchique séquentiel (`hsdruc`). Les lignes «1» à «8» indiquent les résultats selon le nombre de processeurs de la machine parallèle virtuelle. Le «plus grosse tâche» est le plus grand temps mis pour vérifier un modèle.

La complexité du circuit *amd* est trop faible pour pouvoir tirer parti des multiples processeurs : Son plus gros bloc prend 87% du temps dès que deux processeurs sont disponibles.

Notre application distribuée est bien réservée à une certaine classe de circuits : ceux supérieurs au million de transistors.

7.4.2) VRD distribué «multitâche»

Notre VRD utilise un algorithme de calcul de charge simpliste : il place une seule tâche par processeur. Il parait intéressant d'exécuter plusieurs tâches par processeur, en profitant du multi-tâches d'unix, pour multiplexer les phases de calcul et d'entrée/sortie, afin de gagner du temps.

Nous avons construit une version dérivée de notre application, qui place jusqu'à deux tâches sur un même processeur : Elle remplit d'abord tous les processeurs avec une tâche, puis s'il reste des tâches à exécuter, elle rajoute à chaque processeur une deuxième tâche.

Les résultats sont les suivants :

	dlx 1 tâche	dlx 2 tâches
1	25mn 6s	22mn 53s
2	14mn 42s	12mn 49s
3	11mn 49s	9mn 57s
4	9mn 43s	8mn 8s
5	8mn 38s	7mn 47s
6	8mn 29s	
7	8mn 8s	
8	7mn 55s	

Tableau T13. VRD à multiplexage sur le *dlx*

	rapid16 1 tâche	rapid16 2 tâches
1	29mn 59s	22mn 30s
2	15mn 49s	12mn 25s
3	11mn 45s	9mn 27s
4	8mn 58s	7mn 34s
5	9mn 59s	6mn 56s
6	7mn 11s	
7	6mn 27s	
8	6mn 46s	

Tableau T14. VRD à multiplexage sur *rapid16*

	StaCS 1 tâche	StaCS 2 tâches
1	166mn 15s	207mn 33s
2	91mn 40s	96mn 26s
3	65mn 57s	63mn 15s
4	54mn 8s	76mn 19s

Tableau T15. VRD à multiplexage sur *StaCS*

	StaCS 1 tâche	StaCS 2 tâches
5	46mn 42s	46mn 19s
6	42mn 40s	
7	38mn 51s	
8	36mn6s	

Tableau T15. VRD à multiplexage sur *StaCS*

Dans un certain nombre de cas, le VRD à deux tâches est très nettement moins bon. Cinq des 56 tâches (10%) de *StaCS* ont une taille supérieure à la moitié de la mémoire physique disponible dans la granularité retenue. Lorsque deux de ces tâches se retrouvent sur le même processeur, la machine swappe et sa performance chute brutalement. Même dans la granularité la plus fine du circuit *StaCS*, trois tâches répondent à ce critère.

L'utilisation de l'option «multitâche» est donc à manier avec précautions.

7.4.3) Génération automatique d'un fichier de directives optimal

Nous avons doté notre VRD d'une fonction de statistiques. A la fin de chaque exécution, les ressources nécessaires sont présentées à l'utilisateur pour chacun des modèles vérifiés.

- La valeur de la quantité de mémoire nécessaire peut être réutilisée directement par un système de placement / migration.
- L'indication du temps écoulé n'est pas une valeur suffisante pour un système de placement. Pour prendre en compte une architecture hétérogène, il faut pondérer ce temps par la puissance brute (SPECint92) du processeur utilisé. Pour prendre en compte une architecture non-dédiée, il faut pondérer par le pourcentage de CPU pris par d'autres processus utilisateurs. Nous obtenons une valeur en ((temps) * (puissance disponible)) qui représente un nombre d'instructions nécessaires par modèle.

Nous avons complété cette fonction de statistiques par une fonction d'ordonnancement qui réalise l'algorithme LPT [18] avec contraintes. On affecte à chaque noeud de la hiérarchie la nombre d'instructions nécessaires cumulé pour son traitement et celui de ses sous-blocs. Les blocs sont triés par ordre de taille décroissante. Les dépendances de chaque modèle sont ensuite explicitées.

Le fichier de directives résultant pour le circuit *StaCS* est le suivant :

```
#TRUSTME
rf264_3         145728i 25028k
rf264_2         103224i 18012k
rf264_0         102264i 17840k
rf264_1         101208i 18684k
rf264_e         7920i 8008k rf264_3 rf264_2 rf264_0 rf264_1
scmul           359448i 52628k
dpoper          109368i 24912k
dpcmpadr        90600i 17860k
dpalu           89904i 23368k
dpexec          5712i 5648k dpoper dpcmpadr dpalu
stba07f48       70080i 9140k
stba07f08       68496i 8736k
stba07fa8       65496i 9228k
nmxb4_m_dp      528i 420k
bdffe2_dp       528i 448k
dffp_dp         504i 444k
dffi_dp         480i 444k
bmsim_dp        456i 444k
msim_s_dp       432i 424k
bnmxb4_m_dp     432i 444k
mspm_s_dp       408i 416k
bnmxb2_m_dp     408i 420k
nmxb2_dp        408i 420k
bnmxb3_m_dp     408i 444k
bl1_dp          408i 420k
ndrvp_dp        384i 420k
na2_dp          384i 416k
l1_dp           384i 420k
bbmsim_dp       360i 420k
o2_dp           360i 420k
ndrv_dp         360i 416k
l1n_dp          360i 420k
nmxb3_dp        336i 420k
bbl1_dp         336i 416k
n1_dp           336i 416k
dpmmu           74136i 16148k stba07f48 stba07f08 stba07fa8 nmxb4_m_dp bdffe2_dp dffp_dp
dffi_dp bmsim_dp msim_s_dp bnmxb4_m_dp mspm_s_dp bnmxb2_m_dp nmxb2_dp bnmxb3_m_dp
bl1_dp ndrvp_dp na2_dp l1_dp bbmsim_dp o2_dp ndrv_dp l1n_dp nmxb3_dp bbl1_dp n1_dp
rbit_col_6_3    22488i 4084k
decoder_6_3     18768i 5000k
gram            101352i 31392k rbit_col_6_3 decoder_6_3
dpseq           125136i 29152k
scage           121704i 28228k
scexec          97224i 22568k
scmux           77088i 22260k
scctr           71712i 17452k
dpman           46992i 11048k
scsys           37992i 8476k
scseq           32256i 9012k
scman           9480i 3000k
sctag           6936i 2116k
reset           576i 464k
stacs           221880i 45568k rf264_e scmul dpexec dpmmu gram dpseq scage scexec scmux
scctr dpman scsys scseq scman sctag reset
```

Figure F53. Fichier de directives pour *StaCS*

Le premier mot sur une ligne est un nom de modèle à traiter (s'il n'y a pas de directives de placement

/ migration optionnelles). Le chiffre (en vert) suffixé d'un «i» représente le nombre d'instruction nécessaires (temps écoulé fois la puissance disponible du processeur). Le chiffre (en rouge) suffixé d'un «k» représente la taille mémoire nécessaire. Les éventuels noms de modèle à la suite de la même ligne représente les modèles dont ce bloc a besoin.

