Université de Montréal
Faculté des arts et des sciences
Département d'informatique et de recherche opérationnelle

Optimisation lors de la synthèse de circuits à partir de langages de haut niveau

par

François-Raymond Boyer

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en Informatique

Avril, 2001

Université de Montréal

Faculté des arts et des sciences

Cette thèse intitulée :

Optimisation lors de la synthèse de circuits à partir de langages de haut niveau.

présentée par :

François-Raymond Boyer

a été évaluée par un jury composé des personnes suivantes :

Thèse acceptée le :

SOMMAIRE

Notre projet consiste à développer différentes techniques visant à maximiser la vitesse à laquelle un circuit peut traiter des données. Certaines optimisations, présentement faites entièrement à la main en utilisant des langages de bas niveau pour décrire les circuits, pourraient être faites automatiquement ou avec des outils dirigés par le concepteur. Ces techniques permettraient de réduire le temps nécessaire au développement d'un circuit rapide et/ou d'augmenter la performance d'un circuit.

Nous utilisons un algorithme d'ordonnancement utilisé en « software pipelining », pour remplacer la technique habituelle utilisée sur les circuits (la resynchronisation, ou « retiming »). Les circuits résultants ont des registres activés sur différentes phases d'horloge puisque l'ordonnancement peut donner des temps fractionnaires. Nous supportons une combinaison de registres activés sur les front et sur les niveaux de l'horloge, mais montrons qu'il est plus profitable d'utiliser seulement des registres activés sur les niveaux, du point de vue de la tolérance au « clock-skew ». Une méthode de preuve d'équivalence avec le circuit avant optimisation a été développée pour ces circuits qui ne sont pas supportés par les méthodes antérieures.

Nous présentons des bornes théoriques pour la tolérance maximale au « clock-skew » et le nombre de registres requis pour une certaine tolérance. Puis une méthode permettant d'atteindre la tolérance voulue, si elle est possible, sans diminuer les performances du circuit a été développée. Les méthodes antérieures de maximisation de la tolérance impliquaient une diminution de performance non négligeable.

D'autres points sont aussi traités, mais sans aller en profondeur, soit : la resynthèse, le « wave-pipelining » automatique et les horloges à période variable.

Mots clés : performance, resynchronisation, « software pipelining », resynthèse, « clock skew »

TABLE DES MATIÈRES

LISTE DES FIGURES ET TABLEAUX

Figures et tableaux des articles, qui ne se retrouvent pas ailleurs dans le texte.

Article [3] (chapitre 4) :

Article [4] (chapitre 5) :

Article [5] (chapitre 6) :

LISTE DES SYMBOLES

$d(v)$ Le délai maximum du sommet v (de l'élément de calcul v).

$d(p)$ Le délai maximum du chemin p (la somme des d des sommets).

$d_{min}(v)$ Le délai minimum du sommet v.

$d_{min}(p)$ Le délai minimum du chemin p (la somme des d_{min} des sommets).

$d_{min}(a \rightsquigarrow b)$ Le minimum des $d_{min}(p)$ pour tout chemin $a \overset{p}{\rightsquigarrow} b$.

δ La tolérance au « clock-skew ».

E L'ensemble des arcs d'un graphe (les interconnections dans un circuit).

e Un arc dans un graphe.

$v \overset{e}{\rightarrow} v'$ L'arc e entre les sommets v et v'.

G Un graphe (la représentation d'un circuit).

G_s Un graphe d'ordonnancement.

G_r Un graphe qui a subi une resynchronisation r.

maxPath(G, v) Les plus longs chemins dans le graphe G, à partir de la source v.

min c/t Le cycle ayant le plus faible rapport coût/temps.

O( ) L'ordre de ... (borne supérieure).

ω Le débit (l'inverse de la période).

P La période de l'ordonnancement (l'inverse du débit), sauf dans la section 3.2.1.

p Un chemin dans un graphe.

ϕ Une phase.

$v \overset{p}{\rightsquigarrow} v'$ Le chemin p entre les sommets v et v'.

r Une resynchronisation (un « retiming »).

s Une ordonnancement.

Θ( ) L'ordre exacte de ...

V L'ensemble des sommets d'un graphe (les éléments de calcul d'un circuit).

v Un sommet dans un graphe.

$w(e)$ Le poids de l'arc e, souvent un nombre de registres.

$w_s(v \overset{e}{\rightarrow} v')$ Distance (en temps) entre l'ordonnancement de v et de v'.

CHAPITRE 1

INTRODUCTION

L'espace disponible sur les puces électroniques croît actuellement a une très grande vitesse. En effet, les nouveaux procédés de fabrication permettent non seulement de fabriquer des transistors plus petits, mais aussi des plaquettes plus grandes. Cet espace est maintenant si grand qu'on ne sait plus vraiment qu'en faire pour maximiser la vitesse d'exécution. Malheureusement, le développement de grands circuits, avec des langages de bas niveau, est très coûteux en temps humain. Les langages de haut niveau permettent eux de développer à moindre coût humain mais, pour obtenir les performances voulues, il faut souvent modifier le circuit résultant de la synthèse (compilation).

Notre projet consiste à développer différentes techniques visant à maximiser la vitesse à laquelle un circuit peut traiter des données. Certaines optimisations, présentement faites entièrement à la main en utilisant des langages de bas niveau pour décrire les circuits, pourraient être faites automatiquement ou avec des outils dirigés par le concepteur. Ces techniques permettraient de réduire le temps nécessaire au développement d'un circuit rapide et/ou d'augmenter la performance d'un circuit. Ceci s'inscrit dans le cadre d'un plus grand projet qui vise à développer un compilateur ANSI-C optimisant faisant la séparation logiciel/matériel selon des critères de coût et de vitesse.

1.1 Objectifs

Généralement, dans les circuits contrôlés par une seule phase d'horloge, du temps est perdu en plusieurs endroits à attendre le prochain cycle, même sur le chemin critique. Ceci vient du fait que le temps de calcul entre les registres n'est pas partout le même,

mais que le temps entre les activations des registres est partout le même. La resynchronisation tente de diminuer ce problème mais se limite toujours à une seule phase d'horloge, ce qui l'empêche d'atteindre la période optimale. Par contre, un circuit multi-phase pourrait être développé pour que les horloges arrivent toujours exactement au bon moment. Il faudrait alors trouver les phases nécessaires et un circuit équivalent au circuit original mais contrôlé de manière très précise par ces différentes phases. Alors, le problème de la génération des horloges se poserait, ainsi que les variations sur les temps d'arrivés de celles-ci. Il faudrait aussi pouvoir montrer que ce circuit est bien équivalent au circuit original, selon une certaine définition de l'équivalence.

La logique pourrait aussi être optimisée. Présentement il existe des méthodes de resynthèse, qui réorganisent la logique combinatoire entre les registres, pour réduire le temps de calcul. Ces techniques ont le problème de ne pas passer par-dessus les registres pour optimiser le circuit de manière plus globale, et la logique optimisée n'est pas nécessairement sur le chemin critique.

1.2 Plan de la thèse

Le chapitre 2 contient les notions de graphes ainsi qu'une revue des ordres des algorithmes sur ceux-ci, ces notions étant nécessaires à la compréhension du reste de la thèse. Puis le chapitre 3 présente la revue la littérature sur les méthodes d'optimisation de circuits et de logiciels qui seront utilisées ou qui serviront de base de comparaison pour les nouvelles méthodes présentées dans cette thèse.

Dans le corps de la thèse, les points suivants sont étudiés :

- Une alternative à la resynchronisation, multi-phase, et trouvant les phases permettant un débit optimal (chapitre 4).
- La vérification formelle de l'équivalence entre le circuit multi-phase et le circuit original (chapitre 5).
- La minimisation des effets des variations sur les temps d'arrivés de l'horloge (chapitre 6).
- Une technique de resynthèse ciblant le chemin critique et passant par-dessus les registres, qui fonctionne sur nos circuits multi-phases (section 7.2).

- L’adaptation de ces techniques, si nécessaire, pour faire du « wave-pipelining » automatiquement (section 7.3).
- La généralisation des circuits multi-phase aux circuits avec horloge à période variable (section 7.4).

Les trois premiers points sont développés dans trois articles, qu’on retrouve dans les chapitres, tandis que les autres sont discutés, avec moins de profondeur, comme développements possibles (chapitre 7).

Une conclusion, plus globale que celles se trouvant dans chacun des articles, termine le tout (chapitre 8).

CHAPITRE 2

RAPPELS

Ce chapitre contient les notions de graphes nécessaires à la compréhension du reste de la thèse. On a besoin des plus longs chemins dans un graphe ainsi que du cycle ayant le plus faible rapport coût/temps. Différents algorithmes pour résoudre ces problèmes, ainsi que leurs ordres, sont présentés.

2.1 Graphes

Un *graphe* est une paire $\langle V, E \rangle$, où V est l'ensemble des sommets et E est une relation binaire sur V. Un graphe est dit *orienté* si les éléments de E sont des paires ordonnées, alors qu'il est *non orienté* si les paires sont non ordonnées. Un arc (orienté) $e \in E$ de $v \in V$ à $v' \in V$ est la paire ordonnée $\langle v, v' \rangle$ et est souvent noté $v \xrightarrow{e} v'$, alors qu'une arête (non orientée) entre v et v' est la paire non ordonnée $\{v, v'\}$. Un *chemin* dans un graphe est une séquence d'arcs qu'on peut suivre pour aller d'un sommet à un autre, c'est-à-dire que les arcs adjacents dans la séquence doivent avoir un sommet commun. Un chemin est *simple* s'il ne contient pas le même arc plus d'une fois.

Un *multigraphe* permet d'avoir plusieurs arcs (ou arêtes) entre les deux mêmes sommets. Formellement, pour les distinguer, il faut que chaque élément de E ait une étiquette différente. Les éléments de E, pour un graphe orienté, pourraient donc être de la forme $\langle e, v, v' \rangle$ (où e serait l'étiquette).

Les graphes utilisés dans ce texte sont tous des graphes orientés, mais les multigraphes orientés sont aussi permis par les algorithmes décrits.

2.1.1 Connexité

Un graphe non orienté est dit *connexe* si, pour chaque paire de sommets, il existe un chemin qui relie ces derniers. Un graphe orienté est dit *fortement connexe* si de tels chemins existent alors qu'il est dit *faiblement connexe* si le graphe devient connexe seulement lorsqu'on enlève l'orientation des arcs.

2.1.2 Poids

Souvent on ajoute au graphe une fonction qui donne une valeur (un *poids*) à chaque élément de V ou de E (poids sur les sommets ou sur les arcs). Nous étendons ces fonctions pour des chemins dans le graphe. Soit une fonction de poids w, pour tout chemin $v \overset{p}{\rightsquigarrow} v_k = v_0 \overset{e_0}{\to} v_1 \overset{e_1}{\to} \cdots \overset{e_{k-1}}{\to} v_k$, on définit :

$$w(p) = \sum_{i=0}^{k-1} w(v_i) \qquad \text{si } w : V \longrightarrow X \text{ (fonction sur les sommets)}$$

$$w(p) = \sum_{i=0}^{k-1} w(e_i) \qquad \text{si } w : E \longrightarrow X \text{ (fonction sur les arcs)}$$

X est un groupe quelconque (ensemble ayant un opérateur de type « addition ») mais souvent on utilise les entiers ou les réels. Il est à noter que, dans certains articles, le poids d'un chemin, si le poids est sur les sommets, inclut le sommet v_k, alors qu'ici il est exclu.

2.2 Algorithmes sur les graphes

2.2.1 Plus court et plus long chemin

Nous voulons trouver le chemin qui a le plus petit, ou le plus grand poids, entre deux sommets d'un graphe $G = \langle V, E, w \rangle$ (où w est la fonction de poids sur E). L'algorithme utilisé ici est celui de Bellman et Ford, qui est un des algorithmes permettant de trouver les plus courts/longs chemins à partir d'un sommet source vers tous les autres sommets, et le fait en $\mathrm{O}(|V|\,|E|)$. Cet algorithme fonctionne sur des graphes quelconques, contrairement à l'algorithme de Dijkstra qui impose certaines contraintes sur le signe des poids. L'algorithme retourne 'FAUX' si la longueur d'un chemin n'est pas bornée (est infinie positif pour les plus longs chemins, ou infinie négatif pour les plus courts chemins). Ce n'est donc pas un algorithme des plus courts ou plus longs chemins

simples, qui sont des problèmes NP-durs pour des graphes quelconques. Dans ce travail, nous n'avons pas besoin des plus longs chemins simples, mais l'analyse des délais en présence de boucles purement combinatoires peut en avoir besoin ([19]).

minPath, maxPath : Bellman_Ford($G\langle V, E, w\rangle, src$) {
  foreach ($src \xrightarrow{e} j \in E$)
    $s_j = w(e)$;
  repeat $|V|$ **times** {
    foreach ($i \xrightarrow{e} j \in E$)
      $s_j =$ min/max { s_j , $(s_i + w(e))$ } ;
    if (no s_j changed)
      return s ;
  }
  return FAUX ;
}

Les algorithmes de plus courts/longs chemins sont aussi beaucoup utilisés pour résoudre un certain type de système d'inéquations linéaires. Un système constitué uniquement d'inéquations de la forme $x_j \geq x_i + w_{ij}$ peut se représenter par un graphe ayant un sommet pour chaque variable x et un arc de poids w_{ij} , du sommet x_i au sommet x_j , pour chaque inéquation. On trouve une solution de ce système en trouvant les plus longs chemins dans le graphe, l'algorithme retournant 'FAUX' s'il n'y a aucune solution.

2.2.2 Cycle ayant le plus faible rapport coût/temps

Soit un graphe $G\langle V, E, c, t\rangle$, où c et t sont des fonctions de poids sur les arcs. On cherche le cycle $v \overset{p}{\rightsquigarrow} v$ (chemin revenant au même point) qui a le plus petit rapport $c(p)/t(p)$. Ce problème est connu sous les noms « minimum cycle ratio », « minimum cost-to-time ratio cycle » et « tramp steamer ». Un cas particulier, avec $t(e) = 1 \ \forall\, e \in E$, s'appelle « minimum mean cycle problem ». Ces deux problèmes ont une grande importance dans plusieurs domaines, dont l'analyse de performance. Les algorithmes les plus connus pour le problème général sont ceux de Burns [6], Lawler [26] et Howard [10]. L'ordre prouvé de la borne supérieure sur leur temps respectifs d'exécution, est de

$\mathrm{O}(|V|^2\,|E|)$ pour Burns, $\mathrm{O}(|V|\,|E|\,\lg(|V|\,t_{max}))$ pour Lawler et $\mathrm{O}(\min(|V|\,|E|\,\alpha, |V|^2\,|E|\,(w_{max} - w_{min})/\varepsilon))$ ([13]) pour Howard, où $|V|$ est le nombre de sommets, $|E|$ le nombre d'arcs, α le nombre de cycles simples dans *G*, et ε la précision de l'algorithme. Jusqu'à récemment, l'algorithme de Lawler semblait avoir la meilleure borne supérieure théorique des trois, car la seule borne prouvée pour l'algorithme de Howard était exponentielle. Mais en fait, selon une conjecture trouvée dans [11], le nombre d'itérations que fait l'algorithme de Howard est en moyenne $\mathrm{O}(\lg(|V|))$ et en pire cas $\mathrm{O}(|E|)$. Puisque le temps d'une itération de cet algorithme est dans $\Theta(|E|)$, ceci donne un temps moyen dans $\mathrm{O}(|E|\,\lg(|V|))$, si la conjecture est vrai, qui est de beaucoup meilleur aux autres, et un temps en pire cas $\mathrm{O}(|E|^2)$, qui est meilleur que celui de Burns puisque $|E| \leq |V|^2$ (dans un multigraphe ce n'est pas toujours vrai). Mais ces bornes ne sont pas prouvées. Une bonne comparaison des différents algorithmes pour résoudre ces problèmes, ainsi que des temps d'exécutions sur des exemples de différentes tailles, est présentée dans [12] et [13], et, en pratique, l'algorithme de Howard est vraiment le meilleur sur presque tous les exemples, et l'écart devient plus grand quand la taille augmente.

Plusieurs algorithmes sont basés sur la solvabilité des plus longs chemins, dans un graphe qui a comme paramètre le ratio *R* qu'on cherche ([23]). Le graphe utilisé est $G' = \langle V, E, w\rangle$ construit à partir de *G*, avec $w(e) = t(e) - c(e)\,/\,R$. On peut trouver les plus longs chemins dans G' si et seulement si R est plus petit ou égal au plus petit rapport c/t. L'algorithme de Lawler fait simplement une recherche dichotomique pour trouver le plus grand R pour lequel les plus longs chemins existent dans G'. Dans les cas qui nous intéressent, l'intervalle dans lequel se trouve la solution est $[\,1/\,|V|\max(t), \max(c)]$ (dans d'autres cas, c'est facile de vérifier si les bornes sont bonnes et de les ajuster si elles ne le sont pas). Pour avoir une réponse exacte, il faut utiliser un ε assez petit, puis à la fin on trouve le dernier cycle qui fait que *sup* est trop grand et on calcule son rapport c/t. Si le graphe n'est pas fortement connexe, on analyse chaque composante connexe séparément. L'implantation suivante suppose que le graphe est fortement connexe; le choix de la source pour les plus longs chemins n'est pas important.

```
min c/t : Lawler(G⟨V, E, c, t⟩) {
    inf = 1/|V| max(t) ;
    sup = max(c) ;
    while (sup − inf > ε) {
        R = (sup − inf)/2 ;
        if (maxPath(⟨V, E, c, t − c/R⟩, ·) == FAUX)
            sup = R ;
        else
            inf = R ;
    }
    return sup ;
}
```

CHAPITRE 3

REVUE DE LITTÉRATURE

Ce chapitre présente une revue de la littérature sur les méthodes d'optimisation de circuits et la méthode d'optimisation du logiciel que j'applique aux circuits. Les méthodes présentées pour l'optimisation des circuits sont la resynchronisation ainsi que différentes versions multi-phase, la resynthèse et le « wave-pipelining ». La modélisation des délais est aussi présentée, puisque c'est le délai qu'on tente d'optimiser.

3.1 Resynchronisation

La *resynchronisation*[*] ([28]) peut être vue comme une transformation générique s'appliquant sur un graphe ayant une fonction de poids sur les arcs. Dans l'article de Leiserson [28], les poids représentent le nombre de registres, et il contraint donc ceux-ci à être des entiers, mais la transformation se généralise facilement aux autres ensembles ayant une structure de groupe. La resynchronisation est un déplacement qu'on associe à chaque sommet, qui affecte les poids des arcs adjacents, en transférant du poids des arcs sortants sur les arcs entrants. Plus formellement, une resynchronisation, sur un graphe avec poids sur les arcs $G = \langle V, E, w \rangle$ (où $w : E \longrightarrow X$, X étant un groupe quelconque), est une fonction $r : V \longrightarrow X$ (appelée souvent vecteur de resynchronisation) qui le transforme en un graphe $G_r = \langle V, E, w_r \rangle$ où les nouveaux poids $w_r(e)$, pour $v \xrightarrow{e} v'$, sont définis comme suit :

[*] Traduction généralement employée pour « retiming ».

$$w_r(e) = w(e) + r(v') - r(v)$$

Ceci implique directement que pour tout chemin $v \overset{p}{\rightsquigarrow} v'$:

$$w_r(p) = w(p) + r(v') - r(v)$$

L'utilité originale de la resynchronisation ([28]) était le déplacement de registres dans un circuit synchrone de manière à diminuer le délai (temps de calcul) maximum entre deux registres. En diminuant cette distance maximale entre les registres, on permet de réduire d'autant la période d'horloge sans augmenter le nombre de cycles requis pour faire les calculs, ce qui augmente le débit du circuit.

L'idée de la resynchronisation est que, si on enlève un registre à toutes les sorties d'une fonction et qu'on en ajoute un à toutes ses entrées (Figure 1), cette fonction recevra les mêmes données qu'avant, mais un cycle plus tard, et le résultat prendra un cycle de moins avant d'arriver aux destinations. Ceci ne fait donc aucun changement sur ce qui se passe dans le reste du circuit, seulement cette unité fonctionnelle n'effectue pas les calculs au même moment que dans le circuit avant transformation.

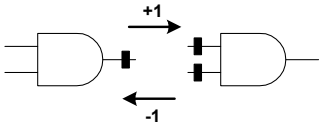


Figure 1. Opérations de base pour la resynchronisation entière.

3.1.1 Représentation du circuit

Le circuit est formé d'éléments de calculs combinatoires séparés par des registres. La représentation utilisée est un multigraphe, avec des poids sur les sommets et sur les arcs, $G = \langle V, E, d, w \rangle$. Les sommets $v \in V$ représentent les éléments de calculs et le poids $d(v) \in \mathbb{Q}$ sur celui-ci, représente le temps de calcul (ou le délai du circuit combinatoire). Ce délai est le temps maximum avant que la sortie ne se stabilise après qu'une entrée ait changé. Les arcs $e \in E$ sont les interconnexions entre les éléments de calculs, pouvant avoir un certain nombre de registres $w(e) \in \mathbb{N}$. Tous les registres sont activés par le front d'une seule horloge.

Dans un graphe quelconque, on a que $|E| \leq |V|^2$. Comme un circuit peut être un multigraphe, le nombre d'arc pourrait théoriquement être plus grand que cette borne, mais en pratique il est beaucoup plus petit. En fait $|E|$ est dans $\Theta(|V|)$ pour la plupart des circuits puisque le nombre d'entrées dans les portes ne dépend pas de la taille du circuit.

