UNIWERSYTET ZIELONOGÓRSKI

WYDZIAŁ ELEKTROTECHNIKI, INFORMATYKI I TELEKOMUNIKACJI

Rozprawa doktorska

mgr inż. Radosław Czarnecki

Kosynteza dynamicznie samorekonfigurowalnych systemów wbudowanych

Promotor:

Dr hab. inż. Stanisław Deniziak, prof. nadzw. PK

ZIELONA GÓRA, 2008

Podziękowanie

Dziękuję Panu dr hab. Stanisławowi Deniziakowi, prof. nadzw. PK, promotorowi tej pracy, za wiele cennych wskazówek i wsparcie przy jej realizacji.

Autor

SPIS TREŚCI

WYKAZ OZNACZEŃ I SKRÓTÓW

α	– stopień wykorzystania powierzchni układu FPGA
c_i	– warunek przypisany do krawędzi warunkowej
C_{ijk}	– proces komunikacyjny (transmisja od zadania v_i do v_j)
CL	– łącze komunikacyjne (ang. Communication Link)
CM_j	– parametr „Memory Load”
CTG	– warunkowy graf zadań
d_{ijk}	– waga krawędzi
DAG	– skierowany i acykliczny graf zadań
ΔE	– miara jakości rozwiązania (globalny współczynnik zysku)
$\Delta\varepsilon$	– lokalny współczynnik zysku
Δt_{zs}	– zysk skuteczności dynamicznej rekonfiguracji prowadzącej do przyspieszenia projektowanego systemu
E_i	– zbiór krawędzi odpowiadających połączeniom komunikacyjnym
E_c	– zbiór krawędzi warunkowych
e_{cij}	– krawędź warunkowa
E_s	– zbiór krawędzi prostych
e_{ij}	– krawędź prosta
GPP	– procesor uniwersalny (ang. General Purpose Processor)
$GPPr$	– procesor wbudowany zajmujący się rekonfiguracją sektorów
λ	– szybkość systemu SOPC
$level(v_j)$	– poziom zadania v_j w hierarchii kolejnych zagnieżdżonych S_w
N	– zbiór wszystkich zadań w grafie
n	– liczba wszystkich węzłów w grafie
PE	– element obliczeniowy (ang. Processing Element)
R	– zbiór komponentów w bibliotece komponentów
r	– liczba wszystkich komponentów w bibliotece komponentów
rr_i	– sygnał rekonfiguracji
RS	– rekonfigurowalny sektor
S	– całkowita powierzchnia systemu SOPC
SK	– ścieżka krytyczna
S_W	– ścieżka warunkowa
SC_j	– powierzchnia zajmowana przez łącze komunikacyjne
Su_j	– powierzchnia GPP lub VC
T_{ij}	– zadanie – skończony zbiór obliczeń

$t_j(v_i)$	– czas wykonania zadania
$t_k(v_i,v_j)$	– czas transmisji przez łącze komunikacyjne
t_{res}	– czas rekonfiguracji sektora
TG, G=(V,E,T)	– graf zadań
τ	– okres czasu, w którym muszą być wykonane wszystkie zadania grafu
V_i	– zbiór węzłów odpowiadających zadaniom
v_{ij}	– węzeł odpowiadający zadaniu T_{ij}
v_{fork}	– węzeł rozwidlający
v_{join}	– węzeł łączący
VC	– wirtualny komponent (ang. Virtual Component)
ZWW	– zadania wzajemnie się wykluczające

SOC	– ang. System on a Chip
SOPC	– ang. System on a Programmable Chip
DRSOPC	– ang. Dynamically Reconfigurable System on a Programmable Chip
SRSOPC	– ang. Self-Reconfigurable System on a Programmable Chip
COSYSOPC	– algorytm kosyntezy systemów SOPC
COSEDYRES	– algorytm kosyntezy systemów DRSOPC/SRSOPC reprezentowanych przez grafy TG
COSEDYRES-CTG	– algorytm kosyntezy systemów DRSOPC/SRSOPC reprezentowanych przez grafy CTG

1 WSTĘP

Szybki rozwój technologii produkcji układów cyfrowych dostarcza nowych możliwości implementacji wbudowanych systemów komputerowych (ang. embedded systems). Funkcje systemu wbudowanego można zaimplementować w formie programu. Zaletą wykorzystania procesorów jest niski koszt takiego rozwiązania. Jednak ze względu na sekwencyjność wykonywania obliczeń przez procesory czas wykonania programu może być zbyt duży. Alternatywnym rozwiązaniem jest zastosowanie wyspecjalizowanych modułów sprzętowych. Funkcje systemu zaimplementowane w układzie/układach ASIC (ang. Application-Specific Integrated Circuit) charakteryzują się dużą szybkością, ale koszt projektowania jest bardzo wysoki. Ponadto funkcjonalność, w przeciwieństwie do procesorów, nie może być zmieniona, a proces wdrożenia jest czasochłonny. W celu znalezienia kompromisu pomiędzy kosztem a szybkością najczęściej systemy wbudowane implementowane są w postaci heterogenicznej[1]. Stosowane są również rozwiązania alternatywne, takie jak konfigurowalne procesory [T07], czy technologia FPGA (ang. Field-Programmable Gate Arrays).

Programowalne układy ***FPGA*** stanowią rozwiązanie pośrednie pomiędzy wyspecjalizowanymi układami ASIC a softwarowymi systemami procesorowymi. Układy te stały się interesujące dla projektantów systemów wbudowanych ze względu na wiele cech, takich jak łatwość oraz szybkość projektowania i wdrażania, duża elastyczność, coraz większa powierzchnia, a przy tym coraz większa szybkość i mniejszy koszt układów. Współczesne układy FPGA charakteryzują się ponadto nowymi możliwościami, takimi jak: wbudowane bloki pamięci RAM, czy sprzętowe mnożarki. Niektóre układy posiadają wbudowane moduły DSP, Gigabit Ethernet, procesory w postaci rdzenia sprzętowego, itp.[T05]. Dostępnych jest również coraz więcej gotowych modułów IP (ang. Intellectual Property) implementujących standardowe funkcje z zakresu przetwarzania sygnałów, telekomunikacji, kryptografii, a także rdzenie procesorów i inne elementy systemów komputerowych. Obecnie intensywnie rozwijane są platformy systemowe pozwalające na implementację całych systemów komputerowych w jednym układzie FPGA. Współczesne układy FPGA pozwalają na dużo większe możliwości projektowania systemów wbudowanych, niż jeszcze kilkanaście lat wcześniej [M05b]. Niektóre firmy (np. Xilinx, Atmel) udostępniają układy FPGA dające możliwość częściowej rekonfiguracji systemu podczas działania [GCSBF06, LBMYB06, A04].

Ze względu na architekturę systemy wbudowane można podzielić na jednoprocesorowe i rozproszone (wieloprocesorowe). W wielu zastosowaniach architektura jednoprocesorowa może mieć zbyt małą moc obliczeniową (np. systemy multimedialne, telekomunikacja). Dlatego coraz częściej stosowanymi rozwiązaniami są architektury rozproszone, składające się z wielu wyspecjalizowanych modułów sprzętowych oraz procesorów. Wówczas część zadań o dużej złożoności obliczeniowej może być wykonana przez zasoby sprzętowe, podczas, gdy pozostałe przez procesory. Architektura

[1] Mieszanej (softwarowo-sprzętowej).

rozproszona pozwala na zrównoleglenie wykonywania wielu zadań, przez co szybkość systemu znacząco rośnie. W rozprawie przyjęto model architektury rozproszonej, gdyż jest to model bardziej ogólny i zgodny z kierunkiem rozwoju współczesnych architektur [H07a].

Proste systemy wbudowane (np. sterowniki) mogą być implementowane z wykorzystaniem standardowych mikrokontrolerów (np. 8051). Obecnie coraz więcej systemów wbudowanych integruje sterowniki z innymi funkcjami realizowanymi przez urządzenia (np. aparaty cyfrowe, TV, komputery pokładowe w samochodach, wbudowane serwery internetowe, telefony komórkowe, funkcje urządzeń PDA, itp.). To powoduje, że złożoność systemów wbudowanych rośnie, a jednocześnie rosną wymagania dotyczące szybkości przetwarzania, niskiego poboru mocy, dużego stopnia scalenia, niezawodności, itp. W związku z tym stosowane są bardziej zaawansowane technologicznie implementacje tych systemów:

- **SOC** (ang. System on a Chip): system zaimplementowany jest w formie jednego układu ASIC, zawierającego wyspecjalizowane moduły sprzętowe, rdzenie procesorów uniwersalnych, pamięci, interfejsy itp.;
- **SOPC** (ang. System on a Programmable Chip): system w całości zaimplementowany jest w jednym układzie FPGA. W celu umożliwienia implementacji softwarowej wybranych funkcji stosowane są układy FPGA z wbudowanymi procesorami uniwersalnymi (w formie rdzenia procesora na stałe wbudowanego w układ FPGA np. PowerPC w układach Virtex II Pro/4/5 firmy Xilinx) lub procesory te są implementowane w logice FPGA przy zastosowaniu dostępnych modułów IP (np. Nios II firmy Altera, MicroBlaze firmy Xilinx);
- **DRSOPC** (ang. Dynamically Reconfigurable SOPC): implementacja pozwalająca na lepsze wykorzystanie powierzchni układów FPGA, poprzez wielokrotne reprogramowanie fragmentów układu dla różnych funkcjonalności, w trakcie działania systemu. ***Dynamiczna rekonfiguracja*** (ang. Run-Time Reconfiguration - RTR) pozwala na wykonanie większej ilości obliczeń w zasobach sprzętowych. Systemy dynamicznie rekonfigurowalne opierają się na koncepcji tzw. wirtualnego sprzętu [SB94]. Dzięki wielokrotnemu wykorzystaniu tych samych zasobów sprzętowych systemy DRSOPC mogą być szybsze od systemów bez dynamicznej rekonfiguracji, przy założeniu tej samej powierzchni układu FPGA. Szczególnym przypadkiem systemu DRSOPC jest system **SRSOPC** (ang. Self-Reconfigurable SOPC). Różnica polega na wbudowaniu w układ FPGA sterownika (może to być specjalizowany sterownik, bądź procesor uniwersalny) zajmującego się reprogramowaniem fragmentów układu. Systemy SRSOPC pozwalają na zintegrowane w jednym układzie FPGA całego systemu, razem z logiką wymaganą do realizacji dynamicznej rekonfiguracji.

Systemy dynamicznie rekonfigurowalne wykorzystują układy FPGA reprogramowalne w systemie (ISR – ang. In-System Reprogrammable). Ze względu na możliwości reprogramowania, można wyróżnić następujące typy układów FPGA:

- układy jednokontekstowe: to układy, które są zawsze reprogramowane w całości (np. układy FPGA firmy Altera). Dynamiczną rekonfigurację można uzyskać poprzez implementację wieloukładową, tzn. podczas gdy część układów pracuje, pozostała część jest reprogramowana. Jednak ze względu na duży czas reprogramowania całych układów, oraz duży koszt implementacji systemów wieloukładowych, efektywne wykorzystanie jednokontekstowych FPGA w systemach dynamicznie rekonfigurowanych jest bardzo trudne;
- układy wielokontekstowe: to układy pamiętające jednocześnie kilka konfiguracji. Rekonfigurowanie polega na przełączaniu pamięci konfiguracji i odbywa się bardzo szybko. Powstały laboratoryjne prototypy takich układów [TCJW97]. Jednak aktualnie nie ma na rynku dostępnych żadnych układów tego typu;
- układy z możliwością częściowego reprogramowania: w układach tych można reprogramować tylko część komórek logicznych, podczas gdy pozostałe pracują w trybie normalnym. Reprogramowanie fragmentów układu jest bardzo szybkie (od kilku μs), dzięki czemu łatwiej jest zrównoleglić reprogramowanie z obliczeniami. Aktualnie dostępnych jest wiele układów częściowo reprogramowalnych, np. układy serii Virtex, Vitrtex II, Virtex 4, Virtex 5, Spartan 3 firmy Xilinx, AT40K firmy Atmel.

W pracy wykorzystuje się częściowo reprogramowalne układy FPGA. Systemy dynamicznie rekonfigurowalne znajdują zastosowanie w wielu dziedzinach takich, jak np. telekomunikacja (szybkie układy pozwalające dynamicznie przełączać różne protokoły sieciowe w zależności od potrzeb), medycyna (przetwarzanie obrazów), kodowanie, statki kosmiczne biorące udział w różnych misjach (potrafiące dynamicznie same rekonfigurować się w zależności od nowych zadań misji lub naprawiać błędy projektowe bez potrzeby powrotu), robotyka, stacje sejsmiczne, różne sterowniki w samochodach, przetwarzanie wideo i wiele innych [GCSBF06].

Wraz z coraz większymi możliwościami technologicznymi możliwe jest uzyskiwanie coraz bardziej złożonych systemów. Projektowanie takich systemów wiąże się jednak z koniecznością opracowania efektywnych narzędzi wspomagających projektanta w ich tworzeniu. Ze względu na czasochłonny i kosztowny proces projektowania systemów komputerowych, jednym z podstawowych kierunków badań staje się opracowanie skutecznych metod automatyzujących ten proces. Głównym etapem projektowania na poziomie systemowym jest ***kosynteza*** (ang. Hardware-Software Co-Synthesis). ***Kosynteza*** jest procesem jednoczesnej syntezy części softwarowej (SW) i sprzętowej (HW) systemu. Dysponując specyfikacją systemu kosynteza HW-SW systemów rozproszonych ma za zadanie [SW97]: ***alokację*** zasobów - wybór odpowiednich zasobów (procesorów, wyspecjalizowanych modułów sprzętowych, łączy komunikacyjnych, itp.); ***przyporządkowanie*** procesów do zasobów - określenie, jakie procesy obliczeniowe i komunikacyjne, wymienione w specyfikacji, będzie wykonywał każdy z zasobów; ***szeregowanie*** procesów obliczeniowych i komunikacyjnych – określenie kolejności wykonywania procesów i transmisji. W wyniku kosyntezy,

na podstawie specyfikacji systemu, generowana jest architektura systemu, która spełnia wszystkie wymagania określone w specyfikacji.

Prace badawcze nad algorytmami automatyzującymi proces kosyntezy skierowane są na optymalizację różnych parametrów związanych z projektowanymi systemami wbudowanymi. Przeważnie są to: maksymalizacja szybkości systemu (jak w niniejszej rozprawie) [BBD05a, M05a], minimalizacja kosztu [DJ98b], minimalizacja powierzchni implementowanego systemu [MD05], ale także minimalizacja poboru mocy systemu [SHE05], czy zwiększenie niezawodności systemu [XLKVI04]. Zwykle przyjmuje się kilka kryteriów optymalizacji. Wówczas stosuje się metody optymalizacji wielokryterialnej [SJ02, ZX06], np. w algorytmie Yena-Wolfa [YW95] wprowadza się tzw. współczynnik czułości uwzględniający dwa parametry: koszt i szybkość. Zatem poszukuje się rozwiązań kompromisowych pomiędzy kosztem a szybkością systemu. Można przyjmować również wartość jednych kryteriów jako wymaganych ograniczeń projektowanego systemu, a optymalizować wg pozostałych. Na przykład szuka się najszybszego systemu nie przekraczającego zadanego kosztu [CA02] lub poszukuje się systemu najtańszego o minimalnej wymaganej szybkości [DJ98b].

W rozprawie zaprezentowane zostaną metody automatycznej kosyntezy systemów SOPC, dynamicznie rekonfigurowalnych wieloprocesorowych systemów SOPC (w tym systemów SRSOPC), a także algorytm, w którym wykorzystuje się informacje o wzajemnie wykluczających się obliczeniach do uzyskania systemów o lepszych parametrach jakościowych. Opracowane algorytmy zwiększają szybkość projektowanego systemu przy zadanym ograniczeniu powierzchni układu FPGA. Dokonane zostanie porównanie efektywności opracowanych algorytmów z istniejącymi metodami.

W rozdziale 2 zaprezentowano przegląd dotychczas stosowanych rozwiązań w zakresie metod automatycznej kosyntezy i projektowania systemów dynamicznie rekonfigurowalnych. W rozdziale 3 przedstawiono motywację, a w rozdziale 4 cel i tezę rozprawy. Krótkie wprowadzenie podstawowych pojęć z zakresu tematyki rozprawy zaprezentowano w rozdziale 5. W rozdziale 6 przedstawiono opracowane algorytmy automatycznej kosyntezy dla systemów SOPC i DRSOPC (SRSOPC) oraz ich analizę wykazującą przydatność metod dla optymalizacji szybkości systemów dynamicznie rekonfigurowalnych. Rozdział ten zawiera również analizę złożoności obliczeniowej oraz wyniki wykonanych eksperymentów mających na celu ocenę jakości opracowanych metod. W rozdziale 7 zaprezentowano praktyczne przykłady demonstrujące możliwości opracowanych metod. Rozdział 8 stanowi podsumowanie rozprawy.

2 METODY KOSYNTEZY SYSTEMÓW WBUDOWANYCH

Zagadnienia związane z automatyczną syntezą systemów wbudowanych wzbudzają zainteresowanie już od wielu lat. Pierwsze systemy wbudowane były prostymi sterownikami i do ich projektowania nie były konieczne wyspecjalizowane narzędzia. Wraz z pojawieniem się coraz bardziej rozbudowanych urządzeń, takich jak kamery i aparaty cyfrowe, telefony komórkowe integrujące wiele różnych funkcjonalności, okazało się, że proste, klasyczne metody projektowania, stosowane dotychczas, nie są już wystarczające. Duża złożoność systemów wbudowanych uniemożliwia ich ręczną optymalizację. Tym bardziej, że coraz większa konkurencyjność na rynku wymusiła konieczność przyspieszenia czasu projektowania takich systemów. Tym samym zaistniała potrzeba automatyzacji procesu projektowania systemów wbudowanych, prowadząca do zmniejszenia kosztów realizacji urządzeń i czasu do ich udostępnienia na rynku. Rozwój systemów zorientowanych na przetwarzanie danych spowodował, że dla zwiększenia ich mocy obliczeniowej tworzy się heterogeniczne architektury wieloprocesorowe, co dodatkowo komplikuje problem jednoczesnego projektowania części sprzętowej i softwarowej takich systemów. Ważnym aspektem przy projektowaniu tych systemów jest ich optymalizacja. Projektanci przy optymalizacji kierują się, oprócz maksymalizacji szybkości, także minimalizacją kosztu, czy w czasach szybkiego rozwoju urządzeń mobilnych – minimalizacją poboru mocy, itp. Wobec powyższych problemów wyłoniony został kierunek badań – projektowanie na poziomie systemowym (ang. Electronic System Level – ESL), którego jednym z głównych nurtów jest kosynteza, czyli jednoczesne projektowanie części sprzętowej i softwarowej systemów w celu uzyskania jak najlepszego rozwiązania (spełniającego zadane kryteria optymalizacji). W ciągu ostatnich kilkunastu lat opublikowano wiele prac dotyczących kosyntezy systemów heterogenicznych. Główne kierunki badań dotyczących kosyntezy systemów zorientowanych na przetwarzanie danych są przedstawione poniżej.

Zwykle w procesie kosyntezy uwzględnia się kilka kryteriów optymalizacji. Można dążyć do uzyskania sytemu najlepszego pod względem jednego kryterium, który jednocześnie spełnia ograniczenia określone przez inne kryterium. Kryteriami optymalizacji najczęściej są: szybkość systemu, jego koszt i pobór mocy. Niektóre metody minimalizują koszt systemu przy zadanej minimalnej szybkości [DJ98b, D04, MD05, YT06], inne maksymalizują szybkość przy ograniczonym koszcie [NB01, CA02, BBD05a, M05a], jeszcze inne minimalizują pobór mocy przy zapewnieniu zadanej minimalnej szybkości systemu [RV02, SK03, WHE03]. Można maksymalizować niezawodność systemu [XLKVI04], czy wprowadzać kryterium minimalizacji efektów termicznych do algorytmu kosyntezy [HXVKI05]. Istnieją również metody kosyntezy z optymalizacją wielokryterialną: jednocześnie koszt i pobór mocy systemu [DJ97, SJ02, SDJ07], koszt i szybkość [YW95], koszt, pobór mocy i powierzchnię [DJ99], czy też równocześnie powierzchnię, pobór mocy i szybkość systemu [ZX06]. Powstały też prace, w których istnieje możliwość wyboru kryterium

optymalizacji. Jedną z takich prac jest [JKH05], gdzie zastosowano algorytm genetyczny, który pozwala na uzyskanie grupy przybliżonych rozwiązań alternatywnych w jednym przebiegu optymalizacji. Na podstawie tych alternatywnych rozwiązań projektant może wybrać najlepsze rozwiązanie wg parametru, który go interesuje: kosztu, szybkości, czy niezawodności. Przy projektowaniu systemów wbudowanych SOPC mamy do dyspozycji układy FPGA o określonej powierzchni, a celem jest znalezienie najszybszego systemu, który zmieści się w tym układzie. Zatem w praktyce dla systemów SOPC maksymalizuje się szybkość projektowanego systemu przy ograniczonej powierzchni wymaganej do implementacji w FPGA. Takie kryteria optymalizacji zostały przyjęte również w niniejszej rozprawie.