La vérification avec utilisation de ce fichier comme fichier de directives d'ordonnancement donne les résultats suivants :

	dlx origine	dlx ordonné
1	25mn 6s	24mn 28s
2	14mn 42s	16mn 16s
3	11mn 49s	12mn 19s
4	9mn 43s	8mn 56s
5	8mn 38s	7mn 55s
6	8mn 29s	6mn 45s
7	8mn 8s	6mn 36s
8	7mn 55s	6mn 28s

Tableau T16. VRD à ordonnancement sur le *dlx*

	rapid16 origine	rapid16 ordonné
1	29mn 59s	23mn 25s
2	15mn 49s	13mn
3	11mn 45s	9mn 42s
4	8mn 58s	7mn 13s
5	9mn 59s	5mn 58s
6	7mn 11s	5mn 20s
7	6mn 27s	5mn
8	6mn 46s	4mn 58s

Tableau T17. VRD à ordonnancement sur *rapid16*

	StaCS origine	StaCS ordonné
1	166mn 15s	162mn 43s
2	91mn 40s	91mn 30s

Tableau T18. VRD à ordonnancement sur *StaCS*

	StaCS origine	StaCS ordonné
3	65mn 57s	66mn 28s
4	54mn 8s	53mn 35s
5	46mn 42s	44mn 54s
6	42mn 40s	41mn 20s
7	38mn 51s	36mn 23s
8	36mn6s	32mn 56s

Tableau T18. VRD à ordonnancement sur *StaCS*

Cette technique d'ordonnancement ammène un gain de performances d'environ 10%. Le fichier de directives du VRD sert à l'ordonnancement (et au placement / migration). La partie consacrée à l'ordonnancement n'est pas d'une grande complexité : Il s'agit principalement de la liste des modèles du circuit, avec sur la même ligne, le nom de ceux dont ils dépendent éventuellement. Les quantités d'instructions et de mémoire nécessaires sont optionnelles. Comme la rédaction d'un tel fichier est aisée pour le concepteur du circuit, nous retiendrons ces valeurs pour la suite de notre étude.

7.4.4) Taux de parallélisme

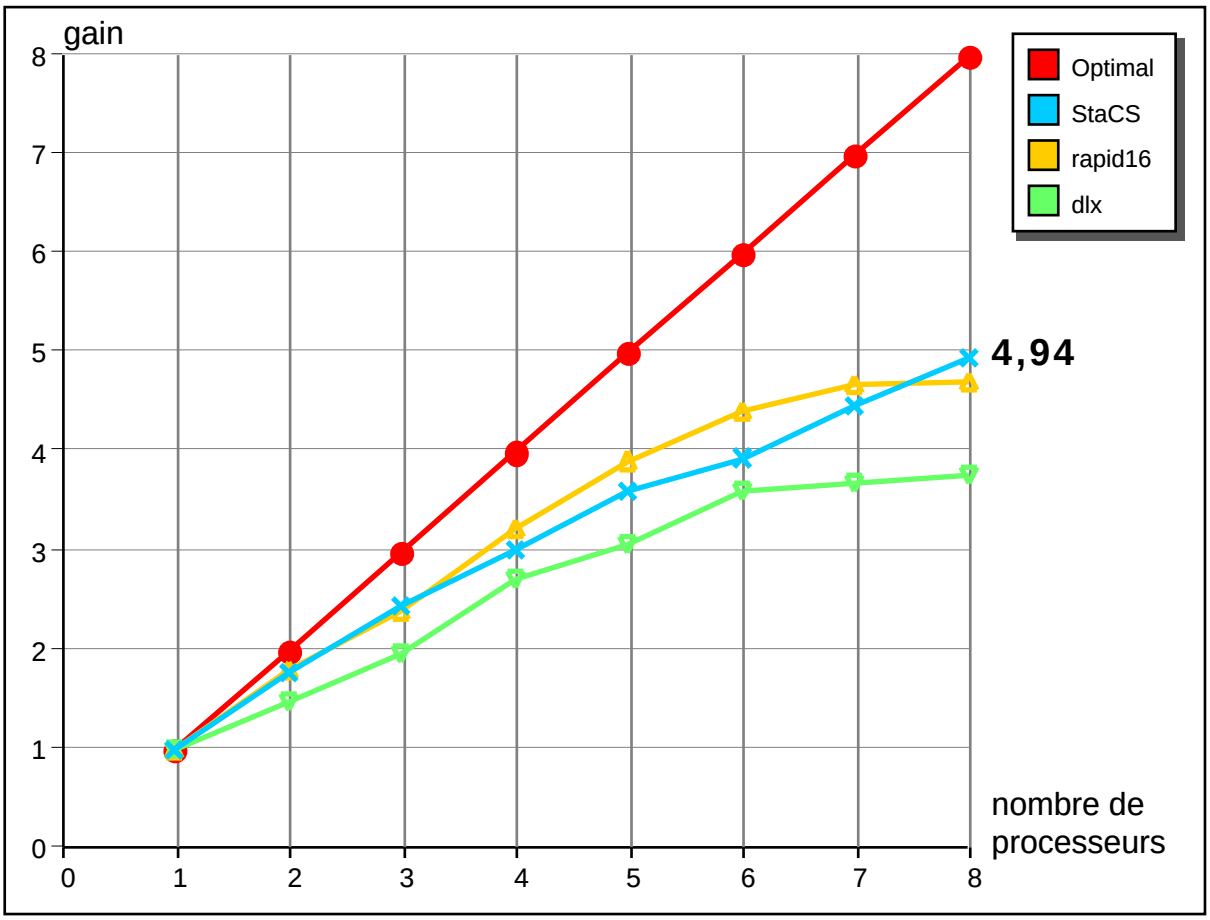

Figure F54. Gain obtenu

L'axe des abscisses représente le nombre de processeurs utilisés. L'axe des ordonnées représente le gain obtenu (*speed-up*) pour une telle configuration.