La Figure 2 montre un circuit pour un corrélateur. Il contient deux types d'éléments de calculs, des comparateurs avec délai de 3 et des additionneurs avec délai de 7. Il y a aussi un élément avec délai de 0, qui représente l'interface avec les autres composantes. Il est nécessaire de mettre cette interface si on veut que les communications avec l'extérieur restent intactes, et par conséquent que le circuit optimisé puisse être utilisé à la place du circuit original dans n'importe quel contexte.

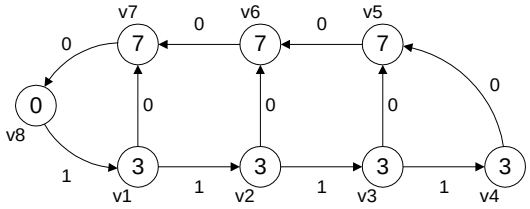


Figure 2. Circuit simple pour un corrélateur.

3.1.2 Minimisation du temps de cycle

Le temps de cycle doit être plus grand ou égal au délai maximal entre un registre et ses successeurs immédiats. Un arc ayant un poids non nul dénote un ou plusieurs registres, et on dit qu'un registre b est un successeur immédiat de a s'il y a un chemin d'arcs de poids nuls entre a et b. Par exemple, dans le circuit de la Figure 2, le chemin v4, v5, v6, v7, v8 n'a aucun registre et le délai de ce chemin est 24, la somme des poids des 5 sommets. Dans ce circuit, c'est le plus long chemin entre deux registres, et qui ne contient aucun autre registre, alors le temps de cycle doit être au moins 24 pour que le circuit fonctionne. Il faut noter que la définition du poids d'un chemin de sommets inclut le dernier sommet dans plusieurs articles, dont l'article original sur la resynchronisation [28], contrairement à la définition présentée dans la section 2.1.2, ceci venant du fait qu'on veut ici calculer la distance entre le registre qui précède le premier sommet et le registre qui suit le dernier sommet.

On tente de trouver le vecteur de resynchronisation qui va minimiser la longueur d'un tel chemin. On commence par trouver toutes les valeurs que pourrait prendre cette

longueur, puis on fait une recherche dichotomique pour trouver la plus petite pour laquelle on peut trouver une resynchronisation valide. La resynchronisation se fait avec des entiers $\in \mathbb{Z}$, mais on impose la contrainte que les poids finaux soient positifs (ou nuls).

Pour trouver les valeurs possibles, on trouve les plus courts chemins entre chaque paire de sommets du graphe du circuit en utilisant comme poids sur les arcs $v \xrightarrow{e} v'$, $\langle w(e), -d(v) \rangle$ (où w est le nombre de registres et d le délai ; section 3.1.1). Pour additionner ces poids on additionne les composantes une à une, et on utilise l'ordre lexicographique pour effectuer la comparaison. Soit le plus court chemin entre v et v' ayant un poids $\langle x, y \rangle$, on définit $W(v, v') = x$ et $D(v, v') = d(v) - y$. Le temps de cycle de la resynchronisation optimale se trouve parmi les valeurs de D.

Il est démontré dans [28] qu'une resynchronisation r d'un graphe $G = \langle V, E, d, w \rangle$ a un temps de cycle inférieur ou égal à c si et seulement si :

$$r(v) - r(v') \leq w(e) \quad \forall\, v \xrightarrow{e} v' \in E$$

$$r(v) - r(v') \leq W(v, v') \quad \forall\, v, v' \in V \text{ t.q. } D(v, v') > c$$

Il suffit de vérifier si ce système d'équations a une solution pour savoir si c est un temps de cycle possible. Cette forme de système d'inéquations peut être résolue par l'algorithme de Bellman-Ford, comme mentionné dans la section 2.2. Le nombre de contraintes étant dans $\mathrm{O}(|V|^2)$, la résolution par cette méthode prend un temps dans $\mathrm{O}(|V|^3)$. Une autre méthode, en $\mathrm{O}(|V|\,|E|)$, est aussi présentée dans [28] :

Commencer avec $r(v) = 0 \quad \forall\, v \in V$;
Répéter $|V| - 1$ fois :
  Calculer G_r en appliquant r sur le graphe initial ;
  Calculer $\Delta(v)$, le plus long délai entre un arc de poids non nul et le sommet v ;
  Pour chaque v t.q. $\Delta(v) > c$, faire $r(v) \leftarrow r(v) + 1$;
Calculer les $\Delta(v)$. Si $\Delta(v) \leq c \quad \forall\, v \in V$ alors r est une solution du système, sinon il n'y a pas de solution.

Pour trouver le temps de cycle minimum on fait une recherche dichotomique, parmi les valeurs de D, pour trouver la plus petite pour laquelle le système d'équations à une solution. Une fois qu'on a trouvé le temps de cycle minimum, on utilise la solution du système d'équations comme vecteur de resynchronisation pour déplacer les registres du circuit. Le temps total est en $\mathrm{O}(|V|\,|E|\,\lg(|V|))$.

Sur l'exemple de la Figure 2, on trouve que le temps de cycle minimum possible est 13. Un vecteur de resynchronisation permettant cette vitesse est $\{\,-1, -1, -2, -2, -2, -1, 0, 0\,\}$, pour les sommets v1 à v8 respectivement. Si on applique cette resynchronisation, on obtient le circuit de la Figure 3.

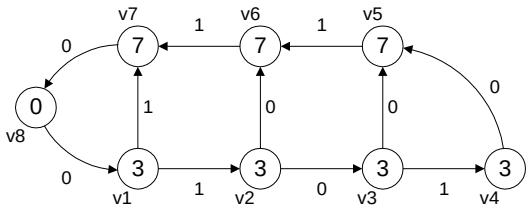


Figure 3. Circuit optimisé par la resynchronisation $\{\,-1, -1, -2, -2, -2, -1, 0, 0\,\}$.

3.2 Resynchronisation multi-phase

La méthode de minimisation du temps de cycle présentée dans la section 3.1 est optimale sous la contrainte que tous les registres sont activés sur le front d'une seule horloge, donc qu'on ne peut que faire une resynchronisation par un nombre entier de périodes d'horloge. En enlevant ces contraintes il est parfois possible d'obtenir des circuits plus rapides.

3.2.1 Phases fixes

Lockyear et Ebeling [29] présentent une extension à la resynchronisation pour gérer les registres sensibles au niveau de l'horloge et aussi permettre plusieurs phases d'horloge, fixées préalablement par le concepteur du circuit. Le fait d'utiliser plusieurs phases permet une resynchronisation par une phase plutôt qu'une période entière, ce qui donne une meilleure résolution, et par conséquent, moins de temps est perdu entre la fin d'un calcul et le rangement du résultat dans un registre. En fait cette méthode va donner la solution optimale, sans temps perdu sur le chemin critique, si les phases fixées au début le permettent.

Dans leur modèle l'horloge Φ a k phases $\{\phi_1 ... \phi_k\}$ qui ont toutes la même période T_Φ. La phase i reste active pendant un temps T_{ϕ_i} et est inactive pendant le reste du cycle. Les phases sont mises en ordre de leur front descendant. La phase i a son front descendant au temps e_i, par rapport à la référence arbitraire $e_k = T_\Phi$. Chaque registre du circuit est nommé et $P(l)$ donne la phase qui contrôle le registre l. Un circuit est dit bien formé si :

W1. Pour chaque chemin $l \overset{p}{\rightsquigarrow} m$, si $w(p) = 0$ alors $P(m) = (P(l) + 1) \bmod k)$

W2. Pour chaque cycle $v \overset{p}{\rightsquigarrow} v$, $w(p) \geq 1$

Lockyear et Ebeling associent chaque arc à un ensemble de phases, ce qui n'est pas nécessaire. Par W1 on a que, sur un chemin, les registres par lesquels on passe ont toujours des phases consécutives. Il n'est donc pas obligatoire de nommer les registres et d'avoir une fonction P sur ces registres. Il suffit d'avoir une fonction sur les arcs, donnant la phase du premier registre ainsi que le nombre de registres, et de connaître k.

Maintenant, on peut appliquer la méthode de minimisation du temps de cycle vue précédemment, mais une resynchronisation de 1 signifie une phase plutôt qu'une période. Les registres à la sortie d'une fonction ont tous la même phase, lorsqu'on enlève ces registres par resynchronisation, on place des registres de cette même phase aux entrées de la fonction.

L'article se concentre seulement sur les horloges symétriques, celles où tous les T_{ϕ_i} sont égaux. Dans la prochaine section une méthode en $\mathrm{O}(|V|\,|E|)$ est présentée, pour le cas de deux phases pas nécessairement symétriques.

3.2.2 Nombre de phases fixe

La méthode de Lockyear et Ebeling (section 3.2.1) nous oblige à fournir nous-mêmes les phases d'horloge du circuit. Ishii, Leiserson et Papaefthymiou [21] présentent une méthode permettant de déterminer ces phases automatiquement, en deux étapes : par resynchronisation et « clock tuning ». Pour k phases, minimiser la période de l'horloge par « clock tuning » (changer les phases et la période, sans déplacer les registres) prend un temps dans $\mathrm{O}(k\,|V|^2)$ et faire la resynchronisation pour atteindre une période fixée (déplacer les registres pour satisfaire une période) est dans $\mathrm{O}(k\,|V|^3)$. Pour atteindre la

période optimale il faut résoudre les deux problèmes en même temps. Une approximation, avec période au plus $(1+\varepsilon)$ fois l'optimal, pour un circuit avec deux phases, peut être trouvée en $\mathrm{O}\big(|V|^3\ \frac{1}{\varepsilon}\log\frac{1}{\varepsilon} + (|V|\,|E| + |V|^2\log|V|)\ \log\frac{|V|}{\varepsilon}\big)$. Pour k phases, l'expression contient un facteur ε^{-k}, qui est beaucoup trop gros pour un petit ε. Cela rend la méthode de Ishii, Leiserson et Papaefthymiou difficilement utilisable pour plus de deux phases. Les auteurs décrivent donc surtout le cas de deux phases.

Le modèle de circuit qu'ils utilisent pour deux phases est un graphe $G = \langle V, E, d, w, \chi\rangle$, comme dans le cas de la resynchronisation à une phase mais avec une nouvelle fonction $\chi : V \longrightarrow \{0,1\}$ qui donne la phase de chaque élément de calcul. Deux registres qui se suivent doivent avoir des phases différentes, donc il est inutile de spécifier les phases de chaque registre. En fait χ est de l'information redondante, il suffit de connaître la phase d'un des éléments pour trouver toutes les autres phases. Un circuit est dit bien formé si :

WF1. Pour tout $e \in E$, $w(e) \geq 0$

WF2. Pour chaque cycle c, $w(c) > 0$

WF3. Pour chaque arc $v \xrightarrow{e} v' \in E$, $w(e) + \chi(v') - \chi(v) \equiv 0 \pmod 2$

L'horloge est décrite par le quadruplet $\pi = \langle \phi_0, \gamma_0, \phi_1, \gamma_1\rangle$. ϕ_0 est le temps pendant lequel la première phase est active, γ_0 est le temps entre la première et la deuxième phase, ϕ_1 est le temps pendant lequel la deuxième phase est active, et γ_1 est le temps entre la deuxième et la première phase (voir la Figure 4). Le « duty ratio » est ϕ_1 / ϕ_0, et la période π est la somme des quatre temps.

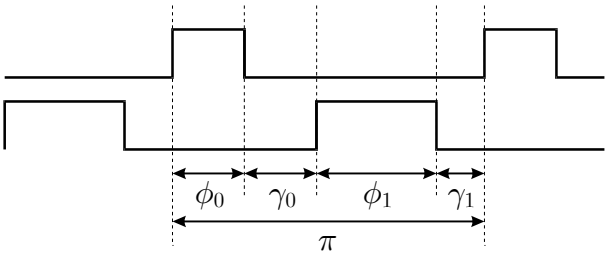


Figure 4. Modèle d'horloge à deux phases.

L'horloge du circuit est correcte si et seulement si le « rise-to-fall time » entre deux registres est toujours plus grand que le temps de propagation entre ces deux registres. Le « rise-to-fall time » est le temps entre l'activation d'un registre et la désactivation d'un de ces successeurs.

La minimisation de la période par « clock tuning » consiste à trouver l'horloge de période minimale $\pi^* = \langle {\phi_0}^*, \gamma_0, {\phi_1}^*, \gamma_1 \rangle$ qui est correcte. Une borne inférieure pour π est $-2 \min {}^{c}/_{t} (\langle V, E, -d, w \rangle)$. On trouve π^* et ${\phi_0}^*$ par programmation linéaire avec les contraintes suivantes en plus de la borne inférieure sur π.

Pour tout $v \in V$, et i entier dans $[\, 1, 3\,|V| - 3\,]$:

Si i impair et $\chi(v) = 1$: $\pi \geq 2(D(v,i) - \phi_0) \,/\, (i+1)$

Si i impair et $\chi(v) = 0$: $\pi \geq 2(D(v,i) + \gamma_0 + \gamma_1 + \phi_0) \,/\, (i+3)$

Si i pair : $\pi \geq 2(D(v,i) + \gamma_{1-\chi(v)}) \,/\, (i+3)$

Où $D(v,i) = \max \{ d(p) : v \overset{p}{\rightsquigarrow} \cdot \text{ et } w(p) = i \}$

Pour chaque v il y a seulement deux contraintes non redondantes, une pour les i impairs et une pour les i pairs. On garde donc seulement $2|V|$ contraintes, en prenant le maximum sur les i pairs et sur les i impairs de toutes les contraintes. Le temps total pour trouver la solution est dans $\mathrm{O}(|V|\,|E|)$, qui est le temps pour trouver le $\min {}^{c}/_{t}$ en utilisant un algorithme qui est dans cet ordre si la longueur de tout chemin avec $|V|$ arcs est dans $\mathrm{O}(|V|)$ ([18]).

L'algorithme suivant trouve en $\mathrm{O}(|V|^3)$ une resynchronisation qui rend l'horloge correcte, s'il en existe un :

$Q = \{$ contraintes sous la forme « $f(r(v)) \geq g(r(u))$ » $\}$;
Commencer avec $r(v) = 0 \;\; \forall\, v \in V$;
Pour chaque contrainte « $f(r(v)) \geq g(r(u))$ » non satisfaite de Q, faire :
  Tant que la contrainte n'est pas satisfaite, faire :
    $r(v) \leftarrow r(v) + 1$;
    Si $r(v) > 3\,|V| - 1$, alors retourner un échec.
  Remettre dans Q toutes les contraintes qui ont $r(v)$ dans la partie de droite ;
Retourner r.

Les contraintes à utiliser dans l'algorithme sont les suivantes :

Pour tout $u \xrightarrow{e} v \in E$, $r(v) + w(e) \geq r(u)$
Pour tout $u, v \in V$:

$$
\begin{aligned}
\text{Si } \chi(u) \neq \chi(v): \quad & \pi(1 + w(p))/2 - d(p) + \phi_{\chi(u)} \\
& + \pi \lfloor r(v)/2 \rfloor + (r(v) \bmod 2)(\gamma_{\chi(u)} + \phi_{1-\chi(u)}) \\
& \geq \pi \lfloor r(u)/2 \rfloor + (r(u) \bmod 2)(\phi_{\chi(u)} + \gamma_{\chi(u)}) \\
\text{Si } \chi(u) = \chi(v): \quad & \pi(2 + w(p))/2 - d(p) - \gamma_{1-\chi(u)} \\
& + \pi \lfloor r(v)/2 \rfloor + (r(v) \bmod 2)(\gamma_{1-\chi(u)} + \phi_{\chi(u)}) \\
& \geq \pi \lfloor r(u)/2 \rfloor + (r(u) \bmod 2)(\phi_{\chi(u)} + \gamma_{\chi(u)})
\end{aligned}
$$

Pour trouver une approximation de la période minimale après resynchronisation, avec « duty ratio » fixe, on fait une recherche dichotomique pour trouver la plus petite période pour laquelle il existe une resynchronisation qui la rend correcte. Ceci prend un temps dans $\mathrm{O}(|V|^3 \log(|V|/\varepsilon))$.

3.2.3 Délais sur l'horloge

Deokar et Sapatnekar [14] définissent l'équivalence entre le délai sur l'horloge (« clock skew ») et la resynchronisation. Pour ce faire, ils calculent premièrement un délai qui doit être appliqué sur l'horloge de chaque registre pour avoir la période désirée. Ensuite ils appliquent leur équivalence pour rapprocher ces délais le plus près possible de zéro, en faisant une resynchronisation. Si les délais ne sont toujours pas nuls, on peut les forcer à zéro en augmentant légèrement la période d'horloge ou on peut ajouter le circuit nécessaire pour implanter ce délai. Malheureusement, le circuit avec les délais sur les horloges n'est pas nécessairement correct puisque cette méthode ne tient pas compte du chemin court. Le chemin court dans un circuit combinatoire est le temps minimal avant que la sortie commence à changer après qu'une entrée ait changé. S'il existe un chemin plus court que le délai mis sur l'horloge, il y aura un problème. Forcer les délais sur l'horloge à zéro donne un circuit à une seule phase, comme dans la section 3.1.

Deokar et Sapatnekar commencent par trouver la période d'horloge optimale en utilisant $\min {}^{c}/_{t}$ et les plus longs chemins à la dernière itération sont les délais à mettre sur l'horloge. Chaque registre du circuit a une sortance de un (on duplique les registres si ce n'était pas le cas). On réduit ensuite les délais pour qu'ils soient le plus près de zéro possible, en utilisant la méthode suivante :

1. Mettre tous les registres avec délais négatifs dans une queue Q.

2. Extraire le registre j à la tête de la queue Q. p est la porte à la sortie de j (le cas où c'est un registre qui est à la sortie est traité autrement). Les délais équivalents sont trouvés pour chaque autre arc entrant dans p, à partir des registres qui alimentent p (possiblement de façon indirecte à travers d'autres portes).

3. Si un des délais trouvés dans l'étape précédente est positif, le délai sur j est mis à zéro sans resynchronisation ; Sinon, on trouve le délai sur j après l'avoir fait passé de l'autre côté de p, par resynchronisation (ce délai est le maximum des délais sur les entrées de p, plus le temps de calcul de p). Si le délai après resynchronisation est plus près de zéro, on effectue cette resynchronisation en déplaçant aussi tous les registres qui alimentent p. Pour déplacer les registres, il faut les dupliquer et laisser les copies aux endroits alimentés par eux mais qui ne sont pas sur le chemin entre eux et p.

4. Tous les registres déplacés qui ont maintenant un délai négatif sont placés à la fin de la queue Q.

5. Si la queue n'est pas vide, aller à l'étape 2.

3.3 « Software pipelining »

Le « software pipelining » est une technique d'ordonnancement des instructions d'une boucle pour avoir une haute performance d'exécution sur des processeurs pouvant exécuter des instructions en parallèle, comme les processeurs super-scalaires et VLIW. Dans le cas des séquences d'instructions sans boucles, le « list scheduling » est efficace et donne de bons résultats. Par contre, en la présence de boucle, on ne veut pas seulement réduire le temps que prend une itération mais aussi réduire le temps entre les itérations. Une bonne introduction à cette technique se trouve dans [2], alors qu'une revue beaucoup plus complète se trouve dans [1].

On représente les dépendances de données dans la boucle avec exactement le même genre de graphe que pour les circuits (section 3.1.1). Les sommets sont les opérations et ont des durées (souvent un nombre de cycles) et les arêtes donnent le flot de données. Les valeurs sur les arêtes représentent les hauteurs de dépendance, plutôt qu'un nombre de registres. Cette hauteur est la distance entre l'itération qui produit la valeur et l'itération qui en a besoin. Une hauteur de zéro signifie que la valeur est requise dans la

même itération, alors qu'une valeur de un signifie qu'elle est requise dans l'itération suivant, etc. La hauteur de dépendance et le nombre de registres sont en fait deux façons différentes de décrire la même chose. C'est ce qui me permet d'utiliser ces méthodes sur des circuits, et inversement on peut utiliser les méthodes pour les circuits en « software pipelining ».

Certaines méthodes sont basées sur l'ordonnancement sans boucle (comme le « list scheduling ») en combinaison avec la resynchronisation, comme ce qui est présenté par Calland et al. dans [9]. Ces techniques utilisent la resynchronisation sur le graphe de dépendances pour déplacer les opérations d'une itération à l'autre, puis font l'ordonnancement d'une itération. En essayant différentes resynchronisations, on tente de minimiser la durée de cet ordonnancement, de manière à pouvoir partir la prochaine itération le plus tôt possible. On peut aussi faire un dépliage du graphe, c'est-à-dire un déroulement partiel de la boucle ([10]), pour obtenir de meilleurs résultats dans certains cas. Ces méthodes ne sont que des heuristiques puisque le problème général d'ordonnancement cyclique, c'est à dire en la présence de boucles, ayant le plus court temps entre les itérations, lorsqu'il y a des contraintes de ressources, est NP-complet ([17]).

Lorsqu'il n'y a pas de contraintes de ressources (on a autant de places et d'unités opératives qu'on veut), il existe une méthode trouvant l'ordonnancement optimal en temps polynomial ([15]). Je montre comment appliquer cette méthode aux circuits dans le chapitre 4.