Metody kosyntezy operują na reprezentacji wewnętrznej (modelu) systemu, tworzonej na podstawie specyfikacji w formie komunikujących się procesów (zadań). Najczęściej przyjmowanym modelem systemu jest graf zadań [DJ97, DJ98b, CV99, DJ99, CA02, D04, BBD05a, ITM05, QSN06, ZX06, FVAGMT07, WKINN07], ale w kilku pracach uwzględniono również specyfikację systemu w formie warunkowego grafu zadań [DE98, XW01, WHE03, BNH05]. Czasami model systemu specyfikowany jest w innej postaci, jak rozszerzony graf przepływu danych i sterowania (ang. Extended Control Data Flow Graph - ECDF) [ZN00], w formie hierarchicznego FSM oraz grafu SDF [LYC02]. Ze względu na to, iż od wielu lat najbardziej popularnym modelem systemu jest graf zadań, w pracy przyjęto również taką reprezentację, z możliwością specyfikacji warunkowego wykonywania zadań.

Pierwsze prace dotyczące kosyntezy koncentrowały się głównie na problemie podziału obliczeń pomiędzy jeden procesor ogólnego przeznaczenia a część sprzętową [EHB93, GM93, W94]. Jednak obecnie wiele systemów wbudowanych są to rozproszone systemy heterogeniczne składające się z wielu procesorów uniwersalnych, procesorów sygnałowych, kontrolerów we/wy, wyspecjalizowanych modułów sprzętowych, analogowych, pamięci, itp. W związku z tym dalsze prace zostały skoncentrowane na systemach rozproszonych.

Istnieją algorytmy umożliwiające znalezienie najlepszego rozwiązania. Do nich można zaliczyć metody oparte o programowanie liniowe całkowitoliczbowe (ang. Integer Linear Programming – ILP) [PP92, NM96, KKS01, FVAGMT07] lub wyszukiwanie wyczerpujące [DH94]. W algorytmie [ZX06] zastosowano model matematyczny do wyznaczania wszystkich możliwych ścieżek w grafie zadań, w celu znalezienia optymalnego rozwiązania. Problem kosyntezy jest jednak NP-zupełny, dlatego zastosowanie takich algorytmów ogranicza się jedynie do bardzo prostych systemów. Większe znaczenie mają heurystyczne metody kosyntezy, wśród których można wyróżnić metody konstrukcyjne i metody rafinacyjne.

Algorytmy konstrukcyjne [DLJ97, BAP98, DJ98a, NB01] działają na zasadzie tworzenia rozwiązania poprzez stopniową rozbudowę systemu w celu implementacji kolejnych zadań i transmisji. Głównym problemem w tych metodach jest oszacowanie wpływu decyzji podejmowanych w każdym kroku na ostateczną jakość rozwiązania. Do tego celu wykorzystywane są różne miary dokonujące

oceny możliwych decyzji projektowych. Miary te mają na celu wyeliminowanie decyzji projektowych prowadzących do konstrukcji systemów nie spełniających zadanych ograniczeń oraz zapewnienie jak największych możliwości optymalizacji w następnych krokach. Oparte są głównie na oszacowaniach na najlepszy i najgorszy przypadek. Jednak zwykle takie miary preferują zasoby najszybsze lub najtańsze, a pomijają pozostałe dostępne zasoby. Algorytm COSYN [DLJ97] w pewnym stopniu ma możliwość modyfikacji wcześniejszych decyzji, ale tylko w przypadku, gdy zostaną naruszone ograniczenia na czas obliczeń i ma to jedynie za zadanie zwiększenie szybkości architektury. Nie może jednak takich nawrotów być zbyt wiele, gdyż czas działania algorytmu znacznie się zwiększa. Jest to algorytm zachłanny, który kierując się jedynie kryterium minimalnego wzrostu kosztu, zwiększa prawdopodobieństwo szybkiego utknięcia w lokalnym minimum kosztu. W algorytmie PA^2 [BAP98] przyjęto specjalne miary mające na celu określenie prawdopodobieństwa wpływu podjętych decyzji na następne kroki w procesie kosyntezy. Miara została zdefiniowana w taki sposób, aby zmniejszyć ilość alokowanych na początku działania algorytmu tanich zasobów i aby zostawić miejsce dla optymalizacji w ostatnich etapach kosyntezy. Algorytm ten uwzględnia tylko sumę kosztów wykonania poszczególnych zadań, nie uwzględnia kosztów jednostkowych, ani alokacji i kosztów kanałów komunikacyjnych. Algorytmy konstrukcyjne są szybkie i można czasem uzyskać dobre wyniki [DLJ97], ale mają skłonność do zatrzymywania się w lokalnych minimach optymalizowanych parametrów.

Algorytmy rafinacyjne startują od przybliżonego rozwiązania początkowego i w kolejnych krokach generują nowe rozwiązania poprzez modyfikacje poprzednich, w celu poprawy optymalizowanych parametrów. Algorytmy rafinacyjne dzielą się na: probabilistyczne i o bezpośrednim wyszukiwaniu. Głównym problemem tych algorytmów jest zapewnienie ich zbieżności. Często stosuje się odpowiednie kryterium wyboru rozwiązania, które nie pozwala na powrót do wcześniej rozpatrywanych rozwiązań. Prowadzi to czasami do zatrzymywania się w lokalnym minimum optymalizacji.

Algorytmy probabilistyczne są algorytmami rafinacyjnymi, w których kolejne rozwiązania są generowane na podstawie poprzednich z uwzględnieniem czynników losowych. Do klasy probabilistycznych algorytmów kosyntezy należą m.in. symulowane wyżarzanie (ang. simulated annealing) [EPKD97] i algorytmy genetyczne [DJ97, CA02, JKH05]. Algorytmy genetyczne [G89] oparte są na biologicznym procesie adaptacji organizmów na drodze genetycznej ewolucji. Rozwiązania reprezentowane są w postaci tzw. chromosomów, w których sąsiednie pozycje opisują najbardziej zbliżone do siebie cechy rozwiązania. W trakcie optymalizacji generowane są nowe pokolenia rozwiązań na drodze reprodukcji, krzyżowania i mutacji. Algorytmy symulowanego wyżarzania [KGV83] oparte są na fizycznym procesie wyżarzania i stopniowego schładzania kryształów w celu otrzymania regularnej struktury uporządkowanych atomów. Niektóre algorytmy łączą cechy algorytmów symulowanego wyżarzania i genetycznych [DJ98b]. Algorytmy probabilistyczne mają zdolność wydobywania się z lokalnych minimów optymalizowanych

parametrów. Ich wyniki silnie zależą jednak od przyjętych wartości parametrów algorytmu. Przykładem takiego algorytmu jest algorytm genetyczny MOGAC [DJ97]. Jak wykazano w [EPKD97] algorytmy podziału zadań pomiędzy sprzęt a oprogramowanie, oparte na algorytmie symulowanego wyżarzania, są mniej efektywne od rafinacyjnego algorytmu o bezpośrednim wyszukiwaniu z tabu (ang. tabu search).

W metodach rafinacyjnych z bezpośrednim wyszukiwaniem [YW95, HW96, D04], ze względu na bardzo dużą liczbę możliwych rozwiązań, bierze się pod uwagę tylko niektóre rozwiązania poprzez stosowanie różnych miar jakości rozwiązań. Wiele istniejących algorytmów tego typu ma też tendencje do zatrzymywania się w lokalnych minimach. Główną przyczyną jest to, że metody rafinacji uwzględniają tylko lokalne zmiany sterowane współczynnikiem zysku, w którym bierze się pod uwagę tylko jeden parametr (np. szybkość lub koszt). W algorytmie [YW95] modyfikacje polegają na przesunięciu tylko jednego zadania z jednego elementu obliczeniowego do innego. W ten sposób nie są możliwe większe (globalne) zmiany w systemie. W związku z tym algorytm może zatrzymywać się w lokalnym minimum optymalizowanych parametrów. Algorytm tabu serach [GTD93] jest też rodzajem iteracyjnej rafinacji, w którym wszystkie zmiany pamiętane są w dwóch rodzajach pamięci (krótkoterminowej i długoterminowej), dzięki którym możliwe są nawroty do stworzonych wcześniej rozwiązań i możliwe jest dokonywanie późniejszych zmian. Algorytm wyszukiwania z tabu został również zastosowany w podziale zadań pomiędzy sprzęt i oprogramowanie [EPKD97], jednak nie udało się uzyskać efektywnego algorytmu do kosyntezy systemów wbudowanych. Żaden z wyżej wymienionych algorytmów nie uwzględnia implementacji w postaci systemów SOPC.

Do klasy rafinacyjnych algorytmów z bezpośrednim wyszukiwaniem należy algorytm EWA [D04]. Algorytm ten wnosi kilka nowych cech, decydujących o jego efektywności, w stosunku do innych algorytmów rafinacyjnych przeznaczonych dla systemów SOC. Zastosowano nowe metody rafinacji. W przeciwieństwie do wielu innych algorytmów kosyntezy, algorytm EWA modyfikuje w pierwszej kolejności alokacje zasobów, zamiast przydziału zadań. W każdym kroku rafinacji wykonywane są modyfikacje takie jak: dodanie jednego zasobu do systemu i przyporządkowanie do niego jak największej liczby zadań oraz usunięcie zasobu i przeniesienie przydzielonych do niego zadań na inne zasoby. Zastosowanie w jednym kroku jednocześnie dodania i usunięcia zasobu pozwala na uzyskanie dodatkowych możliwości rafinacji systemu. Dodanie zasobu zwiększy koszt systemu, a następnie usunięcie w tym samym kroku innego zasobu może spowodować ostatecznie spadek kosztu i w ten sposób algorytm może wydobywać się z lokalnych minimów kosztu. Zastosowane metody rafinacji pozwalają wprowadzić globalne zmiany w systemie przy niewielkim stopniu złożoności tych metod. O wyborze rozwiązania do dalszej rafinacji decyduje funkcja zysku, która nie kieruje się zachłannością, ale uwzględnia się w niej, oprócz minimalizacji kosztu, także tzw. swobodę modyfikacji systemu w następnych krokach algorytmu. Ten dodatkowy współczynnik uwzględnia dla każdego zadania rezerwę czasu na wykonanie, nie przekraczającą zadanego ograniczenia czasowego. Algorytm EWA ma możliwość wydobywania się z lokalnych minimów (w

przeciwieństwie do wielu innych istniejących algorytmów rafinacyjnych) m.in. dlatego, że funkcja zysku uwzględnia również możliwości rafinacji w dalszych krokach, a nie tylko stopień poprawy optymalizowanych parametrów. Algorytm kończy działanie, gdy nie ma już możliwości modyfikacji systemu, dla której zysk jest większy od zera. Uzyskane wyniki wskazują na dużą efektywność algorytmu EWA (są znacznie lepsze niż w [YW95]).

Ostatnio, jednym z intensywniej rozwijanych kierunków badań w dziedzinie kosyntezy są systemy dynamicznie rekonfigurowalne. Jednym z pierwszych algorytmów kosyntezy dynamicznie rekonfigurowalnych systemów wbudowanych jest CORDS [DJ98b]. Algorytm minimalizuje koszt architektury złożonej z układów FPGA i procesorów. W [CV99] zaprezentowano algorytm kosyntezy dla architektury złożonej z jednego procesora i dynamicznie reprogramowanego układu FPGA. Podobną architekturę systemu zastosowano w algorytmie genetycznym [CA02]. Również w [LYC02] zastosowano architekturę z jednym procesorem. W kompilatorze Nimble [LCDHKS00] programy w języku C są odwzorowywane w system złożony z jednego procesora i dynamicznie reprogramowanego FPGA. W [SJ02] zaprezentowano wieloaspektowy algorytm kosyntezy (minimalizujący zarówno koszt systemu, jak i pobór mocy) dla systemów wbudowanych, w którym architektura składa się z dynamicznie reprogramowalnych układów FPGA, procesorów i innych zasobów. Wykorzystywany jest algorytm ewolucyjny[2]. Zastosowano listowy algorytm szeregowania, w którym określa się priorytety zadań w zależności od ograniczeń czasowych i informacji dotyczących rekonfiguracji; następnie szereguje się zadania biorąc pod uwagę wykorzystanie zasobów i stan rekonfiguracji w przestrzeni zarówno czasu, jak i powierzchni. Wszystkie te metody stosują układy reprogramowane w całości i nie wykorzystują możliwości częściowej rekonfiguracji systemów. Algorytm SLOPES [SDJ07] wykorzystuje możliwości układów częściowo reprogramowalnych firmy Xilinx uwzględniając niektóre zasady rekonfiguracji modułowej [3] (więcej informacji na temat rekonfiguracji modułowej zostanie podane w rozdziale 5). Nie bierze się natomiast pod uwagę rozmieszczenia reprogramowalnych modułów w układzie. Wszystkie wyżej wymienione metody stosują architektury wieloukładowe, zatem nie nadają się one dla systemów SOPC. Algorytm [OCP05], minimalizujący pobór mocy, uwzględnia implementację wieloprocesorową w postaci SOPC i wykorzystuje częściowo reprogramowalne układy FPGA (Xilinx), ale stosowany jest bardzo uproszczony model w postaci grafu liniowego. W algorytmie tym stosowana jest metoda rekonfiguracji różnicowej, podobnie jak w [HZ07]. Algorytm [FVAGMT07] uwzględnia możliwości częściowego reprogramowania układów firmy Xilinx, ale jest to metoda oparta o ILP, więc nie nadaje się do bardziej złożonych systemów.

W wielu pracach duży nacisk położono na minimalizację narzutów czasowych spowodowanych reprogramowaniem układu FPGA. W algorytmie [JYLC00] w celu zmniejszenia narzutu czasowego

[2] Algorytmy ewolucyjne są to algorytmy genetyczne, w których chromosomy nie są wektorami bitów, ale bardziej złożonymi strukturami danych.

[3] W rekonfiguracji modułowej reprogramowane są całe bloki komórek FPGA, w przeciwieństwie do rekonfiguracji różnicowej, gdzie reprogramowane są dowolne komórki logiczne.

spowodowanego reprogramowaniem FPGA, wprowadzono tzw. koncepcję wczesnej częściowej rekonfiguracji (ang. Early Partial Reconfiguration - EPR), poprzez wykonanie rekonfiguracji dla operacji tak szybko jak to jest możliwe, aby operacja była gotowa do wykonania, kiedy tylko jest wywołana. Dodatkowo połączono koncepcję EPR z tzw. inkrementowaną rekonfiguracją (ang. Incremental Reconfiguration - IR) w celu dalszej redukcji narzutu czasu reprogramowania. Architektura składa się jednak z jednego procesora i reprogramowanego układu FPGA, nie uwzględnia też możliwości układów częściowo reprogramowalnych. Praca [QSN06] prezentuje nowy model rekonfiguracji, który umożliwia równoległe reprogramowanie. Każdy fragment układu (sektor) ma własną pamięć konfiguracji SRAM, do której dostęp jest niezależny. W ten sposób wiele kontrolerów rekonfiguracji może równolegle sterować reprogramowaniem różnych sektorów, dzięki czemu uzyskuje się przyspieszenie systemu. Metoda ta dotyczy jednak tylko systemów sprzętowych i nie jest przeznaczona dla systemów SOPC. W metodzie [HV05] została zaprezentowana architektura sprzętowa wykorzystująca możliwości częściowego reprogramowania FPGA, ale nie uwzględniono zasad rekonfiguracji modułowej współczesnych układów. Metodę projektowania dla dynamicznie rekonfigurowalnych systemów przedstawiono również w [FSS05]. Architektura zawiera wbudowany w FPGA procesor, który dynamicznie zmienia architekturę systemu, tak, aby spełnione były wszystkie wymagania implementacji takiego systemu. Rozwiązanie wykorzystuje popularne układy FPGA i płytę projektową bez dodatkowych dedykowanych układów, ale nie jest przeznaczone dla systemów wieloprocesorowych.

Opracowano również wiele środowisk do implementacji dynamicznie rekonfigurowalnych systemów. W [KLVBR02] przedstawiono środowisko do implementacji systemów zbudowanych z modułów IP. Moduły te rozmieszczane są w wyznaczonych sektorach układu FPGA, połączonych ze sobą standardową szyną. Dynamiczna rekonfiguracja polega na wymianie modułu IP, natomiast połączenia pozostają bez zmian. Środowisko PaDReH [CCBM04] zawiera kompletny zestaw narzędzi do projektowania, walidacji i implementacji systemów dynamicznie rekonfigurowalnych, w całości implementowanych w sprzęcie. W [EP02] implementacja polega na odwzorowaniu grafu zadań w zadaną architekturę. W środowisku SRP[BRK03] przedstawiono platformę do tworzenia systemów dynamicznie rekonfigurowanych przez procesor wbudowany w układ FPGA (serii Xilinx Virtex II). W takim systemie nie ma potrzeby stosowania dodatkowych układów sterujących reprogramowaniem. W pracy [M05a] zaprezentowano samo-rekonfigurowalną, opartą na agentach, platformę z wielordzeniową rekonfigurowalną architekturą (RMSoC), pozwalającą na dynamiczną rekonfigurację sprzętu przez wbudowany procesor. Wykorzystanie agentów ułatwia tworzenie architektur adaptacyjnych, które mogą zmieniać się w czasie w zależności od potrzeb. Również algorytm [HZ07] wykorzystuje możliwości dynamicznej rekonfiguracji częściowo reprogramowalnych FPGA w systemach adaptacyjnych, ale z zastosowaniem przetwarzania wektorowego. Pozwala to na lepsze wykorzystanie zasobów sprzętowych dla różnych aplikacji przy niewielkich zmianach architektury podczas rekonfiguracji.

Żaden z dotychczas zaprezentowanych algorytmów kosyntezy nie daje zadawalających wyników. Nie ma metody, która uwzględniałaby wszystkie cechy istotne dla współczesnych układów i wymagań. Rozwój technologii układów FPGA powoduje, że niektóre z metod są już nieaktualne. Na przykład, jedynie w algorytmie [BBD05a] uwzględniono ograniczenia dotyczące rozmieszczenia reprogramowalnych modułów w układach FPGA. W pozostałych metodach stosuje się układy reprogramowalne w całości [DJ98b, CV99, CA02], w niektórych nowszych metodach, uwzględniających możliwość częściowej rekonfiguracji, nie uwzględnia się fizycznych ograniczeń związanych z rozmieszczeniem reprogramowanych sektorów we współczesnych układach [HV05]. W wielu metodach stosuje się architektury wieloukładowe, metody te nie nadają się dla systemów SOPC. Żadne, ze znanych z literatury, metod automatycznej syntezy dynamicznie rekonfigurowalnych systemów wbudowanych nie uwzględnia implementacji systemu w formie wieloprocesorowego systemu DRSOPC. Rozwój systemów zmierza w kierunku systemów rozproszonych. Przewiduje się, że w niedalekiej przyszłości, większość produkowanych systemów wbudowanych będą to systemy wieloprocesorowe. W większości prac zakłada się, że dynamiczna rekonfiguracja jest wykonywana z wykorzystaniem zewnętrznych, dodatkowych układów. Niektóre prace przedstawiają sposób implementacji systemów samorekonfigurowalnych, ale nie ma metod kosyntezy dla takich systemów. W żadnym z istotnych algorytmów kosyntezy systemów DRSOPC nie uwzględniono informacji o warunkowym wykonaniu zadań, do optymalizacji rozwiązań.

Głównymi problemami utrudniającymi porównanie istniejących metod kosyntezy są różne założenia dotyczące specyfikacji systemu, docelowej architektury systemu, kryteriów optymalizacji. W większości prac demonstrowane są eksperymenty jedynie dla prostych przykładów (kilka zadań), co uniemożliwia ocenę efektywności poszczególnych metod. W niektórych pracach wykorzystuje się standardowe benchmarki jak E3S [E3S], czy MiBench [GREAMB01] zawierające dane dotyczące pojedynczych zadań. Narzędzie TGFF [DRW98] umożliwia generację grafów dla benchmarków E3S, ale o losowej strukturze, niekoniecznie odzwierciedlającej rzeczywiste systemy. W dostępnych pracach nie są podawane struktury grafów wygenerowanych losowo, co dodatkowo utrudnia porównanie tych metod.