Le gain est limité par le taux d'utilisation des processeurs.

Voici, selon les circuits, les courbes d'utilisation des huit processeurs en fonction du temps :

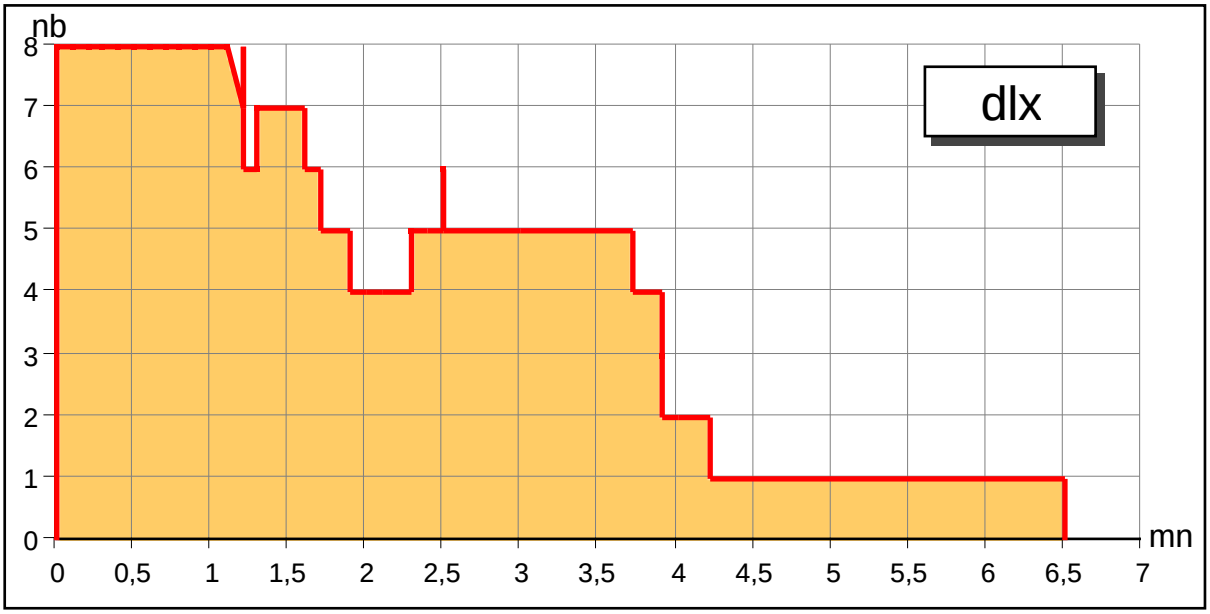

Figure F55. Utilisation des processeurs par le *dlx*

L'axe des abscisses est le temps en minutes. L'axe des ordonnées est le nombre de processeurs utilisés à cet instant.

Pour le circuit *rapid16*, l'usage des huit processeurs est le suivant :

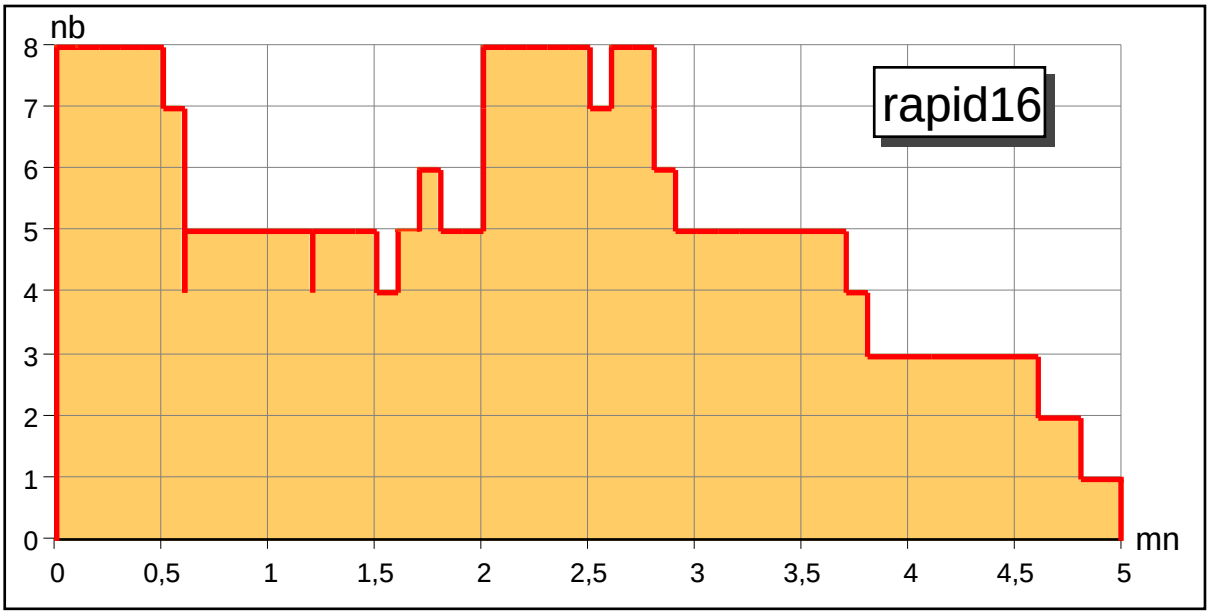

Figure F56. Utilisation des processeurs par *rapid16*

L'axe des abscisses est le temps en minutes. L'axe des ordonnées est le nombre de processeurs utilisés à cet instant.