3.4 Resynthèse

Optimiser en ne faisant que déplacer les registres dans le graphe ne donne pas toujours de très bons résultats (voir ellipf dans le tableau 6 de [3]). Parfois il est préférable de changer la topologie du graphe pour obtenir de meilleurs résultats. La resynthèse transforme donc les circuits combinatoires (expressions booléennes) en des circuits équivalents, en terme de résultats, mais avec des caractéristiques de temps différentes. Un exemple de transformation possible, basée sur la commutativité de l'addition, est $(a+b)+c$ vers $a+(b+c)$. Cette transformation peut être intéressante si a est sur le chemin

critique mais pas c, puisqu'elle diminue le délai entre a est la sortie à une addition plutôt que deux, en augmentant le délai de c.

Dans un circuit ayant des registres, on peut appliquer ces transformations aux circuits entre les registres. Ceci impose des contraintes non nécessaires, puisque parfois on pourrait optimiser plus en regardant la logique se trouvant de l'autre côté du registre. En général la resynthèse sera donc combinée avec la resynchronisation, pour déplacer les registres des endroits qu'on veut optimiser. La puissance des méthodes utilisant l'alternance de resynchronisation et de resynthèse est discutée dans [35] et [37]. Ce dernier montre que, pour deux circuits équivalents, il n'est pas toujours possible de transformer l'un en l'autre par simple resynchronisation et resynthèse et discute de ce qu'il manque pour pouvoir le faire.

Plusieurs méthodes tentent d'intégrer la resynchronisation et la resynthèse, dans le but d'obtenir le circuit le plus rapide se trouvant par ces transformations ([22][33]). En général, l'idée est de trouver un cône de logique qui semble intéressant d'optimiser puis de repousser les registres vers les entrées du cône, de manière à ce qu'il n'y ait aucun registre dans le cône, pour pouvoir faire la resynthèse de cette logique. Si on ne peut pas enlever les registres, à cause de la sortance d'une porte, on peut dupliquer la porte.

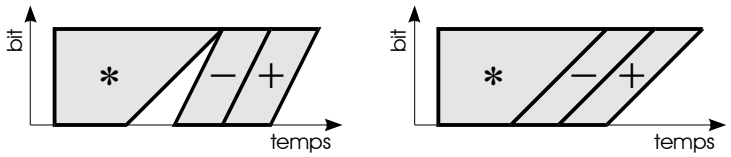
3.5 Modélisation du délai

Les représentations des circuits discutées dans les sections précédentes utilisent un modèle de délais très simple. On parlait seulement du délai maximum avant que les sorties d'un bloc combinatoire soient stables lorsqu'on change les entrées. Le délai minimum avant qu'une des sorties change lorsqu'on change une entrée a aussi été mentionné, puisqu'il faut en tenir compte, mais on utilisait zéro comme borne inférieure. Nous verrons que les techniques, même celles qui sont dites optimales, ne sont pas si précises puisque souvent les délais ne sont pas bien représentés.

3.5.1 Délai pour chaque bit

En général, les éléments de calcul dans un circuit reçoivent plusieurs signaux en entrée et donnent plusieurs signaux en sortie. Les délais entre les entrées et les sorties ne sont pas tous égaux, et parfois il y a une grande différence. Prenons, par exemple, un

additionneur à propagation de retenue (« ripple-carry »). Le délai à partir du bit de poids faible en entrée, vers le bit de poids faible de sortie est petit, alors que vers le bit de poids fort de sortie le délai est grand. Aussi, on peut recevoir le bit de poids fort de l'entrée un certain temps après l'arrivée du bit de poids faible sans que les temps de stabilisation des sorties changent. Le temps que prennent deux additions chaînées n'est donc pas le double du temps que prend une addition. Une méthode permettant d'optimiser des circuits ayant des délais de ce type est présentée par Park et Choi dans [34]. Pour permettre le chaînage à travers les registres, il utilise des bascules actives sur les niveaux. Visualisons cette méthode complexe en représentant les opérations par des formes en deux dimensions qu'on tente d'aligner avec le moins d'espace possible entre elles. Les deux dimensions de ces formes représenteront la position du bit et le temps. Un exemple pour une multiplication, suivie d'une soustraction puis d'une addition, se trouve à la Figure 5a. Souvent les blocs pourraient être déformés plutôt que seulement déplacés, comme illustré à la Figure 5b pour l'addition et la soustraction à propagation de retenue. Park et Choi ne mentionnent pas cette déformation possible, qui peut permettre, dans certains cas, une meilleure optimisation. De plus, ils font un ordonnancement acyclique (sans boucles), ce qui nécessite l'utilisation d'une heuristique pour passer aux cas cycliques traités dans les sections précédentes.



a) placement des blocs. b) déformation des blocs.

Figure 5. Ordonnancement au niveau des bits.

Une fois qu'ils ont un ordonnancement, une horloge à haute fréquence est utilisée pour contrôler une machine à états qui s'occupe d'activer les registres aux bons moments. L'ordonnancement est séparé en étapes de contrôle de longueurs possiblement différentes. Chaque étape est un état de la machine et un compteur ordonne le changement d'état, selon le temps que dure l'étape ; En terme de machines à états, on met beaucoup d'états pour chaque étape, pour qu'elles aient les bonnes durées. Pour n'avoir aucune perte de temps due à la précision de l'horloge, il faut que la période de cette horloge soit un diviseur de la longueur de chaque étape, ce qui n'est pas réaliste en général. Ils utilisent donc une technique pour trouver une période qui n'est pas trop courte et qui ne donne pas

trop de perte de temps. Pour ne pas avoir ces pertes, j'ai développé les horloges à période variable, que j'expliquerai dans la section 7.4.

3.5.2 Faux chemins

Lorsque chaque élément de calcul est une porte logique, on n'a pas besoin d'un modèle comme celui présenté dans la section précédente. Un additionneur est alors formé de beaucoup d'éléments, et la méthode d'ordonnancement s'occupera de les mettre au bon moment. Évidemment, la taille du graphe sera beaucoup plus grande, et si on ne peut pas diviser les blocs qui viennent d'une librairie, il ne sera pas toujours possible d'aller à ce niveau. Mais même en allant au niveau des portes (on pourrait aussi penser au niveau transistors), les méthodes d'ordonnancement peuvent retarder une opération lorsqu'elles croient faussement que ses prédécesseurs n'ont pas eu le temps de se stabiliser. En général on utilise un algorithme des plus longs chemins (plus grand temps de calcul) pour savoir à quel moment on peut exécuter une opération, puisqu'une opération ne peut commencer que lorsque tous ses prédécesseurs ont terminé leurs calculs. Mais parfois, une entrée n'a pas d'influence sur la sortie, ce qui fait que la sortie peut être stable avant cette entrée.

Par exemple, dans le circuit de la Figure 6, on pourrait croire que le délai entre a et g est de 3 si le délai d'une porte est de 1, à cause du chemin a$\rightarrow$e$\rightarrow$f$\rightarrow$g. En regardant en détail, on s'aperçoit que ce n'est pas le cas. Si c est à 1, f n'est pas influencé par a (il est toujours 1), et si b est à 0, e n'est pas influencé par a (il est toujours 0). Donc il ne reste que le cas où $\mathsf{b} = 1$ et $\mathsf{c} = 0$, pour lequel f sera influencé par a. Mais comme dans ce cas d est toujours à 0, g ne sera pas influencé par a. Le chemin a$\rightarrow$e$\rightarrow$f$\rightarrow$g est donc un « faux » chemin. En essayant toutes les transitions possibles sur les signaux d'entrée, on comprend que la sortie se stabilise au plus 2 unités de temps plus tard.

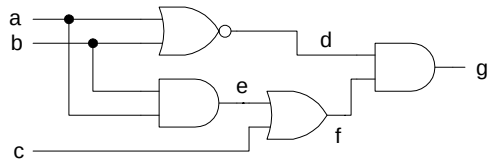


Figure 6. Circuit combinatoire de délai 2 (faux chemin de 3).

Kukimoto et Brayton [24][25] présentent une méthode qui regarde toutes les transitions possibles et qui dit à quel moment une entrée doit être valide pour que la sortie soit stable

au bon moment. Malheureusement, leur technique prend un temps exponentiel même en meilleur cas, mais elle donne une caractérisation assez complète du délai. Avec cette technique, on peut savoir si un module est plus vite qu'un autre dans tous les cas, et de savoir quel est le plus long chemin qui n'est pas « faux ». Ils ont aussi étendu la méthode pour qu'elle soit applicable de manière hiérarchique ([26]) pour ainsi réduire le temps de calcul.

3.6 « Wave-pipelining »

L'effet de « wave-pipelining », dont une bonne revue se trouve dans [6], vient du fait qu'un circuit prend du temps avant que sa sortie change, après que son entrée ait changé. Le circuit a donc une mémoire de durée limitée qui peut parfois être utilisée. Cette mémoire permet de commencer un calcul dans une unité combinatoire, avant que le calcul précédent, dans cette même unité, ait terminé. C'est comme un pipeline, mais sans avoir à ajouter de registres. Il n'y a pas de séparation physique entre les étages, donc il peut être difficile de parler d'étages de pipeline, mais l'effet pipeline est bien là.

La conception d'un tel circuit est en général difficile puisqu'il n'y a pas présentement d'outil de synthèse de haut niveau qui peut en générer. En effet, il faut tenir compte de plusieurs choses lors de la conception. Le but du « wave-pipelining » est généralement d'accélérer le circuit, et non de diminuer le nombre de registres. On tente donc de diminuer le temps maximum avant que la sortie ne se stabilise. Pour avoir un bon pipeline sans avoir à ajouter de registres, les registres n'ayant pas un délai nul dans la réalité, on veut aussi augmenter le temps minimum avant que la sortie ne commence à changer. Par exemple, si le temps minimum est au moins la moitié du temps maximum, on peut utiliser le circuit comme si c'était un pipeline à deux étages, ou presque. Le presque vient du fait que parfois ces circuits ne peuvent pas être arrêtés. Comme la mémoire du circuit est courte, si on ne le fait pas fonctionner assez vite les données peuvent se perdre. Dans notre exemple, si on donne au circuit deux calculs à faire successivement en attendant la moitié du temps maximum entre les deux, puis ensuite on attend trop longtemps avant de lire le résultat, on aura le deuxième résultat et on aura échappé le premier.

CHAPITRE 4

CIRCUIT À DÉBIT MAXIMAL

Le débit maximal d'un circuit, représenté par un graphe comme décrit dans la section 3.1.1, peut être facilement trouvé par les méthodes développées pour le « software pipelining », si on ne permet pas de changer la topologie du graphe. Considérer les sommets comme étant les opérations, et le poids sur les arcs comme étant un nombre d'itérations entre la production d'un résultat et son utilisation, plutôt qu'un nombre de registres, c'est simplement une autre façon de voir la même chose (ça ne change en rien le comportement). Donc les méthodes pour l'ordonnancement des boucles s'appliquent directement aux mêmes circuits sur lesquels s'applique la resynchronisation. Mais, comme le résultat n'est pas limité aux entiers, on ne peut pas prendre ce résultat pour savoir combien de registres il faut placer sur chaque arc. J'ai développé une méthode pour reconstruire un circuit à partir du résultat, qui est présentée en détails dans [3] (article se trouvant après cette introduction du chapitre). Le circuit résultant est un circuit multi-phase, dont le nombre et la position des phases sont déterminées automatiquement. L'algorithme ne tente pas directement de minimiser le nombre de phases mais plutôt de minimiser le nombre de registres, ce qui réduit habituellement le nombre de phases. Cette tentative de minimisation n'est pas optimale, mais le circuit a malgré cela le débit maximal selon la précision de la représentation du circuit original.

Les contributions principales de cet article sont (traduction tirée de l'article) :

- Cette méthode permet l'utilisation mixte de bascules activées sur les fronts et sur les niveaux, le choix et le placement de ceux-ci étant fait par un algorithme en temps linéaire.

- La complexité globale de la méthode est $\mathrm{O}(|V|\ |E|\ \lg(|V|\, d_{max}))$, ou $\mathrm{O}(|V|\ |E|\, d_{max})$ pour des petits délais entiers ([18]). La complexité de la resynchronisation est $\mathrm{O}(|V|\ |E|\log(|V|))$ ([28]). Même si la complexité globale des deux approches est similaire, on évite ainsi de calculer tous les plus courts chemins, ce qui prendrait beaucoup d'espace et de temps. Aucune des autres méthodes de minimisation de la période, décrites dans les ouvrages cités, n'a une borne supérieure plus petite que la nôtre.
- La solution optimale du problème de minimisation de la période d'horloge est toujours atteinte.
- Les éléments de calcul peuvent avoir un délai plus grand que la période d'horloge. Dans ce cas, le débit optimal est atteint en augmentant le nombre d'unités de calcul pour cette fonction.

Les pages suivantes contiennent une copie l'article [3], dans son format original (sauf la numérotation des pages) pour la parution dans *ACM Transactions on Design Automation of Electronic Systems*, Vol. 7, No. 2, Apr. 2002.

Note au lecteur : quelqu'un qui a lu les chapitres 1 et 2 de cette thèse peut sauter la section 1 de l'article, et regarder rapidement les sections 2.1 à 2.3, puis commencer à lire vraiment à partir de la section 2.4.

Optimal design of synchronous circuits using software pipelining techniques

François R. Boyer
and
El Mostapha Aboulhamid
Université de Montréal
and
Yvon Savaria
École Polytechnique de Montréal
and
Michel Boyer
Université de Montréal

We present a method to optimize clocked circuits by relocating and changing the time of activation of registers to maximize the throughput. Our method is based on a modulo scheduling algorithm for software pipelining, instead of retiming. It optimizes the circuit without the constraint on the clock phases that retiming has, which permits to always achieve the optimal clock period. The two methods have the same overall time complexity, but we avoid the computation of all pair-shortest paths, which is a heavy burden regarding both space and time. From the optimal schedule found, registers are placed in the circuit without looking at where the original registers were. The resulting circuit is a multi-phase clocked circuit, where all the clocks have the same period and the phases are automatically determined by the algorithm. Edge-triggered flip-flops are used where the combinational delays exactly match that period, whereas level-sensitive latches are used elsewhere, improving the area occupied by the circuit. Experiments on existing and newly developed benchmarks show a substantial performance improvement compared to previously published work.

Categories and Subject Descriptors: B.6.1 [**Logic Design**]: Design Styles—*Sequential circuits*; B.6.3 [**Logic Design**]: Design Aids—*Optimization/Automatic synthesis*

General Terms: Performance, Algorithms

Additional Key Words and Phrases: retiming, software pipelining, resynthesis

---

A shorter version (6 pages) of this paper appeared in the Proceedings of ICCD '98, pp. 62-67.
Name: François R. Boyer
Affiliation: DIRO, Université de Montréal
Address: 2920 Ch. de la Tour, C.P. 6128, Succ. Centre-Ville, Montréal, (Qué), Canada, H3C 3J7
E-mail: boyerf@IRO.UMontreal.CA

1. INTRODUCTION

In a synchronous circuit, clocked storage elements are used to regulate the data flow and to provide stable inputs while functions (combinational logic) are evaluated. The speed of the circuit is determined not only by the calculation time of these functions, but also by the time wasted waiting for the synchronization point (clock) to arrive. For circuits synchronized by a single periodic event (single clock), the wasted time between two storage elements (registers) is the period duration minus the combinational delay between these storage elements. In this paper we focus on methods for minimizing that wasted time. The proposed method also identifies the circuit path that limits the speed, for further optimization if necessary.

Leiserson and Saxe [1991] reduced the wasted time by moving registers to minimize the maximum combinational delay between two registers and changing the clock period to that value. This register movement does a better repartition of combinational logic, resulting in a tighter fit in the clock period. The method they present, called retiming, is proved to give the register placement that permits the smallest clock period under the constraints that registers are edge-triggered and are all controlled by the same clock. However, it was found in many cases that this solution is not optimal, because registers cannot always be moved so that no time is wasted on the critical path. Indeed, a part of the circuit is always "retimed" by an integral number of periods; better results could be obtained if some form of fractional retiming was applied.

Lockyear and Ebeling [1994] presented an extension to retiming using level-sensitive registers (latches) with multi-phase clocks, the phases being fixed by the designer instead of being computed automatically. The use of multi-phase clocks permits to "retime" a part of the circuit by one phase instead of a whole period, which gives a better resolution and thus a tighter fit to reduce wasted time. That method will give an optimal solution, with no wasted time on the critical path, but only if the phases specified allow it and under the assumption of using a perfect clock.

Deokar and Sapatnekar [1995] define the "equivalence between clock skew and retiming", which they use to minimize the clock period. They first calculate a "skew" that should be applied to the clock input of each flip-flop in order to have the desired period. Then they apply that equivalence to retime the circuit to bring down the "skews" to zero (or as close as possible). If the "skews" are not zero, they can be forced to zero (increasing slightly the clock period) or circuitry can be added to implement that skew on the clock. It is not clear that the circuit with the skews on the clock will be correct because they ignore the short path constraints.

Maheshwari and Sapatnekar [1997, 1998] use retiming to reduce the area (number of registers) for a given clock period. They use a longest path algorithm to find ASAP and ALAP locations of the registers, which permit to reduce the computation time by reducing the size of the linear programming problem to be solved.

Ishii, Leiserson, and Papaefthymiou [1997] present methods to minimize the clock period on a multi-phase level sensitive clocked circuit. They also show how to convert an edge-triggered clocked circuit into a faster level-sensitive one. The method is in two steps: retiming and clock tuning. For a k-phase simple circuit, minimizing the clock period using clock tuning is $O(k\,|V|^2)$ and retiming to achieve a given

clock period for fixed duty-ratios is $O(k\,|V|^3)$, where $|V|$ is the number of computing elements in the circuit. Approximation schemes for solving the two steps at once, to achieve the minimum clock period, are given. For simultaneous retiming and clock tuning, with no conditions on the duty cycles of a two-phase circuit, an approximation, with a period at most $(1+\epsilon)$ times the optimal, can be found in $O\left(|V|^3 \frac{1}{\epsilon} \log \frac{1}{\epsilon} + \left(|V||E| + |V|^2 \log |V|\right) \log \frac{|V|}{\epsilon}\right)$, where $|E|$ is the number of connections between elements. For k-phase, the running time for that approximation contains a factor of ϵ^{-k}, which is impractically large for small values of ϵ.

An optimal solution to the clock tuning problem, on multi-phases level-sensitive circuits, with fixed register placement, is presented by Sakallah, Mudge and Olukotun [1990]. The algorithm uses linear-programming, but no running time or upper bound is given.

Legl, Vanbekbergen, and Wang [1997] extend retiming to handle circuits where not all the registers are enabled at the same time. The idea is that registers can be moved across a logic block only if they are enabled by the same signal. They do not change the enable time of registers.

Work has also been done to speedup loops on parallel processors. The software pipelining method discussed in Van Dongen, Gao, and Ning [1992] gives an optimal schedule of the operations (with no wasted time on the critical cycle) if there are no constraints on the resources (number of operative units). Also, some methods use retiming as a heuristic to find schedules under resource constraints [Bennour and Aboulhamid 1995].

We present a method that uses software pipelining, instead of retiming, to find an optimal schedule of the operations in a circuit, and then a way to reconstruct the circuit from that schedule. Scheduling is much like calculating the clock skews [Deokar and Sapatnekar 1995; Maheshwari and Sapatnekar 1997; Maheshwari and Sapatnekar 1998], but then we do not apply retiming according to that schedule. As said previously, not taking into account the short path problem can cause unpredictable circuit behavior when the skews are not forced to be zero. However, reducing the skews to zero results in a single-phase circuit, the same limitation as the original retiming [Leiserson and Saxe 1991]; we are considering the worst case length for short paths. Once the schedule is done, we place registers with an $O(|E|)$ algorithm, independently of their original placement. Our method produces a circuit with a multi-phase clock; neither the phases nor their count is fixed a priori like in some previous work [Ishii et al. 1997; Lockyear and Ebeling 1994]. Our method is not an approximation, and the running time is low even for circuits with many phases, unlike the work of Ishii et al. [1997]. We do not handle the problem of finding a solution with constraints on the clock phases, which is done in Ishii's work by having a fixed number of phases and permitting to do retiming with fixed duty ratios.

The main contributions of this paper are the following:

—The method is not limited to edge-triggered flip-flops or level-sensitive latches only, but our proposed solution can use a mixture of the two, which is automatically found by a linear algorithm.

—The overall complexity of our method is $O(|V||E| \log(|V| d_{\max}))$, or $O(|V||E| d_{\max})$ for small integral delays [Hartmann and Orlin 1993] where $|V|$ is the number of

computing elements in the circuit, $|E|$ is the number of connections between these elements and $d_{\max}$ is the maximum duration of the computations done by the elements. The complexity of the retiming method is $O(|V||E| \log |V|)$ [Leiserson and Saxe 1991]. Even if the overall complexity of the two approaches is similar, we avoid the computation of all pair-shortest paths, which is a heavy burden regarding both space and time. Our method has an upper bound not higher than any clock minimization method described in previous work cited in this paper.

—The optimal solution to the clock period minimization problem is always achieved.

—Some combinational functions may have a delay greater than the clock period. In this case, the optimal throughput can be reached by increasing the number of functional units realizing the function.

This paper is organized as follows. Section 2 introduces the notations and definitions used in this work. Section 3 presents the main algorithm used as a replacement to the retiming approach. It gives also the main theorems concerning the validity of our approach. Section 4 gives the algorithms for register placement and the automatic selection of edge-triggered or level-sensitive storage. Section 5 extends the method to non-integer clock periods and combinational logic with delays greater than the clock period. Section 6 presents the implementation and experimental results. Section 7 concludes the paper and points to some future work.