3 MOTYWACJA

Rozwój technologii układów FPGA pozwala na implementowanie w nich coraz bardziej złożonych systemów wbudowanych. Rośnie złożoność urządzeń, a jednocześnie istnieje tendencja do ich miniaturyzacji i integrowania wielu różnych funkcjonalności w jednym urządzeniu. Przykładem mogą być telefony komórkowe, kamery cyfrowe, telewizja cyfrowa, itp. Systemy takie mogą być implementowane jako systemy wbudowane, z wykorzystaniem heterogenicznych architektur tworzonych w technologii FPGA. Dzięki temu można uzyskać systemy szybsze (poprzez akcelerację sprzętową krytycznych funkcji) i tańsze. Istnieje również tendencja w kierunku tworzenia architektur wieloprocesorowych, a takie możliwości dają również współczesne układy FPGA. Tym samym rozwijane są systemy SOPC pozwalające na integrację całego systemu w jednym układzie. Dodatkowo, dzięki wykorzystaniu dynamicznej rekonfiguracji systemów implementowanych w układach FPGA, możliwe jest tworzenie systemów jeszcze szybszych poprzez wielokrotne wykorzystanie tych samych zasobów dla różnych funkcjonalności. Technologia FPGA wchodzi w różne dziedziny zastosowań. Do niedawna wykorzystywana głównie w telekomunikacji, przetwarzaniu obrazów, w urządzeniach powszechnego użytku (kamery, aparaty cyfrowe, TV, nawigacja satelitarna, itp.), ale teraz także w medycynie, statkach kosmicznych [FMCOZ07], w przemyśle samochodowym (np. wideo-asystent kierowcy [CZMS07] wykorzystujący możliwości dynamicznej rekonfiguracji częściowo reprogramowalnych FPGA, komputery pokładowe), militariach [MMTDM07], bioinformatyce [HVG07] i w wielu innych dziedzinach.

Ze względu na czasochłonny i kosztowny proces projektowania, coraz bardziej złożonych, wbudowanych systemów komputerowych, jednym z podstawowych kierunków badań staje się opracowanie skutecznych metod automatyzujących ten proces. Problem automatycznej kosyntezy jest bardzo złożony i wciąż stanowi wyzwanie dla naukowców [M05a]. Po pierwsze jest to problem NP-zupełny i nie ma odpowiednich algorytmów potrafiących w satysfakcjonującym czasie znaleźć najlepszą architekturę spełniającą zadane wymagania. Po drugie brakuje efektywnych narzędzi radzących sobie z rzeczywistymi problemami projektowymi np. dla współczesnych układów FPGA. Tylko w niektórych metodach uwzględnia się te problemy [BBD05a, SDJ07], ale są one stosowane dla architektur jednoprocesorowych [BBD05a] i zakładają implementację wieloukładową. Znane metody dla systemów SOPC nie uwzględniają z kolei rzeczywistych problemów układów częściowo reprogramowalnych [OCP05]. Większość metod kosyntezy systemów SOPC/DRSOPC zostało opracowanych dla przestarzałych układów FPGA i nie nadaje się do wykorzystania we współczesnych układach (mających m.in. zupełnie inne parametry i metody reprogramowania). Żadna z istniejących metod kosyntezy dla współczesnych FPGA nie wykorzystuje w procesie optymalizacji możliwości warunkowego wykonywania zadań. Wśród znanych z literatury algorytmów kosyntezy nie można wytypować najlepszego. Nie ma takiego rozwiązania, które można by już stosować w przemyśle.

Problem automatycznej syntezy dynamicznie rekonfigurowalnych systemów sprzętowo-softwarowych zaczął być intensywnie badany dopiero od kilku lat, w związku z pojawieniem się częściowo reprogramowalnych układów FPGA o szybkim czasie programowania (pierwsza praktyczna praca dotycząca kosyntezy dynamicznie rekonfigurowalnych systemów została zaprezentowana w 2005r [BBD05a]). Wykorzystanie dynamicznej rekonfiguracji systemów w technologii FPGA powinno, przy szybkich czasach reprogramowania, umożliwiać projektowanie szybszych systemów, o mniejszej powierzchni. Zastosowanie dynamicznej rekonfiguracji do realizacji nowych, złożonych systemów jest kierunkiem przyszłościowym. W najbliższych latach powinny rozwijać się tzw. architektury „inteligentne" wykorzystujące te możliwości. Powstają już w tym kierunku koncepcje, takie jak np. architektury wieloagentowe [M05a]. Zaczynają być rozwijane architektury adaptacyjne [HZ07], które mogą automatycznie dostosowywać się w zależności od zaistniałych potrzeb. Przykładem takich architektur adaptacyjnych może być wbudowany serwer internetowy, który w zależności od ilości otrzymanych żądań różnych typów protokołów rekonfiguruje się dynamicznie do wykonywania protokołu o największej liczbie żądań ze strony klientów. Do tej pory jednak nie zostały zaprezentowane żadne metody kosyntezy dla dynamicznie samorekonfigurowalnych wieloprocesorowych systemów SOPC implementowanych we współczesnych układach FPGA.

4 CEL I TEZA ROZPRAWY

Celem rozprawy jest opracowanie wyspecjalizowanego algorytmu kosyntezy dynamicznie rekonfigurowalnych systemów SRSOPC. Algorytm będzie maksymalizował szybkość projektowanego systemu SRSOPC przy zadanym ograniczeniu powierzchni układu FPGA. Projektowany system będzie specyfikowany w postaci skierowanego i acyklicznego grafu zadań, gdyż taki model jest również najczęściej przyjmowany w innych tego typu pracach. Opracowane metody będą mogły być wykorzystane dla współczesnych układów FPGA. Będą uwzględniać rzeczywiste ograniczenia współczesnych częściowo reprogramowalnych układów FPGA znanych producentów (m.in. dotyczące zasad rekonfiguracji modułowej). Kosynteza będzie dotyczyła rozproszonych systemów wbudowanych. Opracowane i przedstawione w rozprawie heurystyczne metody kosyntezy projektowanych systemów mają być efektywne pod względem szybkości działania, nawet dla złożonych systemów. Dzięki uwzględnieniu ograniczeń i cech współczesnych FPGA, a także zastosowaniu odpowiednich metod optymalizacji opracowany algorytm powinien być lepszy od tych znanych z literatury zarówno pod względem jakości uzyskanych wyników, jak i możliwości zastosowania w praktyce.

W przypadku systemów SOC istnieje efektywny algorytm kosyntezy EWA [D04], dla którego otrzymano dobre wyniki. Cechy tego algorytmu, pozwalające na wydobywanie się z lokalnych minimów, mogą być również wykorzystane w kosyntezie systemów SOPC. Dlatego też bazą do opracowania nowych algorytmów kosyntezy systemów SOPC będzie algorytm EWA. W pierwszej kolejności zostanie opracowany algorytm kosyntezy systemów SOPC. Dla tego algorytmu, poprzez wykonanie eksperymentów i porównanie wyników syntezy z wynikami uzyskanymi przy pomocy metod znanych z literatury, zostanie dokonana ocena skuteczności stosowanych metod rafinacji rozwiązań. Następnie zostanie opracowany algorytm kosyntezy systemów DRSOPC. Poprzez porównanie wyników syntezy tych samych systemów w formie SOPC i DRSOPC, możliwa będzie ocena skuteczności stosowania dynamicznej rekonfiguracji w systemach wbudowanych. Ostatnia wersja algorytmu kosyntezy systemów SRSOPC będzie uwzględniała informacje o zadaniach wzajemnie się wykluczających. Poprzez uwzględnienie tych informacji powinno być możliwe uzyskanie jeszcze lepszych parametrów jakościowych systemów SRSOPC. Alokacja wzajemnie wykluczających się zadań do tych samych zasobów umożliwia znaczne zmniejszenie się powierzchni zajmowanej przez te zadania.

Wykonane zostaną eksperymenty mające wykazać efektywność opracowanych algorytmów kosyntezy. W celu oceny praktycznej złożoności obliczeniowej algorytmów eksperymenty zostaną przeprowadzone dla losowo wygenerowanych grafów zadań o różnych charakterystykach i różnej złożoności. Dla zademonstrowania praktycznych korzyści z zastosowania opracowanych metod kosyntezy w systemach dynamicznie rekonfigurowalnych i jednocześnie pokazania efektywności tych

metod, przedstawione zostaną praktyczne przykłady systemów wbudowanych reprezentujących różne dziedziny zastosowań. Dokonana zostanie również ocena algorytmów poprzez analizę teoretyczną. W ramach tej oceny zostanie wykonana analiza złożoności obliczeniowej oraz warunków, jakie muszą być spełnione, aby dynamiczna rekonfiguracja była skuteczna (warunków opłacalności dynamicznej rekonfiguracji). Wykazane zostanie na drodze eksperymentalnej i teoretycznej, że dla szerokiej klasy systemów wbudowanych dynamiczna rekonfiguracja jest opłacalna, tzn. prowadzi do przyspieszenia działania projektowanego systemu.

Teza badawcza rozprawy brzmi: **wykorzystanie możliwości dynamicznej rekonfiguracji w projektowaniu szerokiej klasy systemów wbudowanych prowadzi do uzyskania implementacji szybszych niż w przypadku nie stosowania tej techniki, dla tego samego docelowego układu FPGA**.

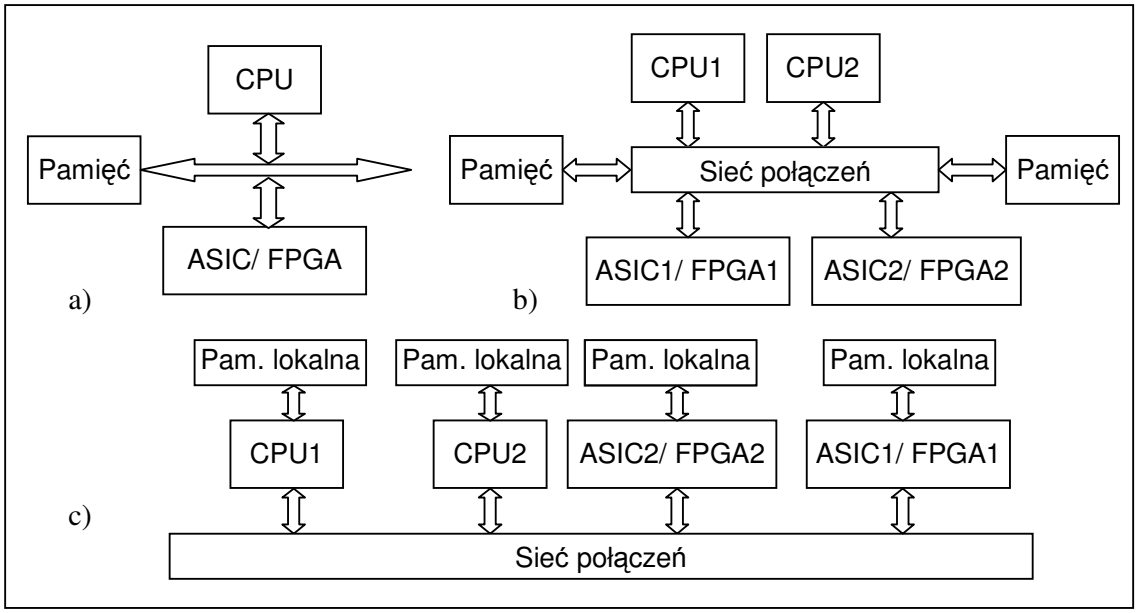
Teza zostanie udowodniona poprzez wykonanie eksperymentów porównujących implementacje tych samych systemów w architekturze SOPC i DRSOPC oraz poprzez analizę teoretyczną. Zademonstrowane zostaną również przykładowe systemy wbudowane i zostanie dokonane porównanie działania opracowanych algorytmów kosyntezy tych systemów implementowanych w postaci SOPC i DRSOPC. Na ich podstawie zostaną pokazane korzyści wynikające ze stosowania dynamicznej rekonfiguracji w systemach wbudowanych.

5 PODSTAWOWE POJĘCIA I DEFINICJE

W tym rozdziale wyjaśnione zostaną najważniejsze pojęcia wprowadzające do tematu niniejszej rozprawy. Podane zostaną założenia dotyczące specyfikacji projektowanego systemu wbudowanego, architektury takiego systemu i sposobów jego implementacji. Przedstawione zostaną również przyjęte ograniczenia i kryteria optymalizacji.

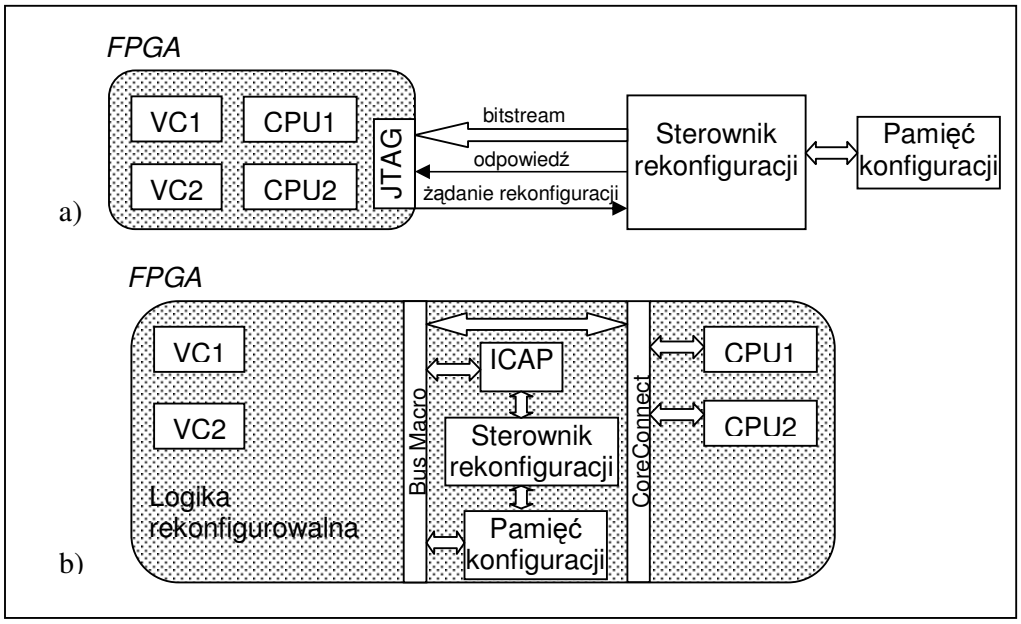
5.1 Architektura projektowanego systemu wbudowanego

Pierwsze heterogeniczne systemy wbudowane składały się z jednego procesora uniwersalnego, oraz jednego wyspecjalizowanego układu ASIC lub FPGA, pełniącego rolę akceleratora sprzętowego (Rys. 5.1-1a) [EHB93, GM93, W94]. Procesy wymagające większych mocy obliczeniowych wykonywane są zwykle przez wyspecjalizowany sprzęt, a pozostałe przez uniwersalny procesor. Bardziej uogólniony model architektury składa się z wielu układów ASIC lub FPGA oraz wielu procesorów [XW00, XLKVI04]. Przykład takiego systemu pokazany jest na Rys. 5.1-1b i c. Architektura jednoprocesorowa z Rys 5.1-1a jest szczególnym przypadkiem wieloukładowej architektury z Rys. 5.1-1b. Wykorzystywane są różne modele pamięci. Może to być pamięć współdzielona (Rys. 5.1-1b) lub pamięć lokalna (Rys. 5.1-1c). Przyjmuje się też różne modele połączeń, np. połączenia bezpośrednie (ang. point-to-point), wspólna szyna [DLJ97, LW99]. W ogólnym przypadku komunikacja pomiędzy procesorami, układami ASIC/FPGA oraz pamięciami współdzielonymi odbywa się poprzez sieć połączeń. Może ona być zrealizowana w postaci szyny (lub kilku szyn), jako sieć z przełącznicą krzyżową (ang. crossbar switch), połączenia typu siatka (ang. mesh), sieci wielostopniowe (ang. multistage network), sieci kompletnie połączone (ang. completely-connected network) i inne. W systemach jednoukładowych obecnie istnieje możliwość realizacji połączeń sieciowych tzw. NoC (ang. Network on Chip) [BM02, JOT04, HCG07]. W rozprawie przyjęto, że komunikacja odbywa się za pomocą wspólnych szyn, a każdy zasób posiada pamięć lokalną.



Rysunek 5.1-1. Architektury systemów heterogenicznych

a) jednoprocesorowa, b) rozproszona z pamięcią współdzieloną, c) rozproszona z pamięcią lokalną



Rysunek 5.1-2. Implementacja systemu DRSOPC

a) system wbudowany z zewnętrznym sterownikiem rekonfiguracji, b) system dynamicznie samo-rekonfigurowalny SOPC

Implementując funkcje sprzętowe systemu SOC w układach ASIC projektanci byli ograniczeni do stałej funkcjonalności, bez możliwości ich zmian w przyszłości. W ostatnich latach, wraz z

upowszechnieniem się technologii układów FPGA, pojawiła się możliwość tworzenia tzw. systemów SOPC, które pozwalają na implementację całych systemów w jednym układzie razem z modułami procesorów uniwersalnych (Rys 5.1-2). Bardzo istotną zaletą jest możliwość dynamicznej rekonfiguracji systemów SOPC. Dynamiczna rekonfiguracja systemów implementowanych w układach FPGA w większości prezentowanych prac jest sterowana przez zewnętrzny sterownik (Rys. 5.1-2a). Taki system nie jest jednak zintegrowany w jednym układzie. Dopiero umieszczenie sterującego reprogramowaniem modułu w układzie FPGA, w postaci wyspecjalizowanego sterownika bądź procesora ogólnego przeznaczenia (Rys. 5.1-2b) pozwala na otrzymanie ***samo-rekonfigurowalnego*** systemu SOPC [BRK03]. Możliwość umieszczenia sterownika rekonfiguracji wewnątrz układu FPGA wymaga jednak specjalnych rozwiązań konstrukcyjnych. Najbardziej popularne współczesne układy FPGA (firmy Xilinx) posiadają specjalny port konfiguracji ICAP (ang. Internal Configuration Access Port), który umożliwia dynamiczną samorekonfigurację systemu przez wbudowany w układ sterownik [BRK03]. Więcej informacji na temat systemów samo-rekonfigurowalnych przedstawione zostanie w podrozdziale 5.4.1.

W układach Xilinx, do komunikacji z wbudowanymi procesorami (PowerPC), a także z procesorami w postaci modułów IP (MicroBlaze), wykorzystywana jest szyna CoreConnect (opracowana przez firmę IBM), natomiast do komunikacji poszczególnych bloków logicznych służą szyny Bus Macro (Rys. 5.1-2).

Podsumowując, ze względu na implementację rozproszone systemy heterogeniczne można podzielić na systemy:

- wieloukładowe (stosowane obecnie tylko w przypadku bardzo dużych systemów),
- SOC,
- SOPC,
- DRSOPC (Rys.5.1-2a), w tym SRSOPC (Rys.5.1-2b).

W niniejszej rozprawie zaprezentowane zostaną metody automatycznej syntezy systemów wieloprocesorowych SOPC i DRSOPC/SRSOPC.

5.2 Graf zadań – abstrakcyjny model systemu

Funkcjonalność projektowanych systemów wbudowanych przedstawiana jest najczęściej w postaci ***grafów zadań*** (ang. Task Graph – TG) [SW97]. Graf zadań (Rys. 5.2-1a) jest uznaną i powszechnie stosowaną reprezentacją specyfikacji na poziomie systemowym funkcji zorientowanych na przetwarzanie danych. Istnieją również rozszerzone wersje grafu zadań umożliwiające reprezentowanie rozejść warunkowych (Rys. 5.2-1b), pętli, czy wywołań podprogramów (Rys. 5.2-1c) [CV01, D05].

DEFINICJA 5.2-1 ZADANIE T_{IJ}.

Zadanie T_{ij} jest to pewien skończony zbiór obliczeń, które mają być wykonane przez projektowany system wbudowany.

Zadaniem może być zarówno pojedyncza operacja, jak np. dodawanie, ale również zbiór obliczeń, np. mnożenie macierzy, FFT. Wszystko zależy od tzw. „granulacji" zadań przyjętej przez projektanta systemu. Projektant może ręcznie wyznaczyć zadania tworząc specyfikację systemu, ale istnieją też metody automatyzujące proces generacji grafu zadań, np. na podstawie opisu systemu w języku SystemC, VHDL, C, itp. [VJ03].W systemie może być wiele zadań tego samego typu.

DEFINICJA 5.2-2 GRAF ZADAŃ G=(V, E, T).

Graf zadań G jest to skierowany i acykliczny graf (DAG) określony trzema parametrami: $G=\{V, E, \tau\}$, gdzie $V = \{v_i; i=1,..., n\}$ reprezentuje zbiór zadań, $E = \{e_{ij}; j=1,...,n\}$ reprezentuje zbiór połączeń komunikacyjnych, a τ jest okresem czasu, w którym muszą być wykonane wszystkie zadania grafu.

Wykonywanie zadań grafu G może być cykliczne (ang. **multi-rate** graph) [DJ98b, MSV00]. Jeśli model systemu składa się z kilku rozłącznych grafów zadań $G_1 \ldots G_g$ o różnych okresach, to w celu analizowania go jako jednego grafu można ustalić wspólny okres dla całego systemu jako najmniejsza wspólna wielokrotność wszystkich okresów $H=NWW(\tau_1,\ldots,\tau_g)$, gdzie g oznacza liczbę grafów opisujących system. Zatem dla systemu opisanego g grafami zadań konieczne jest utworzenie H/τ_i instancji każdego grafu G_i, gdzie i=1,…,g [MSV00].