Pour le circuit *StaCS*, l'usage des huit processeurs est le suivant :

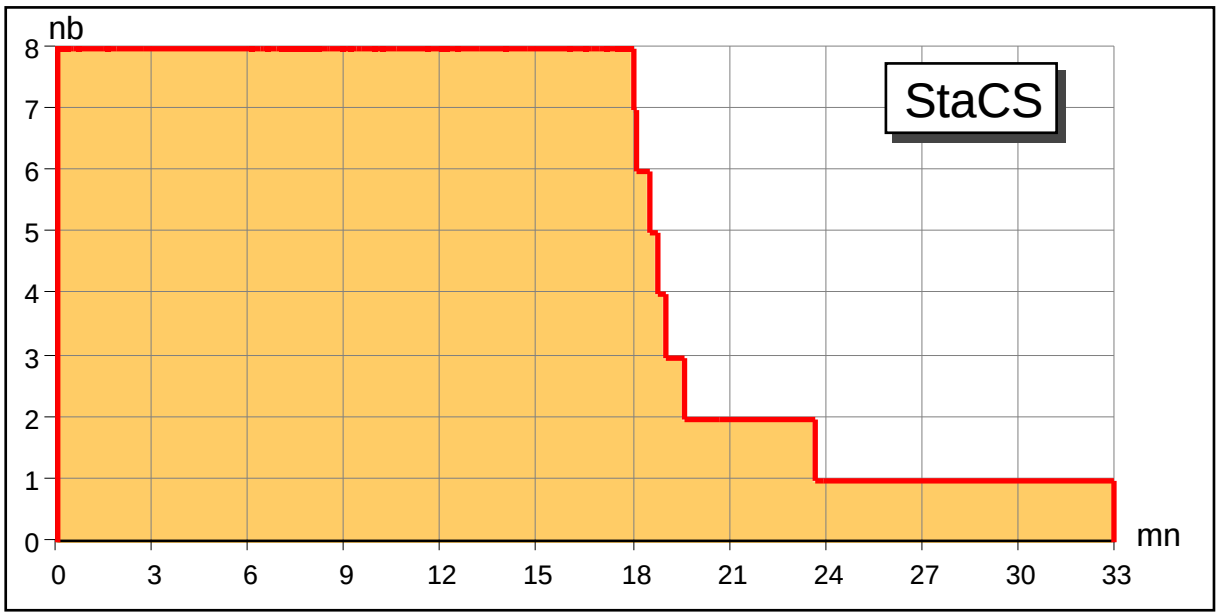

Figure F57. Utilisation des processeurs par le *StaCS*

L'axe des abscisses est le temps en minutes. L'axe des ordonnées est le nombre de processeurs utilisés à cet instant.

Sur un circuit de la complexité de StaCS, et avec la granularité retenue de 56 tâches, tous les processeurs sont maintenus occupés pendant 62% du temps.

Le temps de traitement élevé du dernier modèle (14mn 20s au total) est dû au routage très important dans ce bloc qui est le coeur du circuit et contient 25 gros blocs interconnectés aux 428 plots. 28 % (4mn) du temps de traitement de ce bloc est parallélisé grâce à l'anticipation de génération des couronnes.

On peut néanmoins conclure que l'utilisation de la hiérarchie pré-existante pour paralléliser la vérification est une technique efficace.

7.4.5) Ratio communication/calcul

Les communications efféctuées par le VRD sont :

- les communications directes de tâches à tâches
- le lancement des exécutables et leur configuration
- le chargement des données et l'écriture des résultats

Seul le programme maître recoit des messages. Chaque esclave emet de deux à quatre messages. Dans le cas du circuit StaCS, la reception et le traitement de l'ensemble des messages ne nécessite que 5s sur un temps total de 32mn 56s. Les communications au sens propre occupent 0,2 %. Le pro-

gramme maître ne consomme que très peu de ressources : il peut tourner sur le même processeur qu'un des esclaves.

Le lancement des tâches esclaves ne prend qu'un temps négligeable sur le réseau. C'est toujours le même exécutable qui est appelé 56 fois. Comme il est rappelé dès qu'il a fini, il est encore présent dans le buffer cache de la machine, et n'est chargé par le réseau qu'une fois par machine. La configuration de ces taches nécessite la lecture d'un fichier de règles technologiques. Ce temps reste constant : 1s par tâche = 56s au total.

L'ensemble des opérations sur les fichiers de modèles (eventuellement hiérarchiques) prend 766s. C'est ce temps qui représente l'essentiel des communications.

L'ensemble des vérifications de modèle prend 9681s de temps de calcul. Le ratio communications / calcul total est de **8 %**. Ce rapport ne dépend pas du nombre de processeurs :

	durée totale (rappel)	opérations sur fichiers	temps de calcul total	%
1	162mn 43s	739s	9711s	7,6 %
2	91mn 30s	741s	9472s	7,8 %
3	66mn 28s	743s	9732s	7,6 %
4	53mn 35s	743s	9778s	7,6 %
5	44mn 54s	812s	9985s	8,1 %
6	41mn 20s	834s	9697s	8,6 %
7	36mn 23s	739s	9616s	7,7 %
8	32mn 56s	766s	9681s	7,9 %

Tableau T19. Ratio communication / calcul total

La «durée totale» est le temps mis pour effectuer la vérification complète de *StaCS* sur n processeurs. Le temps «opérations sur fichiers» est la somme de tous les temps de lecture/écriture de fichiers par le VRD. Le «temps de calcul total» est la somme de tous les temps nécessaires pour traiter un modèle. Le «%» est le rapport (opérations sur fichiers) / (temps de calcul total).

Ce ratio est sensiblement constant. Même avec huit machines, il n'y a pas dégradation des performances du VRD due aux communications (surcharge du serveur NFS dédié).

7.4.6) Vérification d'un fichier monolithique (CIF)

Toutes les mesures présentées depuis le début du chapitre ont été effectuées sur la vue physique en utilisant le format de fichier `.ap` de la chaine Alliance. La particularité de ce format est que à chaque modèle de la hiérarchie est associé un fichier propre. Les données sont pré-partitionnées sur le disque. C'est le partage de fichiers par réseau NFS qui s'occupe de mettre à disposition des différen-

tes tâches les données dont elles ont besoin.

Il existe un autre format de fichier pour représenter la vue physique hiérarchique : CIF (*Caltech Intermediate Form*). Ce format contient sensiblement les mêmes informations que le format `.ap`, excepté que les coordonnées sont exprimées en µm, et que tous les modèles nécessaires sont réunis en un seul fichier unique.

Notre VRD, lorsqu'il travaille sur un fichier CIF, doit commencer par partitionner le circuit : c'est à dire créer un fichier pour chaque modèle présent dans la hiérarchie. Il charge la totalité de la hiérarchie en mémoire.

Toutes les mesures présentées depuis le début de ce chapitre ont aussi été effectuées pour un fichier CIF. Les performances sont nettement moins bonnes. Cela n'est pas critique, car le format CIF n'est pas utilisé durant la conception. Le fichier CIF est généré à la fin du processus de conception, et n'est pas censé générer de nouvelles erreurs. Il n'y a pas les incessants aller-retour (vérification-correction) qu'il y a en phase de conception.