2. PRELIMINARIES

In this section, we define the graph representation of a sequential circuit, the "retiming" transformation of an edge-weighted graph, the short and long path constraints, and the basic notion of schedule.

2.1 Input Circuit Definition

As in the original retiming article [Leiserson and Saxe 1991] the input circuit is formed by combinational computing elements separated by registers. We model that circuit as a finite, vertex-weighted, edge-weighted, directed multigraph[1] $G = \langle V, E, d, w \rangle$. The vertices V represent the functional elements of the circuit, and they are interconnected by edges E. Each vertex $v \in V$ has a non-negative rational propagation delay $d(v) \in \mathbb{Q}$ which is the maximum delay before its outputs stabilize; no minimum delay is assumed (we use zero as lower bound on the short path, which is a common conservative approach). Each edge $e \in E$ is weighted with a register count $w(e) \in \mathbb{N}$ representing the number of registers separating two functional elements. We extend the functions d and w to paths in the graph. For any path $v_0 \overset{p}{\rightsquigarrow} v_k = v_0 \overset{e_0}{\rightarrow} v_1 \overset{e_1}{\rightarrow} \dots \overset{e_{k-1}}{\rightarrow} v_k$ we define

$$d^-(p) = \sum_{i=0}^{k-1} d(v_i) \qquad w(p) = \sum_{i=0}^{k-1} w(e_i)$$

Note that, unlike Leiserson and Saxe's definition of $d(p)$ [1991], $d^-(p)$ does not take into account the weight of the last vertex ($d(p) = d^-(p) + d(v_k)$).

[1]Strictly speaking, G should also include the two functions $h : E \to V$ (head of edge) and $t : E \to V$ (tail of edge) such that if $v \overset{e}{\rightarrow} v'$ then $h(e) = v'$ and $t(e) = v$. There may be many edges e with same head and tail.

Fig. 1 shows the graph for the correlator example of Leiserson and Saxe [1991].

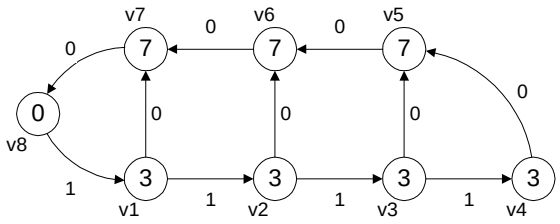


Fig. 1. A simple correlator circuit.

This circuit performs an iterative task: at each clock cycle, the circuit calculates new values from the previously calculated ones. The register count $w(e)$ for $v \xrightarrow{e} v'$ can be thought of as the number of iterations between the time the value is calculated by v and the time it is used by v' (e.g. in Fig. 1, the element v2 uses the result of the previous iteration of element v1). By thinking of the graph as an inter and intra-iteration dependency graph, instead of number of registers, we can use an algorithm for optimal loop scheduling [Van Dongen et al. 1992] to have the maximal throughput, which is limited only by data dependencies and propagation delays. This schedule is not limited by the clock period or the position of the registers (proved in Lemma 5). Register placement is performed at a subsequent step that takes the results of the scheduling step as input.

2.2 Retiming

Retiming [Leiserson and Saxe 1991] can be viewed as a displacement assigned to each vertex, which affects the length (weights) of the edges, taking weight from one side and putting it on the other. More formally, a *retiming* on an edge-weighted graph $\langle V, E, w\rangle$ is a function $r : V \longrightarrow \mathbb{Z}$ (or $V \longrightarrow \mathbb{Q}$ when edge weights $w(e)$ can be fractional, which gives a more general graph transformation) that transforms it into a new *retimed* graph $G_r = \langle V, E, w_r\rangle$, where the weights $w_r(e)$ for $v \xrightarrow{e} v'$ are defined by:

$$w_r(e) = w(e) + r(v') - r(v)$$

which directly implies that, for any path $v \overset{p}{\rightsquigarrow} v'$,

$$w_r(p) = w(p) + r(v') - r(v) \tag{1}$$

2.3 Short and long paths

When you change the inputs of a circuit, after some time the outputs start to change and may oscillate before stabilizing to the appropriate result. The short path is the least time the circuit takes before the output is affected by some input variation. Similarly, the long path is the longest time possible between the inputs change and the outputs stabilize. When designing a sequential circuit, we must have bounds on the short and long paths to know when the result is valid and when registers can be activated. These times are always positive (the outputs change after the inputs), so we can use zero as a lower bound. Using some tighter bounds could

permit more optimizations. The delays in our circuit graph are the upper bounds on the long paths.

2.4 Scheduling and software-pipelining

A *schedule* s [Bennour and Aboulhamid 1995; Van Dongen et al. 1992; Hanen 1994; Hwang et al. 1991; De Micheli 1994] is a function $s : \mathbb{N} \times V \longrightarrow \mathbb{Q}$, where $s_n(v) \equiv s(n, v)$ denotes the time at which the n^{th} iteration of operation v is starting. A schedule s is said to be *periodic* with period P (all iterations having the same schedule), if:

$$\forall n \in \mathbb{N}, \forall v \in V \; s_{n+1}(v) = s_n(v) + P$$

A schedule is said *k-periodic* if there exist integers n_0, k and a positive rational number P such that:

$$\forall n \geq n_0, \forall v \in V \; s_{n+k}(v) = s_n(v) + P\,k$$

Both periodic and k-periodic schedules have the same throughput $\omega = {}^1\!/_P$ (also called "frequency" in some papers, causing some confusion with the clock frequency), but k-periodic schedules have a period of Pk. A schedule is said to be valid iff the operations terminate before their results are needed (whilst respecting resource constraints if any). If the only constraints come from data dependency, s is *valid* iff for all edges $v \xrightarrow{e} v'$,

$$s_n(v) + d(v) \leq s_{n+w(e)}(v').$$

3. SCHEDULING OPERATIONS

In this section, we show how to find the theoretical maximum throughput of a circuit (due to data dependency), and then, how to make a schedule that has a specified throughput. The scheduling is based on a loop-acceleration technique used in software pipelining.

3.1 Maximum throughput

First we must find the critical cycle in the circuit, i.e. the cycle $v \overset{c}{\rightsquigarrow} v$ that limits the throughput. The maximum throughput is [Van Dongen et al. 1992]:

$$\omega = \min_{c \in \mathcal{C}} \left\{ \frac{w(c)}{d^-(c)} \right\}$$

where $\mathcal{C}$ is the set of directed cycles in G.

If there is no cycle (if $\mathcal{C} = \varnothing$), then ω is infinite; this means that we can compute all the iterations at the same time, if we have enough resources to do so. Computing the maximal throughput is a minimal cost-to-time ratio cycle problem [Lawler 1976], which can be solved in the general case in $O(|V||E|\log(|V|d_{\max}))$ where $d_{\max} = \max_{v \in V} d(v)$. The method is based on iteratively applying Bellman-Ford's algorithm for longest paths on the graph $G_l = \langle V, E, w_l \rangle$ derived from G by letting

$$w_l(e) = d(v) - Pw(e) \in \mathbb{Q}$$

where $v \xrightarrow{e} \cdot \in E$ and $P = {}^1\!/_\omega$ is the period. A binary search is used to find the minimal value of P for which there is no positive cycle in G_l [Bennour and

Aboulhamid 1995; Van Dongen et al. 1992]. For small integral delays, we can compute the maximal throughput in $O(|V||E|d_{\max})$ [Hartmann and Orlin 1993]. For the circuit of Fig. 1, the minimal period P is equal to 10, which is interesting compared to retiming [Leiserson and Saxe 1991] that gave a minimal period of 13. This value that we obtain using loop scheduling is the same as that Lockyear and Ebeling [1994] obtained using retiming on a modified graph; their approach, though, may fail to give the optimal throughput if it is not given the right set of phases.

3.2 Schedule of a specified throughput

The graph G_l (whose weight function w_l is described above) is used to find a valid schedule with the specified throughput. Fig. 2 shows the graph for $\omega = {}^1/_{10}$ ($P = 10$), where the vertices are labeled by the length of their longest paths from/to v1.

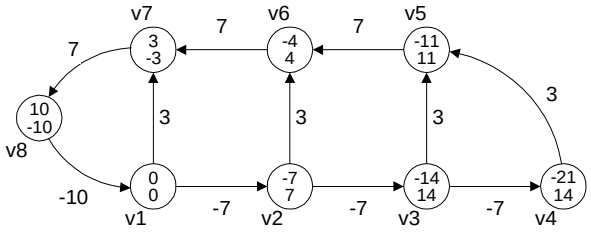


Fig. 2. G_l with the longest paths from/to v1 in the vertices.

The weights denote the minimum distance between the schedule of two vertices. For example the -7 between v1 and v2 means that v1 must be scheduled at most 7 units of time after v2, the 3 between v1 and v7 means that v7 must be scheduled at least 3 units of time after v1, etc. Finding the longest paths in this graph gives a possible schedule with a period of P. A cycle with a positive length gives constraints that cannot be satisfied; the graph will have no positive cycle iff the period is feasible. The longest paths can be found in time $O(|V||E|)$ with Bellman-Ford's algorithm. The ASAP and ALAP schedules can be obtained by finding the longest paths to and from a chosen vertex. To find the longest paths to a vertex, Bellman-Ford's algorithm can be applied on the transposed graph $G_l^\top$ of G_l where all the edges in G_l are made to point in the opposite direction, i.e. what was the head of an edge becomes its tail and conversely. Given a specific vertex v, the ASAP (resp. ALAP) schedule of any other vertex v' is the longest path (resp. minus the longest path) from v to v' in G_l (resp. $G_l^\top$). The longest path from a vertex to itself gives us its mobility. The mobility can also be obtained as the difference between the ALAP and ASAP schedule times or vice-versa. Let $l(v, v')$ be the length of the longest path from v to v' in G_l. Table 1 gives l for the graph in Fig. 2.

The length $l(v, v')$ gives the relative schedule of vertices for the same iteration, that is we have $s_n(v') - s_n(v) \geq l(v, v')$. This permits to determine intervals in which an operation must be scheduled relatively to another operation. Also, because we want a periodic schedule with a period of P, we have that:

$$l(v, v') + Pm \leq s_{n+m}(v') - s_n(v) \leq -l(v', v) + Pm \tag{2}$$

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	**0**	**-7**	**-14**	**-21**	**-11**	**-4**	**3**	**10**
2	**7**	0	-7	-14	-4	3	10	17
3	**14**	7	0	-7	3	10	17	24
4	**14**	7	0	-7	3	10	17	24
5	**11**	4	-3	-10	0	7	14	21
6	**4**	-3	-10	-17	-7	0	7	14
7	**-3**	-10	-17	-24	-14	-7	0	7
8	**-10**	-17	-24	-31	-21	-14	-7	0

Table 1. Longest paths in graph G_l; only the values in bold are calculated.

For example, looking at Table 1 we know that $s_n(\mathsf{v2}) - s_n(\mathsf{v1}) \geq -7$ and $s_n(\mathsf{v1}) - s_n(\mathsf{v2}) \geq 7$ which means that $s_n(\mathsf{v2}) - s_n(\mathsf{v1}) = -7$. Keeping only one line, and the corresponding column, for a vertex that is on the critical cycle, we find the intervals where we can schedule the vertices. This means that we do not need to compute all-pairs longest paths but only the longest path from and to that vertex. Table 2 presents the schedule intervals relative to vertex v1.

3.3 Schedule (multi)graphs

We represent a periodic schedule of operations (vertices), with period P, by a *schedule graph* $G_s = \langle V, E, d, w_s, P\rangle$, where d is as usual a delay function on vertices, and $w_s : E \to \mathbb{Q}$ is a weight function which associates to each $v \xrightarrow{e} v'$ the time distance between the start of operation v and that of operation v' using v's output as input. Schedule graphs are required to satisfy the following condition:

$$\text{If } v \overset{p}{\rightsquigarrow} v' \text{ and } v \overset{p'}{\rightsquigarrow} v' \text{ then } w_s(p) - w_s(p') = P\,k \text{ for some integer } k.$$

i.e. all paths between two fixed vertices have the same length modulo the (possibly fractional) period. To every multigraph $G = \langle V, E, d, w\rangle$ specifying a synchronous circuit and every P periodic schedule s on G (valid or not), there is an associated schedule graph $G_s = \langle V, E, d, w_s, P\rangle$ where $w_s(e)$ for $v \xrightarrow{e} v'$ is defined by

$$w_s(e) = s_{w(e)}(v') - s_0(v)$$

Indeed, $w_s(e) = s_0(v') - s_0(v) + Pw(e)$; if $v \overset{p}{\rightsquigarrow} v'$ and $v \overset{p'}{\rightsquigarrow} v'$, then $w_s(p) - w_s(p') = P\,k$ for $k = w(p) - w(p')$.

Definition 1. The graph G_s is *consistent* iff for all $v \xrightarrow{e} \cdot \in E$, $w_s(e) \geq d(v)$

Fig. 3 shows a consistent G_s for our example, using the ALAP schedule from Table 2 as s. The following lemma shows that the longest path from v to v' in G_l is independent of the placement of the registers in G.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
ASAP	0	-7	-14	-21	-11	-4	3	10
ALAP	0	-7	-14	-14	-11	-4	3	10
Mobility	0	0	0	7	0	0	0	0
Interval	0	-7	-14	[-21,-14]	-11	-4	3	10

Table 2. Schedules and mobility relative to v1.

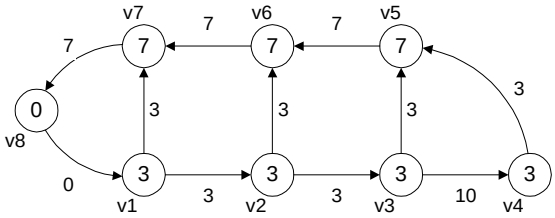


Fig. 3. Schedule graph G_s with $P = 10$.

LEMMA 1. *For all possible retimings r of G, p is a path of maximum length between v and v' in G iff it has the same property in the retimed graph G_r. Moreover $l_r(v, v') = l(v, v') - [r(v') - r(v)]P$, where $l_r(v, v')$ is the maximum length of a path between v and v' in $(G_r)_l$.*

PROOF. From $w_l(e) = d(v) - w(e)P$ for $v \xrightarrow{e} \cdot$, we deduce that for any path $v \overset{p}{\rightsquigarrow} v'$, $w_l(p) = d^-(p) - w(p)P$; moreover, if r is a retiming of G, then $w_r(p) = w(p)+r(v')-r(v)$ and so $w_{rl}(p) = d^-(p)-w_r(p)P = d^-(p)-[w(p)+r(v')-r(v)]P = w_l(p) - [r(v') - r(v)]P$. The result follows from the definitions of $l_r(v, v')$ and $l(v, v')$. □

LEMMA 2. *The graph G_s is the retimed graph derived from $\langle V, E, d, Pw\rangle$ where the retiming function $r(v)$ is $s_0(v)$ i.e. the schedule of iteration 0.*

PROOF. By definition of G_s, $w_s(e) = s_{w(e)}(v') - s_0(v)$ for $v \xrightarrow{e} v'$. Since s is periodic, $s_{w(e)}(v') = s_0(v') + w(e)P$ and so $w_s(e) = Pw(e) + s_0(v') - s_0(v)$. □

The graph with the edge-weights all multiplied by a constant c is called a *c-slow circuit*. The circuit has been slowed down by a factor of c, so that it does the same computation but it takes c times as many clock cycles [Leiserson and Saxe 1991]. Therefore, the graph G_s could be interpreted as a circuit that does the same computations as G. A c-slow circuit can be retimed to have a shorter clock period but the throughput is not higher if we are doing only one computation at a time; multiple interleaved computations can improve the efficiency. This is not our interpretation of that graph and our final circuit is not c-slow, it produces results every clock cycle like the original circuit.

LEMMA 3. *For all periodic schedules s, G_s is consistent iff s is valid.*

PROOF. A schedule s is valid iff the operations terminate before their results are needed, i.e. for all $v \xrightarrow{e} v'$, $s_n(v) + d(v) \leq s_{n+w(e)}(v')$; since s is periodic, this is equivalent to $s_0(v) + d(v) \leq s_{w(e)}(v')$ i.e. $w_s(e) \geq d(v)$ by definition of w_s. □

Notice that retiming a schedule graph always gives a schedule graph. Consequently:

LEMMA 4. *A retiming r of a consistent schedule graph G_s is legal iff it preserves consistency (i.e. $(G_s)_r$ is consistent).*

A consequence of LEMMA 4 is that we can explore different schedules (all with the same period) by retiming the graph G_s to find one that is easier/smaller to implement.

The following result shows that retiming G has no influence on the schedule graphs provided we adjust the schedules accordingly.

LEMMA 5. *For any valid periodic schedule s and any legal retiming r of G, $s'_n(v) = s_n(v) - r(v)P$ is a valid periodic schedule of G_r such that $(G_r)_{s'} = G_s$.*

PROOF. If s is valid then for any $v \xrightarrow{e} v'$, $s_0(v) + l(v, v') \leq s_0(v')$. By LEMMA 1 $l_r(v, v') = l(v, v') - [r(v') - r(v)]P$ and so $s_0(v) - r(v)P + l_r(v, v') \leq s_0(v') - r(v')P$ i.e. $s'_n(v) = s_n(v) - r(v)P$ is a valid schedule of G_r. Moreover, $w_{rs'}(e) = w_r(e)P + s'_0(v') - s'_0(v) = w(e) + r(v') - r(v) + s_0(v') - r(v') - s_0(v) + r(v) = w_s(e)$ □

4. PLACEMENT OF STORAGE ELEMENTS

In this section, we show how the register types (edge-triggered or level-sensitive) and placements are obtained from the schedule graph in order to produce a circuit with the right functionality.

4.1 Register placement

Register placement is derived from a schedule graph G_s. In this section it will be assumed that for every vertex $v \in V$, $d(v) \leq P$. This assumption will be removed in section 5. The easy way to place registers is to place them before each operation and activate them according to the given schedule but this results in a waste of space and works only if $w_s(e) \leq P$. Instead of registering every input of every function we shall chain operations and, as a consequence, reduce the number of registers and controlling signals needed. In fact, we only need a register at each P units of time, considering that in the worst case a short path could be of length zero. Therefore, we must break every path, in the graph, which is longer than P; we can put more than one register on an edge. We use a greedy algorithm called `BreakPath` (Fig. 4), not necessarily optimal, for placing the registers. This algorithm is $O(|E|)$:

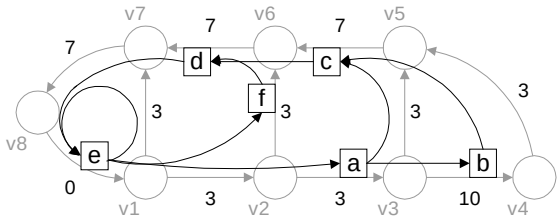
$v.max\_out \equiv \max\{w(e) : v \xrightarrow{e} \cdot\}$
proc `BreakPath`$(v, dist, tim) \equiv$
  if $(v.visited)$ **then exit fi**;
  $v.visited = true$;
  $v.distance = dist$;
  foreach $(v \xrightarrow{e} v')$ **do**
    if $((v'.visited \wedge (dist + w_s(e) > v'.distance))$
      $\vee\ (dist + w_s(e) + v'.max\_out > P))$
      then do
        if $(w_s(e) > P)$
          then put $\lceil w_s(e)/P - 1 \rceil$ registers on edge e
              scheduled at time tim; **fi**
        put one register on edge e
              scheduled at time $(tim + w_s(e)) \bmod P$;
        `BreakPath`$(v', 0, (tim + w_s(e)) \bmod P)$; **od**
    else `BreakPath`$(v', dist + w_s(e), (tim + w_s(e)) \bmod P)$
  fi od.

Fig. 4. Algorithm to place registers

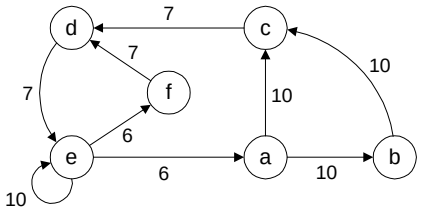
Table 3 gives the register placement and schedule starting `BreakPath` with $(\mathtt{v1}, 0, 0)$. Fig. 5 shows the placement of registers in the final circuit, according to table 3.

Name	*edge*	*Schedule*	*enable time*
a	$1 \rightarrow 3$	6	$[6, 0[$
b	$3 \rightarrow 4$	6	6
c	$5 \rightarrow 6$	6	6
d	$6 \rightarrow 7$	3	$[0, 6[$
e	$8 \rightarrow 1$	0	0
f	$2 \rightarrow 6$	6	$[6, 0[$

Table 3. Register placement and schedule.



a) register placement.



b) delay graph for the placement in a).

Fig. 5. Register placement and delay between them.