Graf zadań może uwzględniać bardziej szczegółowe informacje o projektowanym systemie w postaci warunkowego wykonywania zadań. Niżej przedstawione zostaną najważniejsze pojęcia dotyczące takich grafów.

DEFINICJA 5.2-3 KRAWĘDŹ WARUNKOWA $E_{CIJ} \in E_C$.

Krawędź warunkowa $e_{cij} \in E_c$ jest to krawędź, łącząca węzły v_i i v_j, do której przypisany jest warunek c_j, gdzie v_i jest węzłem „rozwidlającym" (ang. branch fork task), oraz spełnione są warunki:

1) $\forall\, v_j, v_k : v_j, v_k$ są zadaniami wykonywanymi po spełnieniu warunku odpowiednio c_j i c_k, i krawędź e_{cij} łączy v_i z v_j a krawędź e_{ik} łączy v_i z v_k oraz $c_j \wedge c_k = 0$,

2) Jeśli prawdopodobieństwo aktywacji[4] warunku $c_j=P(c_j)$, wówczas zadanie v_j jest wykonywane pod warunkiem c_j z prawdopodobieństwem $P(c_j)$ i $c_j \in C_i=\{c_0,\ldots,c_n\}$, gdzie C_i jest zbiorem wszystkich warunków przypisanych do każdej krawędzi warunkowej wychodzącej od v_i, i $\sum_{k=0}^{n} P(c_k) = 1$.

Pierwszy warunek oznacza, że warunki, które są przypisane do krawędzi warunkowych wychodzących od jednego węzła v_i (branch fork task) muszą się wzajemnie wykluczać. Drugi warunek oznacza, że warunki te muszą być „kompletne", tzn. zawsze musi być spełniony jeden z warunków.

DEFINICJA 5.2-4 KRAWĘDŹ PROSTA $E_{IJ} \in E_S$.

Krawędź prosta $e_{ij} \in E_s$ jest to krawędź, dla której nie przypisano warunku.

Krawędź prosta jest to krawędź, dla której $P(c_k)= P(c_{ij})=1$. Czyli jest to szczególny przypadek krawędzi warunkowej, dla której warunek jest zawsze spełniony [SK03]. Krawędzie warunkowe będą oznaczane w grafie liniami przerywanymi, a krawędzie proste liniami ciągłymi.

DEFINICJA 5.2-5 WARUNKOWY GRAF ZADAŃ CTG=(V,E_S, E_C, T) [WHE03].

Warunkowy graf zadań *CTG* jest to skierowany i acykliczny graf, określony czterema parametrami: $CTG=\{V, E_s, E_c, \tau\}$, gdzie $V = \{v_i; i=1,\ldots, n\}$ reprezentuje grupę zadań, $E_s = \{e_{sij}; j=1,\ldots,n\}$ oznacza zbiór krawędzi prostych, $E_c = \{e_{ckl}; k,l=1,\ldots,n\}$ oznacza zbiór krawędzi warunkowych, τ jest okresem, w którym muszą być wykonane wszystkie zadania grafu, $E_s \cap E_c = \varnothing$ i $E_s \cup E_c = E$.

Zakłada się, że od jednego węzła v_{fork} (*branch fork task*) może wychodzić dowolna liczba krawędzi warunkowych, ale wszystkie warunki muszą być zakończone w jednym wspólnym węźle łączącym v_{join} (ang. branch join task). Warunki w ogólnym przypadku mogą być zagnieżdżone.

DEFINICJA 5.2-6 ŚCIEŻKA WARUNKOWA S_W.

Ścieżka warunkowa S_{Wj} jest to ścieżka w grafie CTG rozpoczynająca się od węzła „rozwidlającego" v_i, zawierająca krawędź warunkową e_{cij} i kończąca się węzłem łączącym v_{join}. Węzeł v_{join} łączy wszystkie ścieżki warunkowe rozpoczynające się węzłem v_i.

Zastosowanie opisu projektowanego systemu w postaci warunkowego grafu zadań pozwala na uwzględnienie bardziej szczegółowych informacji o systemie. Informacje te pozwalają na identyfikację w grafie CTG wszystkich zadań, które się wzajemnie wykluczają, w celu poprawy

[4] Prawdopodobieństwo to może być określone np. poprzez znajomość częstości wykonania zadań odpowiadających węzłom, do których wchodzi krawędź e_{cij}.

parametrów jakościowych projektowanego systemu. Zadania wzajemnie się wykluczające (Def. 5.2-7) nie muszą być wszystkie jednocześnie obecne w układzie, dzięki czemu można zmniejszyć powierzchnię projektowanego systemu.

Niech E_c oznacza zbiór krawędzi warunkowych w grafie zadań, a S_{Wm} i S_{Wn} oznaczają ścieżki w grafie od wspólnego węzła „rozwidlającego", do węzła łączącego te ścieżki warunkowe.

DEFINICJA 5.2-7 ZADANIA WZAJEMNIE SIĘ WYKLUCZAJĄCE (ZWW).

Zadania v_i i v_j są wzajemnie się wykluczające, jeśli $\exists S_{Wm}$ i $\exists S_{Wn}$ i $\exists e_{cm} \in E_c$ i $\exists e_{cn} \in E_c$: $c_m \wedge c_n = 0$ i $e_{cm} \in S_{Wm}$ i $e_{cn} \in S_{Wn}$ i $v_i \in S_{Wm}$ i $v_j \in S_{Wn}$.

Zadania wzajemnie się wykluczające znajdują się na ścieżkach rozpoczynających się od różnych krawędzi warunkowych, wychodzących od jednego wspólnego węzła i kończących się wspólnym węzłem łączącym. Zadania wykonywane na zagnieżdżonej ścieżce warunkowej mogą się wzajemnie wykluczać z zadaniami znajdującymi się na ścieżkach od najbliższego wspólnego węzła „rozwidlającego", ale także z zadaniami na ścieżkach warunkowych wyższych w hierarchii, jeśli tylko iloczyn warunków decydujących o wykonaniu zadań na ścieżkach warunkowych (zawierających potencjalne zadania wykluczające się nawzajem) jest równy 0.

DEFINICJA 5.2-8 PROCES KOMUNIKACYJNY C_{IJ}.

Z każdą skierowaną krawędzią e_{ij} grafu G związany jest proces komunikacyjny c_{ij}, który oznacza transmisję pomiędzy zadaniami v_i i v_j.

Z każdą transmisją związany jest czas transmisji danych przez łącze komunikacyjne określonego typu. Zadanie v_j może się rozpocząć dopiero po zakończeniu wykonywania zadania v_i i po zakończeniu transmisji c_{ij}.

DEFINICJA 5.2-9 WAGA KRAWĘDZI D_{IJ}.

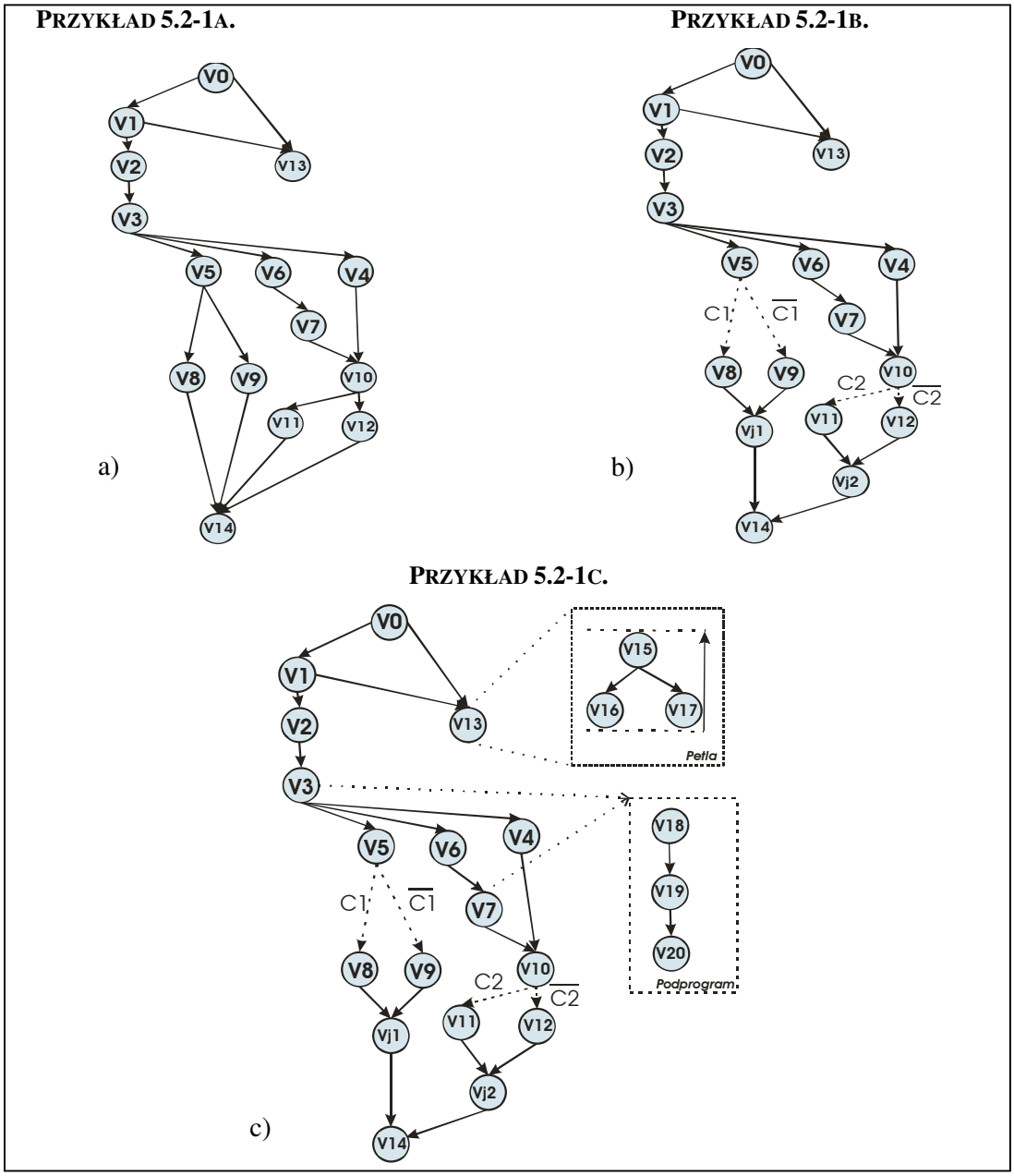
Z każdą krawędzią e_{ij} grafu G związana jest waga d_{ij} określająca ilość przesłanych danych pomiędzy komunikującymi się zadaniami v_i i v_j.

Bardzo istotnym pojęciem przy analizie grafów zadań jest ścieżka krytyczna. Dlatego poniżej przedstawiono jej definicję:

DEFINICJA 5.2-10 ŚCIEŻKA KRYTYCZNA SK.

Ścieżką krytyczną *SK* w grafie zadań *G* nazywana jest ścieżka o najdłuższym sumarycznym czasie obliczeń wszystkich zadań, którym odpowiadają węzły należące do *SK*, i czasów transmisji, którym odpowiadają krawędzie należące do *SK*, spośród wszystkich ścieżek w grafie.

W rozprawie przedstawione zostaną metody kosyntezy dla dwóch typów grafów: ***grafów zadań*** i ***warunkowych grafów zadań***. Zadanie v_{fork} może posiadać jako swoje następniki inne zadania sprawdzające warunek (warunki zagnieżdżone). Przykład 5.2-1a pokazuje graf TG, a 5.2-1b graf CTG, gdzie przerywaną linią oznaczone są krawędzie warunkowe. W grafie z Rys.5.2-1a nie ma możliwości optymalizacji z wykorzystaniem informacji o wykluczających się zadaniach. Natomiast graf z Rys. 5.2-1b zawiera informacje o zadaniach wykonywanych warunkowo, które mogą się nawzajem wykluczać (zadania *v8* i *v9* oraz *v11* i *v12*). Zadania takie przydzielone do jednego zasobu mogą być alternatywne do tych samych zasobów i mogą zajmować tę samą powierzchnię układu pozwalając na poprawę parametrów jakościowych systemu wbudowanego i uzyskanie tym samym tańszego systemu.



Rysunek 5.2. Reprezentacja systemu: a) TG, b) CTG, c) hierarchiczny graf zadań.

Dodatkowo w przykładzie 5.2-1c oprócz możliwości opisania rozejść warunkowych w grafie pokazano także, że możliwe jest opisanie za pomocą hierarchicznego grafu zadań pętli oraz wywołań podprogramów [CV01]. Taki graf można rozdzielić na oddzielne grafy dla każdej pętli i podprogramu. Pętle i podprogramy reprezentowane są wówczas jako grafy TG lub CTG. Pomijane są natomiast dodatkowe informacje związane ze sterowaniem (od grafu „głównego" w hierarchii do grafów reprezentujących podprogramy, czy pętle).

5.3 Zasoby systemowe

Zakłada się, że istnieje biblioteka komponentów sprzętowych i softwarowych, w której określone są wszystkie niezbędne parametry opisujące możliwe implementacje zadań (parametry czasowe, powierzchnia, zajętość pamięci, itp.). W bibliotece komponentów (*PE – Processing Elements*) systemu SOPC znajdują się trzy rodzaje zasobów: rdzenie procesorów uniwersalnych (*ang. General Purpose Processor - GPP*), komponenty sprzętowe (*ang. Virtual Component - VC*) i łącza komunikacyjne (*ang. Communication Link - CL*). Zdefiniowane są parametry jednego lub więcej typów procesorów uniwersalnych implementowanych w postaci modułów IP lub rdzenia wbudowanego oraz parametry komponentów sprzętowych dla FPGA. Zdefiniowane są również parametry łączy komunikacyjnych przenoszących dane pomiędzy komponentami. Przykładowa biblioteka zasobów dla grafu zadań z Rys. 5.2 przedstawiona jest w tabeli 5.3-1. Biblioteka ta zawiera po jednym typie GPP i CL oraz po jednym typie VC dla każdego zadania.

	Powierzchnia FPGA: 2000 CLB, tr=0.86 µs / CLB			
	GPP (Su=203 CLB)		**VC**	
Zadanie	$t_i(v_j)$ [µs]	$C_i(v_j)$	$t_i(v_j)$ [µs]	$S_i(v_j)$ [CLB]
T_0	434	30	89	158
T_1	817	27	94	227
T_2	811	23	59	92
T_3	747	7	57	80
T_4	444	40	54	175
T_5	1020	43	33	411
T_6	755	34	1	319
T_7	335	86	22	283
T_8	250	57	5	482
T_9	655	66	61	152
T_{10}	382	32	94	224
T_{11}	408	24	30	184
T_{12}	945	29	21	109
T_{13}	190	32	26	167
T_{14}	881	8	40	181

CL	SC [CLB]	b	dostępność
B1	10	9kB/µs	GPP, wszystkie VC (HW)

TABELA 5.3-1. BIBLIOTEKA KOMPONENTÓW SPRZĘTOWYCH I PROGRAMOWYCH

5.3.1 Procesory uniwersalne

Rdzenie procesorów ogólnego przeznaczenia (GPP) w systemach SOPC są implementowane w układzie FPGA w postaci modułu IP, np. Nios II firmy Altera, MikroBlaze firmy Xilinx lub w formie rdzenia procesora na stałe wbudowanego w układ FPGA np. PowerPC w układach Virtex II Pro/4/5 firmy Xilinx. Wszystkie zadania przydzielone do jednego GPP wykonywane są sekwencyjnie, tzn. w danym czasie może być wykonywane tylko jedno zadanie.

Każdy moduł GPP_j ma określoną ***powierzchnię*** (Su_j), którą zajmuje w układzie FPGA (w CLB). Natomiast dla wbudowanego rdzenia procesora przyjmuje się powierzchnię Su_j=0. Określa się ***czas wykonania zadania***[5] przez procesor $t_j(v_i)$. Parametr ***memory load*** (CM_j) oznacza ilość pamięci wykorzystywaną przez zadanie v_i wykonywane przez GPP. Jest to pamięć rezerwowana dla danych i instrukcji zadania v_i. Wartości $CM_j(v_i)$ i $t_j(v_i)$ mogą być znane lub mogą być oszacowane metodami przybliżonymi [YW98, HE98].

5.3.2 Komponenty sprzętowe

Niektóre zadania mogą być realizowane w postaci komponentów sprzętowych *VC* w FPGA. Dla komponentów sprzętowych określa się następujące parametry: powierzchnia *VC* (Su_j) [CLB], czas wykonania zadania przez *VC* $t_j(v_i)$. W systemie DRSOPC powierzchnia zajmowana przez komponent sprzętowy może być w trakcie działania aplikacji wykorzystana przez inne zadanie po reprogramowaniu układu FPGA. Wszystkie zadania implementowane jako komponenty sprzętowe mogą być wykonywane równolegle.

5.3.3 Łącza komunikacyjne

Komunikacja pomiędzy procesorami i komponentami sprzętowymi odbywa się przez łącza komunikacyjne *CL*. Charakteryzują się one następującymi parametrami: powierzchnia zajmowana przez łącze SC_j [CLB], przepustowość b_j, dostępność łącza dla poszczególnych typów zasobów.

DEFINICJA 5.3-1 CZAS TRANSMISJI $T_K(V_I,V_J)$.

Czas transmisji przez łącze komunikacyjne CL_k pomiędzy zadaniami v_i i v_j określony jest następującym wzorem:

$$t_k(v_i,v_j) = \begin{cases} \left\lceil \frac{d_{ij}}{b_k} \right\rceil & \text{- gdy zadania } v_i \text{ i } v_j \text{ są przydzielone do różnych zasobów,} \\ 0 & \text{- gdy zadania są przydzielone do tego samego zasobu.} \end{cases} \quad (5.3.1)$$

[5] Jest to ilość czasu, którą potrzebuje zasób na wykonanie zadania

Transfer danych jest wykonywany niezależnie od przetwarzania danych. Zakłada się, że transmisja jest wykonywana w tle, tzn. łącza komunikacyjne są wyposażone w niezależne układy transmisji, np. układy DMA. Dane przesyłane pomiędzy komponentami układu FPGA są pamiętane w buforach [OJ97], a czasy transmisji do pamięci są pomijane. W przypadku współdzielonej pamięci konieczne są dokładne informacje o czasach dostępu do pamięci przez poszczególne zadania przydzielone do zasobów [D02]. Często w przypadku braku takich informacji przyjmuje się, że dostęp do pamięci jest ciągły przez cały czas aktywności zadania. W rozprawie zakłada się, że każdy zasób posiada pamięć lokalną, a więc nie ma problemu szeregowania dostępu do pamięci. Przyjęto, że w architekturze projektowanego systemu można wykorzystać różne typy łączy komunikacyjnych realizowanych jako magistrale [OJ97].

5.4 Systemy dynamicznie rekonfigurowalne

Dynamiczna rekonfiguracja systemów wbudowanych, implementowanych we współczesnych układach FPGA o szybkim czasie programowania, pozwala na wielokrotne wykorzystanie tych samych fragmentów układu FPGA dla różnych funkcjonalności implementowanych sprzętowo. Dzięki temu w układzie FPGA można zaimplementować więcej zadań realizowanych sprzętowo (a co za tym idzie wykonywanych zwykle szybciej od ich realizacji softwarowej), niż bez wykorzystania dynamicznej rekonfiguracji. W rozprawie przyjęto technologię częściowo reprogramowalnych układów FPGA, które są coraz bardziej popularne (działanie aplikacji nie jest przerywane na czas reprogramowania).

Głównym problemem dynamicznej rekonfiguracji jest czas reprogramowania, który ma wpływ na szybkość całego systemu (w szczególności, jeśli czas reprogramowania jest znacznie większy niż czas wykonania zadań). Fakt ten może mieć istotny wpływ na to, iż nie dla wszystkich systemów dynamiczna rekonfiguracja pozwala na uzyskanie szybszych systemów niż bez rekonfiguracji, np. w przypadku, gdy nie da się zrównoleglić rekonfiguracji z obliczeniami. W dalszej części rozprawy zbadane zostaną klasy systemów SOPC, dla których dynamiczna rekonfiguracja przyspiesza szybkość systemu, jak i takie, dla których może nie przynosić zysku.

Współczesne, częściowo reprogramowalne układy FPGA mają zwykle pewne ograniczenia dotyczące rozmieszczenia bloków (fragmentów układu), które mogą być reprogramowane i czasu, w którym blok może być reprogramowany (dla potrzeb badań wykorzystano technologię układów FPGA firmy Xilinx, ponieważ częściowo reprogramowalne układy tej firmy są najbardziej rozpowszechnione). Na przykład w układach firmy Xilinx istnieją dwie metody częściowej rekonfiguracji: modułowa i różnicowa [X04]. W metodzie różnicowej reprogramowane są dowolne komórki logiczne. Ta metoda zalecana jest jedynie w przypadkach, gdy reprogramowane są niewielkie

fragmenty FPGA. Dla systemów wbudowanych bardziej odpowiednia jest rekonfiguracja modułowa, w której reprogramowane są całe bloki komórek FPGA. Narzuca to pewne ograniczenia powodujące, że systemów zaprojektowanych przy pomocy dotychczasowych metod nie da się zaimplementować.