Le traitement du fichier *StaCS* en vue réelle sur une machine virtuelle à huit processeurs prend 182mn 42s (contre 32mn 56s pour la vue symbolique). Il se déroule selon la courbe suivante :

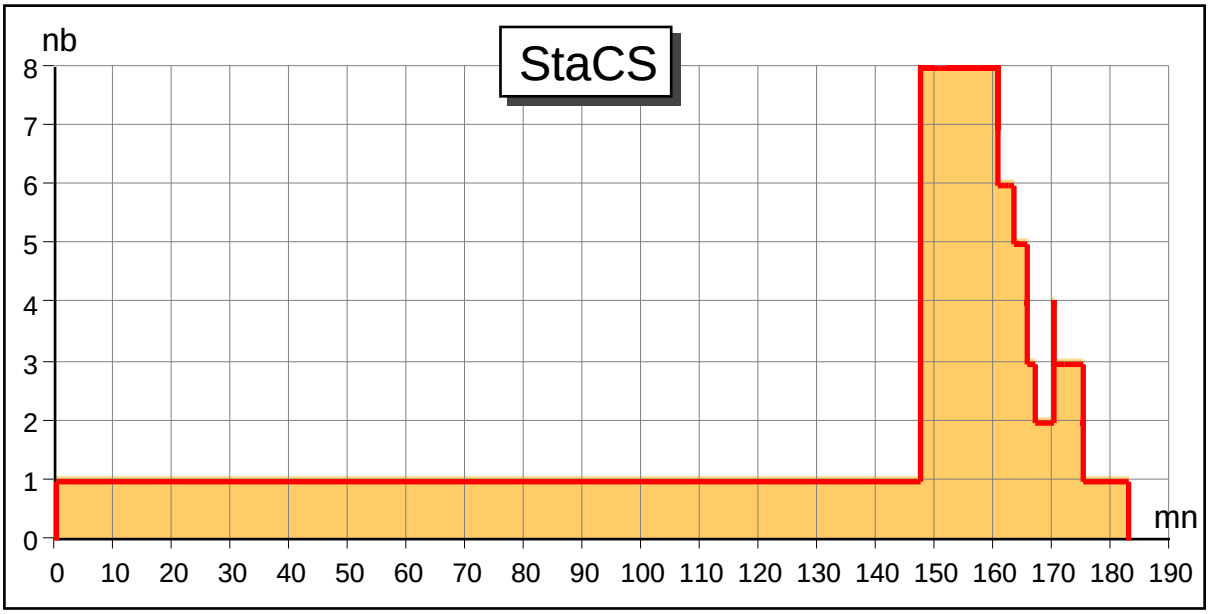

Figure F58. Utilisation des processeurs par *StaCS* au format CIF

L'axe des abscisses est le temps en minutes. L'axe des ordonnées est le nombre de processeurs utilisés à cet instant.

Nous remarquons que 80 % du temps est pris la première tâche (qui effectue le partitionnement). Le chargement du fichier prend 2h 22mn. L'écriture des 46 fichiers de modèles terminaux et des 5 fichiers de modèles non-terminaux prend 5mn.

Ce temps de traitement énorme est dû à un composant logiciel que nous avons réutilisé : le *parser* CIF d'Alliance. La fonction chargée de la lecture des fichiers CIF est extrêmement lente.

7.5) Conclusion

L'apport de la hiérarchie au VRD permet de traiter des circuits qui étaient trop gros pour pouvoir être traités «à plat». Le traitement incrémental facilite les cycles corrections / re-vérifications. L'apport de la distribution permet une diminution significative des temps de calcul. Il est possible d'atteindre un gain d'un facteur 5 avec l'utilisation de 8 machines. L'architecture proposée et validée sur des configurations allant jusqu'à huit processeurs peut être étendue au-dela. Les communications entre tâches ne représentent que 8% du calcul total.

La hiérarchie comme guide du partitionnement de la distribution est efficace si le concepteur définit explicitement les blocs à traiter à plat, ce qui n'est pas difficile pour un concepteur connaissant bien le circuit à vérifier.

a

8) Conclusion générale

Nous avons présenté dans cette thèse une méthode générale permettant la distribution d'outils de vérification de circuits VLSI sur un réseau de stations de travail. Cette méthode repose sur un partitionnement des données guidé par la hiérarchie existante.

La mise en oeuvre de cette méthode nécessite, pour chaque outil de vérification, de définir l'»abstraction» d'un bloc, qui dépend évidemment du type de vérification à effectuer. Nous avons démontré l'efficacité de cette approche dans le cas de la vérification de règles de dessin.

L'usage d'une librairie de compression en ligne (`zlib`) permet de maximiser la bande passante partagée du réseau.

Le stockage des vues abstraites d'une manière semi-permanente sur disque — à l'image des fichiers objets pour un compilateur C — offre une robustesse ainsi qu'un fonctionnement incrémental. Le cycle itératif typique du processus de vérification (correction, re-vérification) rend indispensable cette fonctionnalité.

La portabilité de l'application est assurée par l'emploi de la librairie *PVM* pour les fonctions d'échange de messages, et de gestion de la configuration de la machine parallèle virtuelle.

L'adaptation de l'application aux capacités des stations disponibles est laissée à la discrétion de l'utilisateur par un fichier de description explicite. Des directives d'ordonnancement optionnelles permettent de maximiser l'usage des processeurs. Elles peuvent être complétées par des consignes de placement / migration. L'exploitation d'un graphe de dépendance fourni par l'utilisateur, et l'anticipation de génération des abstractions, permettent de s'affranchir des phases séquentielles du traitement hiérarchique et de maximiser le parallélisme.

Pour privilégier la vitesse de traitement, il faut utiliser, comme limite supérieure de la granularité, la taille de la mémoire physique des stations employées.

Une accélération d'un facteur 5 a été obtenue sur un réseau de 8 processeurs pour un circuit de 875.000 transistors.

Afin de pouvoir utiliser des architectures hétérogènes ou partagées avec des utilisateurs interactifs, il faudrait compléter la librairie de distribution par un gestionnaire de ressources chargé des politiques de placement / migration. Toutefois, dans le cadre de notre étude, — machines homogènes dédiées —, un tel système n'apporterait aucun gain de performance.

Plusieurs outils de vérification — analyseur temporel, simulateurs de fautes — sont en cours de parallélisation au Laboratoire MASI. Ils devraient mettre en oeuvre les principes exposés dans cette thèse.

a

Annexe A : Utilisation de PVM

A.9) maître (hpdruc.c)

Pour s'enregistrer dans PVM : `pvm_mytid ()`

Ensuite, un `pvm_config ()` permet de récupérer le nombre et le nom des machines de la machine parallèle virtuelle.

`pvm_spawn ()` est utilisée pour lancer une tâche esclave sur une machine précise.