4.2 Latch selection

The selection of latch-type registers and their activation time are derived from the delay graph. The idea is the same as above, there must be no path longer than P; that is, there must be no more than P units of time between the enable of a register and the disable of its successors (which is equivalent to the phase non-overlap constraint presented by Sakallah et al. [1990]). Also, a register must be enabled at the time it is scheduled. For example, register a must be enabled at time 6. This means that the maximum time a register can be enabled after (before) its scheduled time is P minus the maximum of the lengths of the edges that go to (exit from) that register. In our example, register e must be enabled exactly at time 0 because there is an edge of length 10 from and to e, so it will be edge-triggered, but register f can stay enabled 4 units of time after 6 and can be enabled 3 units of time before, so it can be level-sensitive, enabled at time 3 and disabled at time $10 \equiv 0 \bmod P$. Table 3 gives a feasible solution for registers' activation time, the intervals being the enable period of the latches (single values are for edge-triggered registers). The implementation of the circuit can be done with a two-phase clock, e and d being clocked by the first phase and a, b, c and f being clocked by the second one; b, c and e are edge-triggered and a, d and f are level-sensitive. This solution has the same period as the one presented by Lockyear and Ebeling [1994], but assuming that the cost of an edge-triggered register is R and that of a latch

is $^{R}/_{2}$, the storage element cost for their circuit is $5R$ whilst ours is $4.5R$. This represents a 10% improvement in the area occupied by the registers. Also, we can use a two-phase clock with an underlap between phases without changing the period.

The algorithm `PlaceLatches` (Fig. 6) checks each register in the delay graph to see if its flip-flops can be replaced by latches and it gives the enable and disable time for each latch.

```
PlaceLatches:
  foreach (v ∈ V) do
    v.max_out = max{w(e) : v -e-> ·}
    v.max_in = max{w(e) : · -e-> v} od
  foreach (v ∈ V) do
    after = P − v.max_in
    before = P − v.max_out
    if ((after ≠ 0) ∨ (before ≠ 0)) then do
      set v as a latch
      v.enable_time = (v.scheduled_time − before) mod P
      v.disable_time = (v.scheduled_time + after) mod P
      foreach (v' -e-> v) v'.max_out = max{v'.max_out, w(e) += after}
      foreach (v -e-> v') v'.max_in = max{v'.max_in, w(e) += before} od
    fi od
```

Fig. 6. Algorithm to place latches.

Each vertex contains its schedule time, which has been given by `BreakPath`. This algorithm is $O(|E|)$. It is an efficient but not necessarily optimal way to place latches and changing the order in which the vertices are processed may give better results.

LEMMA 6. *The circuit resulting from the algorithm PlaceLatches stores valid values in its registers, and will thus have a correct behaviour.*

PROOF. In a delay graph with period P, let B be any register and $A_1, \ldots, A_n$ be all those registers with a vertex to B and let $d_1, \ldots, d_n$ be their respective delays. Register A_i is enabled on the interval $[a_i, b_i[$ and was originally scheduled at time t_i. Register B is enabled on the interval $[a_B, b_B[$ and was originally scheduled at time t_B. An edge-triggered register is modeled with $a_i = b_i$, which means that there is no time where the value passes through the register but we still consider that the input value just before time a_i is stored at time a_i.

We want to prove that if for all i, A_i is valid and stable on time intervals $[t_i, a_i+P[$, then B will be valid and stable on time interval $[t_B, a_B + P[$. This will prove that the latches store valid values if we start with valid values.

By definition, we have that $a_i \leq t_i \leq b_i$ and $a_B \leq t_B \leq b_B$. Also, since there is no path longer than P, $b_B \leq a_i + P$. The validity of the schedule guarantees that $t_i + d_i \leq t_B$, and so $a_i \leq t_i \leq t_i + d_i \leq t_B \leq b_B \leq a_i + P$. Because A_i are valid and stable on time interval $[t_i, t_B[$ (by hypothesis), all inputs of B are valid at time t_B. On time interval $[t_B, b_B[$, B is enabled and its inputs are valid and stable because A_i is valid and stable on time interval $[t_i, b_B[$ (by hypothesis). On time interval $[b_B, a_B + P[$, B is disabled so that it stays stable and was valid at the disable time,

which means that it is still valid. So, register B will be valid and stable on time interval $[t_B, a_B + P[$. □

5. EXTENSIONS

5.1 Functional elements of duration greater than the clock period

Suppose we want to do a scalar product. We can do this using a simple multiply and accumulate circuit as shown in Fig. 7 (additions taking one time unit and multiplications taking two). We should notice that the multiplication (v1) is longer than the period (P). If $d(v) > P$, we have that $s_n(v) + d(v) > s_{n+1}(v)$, which means that we must start the next calculation in vertex v before the current one finishes. There are two ways to accomplish this: pipeline v (Fig. 7b) or put multiple instances of v (Fig. 7c). To be able to pipeline v, we must ask the designer (or a synthesis tool) to split the calculations in v so that each part has a delay $\leq P$.

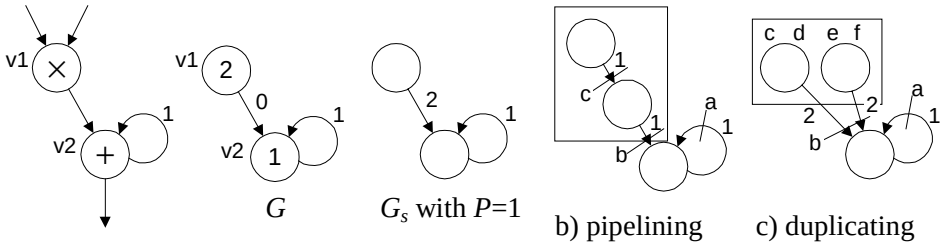


Fig. 7. Scalar product example

To put multiple instances of v, we can modify `BreakPath` so that when $d(v) > P$ it places $\lceil \frac{d(v)}{P} \rceil$ functional elements like v, each one with registered inputs and multiplexing the outputs. At each period, the input registers from the next unit are enabled and the multiplexor chooses the right unit for output. We must adjust $d(v)$ to include the delay of the multiplexor and find a new valid schedule and then restart `BreakPath`. In our example, the input registers and output are scheduled as shown in Table 4.

Cycle	*enable reg.*	*Output*
Even	c, d	e × f
Odd	e, f	c × d

Table 4. Schedule of duplicated unit in the scalar product example.

5.2 Fractional clock duration and k-periodic scheduling

Suppose we want to find an optimal circuit for the circuit graph G in Fig. 8; $P = {}^5/_2$ is fractional and the figure shows the register placement obtained from `BreakPath`. A consistent schedule is to be found in the left part of Table 5. Notice that c_1 and c_2 are but one register c: both have same input and schedule. This first solution may be acceptable but it requires a clock resolution smaller than the time unit used. If all the $d(v)$ are integers then there exists an optimal valid schedule with $s_n(v)$ being all integers. In fact, if all the $d(v)$ are integers, a theorem [Bennour and Aboulhamid

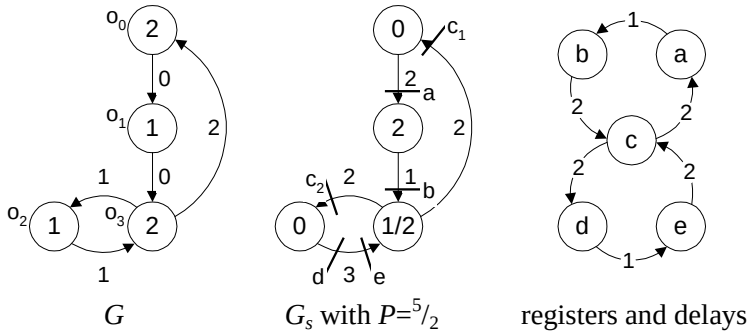


Fig. 8. k-periodic example

1995; Van Dongen et al. 1992] says that if we have a valid periodic schedule s then the schedule s^* defined as $s^*_n(v) = \lfloor s_n(v) \rfloor$ is a valid k-periodic schedule with the same throughput. Applying this result to our first schedule gives the 2-periodic schedule with period $2P = 5$ shown in Table 5: during each period, two output values are calculated, so giving the same throughput as the first schedule.

	periodic ($P = 5/2$)	***k-periodic*** ($k = 2$)		
name	***enable time***	***schedule***$_1$	***schedule***$_2$	***enable times***
a	$[1/2, 2[$	2	4	$[3, 2[$
b	$1/2$	3	0	$\{0, 3\}$
c	$[2, 1/2[$	0	2	$\{0, 2\}$
d	$[1/2, 2[$	2	4	$[3, 2[$
e	$1/2$	3	0	$\{0, 3\}$

Table 5. A schedule for the solution of Fig. 8, and a corresponding k-periodic schedule.

6. EXPERIMENTATION AND IMPLEMENTATION

For our experimentation, we have used a tool that we developed primarily for loop acceleration, called *L.A.* It accepts a description in standard C and produces an internal format where cyclic behaviors are explicit. This intermediate format can be used as input to different algorithms and CAD tools that we intend to develop in the future. To facilitate the development we started from a retargetable C Compiler meant to be modified and retargeted easily [Fraser and Hanson 1995]. The first benchmark is the one presented in [Leiserson and Saxe 1991; Lockyear and Ebeling 1994]. In order to compare our results we also implemented the original retiming method of Leiserson and Saxe [1991]. From Table 6 we see that the acceleration is zero for examples where only one clock phase is needed to have an optimal schedule, but varies from 9% to 100% when more pipelining is possible with multiple phases. We developed the scalar product example and translated from VHDL to C some examples from the HLSynth92 benchmark suite [Benchmarks 1996]. It is interesting to note that the elliptic filter specification in the suite cannot be accelerated, but by re-writing the specification, we obtain an acceleration of 150% using retiming and an additional 9% using our method. The original description used only additions, but after flattening the expressions, we see that the same variable is added multiple times, which could be replaced by a multiplication by a constant. The multiplication

can then be divided in shifts and additions, shifts taking no time as it is only a different interconnection. Then, tree balancing of the expressions was used, to reduce the length of the critical path.

To compare the running time of our algorithm to that of retiming, we ran them on benchmarks from ISCAS89 [Benchmarks 1996]. These circuits are at gate level and have up to 16K gates. We see, in Table 7, that we are around 10 times faster than SIS, except for the larger circuit on which we are 185 times faster. The algorithm used to find maximum throughput has a running time lower than the binary search approach presented in Section 3.1, although the upper bound is not clear.

	Period		***Registers****		***Phases***	***Acceleration***
	Retiming	***L.A.***	***Retiming***	***L.A.***		
correlator	13	10	5	4.5	2	30%
scalar product	2	1	2	4	2	100%
k-periodic	3	$^5/_2$	4	3.5 or 4	2 or 3	20%
diffeq	6	6	7	7	1	0%
ellipf	10	10	13	13	1	0%
modified ellipf	4	$^{11}/_3$	50	25.5	7	9%

Table 6. Benchmarks.
*Register count is the number of edge-triggered plus $^1/_2$ the number of level-sensitive storage.

	Period			***Computation time***	
	Original	***Retiming***	***L.A.***	***Retiming***	***L.A.***
s344	18.8	14.2	14.2	0.8	0.12
s641	30.4	-	-	5.8	0.29
s713	41.6	-	-	7.9	0.38
s1238	28.4	-	-	49.8	1.81
s1423	82.6	73.4	73.4	34.4	2.02
s1488	34.8	32.4	31.9	22.9	3.09
s1494	34.8	32.8	32.8	32.5	4.16
s5378	19.2	14.6	14.6	89.8	15.37
s9234	44.0	-	-	81.7	13.98
s13207	58.4	37.6	36.5	735.8	85.67
s15850	80.4	44.8	44.8	772.4	103.46
s35932	25.0	-	-	3.8 days	30 minutes

Table 7. Computation time (in seconds) of retiming by SIS (retime -ni) vs. scheduling by L.A., on a UltraSPARC-10 with 128MB. A dash says that no optimizations have been made.

7. CONCLUSION AND FUTURE WORK

In this work, we showed that software pipelining techniques are an excellent alternative to retiming techniques in sequential circuit optimization. The resulting circuit has an optimal throughput using multi-phase clocked circuits with a combination of edge-triggered and level sensitive storage. The proved computing complexity is similar to previously published methods but on the benchmarks we are much faster and we have a guarantee of always obtaining the optimal solution regarding the throughput, according to the precision of the graph representation of the circuit. The phases are automatically computed and the registers are placed by a greedy

algorithm. Future work includes the design of an optimal algorithm to maximize chaining and minimize the number of clock phases and of registers. Benchmarks have shown that rewriting of the initial specification using algebraic transformations (like associativity and commutativity) can have a tremendous impact on the final result; we intend to augment our tool using such capabilities. Our work has to be extended to take into account clock skews and to minimize the impact of such phenomena on the overall performances. In addition, the circuit graph could have minimum delays on its edges, which is the time before the output of combinational logic start to change when the inputs are changed. This would allow paths longer than P between registers, which could reduce the number of registers. Tradeoffs between the number of phases, space and throughput have to be explored.

REFERENCES

Benchmarks. 1996. North Carolina State University, Dep. of Comp. Science, Collab. Benchmark Lab., `http://www.cbl.ncsu.edu/benchmarks/Benchmarks-upto-1996.html`.

Bennour, I. E. and Aboulhamid, E. M. 1995. Les problèmes d'ordonnancement cycliques dans la synthèse de systèmes numériques. Technical Report 996 (Oct.), DIRO, Université de Montréal. `http://www.iro.umontreal.ca/~aboulham/pipeline.pdf`.

Deokar, R. B. and Sapatnekar, S. 1995. A fresh look at retiming via clock skew optimization. In *DAC'95* (1995), pp. 304–309.

Van Dongen, V. H., Gao, G. R., and Ning, Q. 1992. A polynomial time method for optimal software pipelining. In *CONPAR'92, Lecture Notes in Computer Sciences, Vol 634* (1992), pp. 613–624.

Fraser, C. W. and Hanson, D. R. 1995. *A Retargetable C Compiler: Design and Implementation*. Benjamin Cummings.

Hanen, C. 1994. Study of a NP-hard cyclic scheduling problem: the recurrent job-shop. *European Journal of Operation Research 72*, 1, 82–101.

Hartmann, M. and Orlin, J. 1993. Finding minimum cost to time ratio cycles with small integral transit times. *Networks; an international journal 23*, 1, 82–101.

Hwang, C.-T., Hsu, Y.-C., and Lin, Y.-L. 1991. Scheduling for functional pipelining and loop winding. In *DAC'91* (1991), pp. 764–769.

Ishii, A. T., Leiserson, C. E., and Papaefthymiou, M. C. 1997. Optimizing two-phase, level-clocked circuitry. *Journal of the ACM 44*, 1 (Jan.), 148–199.

Lawler, E. 1976. *Combinatorial Optimization: Networks and Matroids*. Saunders College Publishing.

Legl, C., Vanbekbergen, P., and Wang, A. 1997. Retiming of edge-triggered circuits with multiple clocks and load enables. In *IWLS'97* (1997).

Leiserson, C. E. and Saxe, J. B. 1991. Retiming synchronous circuitry. *Algorithmica 6*, 1, 3–35.

Lockyear, B. and Ebeling, C. 1994. Optimal retiming of level-clocked circuits using symmetric clock schedules. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 9 (Sept.), 1097–1109.

Maheshwari, N. and Sapatnekar, S. 1997. An improved algorithm for minimum-area retiming. In *DAC'97* (1997), pp. 2–6.

Maheshwari, N. and Sapatnekar, S. 1998. Efficient retiming of large circuits. *IEEE Transactions on VLSI Systems 6*, 1 (March), 74–83.

De Micheli, G. 1994. *Synthesis and optimization of digital circuits*. McGraw-Hill.

Sakallah, K. A., Mudge, T. N., and Oluktun, O. A. 1990. Analysis and design of latch-controlled synchronous digital circuits. In *DAC'90* (1990), pp. 111–117.

CHAPITRE 5

VÉRIFICATION

Après avoir appliqué des transformations pour optimiser un circuit, on voudrait pouvoir prouver que ce circuit fait bien la même chose que le circuit original. Des méthodes efficaces pour vérifier un circuit après la resynchronisation seule ([20]) ou après resynchronisation et resynthèse ([36]) ont été développées, mais elles ne s'appliquent pas aux circuits multi-phase. J'ai donc développé une méthode basée sur celle de Ranjan et al. [36], permettant de vérifier une classe de circuits multi-phase, incluant ceux produits par alternance de resynthèse et de ma méthode pour obtenir le débit maximal. Cette méthode, ainsi qu'une brève revue de certaines autres méthodes, pour comparaison, sont présentées dans [4] (article se trouvant après cette introduction du chapitre).

Les contributions principales de cet article sont (traduction tirée de l'article) :

- La classe des circuits vérifiables est étendue pour couvrir ceux obtenus par des méthodes standard d'ordonnancement, comme celles utilisées en « software-pipelining », au lieu de la resynchronisation. Les circuits obtenus par resynchronisation étant un sous-ensemble strict de la classe qu'on couvre.
- La possibilité de vérifier les circuits multi-phase, en plus des circuits séquentiels à seulement une phase.
- L'absence de contrainte sur la position des registres, ce qui permet plus d'optimisations.

- Le traitement des temps rationnels, en plus des entiers. Ne pas les traiter demande de les convertir en entiers, qui peut ralentir le circuit, ou d'utiliser le déroulage de boucles.

Les pages suivantes contiennent une copie l'article [4], dans son format original (sauf la numérotation des pages) pour *IEEE International Conference on Electronics, Circuits & Systems*, 2000.

Note au lecteur : quelqu'un qui à lu [3] peut sauter la section 2 de l'article.

AN EFFICIENT VERIFICATION METHOD FOR A CLASS OF MULTI-PHASE SEQUENTIAL CIRCUITS

François-R. Boyer[1], El Mostapha Aboulhamid[1], and Yvon Savaria[2]

[1] DIRO, Université de Montréal, 2920 Chemin de la Tour,
C.P. 6128, Succ. Centre-Ville, Montréal, Québec, Canada, H3C 3J7
{boyerf, aboulhamid}@IRO.Umontreal.CA
[2] DGEGI, École Polytechnique de Montréal, Québec, Canada
savaria@VLSI.PolyMtl.CA

ABSTRACT: Currently, many optimizations of sequential circuits, even as simple as retiming, are avoided due to the lack of verification tools that support them. Doing general sequential equivalence to compare the circuits is impractical for circuits of a reasonable size. On the other hand, combinational optimization is part of the design process, because tools and methods are available to ensure correctness and verify combinational circuits. We present a practical method to verify sequential circuits equivalence using combinational equivalence on a transformed circuit of the same size, for a class of circuits. The constraint imposed is that for each loop in the circuit, there must be a point in both circuits that are in correspondence. The circuits can have a different number of clock phases, and they can be transformed by other scheduling algorithms than retiming and multi-phase retiming.

1. INTRODUCTION

Synthesis and optimization of sequential circuits derive an implementation by a transformation of the initial specification, this transformation has to be verified to assert the conformance of the implementation to the specification. If no information is known about the transformation, a general sequential equivalence method must be used, and this can be impractical for circuits of a reasonable size. This means that transformations must have some known properties to be able to verify them in a practical time. On the other hand, to allow the best optimization we would like to impose as few constraints as possible on these transformations.

Currently, many optimizations of sequential circuits, even as simple as retiming, are avoided due to the lack of verification tools that support them. On the other hand, combinational optimization is part of the design process, because tools and methods are available to ensure correctness and verify combinational circuits.

There are many proposed solutions to the sequential equivalence problem. Some methods try to solve the general problem [1][2], but their complexity limits the size of circuits they can process. Some other methods will perform verification with constraints on the transformations, to have a smaller complexity, but they do not permit to verify multi-phases circuits, as those produced by the methods presented in [3] and [4].

The method presented by Ashar [5] exposes some flip-flops in the original circuit, to make the FSM representing the circuit complete-1-distinguishable, before the optimizations are applied. Then, to compare the two FSMs, instead of checking the reachable states in the product machine (in $O(2^{n+m})$), it checks the reachable states of individual machines (in $O(2^n+2^m)$), where m and n are the number of flip-flops in both FSMs. This is still impractical.

Huang [6] uses an approach based on ATPG, reducing the search space to a practical size by finding equivalence between internal signal pairs. Their goal was to prove equivalence after retiming only, so the signal pairs are easy to find, but it may not handle circuits after resynthesis.

Bischoff [7] presents a method composed of three conservative algorithms. First, the circuit is cut into slices manually, then, on each slice, the algorithms are used from the fastest to the slowest until a proof, or a counter proof, is found. The simplest algorithm is normal combinational equivalence checking, which does not allow any sequential elements. The second algorithm uses Timed Ternary BDD to represent function of inputs delayed by some signal events. It allows latches with any function as clock input, but feedback loops are not supported. In addition, different signals are considered independent, so it cannot be used for multi-phases circuits, as most of these circuits are incorrect under those assumptions. If the first two methods fail, general FSM equivalence is used.

Stoffel [2] simplifies the general FSM equivalence using a decomposition of the circuit to simplify the reachable states function. As this decomposition can take a long time, in practice, an approximation is used which will give a superset of the reachable states. Checking with this superset can lead to false negatives.

Aït Mohamed [1] applies model checking to compare circuits. The idea is to use MDGs instead of ROBDDs, which permits symbols and uninterpreted functions instead of converting everything to binary representation. This may help to avoid the state explosion problem, but if the number of variables is large, or if the design is at gate level and cannot be

extracted at a higher level of abstraction, the method can still take an excessive processing time.