DEFINICJA 5.4-1 JEDNOSTKA REKONFIGURACJI - RAMKA.
Ramka jest to najmniejsza jednostka rekonfiguracji w układzie częściowo reprogramowalnym FPGA.

DEFINICJA 5.4-2 DYNAMICZNIE REKONFIGUROWALNY SEKTOR RS.
Dynamicznie rekonfigurowalny sektor *RS* jest to grupa N sąsiednich ramek, gdzie N>0.

Niech N oznacza liczbę sąsiednich ramek, H oznacza wysokość ramki w [CLB] i W oznacza szerokość ramki w [CLB]:

DEFINICJA 5.4-3 POWIERZCHNIA REPROGRAMOWANEGO SEKTORA S_{RS}.
Powierzchnia reprogramowanego sektora zdefiniowana jest następującym wzorem:

$$S_{RS}=N*H*W \tag{5.4.1}$$

WARUNEK 5.4-1 DOPUSZCZALNE WIELKOŚCI SEKTORÓW RS:
W układach firmy Xilinx reprogramowalne sektory mają wysokość H_{Rs} równą wysokości układu FPGA, natomiast szerokość W_{Rs} jest wielokrotnością 4 klasterów (1 lub 2 CLB w zależności od układu FPGA).

Rozmieszczenie sektorów jest ściśle liniowe. Komórki znajdujące się w różnych sektorach komunikują się przez specjalne szyny „bus macro". Sektor może być reprogramowany tylko w czasie, gdy nie ma żadnej transmisji danych przechodzących przez ten sektor. Zatem musi być zawsze spełniony następujący warunek:

WARUNEK 5.4-2 ZASADA POPRAWNOŚCI FIZYCZNEJ IMPLEMENTACJI SRSOPC:
Niech $i=1,\ldots,n$ oznacza numer sektora *RS* w układzie FPGA, j oznacza kolejną konfigurację sektora RS_i, $R_i=\{r_{1,\ldots,} r_j\}$ oznacza zbiór wszystkich przedziałów czasu, w których sektor RS_i jest reprogramowany, gdzie $r_j=<r_p, r_k>$, $T_i=\{t_1,\ldots, t_m\}$ oznacza zbiór wszystkich przedziałów czasów , w których wykonywane są transmisje przechodzące przez sektor RS_i, gdzie $t_m=<t_{pm},t_{km}>$. Musi zachodzić następujący warunek: $\forall r_i \in R_i, t_i \in T_i : r_i \cap t_i = \phi$.

Jest to warunek konieczny implementacji zsyntezowanego systemu we współczesnych częściowo reprogramowalnych układach FPGA przy zastosowaniu metody rekonfiguracji modułowej. Starsze metody kosyntezy nie uwzględniały powyższego warunku. Układy FPGA mogą być reprogramowane

przez zewnętrzny sterownik lub wewnętrznie poprzez specjalny port ICAP, który umożliwia implementację w postaci SRSOPC.

Musi być spełniony jeszcze warunek dotyczący dopuszczalnych wielkości sektorów, aby każdy z dostępnych komponentów *VC* mógł być alokowany w przynajmniej jednym z dostępnych sektorów.

WARUNEK 5.4-3 POWIERZCHNIA NAJWIĘKSZEGO SEKTORA

$$\exists RS: \ max\,(Su_i) \leq S_{RS} \tag{5.4.2}$$

Warunek ten oznacza, że powierzchnia największego sektora nie może być mniejsza od największej powierzchni spośród wszystkich komponentów dostępnych w bibliotece.

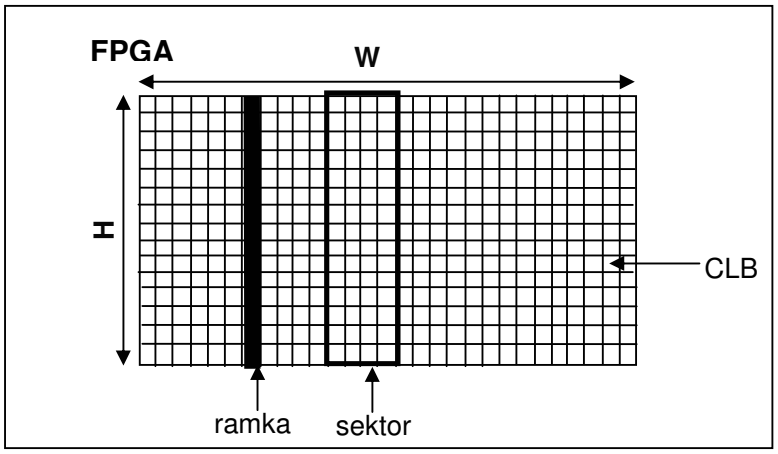
Powyższe rozważania zostały zilustrowane na Rysunku 5.4-1. Moduły sprzętowe (*VC*) są rozmieszczone w sektorach *RS*. Sektor może pomieścić w danej chwili jeden lub kilka *VC* wykonujących zadania równolegle. Po zakończeniu wszystkich zadań, sektor jest reprogramowany w celu alokacji kolejnych modułów *VC*. Rozmiary sektorów są określane na podstawie wielkości modułów *VC* oraz wymagań określonych w warunku 5.4-1 i 5.4-3.

DEFINICJA 5.4-4 CZAS REPROGRAMOWANIA SEKTORA.

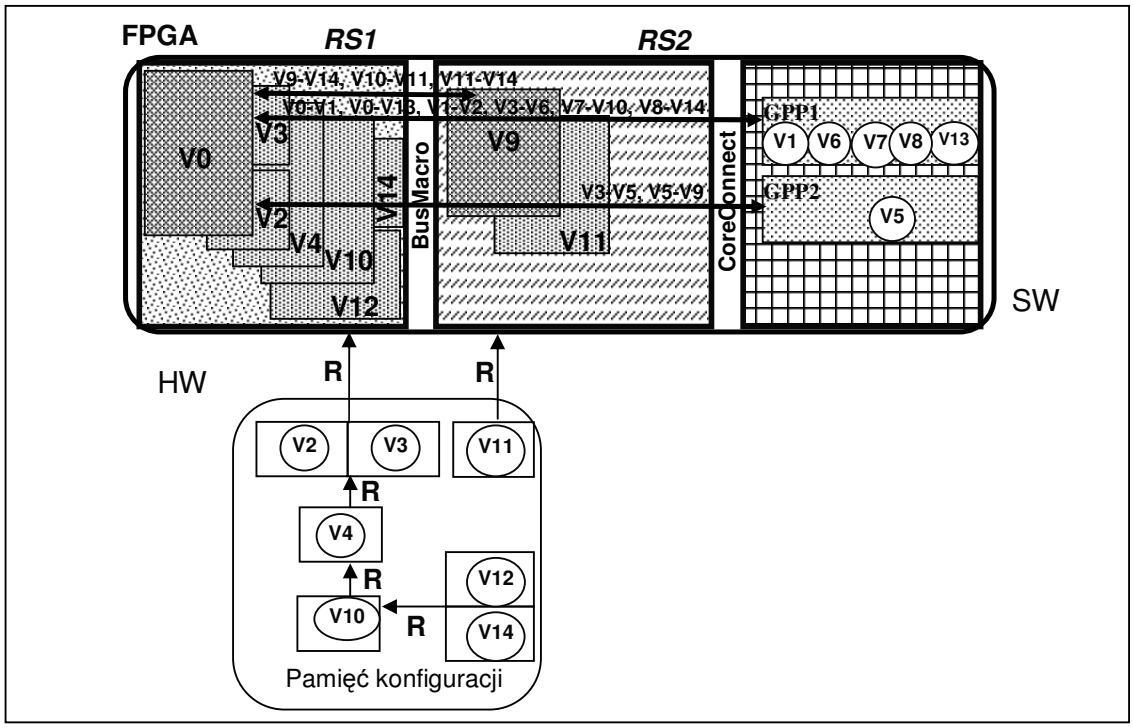
Czas reprogramowania sektora jest określony na podstawie następującego wzoru:

$$t_{res} = t_{rCLB} * S_{RS} \tag{5.4.3}$$

gdzie t_{rCLB} - czas reprogramowania jednej komórki logicznej (na podstawie danych katalogowych).



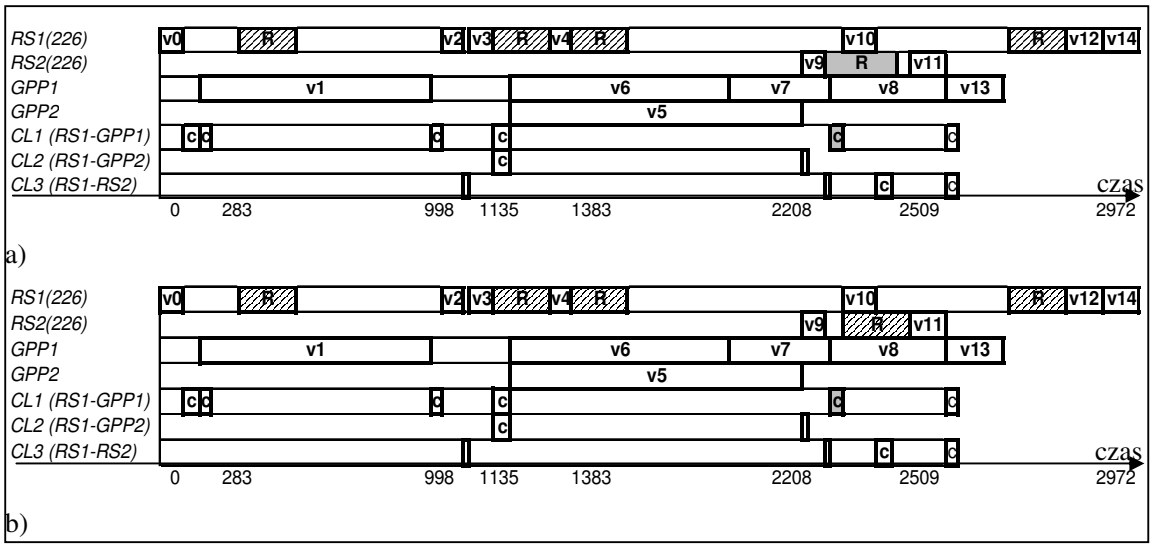
Rysunek 5.4-1. Częściowo reprogramowalny układ FPGA



Rysunek 5.4-2 Rozmieszczenie sektorów w układzie FPGA dla przykładowego systemu opisanego grafem z Rys. 5.2.

PRZYKŁAD 5.4-1.

Na Rys. 5.4-2 pokazano przykładowe rozmieszczenie dynamicznie rekonfigurowanych sektorów *RS1-RS2*, w wyniku kosyntezy systemu specyfikowanego grafem z przykładu 5.2-1, o parametrach podanych w Tab. 5.3-1. Strzałkami zaznaczono wszystkie transmisje pomiędzy zadaniami. Sektory rozmieszczone są zgodnie z zasadami określonymi w warunkach 5.4-1, 5.4-2 i 5.4-3. Pozostałe wolne miejsce w układzie zajmują procesory ogólnego przeznaczenia *GPP1* i *GPP2*. Ze względu na to, że sektor *RS1* jest wielokrotnie reprogramowany, najkorzystniejsze jego położenie jest w skrajnej części układu, takie, aby nie przechodziło przez niego wiele transmisji. Na rysunku zaznaczono również wszystkie transmisje danych. Pomiędzy sektorami w czasie działania aplikacji ma miejsce wiele transmisji, natomiast praktycznie tylko jedna, pomiędzy zadaniami *v7-v10,* może nakładać się w czasie z reprogramowaniem sektora *RS2*. Wówczas może być konieczna modyfikacja uszeregowania transmisji i rekonfiguracji. Na Rys. 5.4-3a pokazano błędne uszeregowanie rekonfiguracji, gdzie rekonfiguracja sektora *RS2* nakłada się w czasie z transmisją danych przechodzącą przez ten sektor. Uszeregowanie takie nie jest zgodne z warunkiem 5.4-2. Na rysunku 5.4-3b pokazano już prawidłowe uszeregowanie transmisji i rekonfiguracji. W tym przypadku prawidłowe uszeregowanie nie zmieni całkowitego czasu wykonania wszystkich zadań sytemu wbudowanego. Więcej na temat metody rozmieszczenia sektorów będzie przedstawione w dalszej części rozprawy.



Rysunek 5.4-3 Wykres Gantta demonstrujący a) uszeregowanie niezgodne z warunkiem 5.4-2, b) prawidłowe uszeregowanie transmisji i rekonfiguracji.

5.4.1 Systemy samo-rekonfigurowalne

W większości istniejących prac zakłada się, że sterowanie rekonfiguracją systemów wbudowanych odbywa się poprzez zewnętrzny układ. Tymczasem wbudowany system SOPC powinien być zintegrowany w całości w jednym układzie FPGA. Sterowanie rekonfiguracją może odbywać się poprzez wyspecjalizowany sterownik zaimplementowany w układzie FPGA, bądź poprzez wbudowany procesor. Taki sposób rekonfiguracji jest możliwy dzięki specjalnemu portowi ICAP, w który wyposażone są współczesne układy FPGA firmy Xilinx.

Zakłada się, że istnieje dodatkowy procesor wbudowany *GPPr*, który zajmuje się tylko reprogramowaniem sektorów. Implementacja dynamicznie rekonfigurowalnych systemów SOPC opiera się na metodzie tworzenia systemów dynamicznie samorekonfigurowalnych SRP zaproponowanej przez firmę Xilinx [BRK03]. Sterowaniem rekonfiguracją zajmuje się osobny procesor wbudowany w FPGA w postaci modułu IP (np. MicroBlaze) lub w formie rdzenia sprzętowego PowerPC. Wszystkie konfiguracje, które mają być programowane w trakcie działania systemu, są przechowywane w pamięci cache. Reprogramowany może być w danym przedziale czasu tylko jeden fragment układu (sektor *RS*) zgodnie z omówionymi wcześniej zasadami. Przyjęto tryb programowania SelectMAP 32-bitowy, który jest obecnie najszybszą metodą reprogramowania. Po zsyntezowaniu procesów implementowanych sprzętowo, generowane są bity konfiguracyjne (bitstream) dla początkowej implementacji systemu oraz bity konfiguracyjne dla poszczególnych konfiguracji wszystkich modułów. Bity konfiguracyjne są generowane przez standardowe narzędzia firmy Xilinx zgodnie z zasadami projektowania modułowego (Modular Design Flow) [X04]. W

momencie startu systemu ładowana jest do układu FPGA konfiguracja początkowa, a następnie system jest dynamicznie rekonfigurowany przez ładowanie ciągów bitów dla poszczególnych modułów, które mają być kolejno reprogramowane. Procedura sterowania rekonfiguracją, wykonywana przez procesor *GPPr*, przedstawiona jest na Rys. 5.4-3.

Algorytm 5.4-1

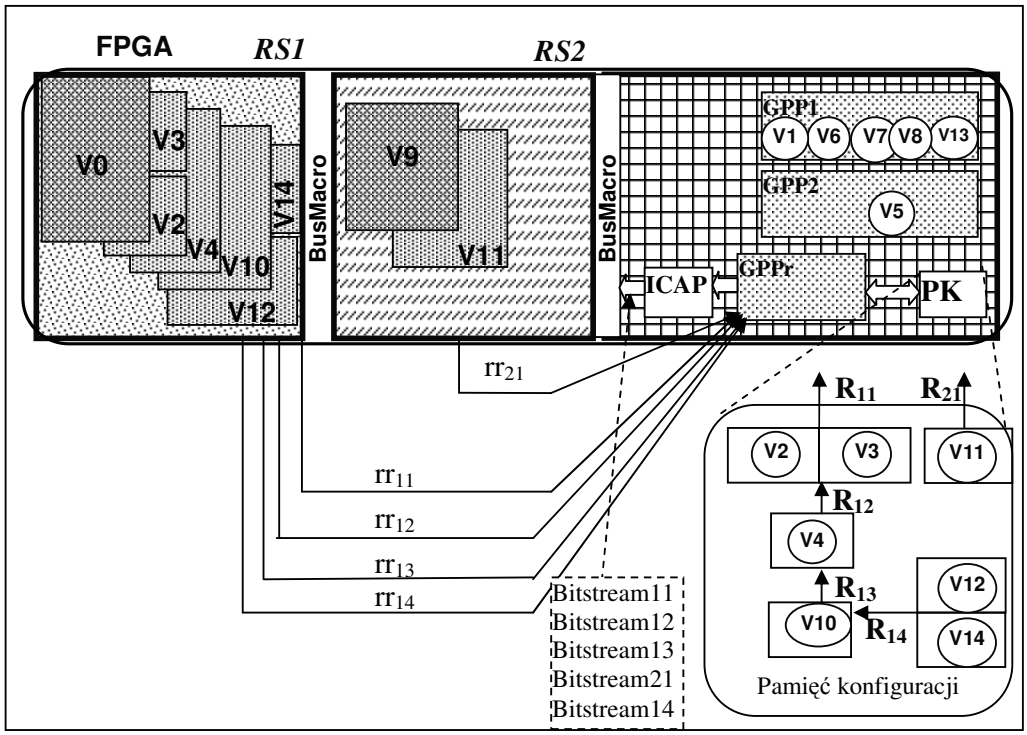
Załaduj początkową konfigurację A_i do FPGA;
wykonuj{
czekaj *dopóki rr_i=true;*
Pobierz następny bitstream dla RS_i z pamięci cache;
Rekonfiguruj RS_i nowymi bitami konfiguracyjnymi;
}dopóki *nie ma oczekujących bitów konfiguracyjnych w pamięci cache;*

Rysunek 5.4-3 Procedura sterowania rekonfiguracją.

Procesor sterujący *GPPr* oczekuje na sygnał rekonfiguracji (rr_i) od sektora RS_i. Po wykonaniu wszystkich zadań w sektorze, ostatnie zakończone zadanie lub ostatnia transmisja aktywuje i przesyła sygnał rr_i do *GPPr*. Na podstawie informacji o ostatnio wykonywanym zadaniu procesor pobiera z odpowiedniego adresu pamięci cache kolejną konfigurację do zaprogramowania. Następnie, na podstawie informacji o położeniu sektora w układzie, system wbudowany jest dynamicznie rekonfigurowany kolejnymi bitami konfiguracyjnymi dla sektora *RS* odpowiadającego wysłanemu sygnałowi rr_i.

PRZYKŁAD 5.4-2.

Rys. 5.4-4 jest uzupełnieniem Przykładu 5.4-1. Na rysunku zaznaczono najważniejsze moduły wymagane do sterowania rekonfiguracją wewnątrz jednego układu FPGA. Są to: procesor ogólnego przeznaczenia *GPPr*, który zajmuje się wyłącznie sterowaniem reprogramowaniem sektorów w FPGA; pamięć konfiguracji (przechowująca wszystkie konfiguracje sektorów) i port ICAP, który umożliwia wewnętrzną rekonfigurację. W tym momencie dynamicznie rekonfigurowany system SRSOPC jest niezależny od zewnętrznych układów. Zaznaczona została również schematycznie zasada działania wbudowanego sterownika rekonfiguracji. Na początku, po wykonaniu zadania *v0*, zadanie to wysyła do sterownika *GPPr* sygnał rr_{11} o konieczności rekonfiguracji sektora, sterownik pobiera z pamięci nową konfigurację sektora i poprzez port ICAP przesyła bitstream z konfiguracją do *RS1*. Następnie podobny przebieg ma miejsce dla kolejnych sygnałów od zadań w sektorach: rr_{12}, rr_{13}, rr_{21} i na końcu rr_{14}.