`pvm_recv ()` permet d'attendre qu'un message arrive (en provenance d'un esclave).

Un `pvm_bufinfo ()` permet de recevoir ce message dans un buffer. `pvm_upkstr ()` permet de récupérer la chaîne de caractères qui est dans ce message.

Finalement, un `pvm_exit ()` permet de sortir.

A.10) esclave (epictete.c)

Pour s'enregistrer dans PVM et récupérer l'adresse du maître : `pvm_parent ()`

`pvm_initsend ()` permet de se préparer à une communication. `pvm_pkstr ()` dépose une chaîne de caractères dans le buffer d'émission. `pvm_send ()` envoie cette chaîne à la tâche mêre.

Finalement, un `pvm_exit ()` permet de sortir.

a

Annexe B: Remarques sur le placement

Dans notre contexte de machines homogènes et dédiées à notre seule application, une politique de placement n'apporte aucun gain de performances. Toutefois, il existe plusieurs cas où une politique de placement serait souhaitable :

- Lorsque les machines ne sont pas homogènes, il faut pouvoir tenir compte de la puissance des processeurs pour placer les tâches les plus exigeantes sur les unités centrales les plus puissantes.
- Lorsque plusieurs vérifications s'exécutent de manière concurrente, elles entrent en compétition pour les ressources du système, provoquant ainsi une dégradation des performances.
- Quand un utilisateur interactif utilise sa station, l'application devrait lui libérer les ressources de sa station.

Nous avons fait plusieurs expérimentations avec gatos. Gatos est une librairie de distribution d'applications similaire à PVM. Elle diffère fortement de PVM, puisqu'elle se place au niveau du système et non de l'application.Grâce à un serveur de charge, gatos offre au concepteur d'applications plusieurs algorithmes de placement : selon la mémoire disponible, selon la charge des processeurs.

Ces expériences ont fait ressortir la **nécessité d'un mécanisme de régulation** de charge qui prévienne les surcharges des unités centrales. Ce mécanisme devrait être basé sur une gestion des ressources disponibles :

La puissance processeur disponible instantanée d'une unité centrale est sa puissance brute (SPECint92) minorée du pourcentage CPU pris par les tâches n'appartenant pas à l'application. Le temps *idle* du système est considéré comme pris par l'application distribuée puisqu'il peut recouvrir les entrée/sorties de l'application.

La mémoire disponible est la mémoire physique diminuée de celle prise par les tâches en cours d'exécution étrangères à l'application. Unix pratique une politique d'occupation de la mémoire physique : dès qu'un utilisateur est connecté, il occupe de la mémoire. Cette mémoire physique ne sera indiquée comme libre que si les tâches inactives de l'utilisateur sont swappées sur le disque. Pour cela, avant d'évaluer la quantité de mémoire physique disponible, il faut exécuter un programme qui réquisitionne le maximum de pages mémoire, provoquant ainsi l'évacuation sur disque des tâches inactives. Lorsque ce libérateur meurt, la mémoire physique est libérée des tâches inactives. Comme une application distribuée vise un gain de performance, il est essentiel de se limiter à la mémoire physique disponible.

Afin de pouvoir effectuer un placement, chaque tâche doit être capable de fournir une estimation (eventuellement basée sur un mécanisme de statistiques) des ressources nécessaires. Elle doit indiquer la quantité de mémoire dont elle a besoin, ainsi que l'energie consommée (puissance disponible * temps nécessaire).

Avec ces deux paramêtres fournis par chaque tâche, le gestionnaire de ressources avec placement peut prendre une décision : les tâches en attente de lancement sont triées dans une file unique par énergie puis taille décroissante. Les processeurs sont aussi triés par puissance disponible instantanée décroissante. Le gestionnaire examine successivement chaque processeur. Si la puissance disponible instantanée diffère fortement de la puissance intrinsèque de la machine, le gestionnaire passe au processeur suivant. Sinon, le gestionnaire prend la première tâche dont la taille est compatible avec la mémoire disponible diminuée des tailles requises par les tâches de l'application déja en cours sur ce processeur.

L'exécution concurrente de plusieurs tâches sur un même processeur ne peut qu'être bénéfique car elle permet de récupérer les temps d'attentes d'entrées/sorties d'une tâche pour les calculs d'une autre. Grâce à cet algorithme, le nombre de tâches par processeur est régulé par la quantité de mémoire disponible.

Un tel gestionnaire de ressources permettrait une cohabition des outils de vérification distribués avec un usage normal des stations de travail.

Nous avons commencé d'évaluer cette approche dynamique avec Gatos. Une librairie de programmation nommée *DAL* [92] a été développée pour rendre la programmation aussi aisée qu'avec PVM. Les premières mesures nous ont permis de constater que les performances du placement dynamique pour une application dans un environnement dédié ne sont que légèrement inférieures à celles obtenues avec PVM. Dans un environnement avec utilisateur interactif et/ou multi-applications, les performances devraient être largement supérieures.

a

Bibliographie

[1] N. H. E. Weste & K. Eshraghian, *Principles of CMOS VLSI design : a systems perspective*, 2nd edition, Addison-Wesley pub., ISBN 0-201-53376-6, Avril 1994

[2] P. Banerjee, *Parallel Algorithms for VLSI Computer-Aided Design*, Prentice Hall pub., ISBN 0-13-015835-6, 1994

[3] W. Stallings, *Computer organization and architecture, principles of structure and function*, 3rd edition, MacMillan Pub. New-York, ISBN 0-02-415495-4, 1993

[4] D. A. Patterson & J. L. Hennessy, *Computer architecture : a quantitative approach*. Morgan Kaufmann Inc. San Mateo California USA, 1990.

[5] L. Lucas, *Conception et réalisation d'un microprocesseur VLIW : méthodologie de conception et implantation VLSI*, Thèse de doctorat de l'université Paris 6, décembre 1995

[6] A. Ettien, *Etude des besoins en ressources d'applications distribuées et stratégie de réduction de charge : cas de FTAM, FTP, et SNMP*, Thèse de doctorat de l'université Paris 6, décembre 1994

[7] J. S. Medou Zengue Ze, *Vérification automatique des règles de dessin des circuits VLSI : règles formelles, approche hiérarchique*, Thèse de doctorat de l'université Paris 6, octobre 1991

[8] F. Pêcheux, *Principe, modèle et réalisation d'un serveur de cohérence dans un environnement de conception VLSI multi-utilisateurs et multi-machines*, Thèse de doctorat de l'université Paris 6, ?