Ranjan et al. [8] presented a verification technique which permits retiming combined with combinational optimization sequential synthesis for a class of circuits. In particular, they require certain constraints to be met on the feedback paths of the latches involved in the retiming process. For a general circuit, these constraints can be satisfied by forcing some latches to be immovable. Equivalence checking after performing repeated retiming and resynthesis on this class of circuit reduces to a combinational verification problem. This paper shows that this class can be extended and that no latches have to be immovable. The proposed method is mainly an extension of [8], with the following contributions:

- The class of circuits is extended to cover those obtained by general scheduling methods, as those used in software pipelining, instead of retiming. The class of retimed circuits being a strict subset of the class we cover.
- The method permits to verify multi-phase circuits, as opposed to single phased sequential circuits.
- The new method does not require any constraints on the position of registers, thus allowing more optimization.
- Rational time can be handled instead of integer time. Not handling it would require that schedules with rational times be converted to integer time, which may slow the circuit or demand to use loop unrolling.

However, some point must be observable for each loop in the circuit graph. That is, the correspondence between a signal in the original circuit and in the circuit to verify must be known at one point in each sequential loop of the circuit. If the method is applied more than once, we should keep a set of correspondence points between successive iterations, otherwise we may be unable to complete the verification. Nevertheless, there may not be any known correspondence between the original circuit and the final one.

Section 2 introduces the circuits and the class of transformations we want to process. Section 3 presents the method for single-phase circuits, proposed by Ranjan et al. [8]. A similar method is then developed for multi-phase circuits in section 4, and a method to consider gate delays, which needs more attention when there is more than one clock phase, is developed in section 5. A complete example of our verification method applied to a multi-phase circuit is presented in section 6.

2. PRELIMINARIES

A circuit is modeled as a set of gates (any calculation element without memory) interconnected by wires that can have memory elements on them. The memory elements are edge-triggered flip-flops and are often called registers. All registers have clocks with the same period, which is considered to be the time unit.

2.1 Combinational Optimization

Between registers, the combinational circuit can be changed for an equivalent one. This transformation does not change the state space, but changes the topology of the circuit. In addition, the intermediate results are not the same, and the signals may not be equivalent anymore, except for those going in and out of registers.

2.2 Retiming

For circuits with a single clock, the retiming technique, presented by Leiserson and Saxe [9], permits to move registers.

Retiming can be seen as a displacement assigned to each gate, which removes registers from one side and puts them on the other. As shown in Fig. 1, retiming of +1 for some gate means "remove one register of each output of the gate and add one to each of its inputs". This transformation does not change the behavior of the circuit, but it changes the time at which calculations are made, moving them from one clock cycle to the other. This also changes the state space, which makes some verification methods unusable.

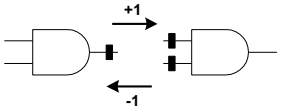


Fig. 1. Primitive retiming operations.

2.3 Fractional "Retiming" and Software Pipelining

It is not possible to move a fraction of register, but we can have multiple clock phases and apply a retiming of one phase instead of a whole cycle [4]. As an alternative, we can place registers and then activate them at the right time using different phases [3]. These methods permit, in some cases, to have circuits with higher throughput than with optimal retiming.

The method we presented in [3] uses modulo scheduling, known in software pipelining, to find an optimal schedule for the operations in the circuit. As this schedule can have fractional time, multiple clock phases are used to implement it. The method is illustrated using the simple circuit of Fig. 2.

The maximal throughput (minimal period) is found using a known algorithm to solve minimal cost-to-time ratio cycle problem, the cost being the number of registers and the time being the combinational delays. That cycle is shown in gray on Fig. 2, and has a throughput of $^1/_{10}$ which gives a period of 10. Note that the best possible period that can be obtained by retiming for this example is 13, which is the optimal period if a single-phase single clocked circuit implementation is targeted.

Then, a valid schedule with that period (10 in our example) is found using Bellman-Ford's longest path algorithm, where the weights on the edges are the delay

of the source vertex minus the number of registers times the period ($w_l(e_{ij}) = d(v_i) - P\, w(e_{ij})$), as shown on Fig. 3. The schedule of a vertex relative to the origin used for the longest path is the length of the path. The As Late As Possible and As Soon As Possible schedules relative to v1 are in Table 1.

To place back registers, and find their clock phase, we use a graph for which the weights on the edges are the time between the start of an operation and the start of the operation that needs its result. More formally, the weight $w_s(e_{ij}) = s(v_j) - s(v_i)$, where $s(v_i)$ is the schedule of vertex v_i. To minimize the number of registers, they are only placed to cut any path that is longer than the period. We give a linear time method that provides good register placement, although the number of registers is not necessarily minimal.

The register placement is shown on Fig. 4. The phase is the distance of the register from the reference point, divided by the period. The phase relative to register 'e' is the fraction besides each register on Fig. 4.

An attractive feature of this method is that it permits to place registers almost anywhere without changing the speed of the circuit (if we neglect the propagation delay through latches). In fact, in the context where clock skews and registers delays are not yet considered, registers can always be added, but, of course, they can't always be removed. The added registers will be used only during the proof, so they can be considered perfect. If real registers are to be added, more precautions should be taken. To add a register, you simply place it on the input of an operation. Then, to find the phase, you add to the phase of some register the length of the path from that register to this new one, divided by the period, and take only the fractional part of the result. For example, in Fig. 4, to place a register between v3 and v5, you place it just before v5 and the distance from 'a' being 3, the period being 10 and the phase of 'a' being $^6/_{10}$, the resulting phase will be $^9/_{10}$.

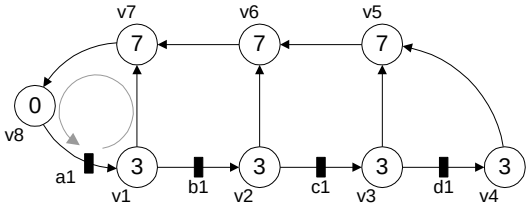


Fig. 2. A simple circuit. Delays of "gates" are shown on them.

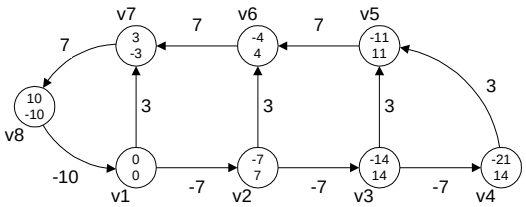


Fig. 3. Longest paths from/to v1 which are the ASAP and (minus) ALAP schedules.

Table 1. Schedules and mobility relative to v1.

Vertex	1	2	3	4	5	6	7	8
ASAP	0	-7	-14	-21	-11	-4	3	10
ALAP	0	-7	-14	-14	-11	-4	3	10
Mobility	0	0	0	7	0	0	0	0
Interval	0	-7	-14	[-21,-14]	-11	-4	3	10

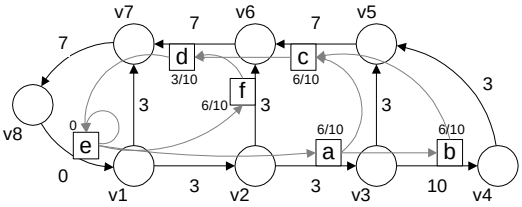


Fig. 4. Register placement for the schedule, with their clock phase.

3. SINGLE-PHASE CIRCUITS

For circuits where all registers are controlled by a single clock, an efficient verification method has been presented by Ranjan [8]. That method permits to verify equivalence with the original circuit after optimization by retiming and resynthesis (combinational optimization). In this section, we summarize the principal concepts presented in [8], which will then be extended to multi-phase circuits in the next section.

The idea is to first transform the circuit to an acyclic representation, then to apply the optimizations on that representation. Finally, to verify the equivalence, both the original and the optimized circuit, in acyclic representation, are transformed to combinational representations and are compared by a combinational equivalence checker.

3.1 Acyclic Sequential to Combinational

In a circuit without loop, the outputs depend only on the inputs. However, because the circuit can contain registers, the output will depend on the inputs at multiple, but finite, different moments. As there is only one clock phase, the time is an integer (the number of cycles).

Clocked Boolean Functions (CBF)

The CBF is a combinational representation of a sequential circuit. It gives the expression for the output in function of time.

For a signal s, the CBF $s(t)$ at time t is :

- If s is the output of a gate G with inputs $y_1 ... y_n$:
  $$s(t) = f_G(y_1(t), ..., y_n(t))$$
- If s is the output of a register : $s(t) = y(t-1)$
- If s is a primary input : $s(t)$ is independent of $s(t')$ for $t \neq t'$

For example, to get the CBF of the circuit in Fig. 5, we proceed as follows. The CBF of each part is:

$$o(t) = c(t) \cdot d(t)$$
$$d(t) = c(t-1)$$
$$c(t) = b(t) \oplus a(t)$$
$$b(t) = a(t-1)$$

Then we substitute everything in $o(t)$ to have $o(t) = (a(t–1) \oplus a(t)) \cdot (a(t–2) \oplus a(t–1))$.
The CBF gives the behavior of the circuit at steady-state, which is when all registers have correct values (after the initialization phase).

THEOREM 1. Canonicity of CBF. [8]
If C_1 and C_2 are acyclic sequential circuits, where all registers are activated at the same time, and F_1 and F_2 are their CBFs, then we have that $F_1 \equiv F_2 \Leftrightarrow C_1 \equiv C_2$. The equivalence between F_1 and F_2 being combinational while the equivalence between C_1 and C_2 is at steady-state.

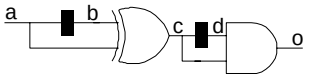


Fig. 5. An acyclic circuit.
Its CBF is $o(t) = (a(t–1) \oplus a(t)) \cdot (a(t–2) \oplus a(t–1))$.

3.2 Circuits with loops

When a circuit contains feedback loops, they must be broken to make the circuit acyclic before its CBF can be computed. The method does not accept purely combinational loops, so there is at least one register in every loop. A certain number of these registers are chosen as cut points, and made observable. The outputs of the chosen registers are now considered as primary inputs to the circuit. The cut points must be the same in the original circuit and the optimized one, so the cut is done before applying optimization transformations. This will put constraints on the possible transformations, as we cannot move a register used as a cut point. Therefore, we may want to use a minimal cut, to reduce the constraints. Once the acyclic version of the circuit has been optimized and verified (with the CBFs), it is "glued back" to have an optimized and verified cyclic circuit.

4. MULTI-PHASE CIRCUITS

Our objective is to verify sequential circuits with multiple clocks, but all having the same period. Some reference time is arbitrarily fixed (often the activation time of some register) and each register has an activation time, in the clock period, relative to that reference. Without loss of generality, the clock period is considered to be the time unit, so the activation time, which is the clock phase, will be a fraction in the interval [0,1[. For example, a register can be activated at time 0 and another at time $^1/_2$. Because events are periodic, and the period is 1, the register activated at time $^1/_2$ is also activated at time 1 $^1/_2$, and at time 2 $^1/_2$...
Registers are modeled as instantaneous, that is, if a register is activated at time 4, it's content will have the old value until time 4, excluded, and the new value from time 4, included. The new value being the value of its input just before the activation time. Note that using the period as the time unit will make it easier to compare different implementations with different period lengths.

4.1 Acyclic Sequential to Combinational

Using the same idea as for single-phase circuits, output signals can be defined as functions of time, but here the time will be a rational number instead of an integer.

Fractional CBF
From the register model described above, a register that has y as input, and that is activated by the clock phase ϕ, will have as output:

$$s(t) = y(\lfloor t - \phi \rfloor + \phi - \varepsilon)$$

This means that $s(t) = y(t - \varepsilon)$ if the fractional part of t is ϕ, and it will keep that value for a whole cycle, until $t + 1$, where it will take the new input value. The value ε can be seen as a positive rational smaller than the time resolution in the system under consideration. In other words for every possible time t in the system $\lfloor t–\varepsilon \rfloor = \lceil t \rceil – 1$.
For the combinational elements, it is exactly as in the single-phase CBFs described in section 3.1. For inputs, it is slightly different, because the phase at which the input changes must be known. Like for the registers, the inputs are changing only once per cycle, at a specified phase. The following can be said of an input s which changes on phase ϕ:

$$s(t) = s(\lfloor t - \phi \rfloor + \phi)$$

and $s(t)$ is independent of $s(t')$ for $\lfloor t - \phi \rfloor \neq \lfloor t' - \phi \rfloor$

Simplification of Floor Operators
If the construction just described is applied blindly, there will be many floor operators everywhere in the expression.
Here are formally defined rules sufficient to simplify the generated expressions. The *Simp* function does the simplification using the following rules:

$Simp(\lfloor \lfloor x \rfloor + y \rfloor) = Simp(\lfloor x \rfloor) + Simp(\lfloor y \rfloor)$	
$Simp(\lfloor -\varepsilon \rfloor) = -1$	
$Simp(\lfloor x–\varepsilon \rfloor) = \lceil x \rceil – 1$	When x is a constant.
$Simp(f) = Map(Simp, f)$	For each argument of a function.

These simple rules will be used to simplify the description of a circuit. At the end of the description of both circuits (original and transformed one) the expressions at the cut points will be single-phased. The problem is then completely reduced to comparison using integer time, which is the work presented in [8].

4.2 Circuits with Loops

As with single-phase circuits, loops are broken to have an acyclic circuit before its CBF is computed. However, with multi-phase circuits, it is possible to cut anywhere, if verification in multiple steps is permitted. Having multi-phase circuits, as those presented in [3], makes it possible to add registers anywhere, as explained in section 2.3. This permits to add a register where a cut is requested, and then prove locally that this

does not change the behavior of the circuit. The cut points can then be chosen anywhere, but we may want to minimize the number of cuts, to minimize constraints on the transformations, or choose some place that seems to be optimized as much as we can.

Local CBF and Local Proofs

To do local proofs, instead of making the CBFs going from outputs to the primary inputs of the circuit, a CBF is made from a register to its predecessors. A register *a* is considered the predecessor of *b* if there exists a path from *a* to *b* without going through a register. The local CBF of each register can be found even if there are loops in the circuit, as there are no combinational loops. To prove that adding, or removing, a register does not change the behavior of the circuit, the local CBFs can be used. The local CBF for the concerned register and the local CBFs of all registers of which it is the predecessor are found, when the register is there and when it is removed. Then we prove that the CBFs are equivalent.

Multiple Steps Proof

Performing proofs in multiple steps, instead of a single global proof, gives a better choice for cut points. The circuit can be cut anywhere and then optimized without necessarily having a register on the cut point, neither in the original circuit nor in the optimized one. In that case, no register has a fixed position. This gives a greater freedom for optimization, and may result in better circuits. In addition, single-phase circuits can also be considered as multi-phase circuits, to be able to do local proofs at the beginning and at the end of the whole proof, even if the final circuit has to be single-phased. Local proofs at the beginning show that adding registers at the cut points does not change the behavior of the circuit, while the proofs at the end are there to show that removing those registers is also correct. In certain cases, depending on the optimizations done, the added registers cannot be removed and must be kept in the final circuit.

5. CONSIDERING GATE DELAYS

All the discussions so far assume that the clock period gives enough time for combinational circuits to compute the next values of registers. Of course, a complete proof should also check the delays and we present the simple delay model we are using.

For a combinational circuit that computes $s = F(y_1, y_2,...)$. If the short path takes at least *min* units of time, and the long path takes less than *max* units of time, *s* can be given in function of time as follows:

$$s(t) = \begin{cases} F(y_1(t-min), y_2(t-min),...) & \text{If } y_1... \text{ are steady on }]t-max, t-min]. \\ \text{Undefined} & \text{Otherwise.} \end{cases}$$

With a delay model where $min = 0$, we can say that $s(t) = F(y_1(t), y_2(t),...)$ if inputs are steady from time $t-max$. That is exactly the $s(t)$ which we used in the CBFs, so it is only required to prove, for each local CBF, that the inputs are steady for at least *max* units of time before the clock phase. This also shows that our CBFs are not always correct in a delay model where $min > 0$. That is, it may not be able to prove equivalence of wave-pipelined circuits. The general case $s(t)$, shown above, must be used in the CBFs if those circuits are to be verified.

6. EXAMPLE OF PROOF

We will verify that the optimized circuit of Fig. 4 is equivalent to the original circuit of Fig. 2. In the original circuit, the four registers (a1, b1, c1 and d1) are all activated by the same clock, and after optimization, the circuit has six registers and three phases (0, $^3/_{10}$ and $^6/_{10}$). For this proof, we will consider that each function is different and unknown. The functions will be named by the vertex name, possibly followed by a letter, to indicate outgoing edge to which it is associated, and then an 'f'. The output of a vertex is noted as its function but without the 'f' and different caps are used to distinguish circuits. The output of a register is simply noted by the name of the register. Using this notation, and simplifications rules from section 4.1, we used Mathematica to find the CBF for each circuit, using register a1/e as cut point. Then we asked Mathematica to prove that they are equivalent.

The Mathematica session is presented on Fig. 6. First, the rules are entered as is. Note that the rules are not added to the "Simplify" of Mathematica, so that only the entered rules, and constants evaluation, will be used to simplify the expressions. Instead of redefining the equality, to be able to compare primary inputs according to the rule, we add a simplification rule, for each primary inputs, that maps equal inputs to the same expression. Therefore, simplification applied to a primary input will give the expression at the time it last changed. The rules for combinational elements, registers and primary inputs used for the description of the multi-phase circuit are from Section 4.1, and those for the single-phase circuit are from Section 3.1.

Now we want to prove that if 'a1' and 'e' have the same value then the next value of 'a1' will also be the same as the next value of 'e'. To do that, the expressions are compared, but we must ensure to compare using the right times. As the time unit is the period of the circuit, the time is already scaled for the comparison, but there may be a phase offset. After simplification, the expression for 'e' has only one phase, as it is the output of a single register, but for the expression to be equal to that of 'a1' we must ensure that they are compared at their according phases. Once they are in phase, any combinational equivalence tools, as those used by [8], can be used to compare the expressions. Here we use Mathematica to compare the next values assuming that they are currently equal.

7. CONCLUSION

We have shown that it is possible to verify the equivalence between multi-phase circuits in a reasonable amount of time, if some points are kept observable. These points, on the other hand, do not have to be on registers if we accept to have proofs in multiple steps. All the proofs that are done are equivalence between two combinational expressions, which is much faster, in general, than equivalence proofs on sequential circuits. Since the final expressions contain only integer time, we have reduced our problem to the verification problem resolved in [8]. From that, we deduce that the experimental results will give the same execution time as in [8].

Therefore, the presented method permits more optimizations, with multi-phase circuits, and these circuits are still verifiable.

To have better optimizations, a resynthesis method specially designed for the multi-phases circuits should be developed. In addition, some work could be done on how the circuit should be cut to have the desired freedom for optimization.

8. REFERENCES

[1] O. Aït Mohamed, E. Cerny, and X. Song, "MDG-based Verification by Retiming and Combinational Transformations", *IEEE Great Lakes Symposium on VLSI*, Lafayette, Louisiana, Feb. 19-21, 1998.

[2] D. Stoffel, and W. Kunz, "Record & Play: A Structural Fixed Point Iteration for Sequential Circuit Verification", *IEEE/ACM Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 9-13, 1997.

[3] F. R. Boyer, E. M. Aboulhamid, Y. Savaria, and I. E. Bennour, "Optimal design of synchronous circuits using software pipelining techniques", *IEEE Int. Conf. Computer Design*, Austin, TX, Oct. 5-7, 1998, pp. 62-67.

[4] A. T. Ishii, C. E. Leiserson, and M. C. Papaefthymiou, "Optimizing two-phase, level-clocked circuitry", *Journal of the ACM*, vol. 44, no. 1, Jan. 1997, pp. 148-199.

[5] P. Ashar, A. Gupta, and S. Malik, "Using Complete-1-Distinguishability for FSM Equivalence Checking", *IEEE/ACM Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 10-14, 1996.

[6] S. Y. Huang, K. T. Cheng, and K. C. Chen, "On Verifying the Correctness of Retimed Circuits", *IEEE Great Lakes Symposium on VLSI*, Ames, Iowa, Mar. 22-23, 1996, pp. 277-280.

[7] G. P. Bischoff, K. S. Brace, S. Jain, and R. Razdan, "Formal Implementation Verification of the Bus Interface Unit for the Alpha 21264 Microprocessor", *IEEE/ACM Int. Conf. Computer Design*, Austin, TX, Oct. 12-15, 1997.

[8] R. K. Ranjan, V. Singhal, F. Somenzi, and R. K. Brayton, "Using Combinational Verification for Sequential Circuits", *Design Automation and Test in Europe*, Munich, Germany, Mar. 9-12, 1999.