Rysunek 5.4-4 Przykładowy system SRSOPC

5.5 Kryteria optymalizacji

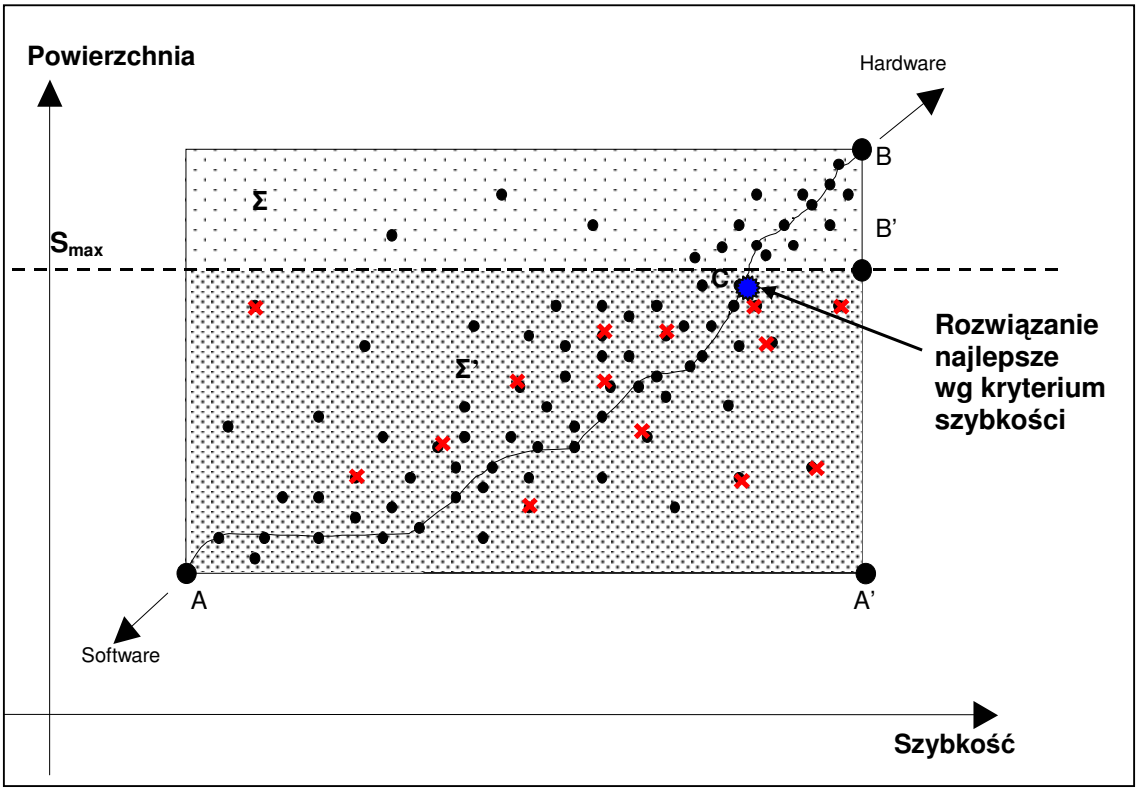
Jak już wspomniano we wstępie, przy projektowaniu systemów wbudowanych można uwzględniać różne kryteria optymalizacji. Najczęściej minimalizowany jest koszt systemu, czyli powierzchnia zajmowana przez system w układzie FPGA, lub maksymalizowana jest szybkość projektowanego systemu. Minimalizuje się również pobór mocy lub maksymalizuje niezawodność systemu.

Celem metody przedstawionej w tej pracy jest znalezienie jak najszybszej, dynamicznie rekonfigurowalnej architektury spełniającej wszystkie wymagania funkcjonalne określone w grafie zadań i zadane ograniczenia powierzchni (określonej przez wielkość docelowego układu FPGA). Architektura docelowa musi ponadto gwarantować możliwość fizycznej implementacji systemu w układzie, a więc uwzględniać wszystkie ograniczenia stawiane przez producentów układów zgodnie z warunkami 5.4-1 i 5.4-2.

Niech architektura systemu składa się z procesorów GPP_i (i=1,…, p), procesora sterującego rekonfiguracją GPP_r o powierzchni Su_r, sektorów RS_i (i=p+1,…, r) i łączy komunikacyjnych CL_j (j=1,…, c). Powierzchnia całkowita systemu jest zdefiniowana następująco:

$$S = \sum_{i=1}^{p} Su_i + \sum_{i=p+1}^{r} S_{RS\,i} + \sum_{j=1}^{c} Sc_j + Su_r \qquad (5.5.1)$$

Rys.5.5-1 pokazuje przestrzeń możliwych rozwiązań znalezionych w procesie kosyntezy systemów SOPC dla maksymalizacji szybkości projektowanego systemu. W przestrzeni Σ zawarte są wszystkie możliwe rozwiązania. Punkt *A* pokazuje rozwiązanie o teoretycznie najmniejszej powierzchni, a *B* najszybsze, jeśli nie uwzględnia się ograniczeń systemu.



Rysunek 5.5-1 Przestrzeń możliwych rozwiązań w procesie maksymalizacji szybkości systemu SOPC

Oprócz samego ograniczenia powierzchni S_{max} docelowego układu FPGA można narzucić projektowanej architekturze dodatkowe ograniczenia. W opracowanym algorytmie kosyntezy można określić również minimalną szybkość λ_{min}, z jaką ma pracować system. Jeśli $\lambda < \lambda_{min}$, wówczas należy zwiększyć wartość S_{max} (czyli wybrać układ o większej powierzchni). Przestrzeń poszukiwanych rozwiązań ogranicza się do pola Σ'. Maksymalizując szybkość systemu startuje się w rozwiązaniu początkowym od rozwiązania najwolniejszego *A* i zajmującego najmniejszą powierzchnię układu FPGA (przeważnie takiego, w którym zadania wykonywane są tylko softwarowo, a więc przez *GPP*). Punkty leżące na odcinku *A-A'* są optymalne pod względem zajmowanej powierzchni, a punkty *A'-B'* optymalne ze względu na szybkość systemu. W rzeczywistości uzyskanie rozwiązań w tych punktach jest zwykle niemożliwe, bo minimalizacja kosztu (powierzchni systemu) wyklucza się z maksymalizacją szybkości. Im architektura jest szybsza, tym jej powierzchnia jest większa. Wśród znalezionych rozwiązań, w przestrzeni Σ' może być wiele rozwiązań, które teoretycznie spełniają ograniczenia, ale w praktyce ich fizyczna realizacja w układzie będzie niemożliwa. Takie rozwiązania

również muszą być wykluczone (na rysunku zaznaczone krzyżykami). Biorąc pod uwagę w/w rozważania najlepsze rozwiązanie w przestrzeni możliwych rozwiązań dla kryterium maksymalizacji szybkości zaznaczono na Rys 5.5-1 jako punkt C.

6 ALGORYTMY KOSYNTEZY SYSTEMÓW SOPC

W tej części rozprawy przedstawiona zostanie propozycja algorytmu kosyntezy dla systemów SRSOPC maksymalizujących szybkość projektowanego systemu przy zadanym ograniczeniu na zajmowaną powierzchnię w układzie FPGA. Jak już zostało przedstawione w rozdziale 2, algorytmy kosyntezy można podzielić na konstrukcyjne i rafinacyjne. Algorytmy konstrukcyjne praktycznie nie są już rozwijane ze względu na ich skłonność do zatrzymywania się w lokalnych minimach optymalizowanych parametrów, a przez to uzyskiwane wyniki są dalekie od optymalnych. Obecnie, zdecydowana większość algorytmów kosyntezy to algorytmy rafinacyjne. Wiele z nich ma możliwość wydobywania się z lokalnych ekstremów, w przeciwieństwie do algorytmów konstrukcyjnych. Algorytmy rafinacyjne można podzielić na probabilistyczne i o bezpośrednim wyszukiwaniu. Te pierwsze (algorytmy genetyczne, symulowane wyżarzanie itp.) były już intensywnie badane, ale czas obliczeń tych algorytmów jest dosyć duży. Ze względu na wady algorytmów konstrukcyjnych i rafinacyjnych probabilistycznych, algorytm kosyntezy systemów SRSOPC prezentowany w rozprawie będzie należał do klasy algorytmów rafinacyjnych o bezpośrednim wyszukiwaniu. Szkielet takiego algorytmu przedstawiono na Rys. 6-1.

Algorytm 6.1-1

Generuj rozwiązanie początkowe R^{Akt}*;*
Powtarzaj {
$R^{best}=R^{Akt}$*;*
zysk=0;
dopóki *((R'=Rafinacja(*R^{Akt}*))≠0)* ***wykonuj{***
zysk(R')=Jakość(R')-Jakość(R^{Akt}*);*
jeżeli*(zysk(R')>0)* ***to*** $R^{Akt}=R'$*;*
}
}dopóki *(zysk>0);*

Rysunek 6-1 Szkielet algorytmu rafinacyjnego o bezpośrednim wyszukiwaniu.

Rafinacja jest sterowana pewnym współczynnikiem zysku. Zysk jest to różnica w jakości dwóch porównywanych ze sobą rozwiązań. O jakości rozwiązania może decydować kilka parametrów opisujących projektowany system (np. jego szybkość, koszt). Algorytmy startują od rozwiązania początkowego, w każdym kroku rafinacji analizowane są modyfikacje aktualnego rozwiązania i do następnego kroku wybierane jest rozwiązanie dające największy zysk. Algorytm kończy działanie, gdy żadna z rozpatrywanych modyfikacji nie prowadzi do ulepszenia rozwiązania. Kluczowymi elementami decydującymi o skuteczności algorytmu rafinacyjnego są: wybór rozwiązania początkowego, stosowane metody rafinacji oraz stosowana miara zysku. Elementy te, odpowiednio

zdefiniowane, pozwalają na to, iż algorytm będzie zbieżny[6] (źle zaprojektowane algorytmy rafinacyjne mogą nie być zbieżne), oraz umożliwiają uzyskanie rozwiązań o dobrej jakości w dosyć krótkim czasie.

Punktem wyjścia do opracowania tego algorytmu będzie algorytm kosyntezy dla systemów SOC [D04]. Jest to również algorytm rafinacyjny mający zdolność wydobywania się z lokalnych minimów. Dla tego algorytmu uzyskano wyniki znacznie lepsze niż w metodach znanych z literatury. Zatem jest to algorytm już sprawdzony i istnieje duże prawdopodobieństwo, że analogiczny algorytm dla systemów SOPC będzie działał równie skutecznie. Nie jest jednak możliwe bezpośrednie zastosowanie go dla systemów SRSOPC, a także nie jest możliwa prosta modyfikacja poprzez uwzględnienie innych kryteriów optymalizacji. Wynika to m.in. z następujących różnic w projektowaniu systemów SOC i SOPC:

- celem kosyntezy dla SOC jest minimalizacja kosztu projektowanego systemu przy uwzględnieniu minimalnej szybkości, a dla systemów SOPC chcemy maksymalizować szybkość systemu przy zadanym maksymalnym koszcie (powierzchni układu FPGA),

- w systemach SOC (algorytm EWA) nie ma ograniczenia na koszt systemu, a w systemie SOPC istnieje ograniczenie powierzchni układu FPGA, które należy uwzględnić. Tym samym ograniczenie to może spowodować zmniejszenie ilości rozpatrywanych rozwiązań i w pewnych przypadkach nawet blokować rafinację – jest mniejsza swoboda w poszukiwaniu kolejnych rozwiązań,

- algorytm EWA nie uwzględnia specyficznych cech systemów SOPC i SRSOPC, takich jak: stały koszt systemu SOPC (koszt układu FPGA), podczas gdy w algorytmie EWA koszt zmienia się w zależności od architektury, możliwości rekonfigurowania części sprzętowej systemu, ograniczenia układów FPGA (warunki 5.4-1, 5.4-2), itp.

Wszystkie wyżej wymienione różnice powodują, że niemożliwa jest bezpośrednia adaptacja algorytmu dla innej technologii i innych wymagań. Nie wystarczy np. jedynie zamienić optymalizowany parametr (tzn. koszt na szybkość systemu). Algorytmy rafinacyjne dla SOC i dla SRSOPC będą różniły się w każdym elemencie, począwszy od wyboru rozwiązania początkowego. Algorytm EWA startuje od rozwiązania najszybszego, które jest jednocześnie najdroższe. W przypadku FPGA zwykle takie rozwiązania przekraczają dopuszczalną powierzchnię i nie powinny być rozpatrywane. Inne muszą być również same metody rafinacji uwzględniające wcześniej wspomniane cechy systemów SOPC i SRSOPC. Inaczej powinna być obliczana funkcja zysku uwzględniająca odmienne założenia i ograniczenia (tak, aby uwzględnić możliwość rafinacji w dalszych krokach). W algorytmie powinna pozostać zasada stosowania równie prostych metod modyfikacji (dodawanie i usuwanie zasobów) i takich, które powodują istotne zmiany architektury systemu oraz nie prowadzą do zatrzymywania się algorytmów w lokalnych minimach optymalizowanych parametrów.

[6] Algorytm rafinacyjny jest zbieżny, gdy nie wraca cyklicznie do wcześniej rozpatrywanych rozwiązań.

W celu uzyskania algorytmu kosyntezy dla systemów SRSOPC opracowano i zaimplementowano kolejno trzy wersje algorytmu, będące ewolucją zmierzającą do opracowania ostatecznej wersji algorytmu. Przyjęto następujący sposób postępowania:

1. **Opracowanie algorytmu dla systemów SOPC: COSYSOPC**. Etap ten ma na celu opracowanie skutecznych metod rafinacji dla systemów wbudowanych w jeden układ FPGA. Na tym etapie możliwe będzie sprawdzenie skuteczności wyboru rozwiązania początkowego, przyjętych metod rafinacji i funkcji kosztu poprzez porównanie z algorytmami dla systemów SOPC znanymi z literatury. Architektura systemu SOPC jest szczególnym przypadkiem architektury dynamicznie rekonfigurowalnej. Układ FPGA może być w całości reprogramowany, a w szczególnych przypadkach cały system mieści się w jednym układzie bez konieczności rekonfiguracji. Dlatego należy sądzić, że rafinacja dla systemu SOPC będzie również skuteczna dla DRSOPC.
2. **Opracowanie algorytmu dla systemów DRSOPC: COSEDYRES**. Algorytm ten będzie rozszerzeniem algorytmu COSYSOPC o możliwość dynamicznej rekonfiguracji. Opracowanie tego algorytmu i porównanie z algorytmem COSYSOPC pozwoli sprawdzić skuteczność wykorzystania dynamicznej rekonfiguracji w systemach DRSOPC. Algorytm będzie wykorzystywał technologię układów częściowo reprogramowalnych, zatem konieczne będzie uwzględnienie dodatkowych ograniczeń takich układów. Przede wszystkim konieczne jest uwzględnienie dynamicznej rekonfiguracji w metodach rafinacji i współczynniku zysku, czyli m.in. wpływu dodatkowego czasu reprogramowania na szybkość systemu, wielokrotnego wykorzystania tych samych powierzchni układu (konieczność uwzględnienia sektorów o różnych możliwych powierzchniach, itp.). Ponadto algorytm COSEDYRES powinien uwzględniać ograniczenia wyspecyfikowane w warunkach 5.4-1 i 5.4-2.
3. **Opracowanie algorytmu dla SRSOPC z uwzględnieniem zadań wzajemnie się wykluczających: COSEDYRES-CTG**. Będzie to ostateczna wersja algorytmu kosyntezy ***COSEDYRES***. W praktyce wiele rzeczywistych systemów działa w ten sposób, że niektóre grupy zadań są alternatywne. Wykorzystanie informacji o wykluczających się zadaniach może istotnie zwiększyć możliwość optymalizacji w kosyntezie systemów dynamicznie rekonfigurowalnych. Przydzielenie takich zadań do jednego zasobu może zmniejszyć koszt lub zwiększyć szybkość systemu (przyporządkowanie ich do jednego sektora może zmniejszyć koszt systemu, a poprzez implementację sprzętową większej liczby zadań również zwiększyć szybkość). W algorytmie tym, w związku z możliwym istnieniem zadań wzajemnie się wykluczających, oprócz konieczności uwzględnienia zmian w szeregowaniu zadań należy zmodyfikować również funkcję kosztu i metody rafinacji.

W pierwszym podrozdziale zostanie opisany algorytm COSYSOPC maksymalizujący szybkość systemu. Następnie zaprezentowany zostanie algorytm COSEDYRES dla architektur dynamicznie samorekonfigurowalnych opartych na współczesnych układach częściowo reprogramowalnych FPGA.

W ostatnim podrozdziale przedstawiona zostanie finalna wersja algorytmu COSEDYRES dla systemów SRSOPC, uwzględniająca zadania wzajemnie się wykluczające.

6.1 Kosynteza systemów SOPC

W tej części zostanie omówiona metoda kosyntezy systemów SOPC. Zostanie przedstawiona większa efektywność tej metody w odniesieniu do metod znanych z literatury. Algorytm będzie również stanowić punkt odniesienia do oceny skuteczności dynamicznej rekonfiguracji w systemach projektowanych metodą opisaną w podrozdziale 6.2. Opracowanie takiego algorytmu pozwoli na ocenę skuteczności stosowanych metod rafinacji dla systemów SOPC. Zakłada się, że dostępna powierzchnia układu FPGA wynosi S_{max}. Celem prezentowanego rafinacyjnego algorytmu kosyntezy jest znalezienie jak najszybszego systemu SOPC przy zadanej powierzchni docelowego układu FPGA.

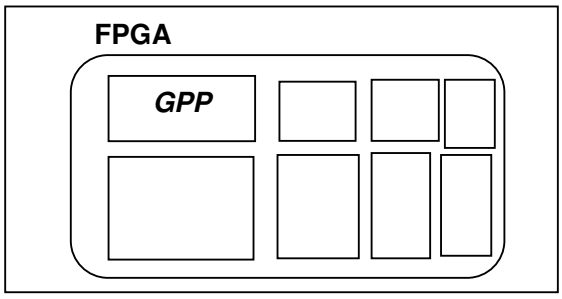
6.1.1 Rozwiązanie początkowe

W rozwiązaniu początkowym wszystkie zadania grafu G przydzielone są do jednego procesora uniwersalnego *GPP* o najmniejszej powierzchni. System zajmuje wtedy najmniejszą powierzchnię układu FPGA, choć jest przeważnie najwolniejszy (wszystkie zadania wykonywane są sekwencyjnie). Rozwiązanie takie pozwala na największą „swobodę" w poszukiwaniu kolejnych rozwiązań drogą rafinacji (największa wolna powierzchnia FPGA). Ważnym aspektem przy wyborze rozwiązania początkowego jest to, aby rozwiązanie początkowe nie ograniczało przestrzeni poszukiwań, czyli aby było możliwe wybranie na drodze rafinacji rozwiązania o dowolnej architekturze.

TWIERDZENIE 6.1-1.
Załóżmy, że wybrano rozwiązanie początkowe A^0 o architekturze złożonej z jednego modułu *GPP* o najmniejszej powierzchni i niech podczas rafinacji będą stosowane tylko metody rafinacji polegające na usunięciu i/lub dodaniu jednego zasobu:
Teza:
Rozwiązanie początkowe A^0 umożliwia uzyskanie na drodze rafinacji każdego rozwiązania o architekturze zajmującej powierzchnię nie większą niż S_{max}.

DOWÓD:
W celu udowodnienia twierdzenia 6.1-1 zostanie przeprowadzona analiza procesu rafinacji „od tyłu", tzn. rozpoczynając od rozwiązania docelowego, a kończąc na rozwiązaniu początkowym. W procesie generacji rozwiązania początkowego sprawdzane są w pierwszej kolejności skrajne przypadki. Jeśli

możliwe jest wykonanie wszystkich zadań sprzętowo, to algorytm od razu kończy działanie, bo jest to rozwiązanie najszybsze. Kolejny skrajny przypadek będzie miał miejsce, jeśli do modułu *GPP* o najmniejszej powierzchni, który jest jednocześnie jedynym typem procesora dostępnym w bibliotece, nie jest możliwe dodanie żadnego innego komponentu z biblioteki, ze względu na ograniczenie powierzchni FPGA. Algorytm wtedy również od razu kończy działanie. Biorąc pod uwagę, że kosynteza jest potrzebna tylko w systemach mieszanych zakłada się, że w rozwiązaniu docelowym otrzymanym w wyniku rafinacji znajduje się przynajmniej jeden moduł *GPP*. Pozostałymi modułami mogą być komponenty *VC* lub inne *GPP*. Takie rozwiązanie przedstawiono na Rys. 6.1-1.

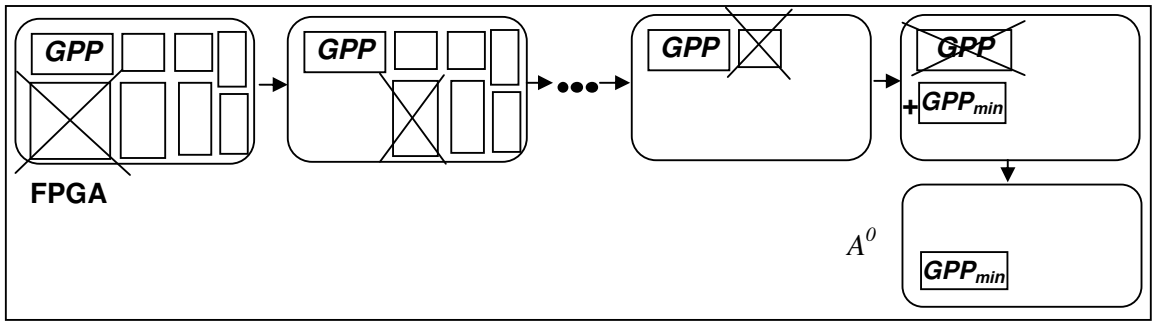


Rysunek 6.1-1 Rozwiązanie docelowe poszukiwane przez algorytm rafinacji

Dla takiego rozwiązania możliwe są następujące modyfikacje prowadzące do uzyskania na drodze „wstecznej" rafinacji rozwiązania początkowego z jednym, najtańszym procesorem:

1) Usuwanie pojedynczych *VC*. W końcowym rozwiązaniu pozostanie jeden *GPP*.
2) Usuwanie pojedynczych *GPP*, aż uzyska się architekturę składającą się z jednego *GPP*.
3) W architekturze jest już tylko jeden *GPP* (nie o najmniejszej powierzchni) i algorytm ma dojść drogą „wstecznej" rafinacji do najmniejszego modułu GPP_{min}. Uzyskanie takiej architektury jest możliwe poprzez dodanie modułu GPP_{min} (nawet kosztem chwilowego przekroczenia S_{max}), a następnie usunięcie *GPP*. Wówczas w architekturze pozostanie GPP_{min}.