[9] L. Ben Ammar. *Compilation de chemin de données optimisés pour circuits VLSI*. Thèse de doctorat de l'université Paris 6, mai 1994.

[10] Advanced Micro Devices. *The Am2900 family databook with related support circuits*. AMD Inc., California, P. 21-25, 1978

[11] P. Bazargan-Sabet, A. Greiner & M.-M. Louërat. *An educationnal VLSI design project : Implementation of the 32-bit DLX microprocessor*. Proc. advanced training course : "mixed design of VLSI circuits", p180-182, avril 1994

[12] A. Greiner & F. Pêcheux, *ALLIANCE : a complete set of CAD tools for teaching VLSI design*, 3rd Eurochip workshop on VLSI design training, pp230-237, 1993

[13] N. Hedenstierna & K. O. Jeppson, *The use of inverse layout trees for hierarchical design rule checking*, Proc. 26th DAC'89, Las Vegas, pp 508-512, Jun. 1989

[14] N. Hedenstierna & K. O. Jeppson, *A parallel hierarchical design rule checker*, ,1992

[15] A. Greiner & J. P. Leroy, *A symbolic layout view in Edif for process independent design*, Fourth european Edif forum, Daresbury, Cheschire, Oct. 1990

[16] K. P. Belkhale & P. Banerjee, *PACE: a parallel VLSI extractor on the Intel hypercube multiprocessor*, Proc. ICCAD, Santa-Clara, pp 326-329, Nov. 1988

[17] K. P. Belkhale & P. Banerjee, *PACE2: an improved parallel VLSI extractor with parameter extraction*, Proc. ICCAD, Santa-Clara, pp 526-529, Nov. 1989

[18] K. P. Belkhale & P. Banerjee, *A parallel algorithm for hierarchical circuit extraction*, Proc. ICCAD, Santa-Clara, pp 236-239, Nov. 1990

[19] K. P. Belkhale, R. J. Brouwer & P. Banerjee, *Task scheduling for exploiting parallelism and hierarchy in VLSI CAD Algorithms*, IEEE transactions on computer-aided design of integrated circuits and systems, pp 557-567, Mai 1993

[20] R. Widdowson & K. Ferguson, *Parallel polygon operations using loosely coupled workstations*, pp276-279, 1988

[21] R. Kane & S. Sahni, *A systolic design rule checker*, Proc. 21st DAC'84, pp243-250, 1984

[22] E. C. Carlson & R. A. Rutenbar, *Mask verification on the connection machine*, Proc. 25th DAC'88, pp134-140, 1988

[23] G. E. Bier, *An algorithm for design rule checking on a multiprocessor*, Proc. 22nd DAC'85, pp 299-303, 1985

[24] B. M. Riess, K. Doll & F. M. Johanes, *Partitionning very large circuits using analytical placement techniques*, 94?

[25] H. Sawada, *A hierarchical circuit extractor based on new cell overlap analysis*, IEEE pp 240-243, 1990

[26] G. Pelz, *An interpreter for general netlist design rule checking*, Proc. 29th DAC'92, pp 305-310, 1992

[27] G. S. Taylor & J. K. Ousterhout, *Magic's incremental design rule checker*, Proc. 21st DAC'84, pp 160-165, 1984

[28] P. Quinton, *Les architectures parallèles et leur mise en oeuvre matérielle*, Technologies matérielles futures pour l'ordinateur, Editions du CNRS, 1993

[29] F. Raimbault, P. Quinton, D. Lavenier, *Architectures systoliques et parallèlisme de données; l'environnement de programmation ReLaCS*, PI-727, IRISA, mai 1993

[30] C. Dezan, P. Quinton, *Verification of regular architectures using Alpha : a case study*, PI-823, IRISA, mai 1994

[31] P. Quinton, S. Rajopadhye, D. Wilde, *Deriving imperative code from functional programs*, PI-905, IRISA, jan 1995

[32] F. André, M. Le Fur, Y. Mahéo, J.-L. Pazat, *The Pandore compiler : overview and experimental results*, PI-869, IRISA, oct. 1994

[33] F. André, M. Le Fur, Y. Mahéo, J.-L. Pazat, *Parallelization of a wave propagation application using a data parallel compiler*, PI-868, IRISA, oct. 1994

[34] M. Tabusse, *Démarrez votre conception d'ASIC en langage comportemental*, Electronique n° 35, pp41-50, février 1994

[35] K. Usami & J. Iwamura, *Optimized design method for full-custom microprocessors*, IEEE 1989 custom integrated circuits conference, 1989

[36] J. Shandle, *Technology drivers*, Electronic Design, février 1995

[37] A. Greiner, L. Lucas, F. Wajsbürt & L. Winckel, *Design of a high complexity superscalar microprocessor with the portable IDPS ASIC library*, Proc. EDAC'93, pp9-13, février 1994.

[38] A. Greiner, L. Lucas, F. Wajsbürt, *Design of a high complexity superscalar microprocessor using the ALLIANCE CAD system*, 7th IEEE international ASIC conference and exhibit, pp 223-226, Rochester, New-York, septembre 1994

[39] ?, *Efficient and tunable data structure for VLSI CAD algorithms on rectilinear layouts*, ?, 1994 ?

[40] M.-M. Louërat, *Compte-rendu d'avancement projet dlxm nov. 1994*, Doc. interne, Lab. MASI, novembre 1994

[41] F. Scherber & E. Barke, *PALACE: A parallel and hierarchical layout analyzer and circuit extractor*, Proc. EDTC'96, pp. 357-361, mars 1996

---

[42] S. Ragsdale, *Parallel programming*, MacGraw-Hill pub., ISBN 0-07-051186-1, 1991

[43] W. R. Stevens, *Unix network programming*, Prentice Hall Pub., ISBN 0-13-949876-1, 1990

[44] D. E. Comer, D. L. Stevens, *Internetworking with TCP/IP, volume III, client-server programming and applications, BSD socket version*, Prentice Hall Pub., ISBN 0-13-020272-X, 1993

[45] J. Bloomer, *Power programming with RPC*, O'Reilly & Associates Inc., ISBN 0-937175-77-3, 1992

[46] J. S. Kowalik & L. Grandinetti, *Software for parallel computation*, Springer-Verlag pub., ISBN 0-387-56451-9, novembre 1992

[47] G. R. Andrews, *Concurrent programming, principles and practice*, Benjamin/Cummings pub., ISBN 0-8053-0086-4, 1991

[48] C. Lavault, *Evaluation des algorithmes distribués, analyse, complexité, méthodes*, Ed. Hermes, ISBN 2-86601-460-X, 1995

[49] A. Geist, A. Beguelin, J. Donguarra, W. Jiang, R. Manchek, V. Sunderam, *PVM : Parallel Virtual Machine : A users' guide and tutorial for networked parallel computing*, MIT Press, ISBN 0-262-57108-0, 1994

[50] W. Rosenberry, D. Kenney & G. Fisher, *Understanding DCE*, O'Reilly & Associates Inc., ISBN 1-56592-005-8, septembre 1993

[51] J. Shirley, *Guide to writing DCE Applications*, O'Reilly & Associates Inc., ISBN 1-56592-004-X, juin 1992

[52] W. Rosenberry & J. Teague, *Distributing Applications across DCE and Windows NT*, O'Reilly & Associates Inc., ISBN 1-56592-047-3, novembre 1993

[53] S. Mullender, *Distributed systems*, New-York, ACM Press, Addison-Wesley, 1989.