[9] C. E. Leiserson, and J. B. Saxe, "Retiming synchronous circuitry", *Algorithmica*, vol. 6, no.1, 1991, pp. 3-35.


```
Simpl[⌊x_⌋ + y_] := Simpl[⌊x⌋] + Simpl[⌊y⌋];
Simpl[⌊-ϵ⌋] := -1;
Simpl[⌊x_ - ϵ⌋] := ⌈x⌉ - 1;
Simpl[f_] := Map[Simpl, f];

PrimaryInput[in_, ϕ_] := (Simpl[in[t_]] := in[Simpl[⌊t - ϕ⌋ + ϕ]]);
Register[in_, time_, ϕ_] := in[⌊time - ϕ⌋ + ϕ - ϵ];

PrimaryInput[e, 0];
v1h[t_] := v1hf[e[t]];   v1d[t_] := v1df[e[t]];
v2h[t_] := v2hf[v1d[t]]; v2d[t_] := v2df[v1d[t]];
v3h[t_] := v3hf[a[t]];   v3d[t_] := v3df[a[t]];
v4h[t_] := v4hf[b[t]];
v5[t_] := v5f[v3h[t], v4h[t]];
v6[t_] := v6f[f[t], c[t]];
v7[t_] := v7f[v1h[t], d[t]];
a[t_] := Register[v2d, t, 6/10];
b[t_] := Register[v3d, t, 6/10];
c[t_] := Register[v5, t, 6/10];
d[t_] := Register[v6, t, 3/10];
f[t_] := Register[v2h, t, 6/10];
nexte[t_] := Register[v7, t, 0];

Simpl[nexte[t]] // TraditionalForm

v7f(v1hf(e(⌊t⌋ − 1)), v6f(v2hf(v1df(e(⌊t⌋ − 2))),
   v5f(v3hf(v2df(v1df(e(⌊t⌋ − 3)))), v4hf(v3df(v2df(v1df(e(⌊t⌋ − 4))))))))

V1h[t_] := v1hf[a1[t]];   V1d[t_] := v1df[a1[t]];
V2h[t_] := v2hf[b1[t]];   V2d[t_] := v2df[b1[t]];
V3h[t_] := v3hf[c1[t]];   V3d[t_] := v3df[c1[t]];
V4h[t_] := v4hf[d1[t]];
V5[t_] := v5f[V3h[t], V4h[t]];
V6[t_] := v6f[V2h[t], V5[t]];
V7[t_] := v7f[V1h[t], V6[t]];
b1[t_] := V1d[t - 1];
c1[t_] := V2d[t - 1];
d1[t_] := V3d[t - 1];
nexta1[t_] := V7[t - 1];

nexta1[⌊t⌋] // TraditionalForm

v7f(v1hf(a1(⌊t⌋ − 1)), v6f(v2hf(v1df(a1(⌊t⌋ − 2))),
   v5f(v3hf(v2df(v1df(a1(⌊t⌋ − 3)))), v4hf(v3df(v2df(v1df(a1(⌊t⌋ − 4))))))))

Simpl[nexte[t + ϕ]] == nexta1[⌊t⌋] /. {e → a1, ϕ → 0}

True
```

Fig. 6. Mathematica session of the proof: returns True.

CHAPITRE 6

TOLÉRANCE AU « CLOCK-SKEW »

Quand on dessine un circuit, en général on suppose que l'horloge arrive correctement en même temps à tout ce qui en a besoin. Comme le réseau de distribution de l'horloge est grand, en réalité l'horloge arrive à des temps qui dépendent non seulement de la topologie du réseau, qui est connue avant la fabrication, mais aussi des variations dans la fabrication, de la température de l'environnement et des variations sur la source de courant. Il faut donc adapter les circuits pour qu'ils soient tolérants aux imprécisions de l'horloge.

Les méthodes antérieures tentaient d'augmenter la tolérance en utilisant soit la resynchronisation, soit en modifiant les délais sur l'horloge, soit avec une combinaison des deux. La méthode présentée dans [16] fait les deux en même temps de manière à trouver la solution optimale. Malheureusement cette méthode nécessite la résolution d'un problème de programmation mixte entier et linéaire qui n'a pas de solution connue en temps polynomial. Un circuit d'assez petite taille, comme le s713, passe un peu plus de 13 heures dans l'algorithme.

En utilisant un algorithme de placement de registres semblable à celui que je présente dans [3], on peut avoir un circuit avec la tolérance voulue en un temps linéaire en le nombre d'arcs. Cette nouvelle méthode de placement de registres, présentée dans [5] (article se trouvant après cette introduction du chapitre), permet d'avoir un résultat rapidement et elle permet de dépasser de beaucoup l'optimum de tolérance trouvé par la méthode de [16] dans certains cas. Ce dépassement de l'optimum vient, encore une fois,

du fait qu'on utilise plusieurs phases d'horloge, mais aussi du fait que tous les registres sont actifs sur les niveaux d'horloge plutôt que sur les fronts.

Un résultat important de cet article est la borne supérieure théorique pour la tolérance, lorsque le circuit tourne au débit maximal, qui est $(P - \max_{v \in V} \{d(v) - d_{min}(v)\})/4$. Beaucoup des méthodes précédentes ne permettent aucune tolérance lorsque le circuit tourne au débit maximal.

Les contributions principales de cet article sont (traduction tirée de l'article) :

- Permet une tolérance au « clock-skew » et au « clock jitter » même si le circuit tourne à la période d'horloge optimale. La méthode de [16] doit baisser le débit du circuit pour augmenter la tolérance.
- Permet plus de tolérance que n'importe quelle méthode basée sur la resynchronisation et le « clock skew scheduling ».
- Démontre l'existence d'un compromis entre le nombre de registres (complexité du matériel) et la tolérance.
- La méthode a un temps d'exécution polynomial, et les résultats expérimentaux montre qu'elle est très rapide.

Les pages suivantes contiennent une copie l'article [5], dans son format original (sauf la numérotation des pages) qui a été soumis à *European Conference on Circuit Theory and Design*, Espoo, Finlande, 28–31 août 2001.

Note au lecteur : quelqu'un qui a lu [3] peut sauter la section 2 de l'article.

Minimizing Sensitivity to Clock Skew Variations Using Level Sensitive Latches

François-R. Boyer*, El Mostapha Aboulhamid*, and Yvon Savaria†

***Abstract* — We propose a method for improving the tolerance of synchronous circuits to delay variations on the clock distribution. Instead of retiming and clock skew scheduling applied to edge-triggered flip-flops, as used by most other methods, we use level-sensitive latches placed based on a schedule of the operations. The resulting circuit can have a non-zero tolerance even at the optimal clock period, which is impossible with edge-triggered flip-flops.**

1 Introduction

As clock frequencies get higher and the clock distribution network gets larger, the fluctuation on the arrival time of the clock signals gets important compared to the clock period. Clock distribution networks can be made more precise [4], but they are not perfect. For a circuit to work properly at high speed, it must be designed to have tolerance to delay fluctuations.

Previous methods trying to maximize the tolerance used mainly retiming and clock skew scheduling [3]. Each of these approaches used separately will not guarantee maximum tolerance, and combining them is not trivial. The method in [5] finds an optimal combination of these two, using a mixed-integer linear program, for which there is no known polynomial time algorithm. Also, only single-phase edge-triggered circuits are supported. We will show that the optimal solution they find is no more optimal if level-sensitive latches and multiple phases are permitted.

We present a new way to have a higher tolerance to the timing variations on the clock, placing level-sensitive latches according to a schedule found by a software-pipelining technique. This is an extension to what we presented in [1], where the goal was to achieve the optimal clock period, but where clocks were supposed to be perfect.

The main contributions of this work are:

- It permits tolerance to clock skew and clock jitter while running at optimal clock period. Method [5] trades clock frequency for higher tolerance.
- It permits more tolerance than any method based on retiming and clock skew scheduling.
- It demonstrates the existence of a tradeoff between register count (hardware complexity) and tolerance.
- The method has a polynomial execution time, and it is very fast as illustrated by the experimental results.

Combining small time complexity and possibility of tradeoff between tolerance and hardware complexity permits an exploration of the design space, where a satisfying tolerance, not necessarily optimal, is obtained, while keeping the register count within an acceptable limit.

* DIRO, Université de Montréal, Québec, Canada.
E-mail: {boyerf, aboulhamid}@IRO.UMontreal.CA.
† DGEGI, École Polytechnique de Montréal, Québec, Canada.
E-mail: savaria@VLSI.PolyMtl.CA.

2 Preliminaries

2.1 Input Circuit Definition

As most methods do, we support single-phase edge-triggered circuits formed by combinational computing elements separated by registers, similarly to the original retiming article [7]. With the reference model, circuits are represented as a finite, vertex-weighted, edge-weighted, directed multigraph $G = \langle V, E, d, w \rangle$. The vertices V represent the functional elements of the circuit, and they are interconnected by edges E. Each vertex $v \in V$ has a propagation delay $d(v) \in \mathbb{Q}$, which is the maximum delay before its outputs stabilize. Each edge $e \in E$ is weighted with a register count $w(e) \in \mathbb{N}$, representing the number of registers separating two functional elements. All registers are edge-triggered and controlled by the same clock.

We extend the d and w functions for paths in the graph. For any path $v \overset{p}{\rightsquigarrow} v_k = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \dots \xrightarrow{e_{k-1}} v_k$, we define:

$$d(p) = \sum_{i=0}^{k-1} d(v_i) \qquad w(p) = \sum_{i=0}^{k-1} w(e_i)$$

For registers a on an input arc of operation v, and b on an input arc of v', we define $a \overset{p}{\rightsquigarrow} b$ as $v \overset{p}{\rightsquigarrow} v'$.

We also support lower bounds on the delay of an element, which requires another vertex-weight $d_{\min}(v) \in \mathbb{Q}$. To simplify figures and readability, the minimum delay is considered zero in the examples.

2.2 Scheduling and software-pipelining

A *schedule* s is a function $s : \mathbb{N} \times V \to \mathbb{Q}$, where $s_n(v) \equiv s(n, v)$ denotes the time at which the n^{th} iteration of operation v is starting. A schedule s is said to be *periodic* with period P (all iterations having the same schedule), if:

$$\forall n,\ \forall v \in V,\ s_{n+1}(v) = s_n(v) + P$$

A schedule is valid iff the operations terminate before their results are needed (whilst respecting resource constraints if any). If the only constraints come from data dependency, s is *valid* iff for all edges $v \xrightarrow{e} v'$,

$$s_n(v) + d(v) \le s_{n+w(e)}(v').$$

A periodic schedule that satisfies those constraints can be found in $O(|V|\,|E|)$ using a longest path algo-

rithm like Bellman-Ford, for any valid period. The problem of finding the minimal period is the same as the well known minimal cost-to-time ratio cycle problem, the cost being the number of registers and the time being the combinational delays, which can be solved in $O(|V| |E| \log(|V| d_{max}))$ using Lawler's algorithm [6]. A cycle is called *critical* if its ratio is the same as the minimum ratio, and a path is critical if all its edges are on a critical cycle. Note that the optimal period is the inverse of the minimum ratio.

For example, the circuit of Fig. 1 (the same as in [7]) has a clock period of 24. Using retiming techniques we can obtain a single phase clocked circuit with a period of 13. An optimal period of (3+7)/1=10 can be achieved. This minimal period is determined by the critical cycle shown with the gray arrow [1]. Meeting this performance necessitates a multi-phase clock; and a feasible register placement is shown in Fig. 2. On non-critical edges, more than one register may be required to respect the desired maximum distance between registers; this can increase tolerance to timing variations as will be explained later.

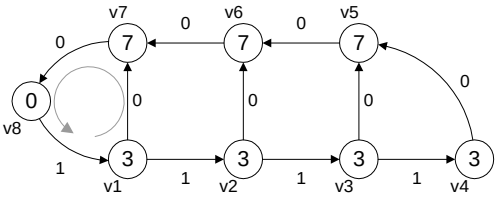


Fig. 1. A simple circuit with delays on vertices and register count on edges. A critical cycle is shown in gray.

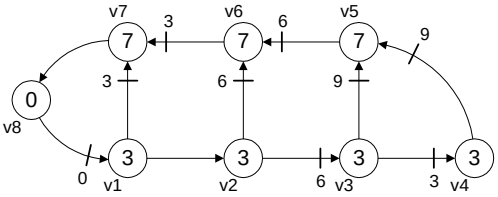


Fig. 2. A possible way to place registers in Fig. 1, with a period of 10. The phases of the registers are indicated beside them.

3 Clock and Register Constraints

We consider that registers can be activated at any time using multiple clock phases. Each register is activated at a specific phase of the clock, according to the schedule, all clock signals having the same period. This permits to follow any periodic schedule.

3.1 Edge-triggered Registers

In a circuit with edge-triggered registers, a register must not be activated before the combinational circuit preceding it has stabilized. That is, a register placed on an input of operation v must be activated following a valid schedule for v. There is another constraint, as the circuit calculates values at each cycle. All registers must be activated before the result start to change for the next result. Therefore, for each path $a \overset{p}{\rightsquigarrow} b$, between registers a, and b, with no other register between them, we have:

$$s_n(a) + d(p) \le s_{n+k}(b) \le s_{n+1}(a) + d_{min}(p)$$

where k is 0 or 1 if b depends on a from the same or from the previous iteration, respectively.

This forces to place at least one register on each path p longer than the period $P + d_{min}(p)$, for periodic schedules. Algorithm `BreakPath` [1] resolves this problem when d_{min} is considered to be zero.

If there is no slack in the schedule, the time at which registers are activated must be exact, as any deviation will make the schedule invalid. A circuit using edge-triggered registers on a critical path has zero tolerance to schedule variations when running at the optimal clock period. The method in [5] finds the maximum tolerance for single-phase edge-triggered circuits with a relaxed period, if only retiming and clock skew scheduling is permitted.

For example, on the circuit of Fig. 3 (containing only a critical cycle of Fig. 1) at the optimal period of 10, registers can be placed at either a or b, or both a and b using two clocks as shown on Fig. 4.

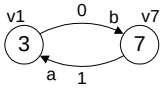


Fig. 3. The critical cycle from Fig. 1, with the two possible positions for registers (a, b).

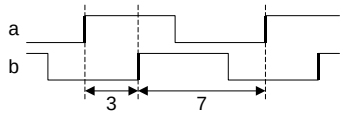


Fig. 4. Clocks when both a and b are edge-triggered registers in Fig. 3.

3.2 Level-sensitive Registers

With level-sensitive registers, the constraints seem similar, but are in fact more flexible, as will be shown in the following. A value must not be latched (by the disabling edge of the clock) before it has stabilized, but the register can be enabled some time before (by the enabling edge of the clock). To correctly follow the schedule of operations, and not delay them, a latch placed on an input of operation v must be enabled before the schedule of v. The register must still be enabled at the schedule of v and must be disabled before the value starts to change for the next one. Then it must stay disabled long enough for a valid value to propagate to other registers. If we have a valid schedule for an edge-triggered circuit, we can use level-sensitive latches if the following constraints can be met with a non-null enabling period (the enable time different from the disable time, for each register). For each path $a \overset{p}{\rightsquigarrow} b$, between registers a and b, with no other register between them,

$$s_{n+k}(b_e) \le s_{n+k}(b) \le s_{n+k}(b_d) \le s_{n+1}(a_e) + d_{min}(p),$$

where a_e and b_e are the enabling time of the latches, and a_d and b_d is the disabling time of the latches; the other values are as in the edge-triggered case.

Here we consider the value of register a to be valid at time $s_n(a)$, even if it is disabled only at a later time: $s_n(a_d)$. As the original edge-triggered schedule was valid, we can prove that this is true [1].

Even if there is no slack in the schedule of operations, there may be slack on the enable and disable time of registers. Since no clock is sent according to the schedule of operations, small errors in clocks arrival times may be tolerated.

With the same example as in the edge-triggered case (Fig. 3), using level sensitive latches permits to have imperfect clocks even at the optimal period, as shown on Fig. 5.

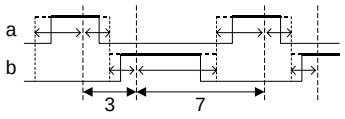


Fig. 5. Clocks when level-sensitive registers are at both a and b in Fig. 3. All clock edges can move as indicated by the small arrows, without changing the circuit's behavior.

4 Maximum Tolerance to Clock Variations

This section analyses how much error can be tolerated on the enable time and the disable time of a register. We say that a circuit has a tolerance of δ if all clock edges can be randomly moved by $\pm\delta$ time units from their nominal values, without changing the result produced by the circuit. Note that in [5], the tolerance τ is defined as the width of the interval, thus $\tau = 2\delta$.

4.1 Tolerance at Optimum Clock Period

THEOREM 1. At optimum clock period, the maximum tolerance for registers a and b on a critical path, without registers between them, is $(P - (s_{n+k}(b) - s_n(a)) + d_{min}(a \rightsquigarrow b))/4$, where $d_{min}(a \rightsquigarrow b)$ is the minimum $d_{min}(p)$ for all paths $a \overset{p}{\rightsquigarrow} b$.

PROOF. On a critical path, when the circuit runs at maximum frequency, the schedule must be followed exactly. From previous section, we must satisfy the following constraint for all paths $a \overset{p}{\rightsquigarrow} b$:

$$s_{n+k}(b_e) \le s_{n+k}(b) \le s_{n+k}(b_d) \le s_{n+1}(a_e) + d_{min}(p)$$

The lower bound on $d_{min}(p)$ is $d_{min}(a \rightsquigarrow b)$, from the definition, and as the schedule in periodic $s_{n+1}(a_e) = s_n(a_e) + P$ (if there is no error on the clock). So we have that:

$$s_{n+k}(b_e) \le s_{n+k}(b) \le s_{n+k}(b_d) \le s_n(a_e) + P + d_{min}(a \rightsquigarrow b)$$

To have a tolerance of δ, we must be able to move all clock edges by that value and still satisfy the constraint. We want to maximize δ under:

$$s_{n+k}(b_e) + \delta \le s_{n+k}(b) \le s_{n+k}(b_d) - \delta$$
$$s_{n+k}(b_d) + \delta \le s_n(a_e) + P + d_{min}(a \rightsquigarrow b) - \delta$$
$$s_{n+1}(a_e) + \delta \le s_{n+1}(a)$$

We obtain the maximum δ when the inequations are at equality. Putting them together, we get:

$$(s_{n+k}(b) + \delta) + \delta = (s_n(a) - \delta) + P + d_{min}(a \rightsquigarrow b) - \delta$$
$$\Leftrightarrow \delta = (P - (s_{n+k}(b) - s_n(a)) + d_{min}(a \rightsquigarrow b))/4 \quad \square$$

The tolerance to clock edge deviations depends on the distance between registers; a shorter time between them gives a higher tolerance. The maximum tolerance will be obtained when registers are placed on all edges in the graph.

LEMMA 1. At optimum clock period, the maximum tolerance on critical paths is $(P - \max_{v \in V} \{ d(v) - d_{min}(v) \})/4$.

PROOF. To minimize the distance between registers and have a valid schedule, the distance will be the delay of one operation. LEMMA 1 follows directly from THEOREM 1, as the longest operation will determine the maximum distance between registers. $\square$

LEMMA 2. At optimum clock period, the required number of registers on each cycle of the graph will be multiplied by at least $1/(1 - 4\,\delta/P)$ if all paths are critical and d_{min} is zero.

PROOF. From THEOREM 1, $\delta = (P - dist)/4$, where $dist$ is the distance between registers with a path between them. In the optimum case, the registers will have equal distances; a cycle containing m registers will have a length of $m \cdot dist$. P is optimal and all paths are critical, so for any cycle $P = m \cdot dist/n$, where n is the original number of registers. We obtain $\delta = (P - P \cdot n/m)/4$, which means that $m/n = 1/(1 - 4\,\delta/P)$. $\square$

For our example of Fig. 3, the maximum tolerance is $(10 - 7)/4 = ¾$, if d_{min} is considered zero. Looking at Fig. 5, we see that to maximize the length of the four arrows in the section lasting 3 time units, each arrow must be one fourth of the 3 time units. This is also the maximum tolerance for the circuit of Fig. 2, as the longest delay between registers is the same.

To show how more registers will give more tolerance, we will use the circuit of Fig. 6. The circuit has an optimal period of 16 and would have no tolerance at that speed if one edge-triggered register were used. If level-sensitive registers are placed at a and c, a tolerance of $(16 - 8)/4 = 2$ can be achieved, and if registers are added at b and d, that tolerance can go up to $(16 - 4)/4 = 3$. In that later case, the clocks will be as shown in Fig. 7.

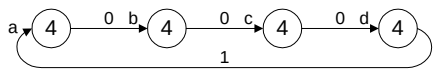


Fig. 6. Simple loop circuit to show possible tolerance.

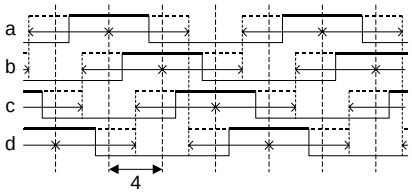


Fig. 7. Clocks with highest tolerance for circuit Fig. 6 with period 16.

4.2 Tolerance with a Relaxed Period

If we give some slack time to the circuit, using a longer period or on non-critical paths, the circuit could have a higher tolerance. As a small error on $s_n(a)$ can be accepted, the tolerance could be higher than half the width of the enabling interval (between a_e and a_d) of the register. Of course we must still guarantee that $s_n(a_e) \le s_n(a_d)$, which is not a problem because whatever the variations on the delays are, the transitions on a wire will always keep the same order. We do not currently have a method to optimize tolerance on non-critical paths, they are optimized as if they were also critical, which may underestimate the actual tolerance.

5 Experimental Results

We have implemented a method to place registers based on THEOREM 1. The distance between registers must be bounded by a value lower than the optimal period in order to achieve a non-zero tolerance. It has been applied to some .edif circuits from LGSynth93 benchmark suite [8]. To compare the results with those presented in [5], we tried to use the same circuits and delay model, but their delays are a bit randomized. Gate delays are $a + b\cdot(fanout \pm ½)$, where a and b depends on the gate type and come from the library iwls93.mis2lib. The library gives a and b for the rise and fall time, for each case we add or subtract ½ from the number of connections to the output of the gate (*fanout*), then the highest and lowest values of the four calculated delays are used as maximum and minimum delays, respectively.