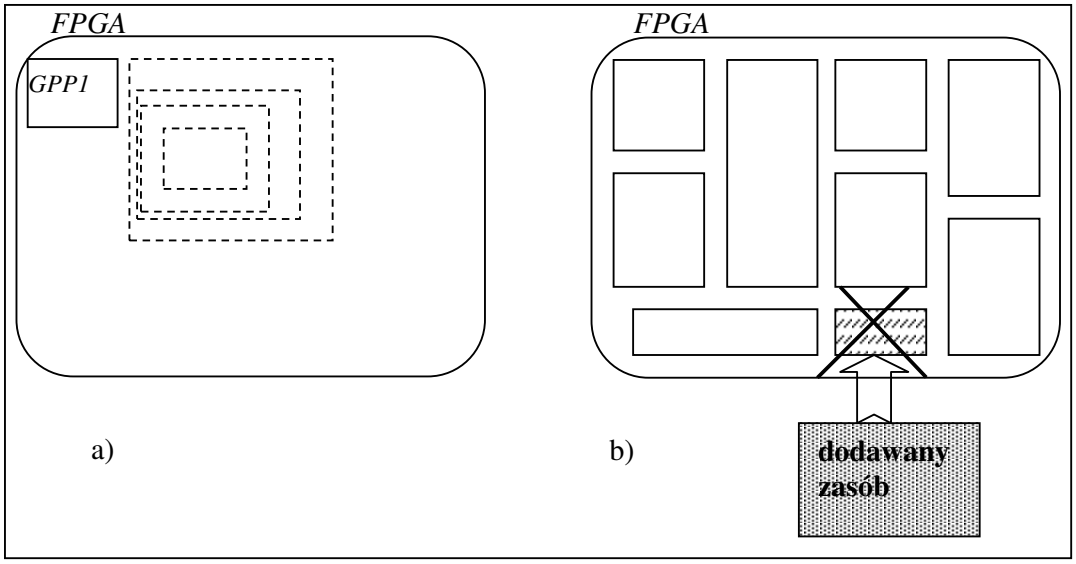
Powyższą analizę zilustrowano na Rys. 6.1-2.



Rysunek 6.1-2 Rafinacja „wsteczna" rozwiązań od rozwiązania docelowego do początkowego.

Zatem startując od rozwiązania początkowego A^0 możliwe jest, poprzez pojedyncze metody rafinacji, takie jak usunięcie i/lub dodanie jednego zasobu, uzyskanie na drodze rafinacji każdej architektury spełniającej warunek $S(A) \leq S_{max}$. ■

W stosunku do architektury początkowej A^0, jak na Rys. 6.1-3a, możliwe jest usunięcie procesora i dodanie szybszego procesora lub dodanie każdego zasobu sprzętowego, który zmieści się w układzie FPGA. Krok ten nie blokuje możliwości żadnej zmiany architektury z udziałem każdego dostępnego zasobu. W przypadku architektury początkowej o powierzchni zbliżonej do powierzchni docelowego układu S_{max} mogą nie być możliwe niektóre modyfikacje, jak dodawanie dużych modułów sprzętowych (najczęściej najszybszych), nawet po wcześniejszym usunięciu (w tym samym kroku) innego zasobu. Nie można dojść do każdej architektury, a więc pewne rozwiązania, które mogą przyspieszyć projektowany system, nie będą rozpatrywane. Kontrprzykład wybranego rozwiązania początkowego pokazano na Rys. 6.1-3b. Jeśli wybrano by takie rozwiązanie początkowe, to otrzymanie jeszcze szybszego systemu mogłoby nie być możliwe. Na przykład w sytuacji, gdy jedynie usunięcie zasobu o małej powierzchni i następnie dodanie w jego miejsce innego (szybszego) prowadziłoby do uzyskania lepszej architektury (zysk>0), ale dodawany zasób nie mieściłby się już w układzie FPGA. Takie rozwiązania blokują możliwości uzyskania jeszcze lepszych architektur w kolejnych krokach.

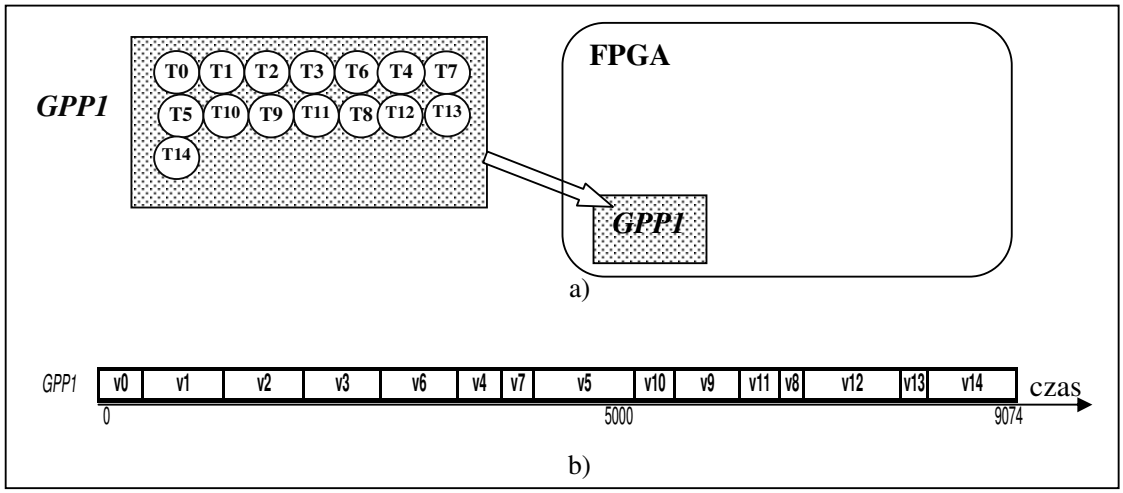


Rysunek 6.1-3 Wybrane rozwiązanie początkowe: a) najlepsze – pozwalające na dojście do każdej architektury, b) blokujące możliwość dojścia do rozwiązania optymalnego.

PRZYKŁAD 6.1-1.

Rysunek 6.1-4 pokazuje rozwiązanie początkowe dla systemu opisanego grafem z przykładu 5.2-1a, przy założeniu, że dostępna powierzchnia układu wynosi 2000 CLB. Zadania wykonywane są sekwencyjnie przez procesor *GPP*, w kolejności jak na rysunku (z uwzględnieniem wszystkich zależności w grafie). Duża część

powierzchni FPGA jest jeszcze niewykorzystana. Czas wykonania zadań dla takiej architektury wynosi 9074 μs, a zajmowana powierzchnia 203 CLB (tylko powierzchnia modułu procesora *GPP*).



Rysunek 6.1-4 Rozwiązanie początkowe algorytmu COSYSOPC

a) architektura systemu, b) uszeregowanie zadań.

6.1.2 Kryteria optymalizacji

Koszt systemu SOPC jest stały i jest równy kosztowi najtańszego docelowego układu FPGA, dla którego zostanie znalezione rozwiązanie o zadowalającej szybkości. Powierzchnia projektowanego systemu SOPC liczona jest jako suma powierzchni wszystkich modułów użytych do implementacji tego systemu. Niech architektura systemu składa się z modułów procesorów GPP_i (i=1,…, p), komponentów sprzętowych VC_j (j= p+1,…, r) i łączy komunikacyjnych CL_k (k=1,…, c).

DEFINICJA 6.1-1 POWIERZCHNIA SYSTEMU SOPC(S)

Powierzchnia całkowita systemu SOPC jest zdefiniowana następująco:

$$S = \sum_{i=1}^{p} Su_i + \sum_{j=p+1}^{r} Su_j + \sum_{k=1}^{c} Sc_k \qquad (6.1.1)$$

Dla każdego rozpatrywanego rozwiązania musi być spełniony warunek: $S \leq S_{max}$ (gdzie S_{max} – powierzchnia układu FPGA).

Niech $v_i \in V$, $e_l \in E$ i niech czas wykonania zadań systemu SOPC zdefiniowany będzie następująco:

$$T=max(max(t_k(GPP_1),\ldots, t_k(GPP_p)), max(t_k(VC_{p+1}),\ldots, t_k(VC_r)), max(t_k(CL_1),\ldots, t_k(CL_c))) \qquad (6.1.2)$$

gdzie $t_k(PE_j)$ jest to czas zakończenia wykonywania zadania, uszeregowanego jako ostatnie, przez zasób PE_j (VC_j lub GPP_j), a $t_k(CL_j)$ jest to czas zakończenia transmisji uszeregowanej jako ostatniej na łączu komunikacyjnym CL_j.

DEFINICJA 6.1-2 SZYBKOŚĆ SYSTEMU SOPC(λ)

Szybkość projektowanego systemu, mierzona w liczbie wykonań grafu na sekundę [G/s], zdefiniowana jest następująco:

$$\lambda = 1/T \tag{6.1.3}$$

W algorytmie COSYSOPC maksymalizowana jest szybkość λ projektowanego sytemu, czyli minimalizowany jest czas wykonania wszystkich zadań *T*.

DEFINICJA 6.1-3 OPTYMALNY SYSTEM SOPC (A^{OPT})

Optymalny system SOPC jest to system, dla którego szybkość λ jest największa z możliwych i spełniony jest warunek $S \leq S_{max}$.

6.1.3 Miara jakości rozwiązania

Miarę jakości rozwiązań określa zysk rafinacji. Celem optymalizacji w opisywanym algorytmie kosyntezy jest maksymalizacja szybkości, a więc głównym parametrem określającym zysk powinna być właśnie szybkość systemu. Jednak kierowanie się tylko jednym parametrem zwykle prowadzi do szybkiego zatrzymywania się algorytmów w lokalnych minimach optymalizowanych parametrów. Zatem miara jakości powinna być zdefiniowana w ten sposób, aby uwzględniała globalne możliwości dalszych rafinacji w kolejnych krokach. W systemach implementowanych w jednym układzie FPGA im większa jest dostępna wolna powierzchnia FPGA, tym większe są możliwości dodawania nowych modułów (*VC* lub *GPP*), a tym samym większe możliwości zrównoleglenia obliczeń i zwiększenia szybkości. Jeśli wolna przestrzeń jest zbyt mała, to mniejsze są również możliwości sprawdzenia nowych rozwiązań z innymi komponentami, które być może wykonałyby pewne zadania szybciej. Algorytm kończy wówczas działanie nie znajdując nowych, lepszych rozwiązań. Należy zatem wprowadzić do miary jakości rozwiązania dodatkowy parametr, który będzie miał za zadanie zahamować zachłanność algorytmu. Parametr ten powinien uwzględniać stopień wykorzystania powierzchni układu.

DEFINICJA 6.1-4 STOPIEŃ WYKORZYSTANIA POWIERZCHNI UKŁADU (α)

Stopień wykorzystania powierzchni układu FPGA określony jest następująco:

$$\alpha = S_{max} - S_{akt} \tag{6.1.4}$$

gdzie S_{akt} jest powierzchnią zajmowaną przez aktualnie analizowane rozwiązanie (system SOPC).

TWIERDZENIE 6.1-2.

Niech A^{R1} i A^{R2} będą dwoma porównywanymi rozwiązaniami znalezionymi w wyniku rafinacji systemu SOPC i niech:

$\Delta\lambda_1 = \lambda(A^{R1}) - \lambda(A^{pop})$, $\Delta\lambda_2 = \lambda(A^{R2}) - \lambda(A^{pop})$,

$\Delta\alpha_1 = \alpha(A^{R1}) - \alpha(A^{pop})$, $\Delta\alpha_2 = \alpha(A^{R2}) - \alpha(A^{pop})$, gdzie A^{pop} jest rozwiązaniem wybranym jako najlepsze w poprzednim kroku rafinacji, $\lambda(A)$ oznacza wartość λ dla systemu A, $\alpha(A)$ oznacza wartość α dla systemu A, $\Delta\alpha_1 < 0$, $\Delta\alpha_2 < 0$.

Teza:

W czasie rafinacji systemu wybór rozwiązań, dla których $\frac{\Delta\lambda}{-\Delta\alpha}$ jest największe, ma największe prawdopodobieństwo osiągnięcia rozwiązania optymalnego A^{opt}.

DOWÓD:

Rafinacja generuje nowe rozwiązania znajdujące się w otoczeniu rozwiązania modyfikowanego. Zatem prawdopodobieństwo (P(A)) dojścia do rozwiązania A^{opt} będzie większe, jeśli algorytm będzie wybierał rozwiązania najszybsze (najbliższe rozwiązania A^{opt}), ponieważ wtedy jest największe prawdopodobieństwo, że w następnych krokach rozwiązanie A^{opt} znajdzie się w zasięgu rozwiązań rozpatrywanych na drodze rafinacji (w ramach jednego kroku). Czyli dla aktualnie rozpatrywanego rozwiązania A prawdopodobieństwo znalezienia A^{opt} jest tym większe, im większe jest λ. Wobec tego, jeśli $\Delta\alpha_1 = \Delta\alpha_2$, to powinno być wybierane rozwiązanie szybsze, czyli $P(A^{R1}) > P(A^{R2})$, jeżeli $\lambda(A^{R1}) > \lambda(A^{R2})$.

Jednocześnie, przy tej samej szybkości rozwiązań, rozwiązania, dla których powierzchnia S będzie mniejsza, będą dawały większe prawdopodobieństwo dojścia do A^{opt} (mniejsza powierzchnia rozwiązania oznacza większą powierzchnię FPGA, którą można wykorzystać do implementacji sprzętowej, czyli do zwiększenia λ). Czyli rozwiązania o największym współczynniku $\frac{\lambda}{S}$ dają największe prawdopodobieństwo uzyskania rozwiązania A^{opt}. A ponieważ, im S jest mniejsze, tym współczynnik swobody α jest większy, zatem prawdopodobieństwo znalezienia A^{opt} jest większe dla rozwiązań o większym współczynniku α. Wobec tego, jeśli $\Delta\lambda_1 = \Delta\lambda_2$, to powinno być wybierane rozwiązanie o mniejszej powierzchni, czyli $P(A^{R1}) > P(A^{R2})$, jeżeli $\alpha(A^{R1}) > \alpha(A^{R2})$.

Załóżmy ,że $\frac{\Delta\lambda(A^{R1})}{-\Delta\alpha(A^{R1})} > \frac{\Delta\lambda(A^{R2})}{-\Delta\alpha(A^{R2})}$.

Czyli: $\frac{\Delta\lambda(A^{R1})}{-(S_{max} - S(A^{R1}))} > \frac{\Delta\lambda(A^{R2})}{-(S_{max} - S(A^{R2}))} \equiv \frac{\Delta\lambda(A^{R1})}{S(A^{R1}) - S_{max}} > \frac{\Delta\lambda(A^{R2})}{S(A^{R2}) - S_{max}}$

Stąd wynika, że: $\frac{\Delta\lambda(A^{R1})}{S(A^{R1})} > \frac{\Delta\lambda(A^{R2})}{S(A^{R2})} \equiv \frac{\lambda(A^{R1}) - \lambda(A^{pop})}{S(A^{R1})} > \frac{\lambda(A^{R2}) - \lambda(A^{pop})}{S(A^{R2})} \equiv \frac{\lambda(A^{R1})}{S(A^{R1})} > \frac{\lambda(A^{R2})}{S(A^{R2})}$.

Wobec tego wybór rozwiązania, dla którego $\frac{\Delta\lambda}{-\Delta\alpha}$ ($\Delta\alpha<0$) jest największe, oznacza, że wybierane jest rozwiązanie o największym stosunku $\frac{\lambda}{S}$. ■

Dla $\Delta\alpha>0$ wybór rozwiązania jest oczywisty, bo przy takim samym przyroście szybkości $\Delta\lambda$ wybierane jest zawsze rozwiązanie o większym $\Delta\alpha$.

Powinno preferować się rozwiązania o jak największym przyroście szybkości, ale równocześnie zmniejszające zajętość powierzchni układu. Wówczas z rozwiązań o takim samym przyroście szybkości wybrane zostanie to, które zmniejsza w jak największym stopniu zajętość powierzchni układu FPGA, czyli o największym współczynniku swobody rozwiązań α. W drugiej kolejności powinny być wybierane rozwiązania powodujące największy przyrost szybkości, ale nie zwiększające powierzchni zajmowanej przez architekturę ($\alpha=0$). Dopiero w dalszej kolejności mogą być wybierane rozwiązania, które zwiększają zajmowaną powierzchnię ($\alpha<0$), ale nadal powodują przyrost szybkości. Rozwiązania nie prowadzące do przyrostu szybkości są odrzucane. Miara przyrostu jakości (zysk) w algorytmie COSYSOPC musi uwzględniać również przypadki, gdy $\Delta\lambda\leq 0$ oraz $\Delta\alpha\geq 0$, zatem:

DEFINICJA 6.1-5 MIARA PRZYROSTU JAKOŚCI ROZWIĄZANIA (ΔE)

Miara przyrostu jakości rozwiązania ΔE określona jest następująco:

$$\Delta E=\begin{cases}\Delta\alpha*\Delta\lambda & ,\text{gdy } \Delta\alpha>0\\ \Delta\lambda & ,\text{gdy } \Delta\alpha=0 \text{ i } \Delta\lambda>0\\ \Delta\lambda/(-\Delta\alpha) & ,\text{gdy } \Delta\alpha<0\\ 0 & ,\text{gdy } \Delta\lambda\leq 0\end{cases} \qquad (6.1.5)$$

gdzie: $\Delta\alpha=\alpha(A^{akt})-\alpha(A^{pop})$, a $\Delta\lambda=\lambda(A^{akt})-\lambda(A^{pop})$, $\alpha(A^{akt})$ i $\lambda(A^{akt})$ są parametrami dla aktualnego rozwiązania, $\alpha(A^{pop})$ i $\lambda(A^{pop})$ są parametrami poprzedniego najlepszego rozwiązania, z którym aktualne rozwiązanie jest porównywane.

W zysku ΔE zawsze uwzględniona jest zmiana szybkości systemu. Jeśli szybkość nie została zwiększona w nowym rozwiązaniu, to takie rozwiązanie nie jest uwzględniane (jest odrzucane). Uwzględniane są cztery przypadki. Pierwszy przypadek ($\Delta\alpha>0$) odpowiada modyfikacji, po której zwiększa się szybkość systemu i dostępna powierzchnia układu FPGA (np. zamiana komponentu większego na mniejszy lub usunięcie komponentu i szybsze wykonanie zadań przez pozostałe komponenty). Drugi przypadek ($\Delta\alpha=0$ i $\Delta\lambda>0$) ma miejsce, gdy modyfikacja nie zmienia powierzchni zajmowanej przez system, a zwiększa jego szybkość (najczęściej przeniesienie zadań pomiędzy komponentami, zmiana uszeregowania zadań). Trzeci przypadek ($\Delta\alpha<0$) dotyczy sytuacji, gdy modyfikacja zwiększa szybkość, ale również powierzchnię zajmowaną przez system w FPGA. Ostatni przypadek określa, że jeśli nie ma przyrostu szybkości to rozwiązanie jest odrzucane ($\Delta E=0$).

Pierwszy przypadek jest najkorzystniejszy, jeśli chodzi o możliwości dalszej rafinacji i prawdopodobieństwo znalezienia jeszcze lepszych rozwiązań, dlatego wartość ΔE w tym przypadku powinna być największa. Modyfikacje z przedostatniego przypadku powodują zmniejszanie się wolnej przestrzeni FPGA dla kolejnych rozwiązań, a więc wartość ΔE jest wtedy tym mniejsza, im $\Delta\alpha$ jest większe.