[54] Majidmehr, *Optimizing Unix for performance*, Prentice Hall ISBN: 0-13-111551-0, oct 1995

[55] , *Tuning Unix*, O'Reilly & Associates Inc.,

[56] , *High Performance Computing*, O'Reilly & Associates Inc.,

[57] B. Folliot, *Journées de recherche sur le placement dynamique et la répartition de charge : application aux systèmes répartis et parallèles*, Lab. MASI, mai 1995

[58] B. Folliot, *Méthodes et outils de partage de charge pour la conception et la mise en oeuvre d'applications parallèles dans les systèmes répartis hétérogènes*, Thèse de doctorat de l'université Paris 6, décembre 1992

[59] P. Sens, *Conception et mise en oeuvre d'une plate-forme logicielle de tolérance aux fautes pour le support d'applications réparties*, Thèse de doctorat de l'université Paris 6, décembre 1994

[60] B. Folliot, P. Sens & P.-G. Raverdy, *Plate-forme de répartition de charge et de tolérance aux fautes pour applications parallèles en environnement réparti*, Calculateurs parallèles, vol 7 n°4, pp 345-366, 1995

[61] J.-L. Gailly & M. Adler, ftp://quest.jpl.nasa.gov:/beta/zlib/zlib-0.71.tar.gz, mai 1995

[62] L. Dikken, F. van der Linden, J. Vesseur & P. Sloot, *DynamicPVM : Dynamic load balancing on parallel systems*, Shell Netherlands, Proc. high-performance computing and networking '94 pp 273-277, 1994

[63] L. Dikken, *DynamicPVM: task migration in PVM*, Report n° ICS/155.1, Shell Research, Amsterdam, novembre 1993

[64] W. Chuang, *PVM light weight process package*, Computation structures group memo 372, M.I.T., décembre 1994

[65] *Load Sharing Facility*, Platform computing corp., Toronto, Canada

[66] E. Violard & G.-R. Perrin, *PEI: a single unifying model to design parallel programs*, University of Franche-Comté, Besançon, 1993

[67] A. Platonoff, *Contribution à la distribution automatique des données pour machines massivement parallèles*, Thèse de doctorat de l'université Paris 6, mars 1995

[68] P. Cadinot, N. Dorta, B. Folliot & P. Sens, *Swap réparti*, Lab. MASI, février 1996

[69] R. Boutaba & B. Folliot, *Presentation of a multicriteria load balancing*, Workshop on dynamic object placement and load balancing, ECOOP 92, Netherlands, juin 1992

[70] E. S. H. Hou, H. Ren & N. Ansari, *Efficient multiprocessor scheduling based on genetic algorithms*, Chaper 12, 1992

[71] A. Bricker, M. Litzkow & M. Livny, *Condor technical summary v4.1b*, University of Wisconsin, janvier 1992

[72] J. Pruyne & M. Livny, *Providing ressource management services to parallel applications*, Proc. 2nd workshop on environments and tools for parallel scientific computing, pp 152-161, mai 1994

[73] M. Litzkow & M. Livny, *Experience with the Condor distributed batch system*, University of Wisconsin,

[74] , *Checkpoint & Migration of Unix processes in the Condor distributed processing system*,

[75] J. Pruyne & M. Livny, *Parallel processing on dynamic resources with CARMI*, University of Wisconsin, 1995

[76] IBM, *LoadLeveler*, RS/6000 SP products, http://www.ibm.com/, 1995

[77] T. P. Green, *DQS 3.1 installation manual*, Supercomputer computations research institute, Florida, juillet 1994

[78] S. S. Rao, *Prospero Resource Manager 1.0 user's guide*,

[79] S. Dowaji & C. Roucairol, *Priority of tasks and performances of branch-and-bound load balancing strategies*, Lab. PRISM, 1995

[80] H. Ilmberger, *Analysis and debugging of PVM Programs with VISIT*, Siemens AG, München, septembre 1994

[81] F. Zhang & E. H. D'Hollander, *Generating parallel loops from serial code for PVM*, University of Gent, Belgium, 1994

[82] M. Resh, A. Geiger & M. Sang, *Developing PVM-code on various hardware platforms: portability and performance*, Université de Stuttgart, 1995

[83] L. Brunie & L. Lefèvre, *DOSMOS : a distributed shared memory based on PVM*, Lab. LIP, ens-lyon, septembre 1994

[84] X.-F. Vigouroux, B. Tourancheau & F. Desprez, *Research using PVM in France*, Université de bordeaux, ens-lyon, septembre 1994

[85] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, S. Otto & J. Walpole, *PVM: Experiences, current status, and future direction*, ACM, 1993

[86] G. Stellner, *Consistent checkpoints of PVM applications*, Université de München, 1995

[87] M. M. Theimer & K. A. Lantz, *Finding idle machines in a workstation-based distributed system*, IEEE Transactions on software engineering, pp 1444-1458, novembre 1989

[88] A. Oram & S. Talbott, *Managing projects with make*, O'Reilly & Associates Inc., 1993

[89] B. Schmidt, 1994

[90] M. BenaÏchouche, V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor, & C. Roucairol, *BOB: une plate-forme unifiée de développement pour les algorithmes de type Branch-and-Bound*, Rapport de recherche n° 95/12 du Laboratoire Prism, UVSQ, http://www.prism.uvsq.fr/, mai 1995

[91] A. Alves, *Parallel computing - Windows style* (PVM for Windows), Byte, mai 1996

[92] L. Poujoulat, *DAL, Distributed Algorithms Library*, Lab. MASI, rapport de stage de DEA, juin 95