In the first part of Table 1, we obtain the same tolerance as in [5] without deviating from optimal clock period. A zero tolerance would be obtained with the method [5]. The last two circuits are there to show that our method is fast even on larger circuits. The columns $|V|$ and $|E|$ give the size of the circuit graph in number of vertices and edges, P is the optimal period and the period at which we want to run the final circuit. The targeted tolerance is δ, and the register count in the resulting circuit with that tolerance is ×reg times that of the original circuit (not the one at optimal period). The time taken for the optimization and register placement, in seconds on a PII 450MHz, is shown in the column CPU (s). The time for the placement of registers to achieve a specified tolerance is about 12% of that total time. *To compare with the time taken by the method in [5], the circuits s713 and dk512 took 13.6 hours and 21.5 hours, respectively, also on a PII (as stated in their paper).* Our method is fast but does not currently minimize the number of registers, and it does not exploit the slack on some paths.

The number of registers to achieve a certain tolerance is shown on Fig. 8. Both axis have been normalized: the tolerance is the fraction of the period and the register count is the factor by which the number of registers increased compared to the original circuit. The solid line shows the theoretical minimum number from LEMMA 2. The dots are the values obtained by trying different tolerances on the benchmark circuits.

6 Conclusion and Future Work

We showed that a circuit with a tolerance higher than that of any edge-triggered circuit could be obtained using level-sensitive latches activated by different clock phases. The tolerance cannot be over ¼ the period, when running at the optimal period, but can be non-zero, which is impossible with edge-triggered circuits. A higher tolerance requires more registers, which leads to possible tradeoffs. Currently the algorithm does not minimize the number of registers and does not exploit slack times on non-critical path, or when running the circuit at a relaxed clock period. We plan to develop an algorithm that gives higher tolerance with fewer registers, using those slack times.

Circuit	$\|V\|$	$\|E\|$	P	δ	×reg	CPU (s)
bbtas	35	75	4.63	0.315	2	0
dk14	73	199	6.16	0.7	1.25	0
dk512	44	122	4.83	0.35	3.2	0.016
s208	43	93	4.22	0.835	28.17	0.016
s713	397	575	47.34	1.89	2.05	0.125
s9234.1	2931	4057	30.25	2	2.10	0.750
s38417	23985	33248	27.78	2	2.36	8.703

Table 1. Tolerance (δ) and register increase (×reg) for some circuits.

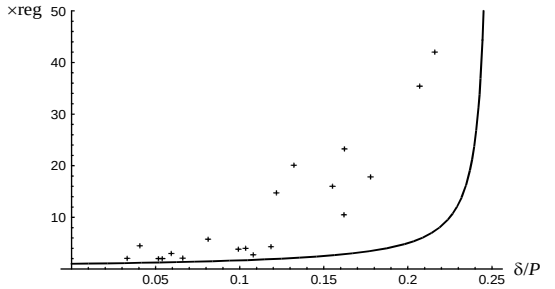


Fig. 8. Register count for different tolerance from 0 to ¼ the period. Solid line is the bound from LEMMA 2; dots are actual results.

References

[1] F. R. Boyer, E. M. Aboulhamid, Y. Savaria, and M. Boyer, "Optimal design of synchronous circuits using software pipelining techniques," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 2, April 2002, in press, available on www.acm.org.

[2] F. R. Boyer, E. M. Aboulhamid, and Y. Savaria, "An efficient verification method for a class of multi-phase sequential circuits," *IEEE International Conference on Electronics, Circuits & Systems*, December 2000, in press.

[3] J. P. Fishburn, "Clock skew optimization," *IEEE Transactions on Computers*, vol. 39, pp. 945-951, July 1990.

[4] E. G. Friedman, "*Clock distribution networks in VLSI circuits and systems*," IEEE press, 1995.

[5] E. G. Friedman, X. Liu, and M. C. Papaefthymiou, "Minimizing sensitivity to delay variations in high-performance synchronous circuits," *Proceedings of Design Automation and Test in Europe*, 1999, pp. 643-649.

[6] E. Lawler, "*Combinatorial Optimization: Networks and Matroids,*" Saunders College Publishing, 1976.

[7] C. E. Leiserson, and J. B. Saxe, "Retiming synchronous circuitry. *Algorithmica*," vol. 6, no. 1, 1991, pp. 3-35.

[8] http://cbl.ncsu.edu/CBL_Docs/lgs93.html

CHAPITRE 7

DÉVELOPPEMENTS POSSIBLES

Cette section contient des points qui ont étés travaillés mais sur lesquels aucun article n'a été écrit. Ce sont donc des voies ouvertes avec quelques résultats, pour la continuité des recherches sur le sujet.

7.1 Accélération de la resynchronisation

J'ai effectué sur des circuits de grande taille des tests comparant la resynchronisation faite par SIS avec ma méthode (version optimisée de [3]). Sur ces circuits, les plus gros de ISCAS98, ma méthode est beaucoup plus rapide. Pour le circuit s35932 par exemple, le résultat est obtenu en moins de deux secondes, comparé aux 3,8 jours que SIS prend pour la resynchronisation. Comme la resynchronisation semble donner des débits très près de l'optimal que ma méthode donne, si le circuit à optimiser est au niveau des portes logiques (il est montré dans [14] que la différence n'est pas plus que le délai d'une porte), il semble que dans beaucoup de cas on aimerait mieux avoir un circuit à une seule phase. Un circuit à une phase est plus simple à implanter et tous les outils actuels permettent de travailler sur ces circuits. Il serait donc intéressant de développer une méthode pour trouver un ordonnancement nécessitant une seule phase, mais beaucoup plus rapidement que la resynchronisation. La recherche dichotomique que la resynchronisation fait parmi les $|V|^2$ valeurs obtenues en trouvant tous les plus longs chemins est très longue. En fait, il semble qu'une seule itération de cette recherche soit souvent plus longue que ma méthode au complet. L'accélération obtenue risque d'être semblable à ce qui est dans

[14] (regarder la Table 1 de l'article), et il y a d'autres méthodes comme celle de [31], alors ceci a été mis de côté.

7.2 Resynthèse

Les méthodes de resynthèse sont problématiques à cause des registres dans le circuit. L'optimisation se fait donc seulement entre les registres, en commençant par déplacer les registres à l'extérieur des parties qui semblent être à optimiser. Le délai entre les registres est diminué le plus possible, en supposant que ça va diminuer aussi le délai maximal après avoir fait une resynchronisation pour replacer les registres. Avec la méthode que j'ai présentée dans [3], il n'y a pas de registre dans le circuit durant la phase d'ordonnancement. Ce n'est qu'à la fin qu'on place les registres, et ce placement n'a aucun impact sur le débit du circuit. Aussi, avec cette technique, on sait quels arcs sont critiques et lesquels ne le sont pas, alors il est plus facile d'identifier les parties importantes à optimiser, et de déplacer des sommets à l'extérieur du chemin critique.

Après avoir fait une transformation, un test simple permet de vérifier si le débit du circuit a augmenté. Il suffit de calculer les plus longs chemins à partir d'un sommet sur un chemin critique. S'ils n'existent pas c'est que le débit a diminué, si le chemin du sommet à lui-même est zéro le débit est resté le même, sinon on peut augmenter le débit. Il est donc assez facile d'utiliser une méthode du genre « hill climbing » pour trouver un optimum local.

Lemme 1. On peut utiliser le graphe $\langle V, E, d, w_s/P\rangle$ (les w_s étant les poids dans le graphe d'ordonnancement) comme entrée à l'algorithme d'ordonnancement optimal.

Preuve. Le graphe d'ordonnancement G_s est une resynchronisation de $\langle V, E, d, P\,w\rangle$ (le lemme 2 de [3]), par conséquent le graphe $\langle V, E, d, w_s/P\rangle$ est une resynchronisation du graphe G du circuit initial. Comme appliquer une resynchronisation sur le graphe G ne pose aucun problème (le lemme 5 de [3]), ce graphe est aussi valide comme entrée. Par contre, on ne peut plus considérer que w est un nombre de registres, puisqu'il n'est pas nécessairement entier. □

7.3 « Wave-pipelining »

Le nouvel algorithme de placement de registres, qui permet la tolérance au « clock-skew » ([5]), utilise le fait que le délai minimum d'un élément de calcul n'est pas nécessairement zéro, et calcule le chemin court. Plutôt que de placer les registres pour qu'il n'y ait aucun chemin plus long que P dans G_s, on les place pour qu'il n'y ait aucun chemin p de longueur supérieure à $P + d_{min}(p)$. Cet algorithme fait donc du « wave-pipelining », en permettant un temps plus grand que P entre l'activation de registres qui se suivent. Par contre pour avoir un meilleur effet de « wave-pipelinig » il faut tenter de maximiser $d_{min}(p)$, comme mentionné dans la section 3.3, et ce n'est pas présentement implanté.

Pour maximiser $d_{min}(p)$ et minimiser $d(p)$ sur un chemin critique, pour avoir un circuit rapide, on peut utiliser une méthode de resynthèse. Par contre, sur un chemin que n'est pas critique, on peut se permettre d'augmenter $d(p)$ pour maximiser $d_{min}(p)$ sans diminuer le débit du circuit. On peut donc ajouter des éléments de délai qui ne font qu'augmenter $d(p)$ et $d_{min}(p)$, dans les cas où c'est plus profitable que d'ajouter un registre.

7.4 Horloge à période variable

L'horloge à période variable est un concept qui semble totalement nouveau et qui pourrait avoir une grande influence sur notre manière de voir les circuits synchrones (contrôlés par une horloge) par rapport aux circuits asynchrones (sans horloge). Je présente cette idée qui peut paraître bizarre.

Pour obtenir les meilleures performances possibles lors de la synthèse de haut niveau, l'ordonnancement ne devrait pas être contraint par la période d'horloge. Park et Choi [34] et moi-même [3] décrivons de telles méthodes qui ne regardent que le délai du circuit pour faire l'ordonnancement, puis qui tentent d'ajuster une horloge au résultat. La méthode d'ajustement de l'horloge est très différente dans ces deux articles. J'utilise des horloges multi-phases, une méthode directe pour permettre la resynchronisation fractionnaire. Park et Choi font de l'ordonnancement acyclique avec contraintes de ressources, et la méthode évidente pour ajuster l'horloge est alors d'utiliser une horloge

rapide et de faire les opérations en plusieurs cycles. Cette dernière méthode introduit des pertes de temps dans les cas où la période d'horloge n'est pas un diviseur de tous les temps de l'ordonnancement. Par contre, le plus grand commun diviseur est habituellement une période trop courte, alors Park et Choi donnent une méthode pour minimiser les pertes en respectant une borne supérieure sur la fréquence d'horloge.

Je présente une nouvelle méthode, qui permet de n'avoir aucune perte de temps tout en ayant une seule horloge qui contrôle le circuit. Si on généralise la méthode que je présente dans [3] à des ordonnancements non périodiques, au lieu d'utiliser une horloge à haute fréquence, on utilise une horloge à période variable, dont la période est ajustée par le contrôleur pour que la distance entre les opérations soit toujours de un « cycle ». Comme la période est variable, les « cycles » ont des durées différentes, et notre ordonnancement est respecté. Cette technique permet aussi de faire de l'ordonnancement à l'exécution, qui peut changer selon l'évaluation d'expressions conditionnelles. Ceci permet de faire l'équivalent des circuits asynchrones « self-timed » mais avec un circuit synchrone, en autant qu'on ait une évaluation du temps que prennent les opérations.

Les principales contributions qu'apporterait un tel article sont :

- On permet une meilleure résolution des temps d'ordonnancement, avec une horloge plus lente.
- La machine à états contrôlant le circuit n'a qu'un état par étape de contrôle (pas besoin de compteur opérant à haute vitesse).
- Les temps des périodes peuvent être changés dépendant des résultats précédents, pour faire des circuits rapides ayant des expressions conditionnelles.

Avec la technologie actuelle CMOS 0.25µm (qui est même déjà remplacé par le 0.18µm), la résolution est autour de 50ps ([8]). La même résolution demande une horloge jusqu'à 20GHz si la période est fixe, mais c'est impossible de faire tourner le circuit de contrôle à une telle fréquence d'horloge avec cette même technologie.

Si on pousse encore plus loin l'idée, on pourrait avoir des tables pour la longueur des opérations en fonction de divers paramètres, comme la température du circuit par exemple (puisque plus la température est élevée, plus un circuit est lent).

CHAPITRE 8

CONCLUSION

Mes contributions se sont effectuées sur trois grandes lignes. La première présente une nouvelle manière de voir l'ordonnancement des opérations dans un circuit et le placement des registres. Cette approche permet d'ajouter autant de registres qu'on le désire, n'importe où dans un circuit, sans en changer le comportement (possiblement en augmentant la période, considérant qu'un registre n'a pas réellement un délai nul). Ces registres peuvent être activés sur les fronts ou les niveaux, mais doivent être contrôlés par une certaine phase bien calculée de l'horloge. Cette possibilité d'ajouter des registres à n'importe quel endroit est l'un des pré-requis nécessaires pour les deux autres grandes lignes : la preuve d'équivalence, et la tolérance aux imprécisions des temps d'arrivés de l'horloge.

Nous avons ensuite élargi la possibilité de prouver l'équivalence de deux circuits aux circuits générés par notre méthode d'optimisation, ce qui nous a permis de vérifier le bon placement des registres, et ce qui enlève une barrière pour l'utilisation industrielle de la méthode. Une méthode ne sera pas réellement utilisée tant qu'on ne peut pas montrer par une méthode indépendante que le résultat est correct. On peut assez facilement prouver qu'une certaine méthode est correcte, mais il est beaucoup plus difficile de prouver que l'implantation est correcte. Si la méthode de preuve est implantée de manière indépendante, les chances qu'elle se trompe sur les mêmes cas que la méthode d'optimisation sont très faibles. Cette étape à aussi été importante pour se convaincre nous-mêmes que nous n'avions pas fait d'erreur dans nos affirmations sur le placement des registres.

Dans un monde réel l'horloge n'est pas parfaite comme le considérait la méthode de placement de registres. Il était donc important de voir si notre technique pouvait vraiment s'appliquer dans ce contexte imparfait. Les résultats sont très positifs, non seulement nous pouvons tolérer des variations sur les temps d'arrivées de l'horloge plus grandes qu'avec les autres méthodes, mais aussi notre algorithme trouvant le circuit avec une certaine tolérance est beaucoup plus rapide.

Plusieurs autres développements intéressants sont possibles à partir de ces idées. Quelques-uns ont déjà été discutés dans le chapitre 7. La resynthèse et le « wave-pipelining » automatique seraient des points à développer puisqu'un bon placement de registre n'est pas nécessairement suffisant pour obtenir les performances voulues. De plus, l'horloge à période variable, qui est une généralisation des nos horloges multi-phase, pourrait offrir une nouvelle flexibilité aux circuits.

Un autre développement possible, pour supporter les expressions conditionnelles, n'est que mentionné dans la section 7.4. La méthode d'ordonnancement utilisée pour générer nos circuits suppose que chacune des unités fonctionne à tous les cycles, et ne supporte donc pas directement l'exécution conditionnelle. On peut assez facilement faire un ordonnancement en pire cas, mais si on veut un meilleur ordonnancement, il faut qu'il soit dynamique (qu'il change selon les données). Il faudrait donc aussi regarder cette avenue.

CHAPITRE 9

BIBLIOGRAPHIE

[1] V. H. Allan, R. B. Jones, R. M. Lee, et S. J. Allan. Software Pipelining. *ACM Computing Surveys*, Vol. 27, No. 4, Dec. 1995, pp. 367–437.

[2] I. E. Bennour, et E. M. Aboulhamid. Les problèmes d'ordonnancement cycliques dans la synthèse des systèmes numériques. Université de Montréal, Montréal, Publication 996, Oct. 1995. http://www.iro.umontreal.ca/~aboulham/pipeline.pdf

[3] F. R. Boyer, E. M. Aboulhamid, Y. Savaria, et M. Boyer. Optimal design of synchronous circuits using software pipelining techniques. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 7, No. 2, Apr. 2002.

[4] F. R. Boyer, E. M. Aboulhamid, et Y. Savaria. An Efficient Verification Method for a Class of Multi-phase Sequential Circuits. *IEEE International Conference on Electronics, Circuits & Systems*, 2000.

[5] F. R. Boyer, E. M. Aboulhamid, et Y. Savaria. Minimizing Sensitivity to Clock Skew Variations Using Level Sensitive Latches. Soumis à *European Conference on Circuit Theory and Design*, Espoo, Finlande, 28–31 août 2001, 4 pages.

[6] W. P. Burleson, M. Ciesielski, F. Klass, et W. Liu. Wave-Pipelining: A Tutorial Research Survey. *IEEE Transactions of VLSI Systems*, Vol. 6, No. 3, Sept. 1998.

[7] S. M. Burns. Performance analysis and optimization of asynchronous circuits. Ph.D. thesis, California Institute of Technology, 1991.

[8] D. E. Calbaze et Y. Savaria. New trends in Direct Digital Frequency Synthesis Design.

[9] P.-Y. Calland, A. Darte, et Y. Robert. Circuit Retiming Applied to Decomposed Software Pipelining. *IEEE Transactions of Parallel and Distributed Systems*, Vol. 9, No. 1, Jan. 1998.

[10] L.-F. Chao and E. H.-M. Sha. Scheduling Data-Flow Graphs via Retiming and Unfolding. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 12, Dec. 1997, pp. 1259–1267.

[11] J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. McGettrick, et J.-P. Quadrat. Numerical computation of spectral elements in max-plus algebra. In *Proc. IFAC Conference on System Structure and Control*, 1998.

[12] A. Dasdan et R. K. Gupta. Faster Maximum and Minimum Mean Cycle Algorithms for System-Performance Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 17, No. 10, 1998, pp. 889–899.

[13] A. Dasdan, S. S. Irani, et R. K. Gupta. Efficient Algorithms for Optimum Cycle Mean and Optimum Cost to Time Ratio Problems. *DAC'99*, 1999.

[14] R. B. Deokar et S. Sapatnekar. A fresh look at retiming via clock skew optimization. *DAC'95*, 1995, pp. 304–309.

[15] V. Van Dongen, G. R. Gao, et Q. Ning. A polynomial time method for optimal software pipelining. *CONPAR'92, Lectures Notes in Computer Sciences*, Vol. 634, 1992, pp. 613–624.

[16] E. G. Friedman, X. Liu, et M. C. Papaefthymiou. Minimizing Sensitivity to Delay Variations in High-Performance Synchronous Circuits. *Proceedings of Design Automation and Test in Europe*, 1999, pp. 643–649.

[17] C. Hanen. Study of a NP-hard cyclic scheduling problem: the recurrent job-shop. *European Journal of Operation Research*, Vol. 72, No. 1, 1994, pp. 82–101.

[18] M. Hartmann et J. Orlin. Finding minimum cost to time ratio cycles with small integral transit times. *Networks; an international journal*, Vol. 23, 1993, pp. 567–574.

[19] Y.-c. Hsu, S. Sun, et D.H.C. Du. Finding The Longest Path in Cyclic Combinational Circuits. *IEEE International Conference on Computer Design*, 1998, pp. 530–535.

[20] S. Y. Huang, K. T. Cheng, et K. C. Chen. On Verifying the Correctness of Retimed Circuits. *Proceedings of The Great Lakes Symposium on VLSI*, 1996, pp. 277–280.

[21] A. T. Ishii, C. E. Leiserson, et M. C. Papaefthymiou. Optimizing two-phase, level-clocked circuitry. *Journal of the ACM*, Vol. 44, No. 1, January 1997, pp. 148–199.

[22] P. Kalla et M. J. Ciesielski. Performance Driven Resynthesis by Exploiting Retiming-Induced State Register Equivalence. *DATE'99*, 1999.

[23] R. M. Karp et J. B. Orlin. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete Appl. Math.*, Vol. 3, pp. 37–45, 1981.

[24] Y. Kukimoto et R. K. Brayton. Exact Required Time Analysis via False Path Detection. *DAC'97*, 1997.

[25] Y. Kukimoto et R. K. Brayton. Delay Characterization of Combinational Modules. *IWLS'98*, 1998, pp. 446–451.

[26] Y. Kukimoto et R. K. Brayton. Hierarchical Functional Timing Analysis. *DAC'98*, 1998.

[27] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Saunders College Publishing, 1976.

[28] C. E. Leiserson et J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, Vol. 6, No.1, 1991, pp. 3–35.

[29] B. Lockyear et C. Ebeling. Optimal retiming of level-clocked circuits using symmetric clock schedules. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 9, September 1994, pp. 1097–1109.

[30] N. Maheshwari et S. Sapatnekar. An improved algorithm for minimum-area retiming. *DAC'97*, 1997, pp. 2–6.

[31] N. Maheshwari et S. Sapatnekar. Efficient retiming of large circuits. *IEEE Transactions on VLSI Systems*, Vol. 6, No. 1, March 1998, pp. 74–83.

[32] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[33] P. Pan. Performance-driven Integration of Retiming and Resynthesis. *DAC'99*, 1999.

[34] S. Park et K. Choi. Performance-Driven Scheduling with Bit-Level Chaining. *DAC'99*, 1999.

[35] R. K. Ranjan, V. Singhal, F. Somenzi, et R. K. Brayton. On the Optimization Power of Retiming and Resynthesis Transformations. *Proceeding of IEEE/ACM International Conference on Computer-Aided Design*, 1998.

[36] R. K. Ranjan, V. Singhal, F. Somenzi, et R. K. Brayton. Using Combinational Verification for Sequential Circuits. *Proceeding of Design Automation and Test in Europe*, 1999.

[37] H. Zhou, V. Singhal, et A. Aziz. How Powerful is Retiming? *IWLS'98*, 1998.