6.1.4 Metody rafinacji systemu

Stosowane metody rafinacji decydują o złożoności obliczeniowej algorytmu. W każdym kroku algorytmu rozpatrywane są różne modyfikacje aktualnego rozwiązania i do następnego kroku wybierane jest rozwiązanie dające największy zysk ΔE. Zbyt duża liczba rozpatrywanych modyfikacji systemu powoduje znaczny wzrost złożoności obliczeniowej, a w rezultacie uniemożliwia stosowanie algorytmu dla bardziej złożonych systemów. Z drugiej strony stosowanie zbyt prostych metod rafinacji powoduje zatrzymywanie się algorytmu w lokalnym ekstremach optymalizowanych funkcji, co będzie skutkowało znajdowaniem rozwiązań o nienajlepszych parametrach jakościowych. W wielu prezentowanych algorytmach kosyntezy uwzględniane są właśnie takie proste modyfikacje, jak przenoszenie zadań z jednego zasobu do innego, dodawanie, czy usuwanie zasobu z architektury [YW95]. W algorytmie COSYSOPC będą stosowane proste metody rafinacji, ale umożliwiające wykonanie istotnej modyfikacji architektury systemu (podobnie jak w [D04]), takie, które pozwolą na zmniejszenie prawdopodobieństwa zatrzymywania się w lokalnych minimach. Uwzględniane są następujące modyfikacje:

1. Dodanie zasobu do architektury i przeniesienie do niego tych zadań, dla których po przeniesieniu z innego zasobu, zysk będzie dodatni. Zadania są przenoszone do momentu, aż zysk będzie największy. Procesory, które nie mają przyporządkowanych żadnych zadań są automatycznie usuwane z architektury.
2. Usunięcie zasobu i przeniesienie wcześniej przyporządkowanych mu zadań do innych zasobów. Zadania są przenoszone do tych zasobów, dla których zysk będzie największy.

Obie modyfikacje (dodanie i usunięcie zasobu) mogą być wykonane w jednym kroku algorytmu. W ten sposób możliwe jest także przesunięcie grupy zadań z jednego zasobu do innego. Wykonanie obu modyfikacji w jednym kroku pozwala na globalne zmiany architektury systemu. Należy zauważyć, że możliwe są również pojedyncze zmiany, jak przesunięcie tylko jednego zadania pomiędzy zasobami, np. poprzez usunięcie zasobu *PE*, przeniesienie z niego zadania v_i do innego zasobu, a następnie dodanie zasobu *PE* i przyporządkowanie do niego innych zadań. Eksperymenty [D04] pokazują, że takie modyfikacje razem z wcześniej zdefiniowaną miarą jakości rozwiązania pozwalają na wydobywanie się algorytmu z lokalnych ekstremów.

W przypadku, gdy do jednego zasobu przydzielono więcej niż jedno zadanie, konieczne jest ich uszeregowanie. W algorytmie przyjęta została metoda szeregowania listowego [D98]. Priorytety

przydzielane są zadaniom dynamicznie, na podstawie informacji o czasie wykonania zadań przez zasoby, do których aktualnie są przydzielone. Brane są pod uwagę całkowite czasy wykonania zadań na ścieżkach krytycznych *SK* (Def. 5.2-10) począwszy od zadania v_i, dla którego liczony jest priorytet, aż do ujścia grafu, dla aktualnej alokacji zasobów. Z tego powodu priorytety muszą być przeliczane przy każdej modyfikacji architektury. Wybrana została metoda szeregowania listowego, gdyż jej złożoność jest niewielka, a wprowadzenie bardziej złożonej metody szeregowania, przy konieczności jej wykonywania dla każdej modyfikacji, mogłoby znacznie zmniejszyć szybkość algorytmu.

Równocześnie z szeregowaniem zadań przydzielane są transmisje do łączy. Generalnie, jeśli podczas szeregowania zadań konieczna jest komunikacja pomiędzy alokowanym zadaniem v_i a zadaniem v_j już przyporządkowanym do innego zasobu *PE*, to algorytm stara się przydzielić transmisję do istniejącego, najmniej wykorzystywanego łącza komunikacyjnego i takiego, w którym ostatnia transmisja była wykonywana najwcześniej. Sprawdzana jest również możliwość dodania do architektury nowego łącza i wykorzystania go do transmisji v_i-v_j. Jeśli okaże się, że zysk z nowym łączem jest większy, to takie łącze jest dodawane. Podobnie jak w przypadku procesorów, czy komponentów sprzętowych, jeśli łącze komunikacyjne w architekturze nie jest wykorzystywane, jest ono z niej automatycznie usuwane.

6.1.5 Algorytm COSYSOPC

Niech $PE(RT_i)$ będzie dostępną w bibliotece jednostką wykonawczą, czyli rdzeniem procesora GPP_i lub komponentem sprzętowym VC_i. Zarys algorytmu kosyntezy COSYSOPC maksymalizującego szybkość projektowanego systemu SOPC przedstawiony jest na Rys. 6.1-5.

Algorytm 6.1-2

Utwórz architekturę początkową A;
Oblicz powierzchnię S_{akt}; Oblicz szybkość λ systemu A;
powtarzaj
Z=0;
dla *każdego $PE(RT_i)$* ***wykonuj {***
jeżeli *(liczba zasobów w A)≥2)* ***to {***
dla *każdego $PE_j \in A$* ***wykonuj {***
A'=A- PE_j;
dla *każdego $v_k \in PE_j$* ***wykonuj {***
Znajdź $PE_l \in A'$ dające największą wartość $\Delta\varepsilon$ po przydzieleniu do niego zad v_k;
Przyporządkuj v_k do PE_l
}
jeżeli *$\Delta E>Z$* ***to {***
$Z=\Delta E$; $A^{best}=A'$;
}
}
}
$A''=A' \cup PE(RT_i)$;
powtarzaj
Znajdź zadanie v_k o największym współczynniku $\Delta\varepsilon$ po przydzieleniu do $PE(RT_i)$;
jeżeli *$\Delta\varepsilon >0$* ***to*** *przyporządkuj v_k do $PE(RT_i)$*
dopóki *nie ma już zadań po przeniesieniu których $\Delta\varepsilon >0$;*
A''=A''- PEs bez przydzielonych zadań;
jeżeli *$\Delta E>Z$* ***to {***
$Z=\Delta E$; $A^{best}=A''$;
}
}
jeżeli *$Z>0$* ***to*** *$A=A^{best}$;*
dopóki *$Z>0$;*

Rysunek 6.1-5 Zarys algorytmu kosyntezy COSYSOPC

Zmienna Z jest to globalny współczynnik zysku. Jeśli w całym kroku nie zostanie znalezione rozwiązanie lepsze, czyli nie zostanie spełniony warunek $\Delta E>0$, to algorytm kończy działanie. W

zewnętrznych pętlach sprawdzane są wszystkie możliwości usunięcia zasobu z aktualnie analizowanej architektury i dodania zasobu. Przy czym, aby usunąć zasób, w architekturze muszą być co najmniej 2 zasoby. Usunięcie zasobu może prowadzić do zwiększenia dostępnej powierzchni w układzie dla zasobu szybszego, ale o większej powierzchni, który może być dodany w tym samym kroku rafinacji. W wewnętrznych pętlach wykonywane jest przenoszenie zadań i liczony jest tzw. zysk lokalny $\Delta\varepsilon$. Współczynnik $\Delta\varepsilon$ jest obliczany tak samo jak ΔE i jest sprawdzany w wewnętrznych pętlach przy dodawaniu/usuwaniu procesora/komponentu sprzętowego podczas przenoszenia zadań. Rozważane są tylko rozwiązania dające dodatni zysk i spełniające ograniczenia powierzchni. Ogranicza to zakres możliwych poszukiwań, a tym samym złożoność algorytmu. Architektura, dla której otrzymano największy zysk (A^{best}) jest zapamiętywana jako wejściowa dla następnego kroku rafinacji.

Jednym z głównych problemów w algorytmach rafinacyjnych jest zapewnienie zbieżności algorytmu.

TWIERDZENIE 6.1-3.

Algorytm kosyntezy COSYSOPC jest zbieżny.

DOWÓD:

Własność I: Algorytm nigdy nie rozpatruje żadnego rozwiązania więcej niż jeden raz.

Niech rozwiązanie A_0 będzie wybrane jako najlepsze w dowolnym kroku algorytmu. W następnym kroku wszystkie rozwiązania, dla których zysk w stosunku do rozwiązania A_0 jest mniejszy od 0 ($\Delta E \leq 0$) są odrzucane, to znaczy, że w każdym kolejnym kroku wybierane są tylko rozwiązania o różnych wartościach λ i/lub α (dla których $\Delta E > 0$), czyli zawsze inne. Załóżmy, że algorytm cofnie się do rozwiązania A_0, czyli, że w pewnej sekwencji kroków zostaną wybrane rozwiązania: $A_0, A_1, \ldots, A_{n-1}$, A_n *i* $A_0 = A_n$. Wtedy musi być spełniona równość $\sum_{i=0}^{n-1} \Delta E(A_i, A_{i+1}) = 0$, ponieważ zysk pomiędzy rozwiązaniem A_0 i A_n byłby zerowy (to samo rozwiązanie). Ze względu na to, że $\Delta E(A_0, A_1) > 0$, $\Delta E(A_1, A_2) > 0$, …, $\Delta E(A_{n-1} \rightarrow A_n) > 0$, to $\sum_{i=0}^{n-1} \Delta E(A_i, A_{i+1}) > 0$, czyli $A_0 \neq A_n$.

Czyli algorytm nie będzie się cofał do poprzednio rozpatrywanych rozwiązań.

Własność II: Algorytm rozpatruje skończoną liczbę rozwiązań.

Dla *r*-typów zasobów i *n*-zadań algorytm może alokować maksymalnie *n*-zasobów w architekturze (zasoby, którym nie przydzielono zadań są usuwane). Typy zasobów w architekturze mogą się powtarzać. Zatem liczbę wszystkich możliwych architektur można oszacować przez $\boldsymbol{r^n}$. Zadania można przydzielić do *n*-zasobów w architekturze (w jednym zasobie maksymalnie *n* zadań) na $\boldsymbol{n^n}$ sposobów. Wobec tego liczbę wszystkich możliwych rozwiązań można ograniczyć od góry poprzez

wartość $r^n n^n$. Taka liczba rozwiązań w praktyce nigdy nie wystąpi, bo uwzględnione są tu m.in. przypadki z niewykorzystanymi zasobami[7].

Ponieważ algorytm rozpatruje skończoną liczbę rozwiązań i każde rozwiązanie jest rozpatrywane co najwyżej jeden raz, zatem algorytm jest zbieżny. ■

6.1.6 Analiza złożoności obliczeniowej

W tym podrozdziale zostanie przedstawiona teoretyczna analiza złożoności obliczeniowej algorytmu COSYSOPC. Ocena złożoności obliczeniowej w/w algorytmu zostanie dokonana z zastosowaniem notacji asymptotycznej „O" [P02]. Niech *f* i *g* będą funkcjami z **N** w **N**.

DEFINICJA 6.1-6 NOTACJA „O".

f(n)=O(g(n)), jeżeli istnieją takie liczby naturalne *c* i n_0, że dla każdego $n \geq n_0$ zachodzi $f(n) \leq c*g(n)$.

Funkcja *g(n)* w definicji oznacza złożoność problemu. W algorytmie COSYSOPC wielkość problemu jest określona przez liczbę zadań w grafie $\boldsymbol{n \in N}$ i liczbę dostępnych zasobów $\boldsymbol{r \in R}$, gdzie *N* jest zbiorem wszystkich węzłów, a *R* jest zbiorem wszystkich typów komponentów w bibliotece komponentów. Dlatego funkcja określająca złożoność problemu zawiera dwa argumenty *n* i *r*. Zatem złożoność będzie określona jako ***O(g(n, r))***. W przypadku algorytmu COSYSOPC **obliczeniem elementarnym** będzie **generacja jednego rozwiązania**. Generacja jednego rozwiązania jest wykonywana na podstawie poprzedniego rozwiązania, w stosunku, do którego wykonywana jest taka sama liczba modyfikacji (usunięcie i dodanie jednego zasobu). Liczba przesunięć zadań z usuwanego (na pozostałe zasoby w architekturze) lub do dodawanego zasobu nie jest większa niż *n*. Wobec powyższego generacja jednego rozwiązania odbywa się w stałym czasie. **Złożoność obliczeniowa** będzie wyrażona w **maksymalnej liczbie rozpatrywanych rozwiązań** (*L*). Złożoność będzie określona funkcją wielomianową na najgorszy przypadek, czyli uwzględniającą maksymalną liczbę

---

[7] Dokładna liczba możliwych rozwiązań wynosi:

$$l = r*(1+n+\frac{\binom{n}{2}}{\left\lfloor \frac{n}{2} \right\rfloor}*\frac{\binom{n-2}{2}}{\left\lfloor \frac{n-2}{2} \right\rfloor}*\ldots*\left|(-1)^n-2\right|+\frac{\binom{n}{2}}{\left\lfloor \frac{n}{2} \right\rfloor-1}*\frac{\binom{n-2}{2}}{\left\lfloor \frac{n-2}{2} \right\rfloor-1}*\ldots*\left(4+\frac{1-(-1)^n}{2^2}\right)+$$

$$+\frac{\binom{n}{2}}{2}*\binom{n-2}{2}+\binom{n}{2}+\frac{\binom{n}{3}}{\left\lfloor \frac{n}{3} \right\rfloor}*\frac{\binom{n-3}{3}}{\left\lfloor \frac{n-3}{3} \right\rfloor}*\ldots*\frac{\binom{3}{3}}{1}+\ldots+\binom{n}{3}+\ldots+\frac{\binom{n}{3}}{\left\lfloor \frac{n}{3} \right\rfloor}*\frac{\binom{n-3}{2}}{\left\lfloor \frac{n-3}{2} \right\rfloor}*\ldots*\left|(-1)^n-2\right|+\ldots$$

$$+\binom{n}{n-1})$$

gdzie n - liczba wszystkich zadań w grafie, r – liczba komponentów dostępnych w bibliotece komponentów, a kolejne składniki sumy oznaczają wszystkie kolejne kombinacje zadań w zasobach architektury (począwszy od rozwiązania z jednym zasobem, następnie po jednym zadaniu w każdym zasobie, maksymalnie 2 zadania w zasobie, 3 zadania, itd.).

kroków. Na całkowitą liczbę rozwiązań składają się: liczba rozwiązań rozpatrywanych w jednym kroku (L_{kl}) i maksymalna liczba kroków (L_k), czyli:

$$\boldsymbol{L = L_{kl} * L_k}$$

W celu oszacowania złożoności algorytmu COSYSOPC przyjęto następujące **założenia**:

1. *Liczba węzłów w grafie wynosi* $\boldsymbol{n=|N|}$.

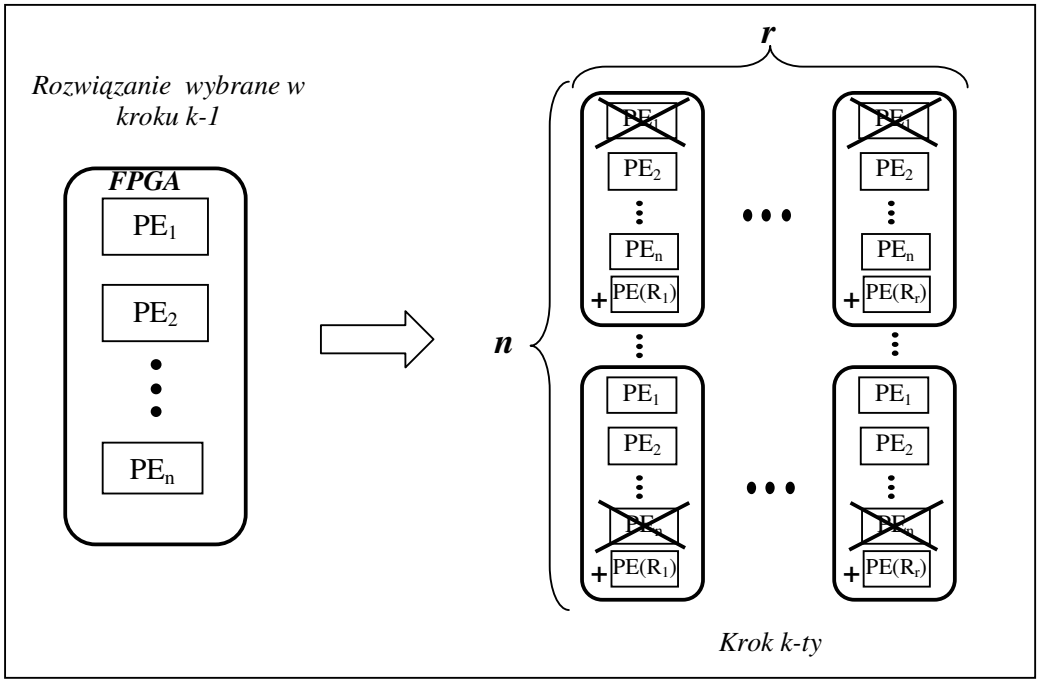
   Z tego założenia wynika również, że maksymalna liczba zasobów alokowanych w architekturze jest równa liczbie wszystkich węzłów w grafie, czyli *n* (każde zadanie w innym zasobie).
2. *Liczba dostępnych typów zasobów w bibliotece wynosi* $\boldsymbol{r=|R|=r_P+r_H}$, gdzie

   $\boldsymbol{r_P}$ – liczba typów procesorów *GPP*, $\boldsymbol{r_H}$ - liczba możliwych typów komponentów *VC* dla każdego zadania. W dalszej części rozważań indeksami *P* i *H* będą oznaczone odpowiednio procesory i komponenty sprzętowe.
3. *Brak ograniczeń w FPGA.*

   Dla celów analizy złożoności przyjęto nieograniczoną powierzchnię układu FPGA, aby liczba badanych rozwiązań nie była ograniczona powierzchnią układu.
4. *Brak zależności pomiędzy zadaniami*. Wtedy liczba różnych kombinacji rozwiązań będzie największa.
5. Zadania v_i: i=1…n, realizowane jako *VC*, uszeregowane są odpowiednio od największej wartości stosunku λ/S do najmniejszej (dla każdego typu *VC*).

W pierwszej kolejności oszacowana zostanie złożoność jednego kroku rafinacji poprzez określenie liczby rozwiązań rozpatrywanych przez algorytm w jednym kroku (L_{kl}), a następnie, poprzez oszacowanie maksymalnej liczby kroków wykonywanych przez algorytm (L_k), oszacowana zostanie złożoność całego algorytmu (na najgorszy przypadek).

W każdym kroku sprawdzane są wszystkie typy zasobów dostępnych w bibliotece, a więc *r* zasobów (r_P+r_H). Zgodnie z algorytmem 6.1-2, dla każdego $PE(R_i) \in R$ wykonywana jest próba usunięcia każdego zasobu dostępnego w architekturze (maksymalnie *n*), a następnie wykonywana jest próba dodania zasobu $PE(R_i)$ do architektury. W algorytmie 6.1-2 najbardziej zewnętrzna pętla, w pętli głównej, wykonywana jest *r*-razy, natomiast następna w hierarchii pętla wewnętrzna wykonywana jest maksymalnie *n*-razy. Maksymalna liczba wszystkich rozwiązań rozpatrywanych w jednym kroku ($\boldsymbol{l_{kl}}$) wynosi zatem $\boldsymbol{r*n}$. Przypadek, w którym liczba rozwiązań w jednym kroku będzie największa zilustrowano na rys. 6.1-6. Będzie miał miejsce wówczas jeśli w poprzednim rozwiązaniu liczba zasobów w architekturze była równa *n*. Wtedy możliwe modyfikacje tego rozwiązania będą polegały na usunięciu jednego z *n*-zasobów i dodaniu jednego z *r*-zasobów. W każdym innym przypadku liczba rozwiązań rozpatrywanych w jednym kroku będzie mniejsza.



Rysunek 6.1-6 Maksymalna liczba rozwiązań w jednym kroku.

OBSERWACJA 6.1-1

Złożoność jednego kroku algorytmu COSYSOPC w najgorszym przypadku wynosi ***O(r*n)***.

Określenie liczby kroków koniecznych do znalezienia przez algorytm możliwie najlepszego rozwiązania nie jest już tak proste i wymaga głębszej analizy, która zostanie przedstawiona w dalszej części.

OBSERWACJE 6.1-2

1) Algorytm nie wraca do rozwiązań wolniejszych. W każdym kroku wybierane jest rozwiązanie, dla którego zysk $\Delta E>0$. Zgodnie ze wzorem 6.1-5 w każdym kolejnym kroku zostanie wybrane tylko takie rozwiązanie, dla którego $\Delta\lambda>0$, czyli szybkość będzie większa od poprzedniego rozwiązania. Algorytm nie zawsze wybiera jednak od razu rozwiązania najszybsze, ale takie, które przyspieszając system, w jak najmniejszym stopniu zwiększają jego powierzchnię.
2) Dla każdego zadania v_i należy uwzględnić, oprócz takich parametrów jak ***n*** i ***r***, również parametry: $\boldsymbol{\lambda(v_i)}$, $\boldsymbol{S(PE(v_i))}$, które mają wpływ na ilość kroków algorytmu.
3) Istotna, ze względu na liczbę rozpatrywanych rozwiązań, jest struktura grafu zadań. Im więcej jest zależności pomiędzy zadaniami w grafie, tym liczba możliwych modyfikacji architektury jest mniejsza. Wówczas liczba kroków jest mniejsza, gdyż nie jest opłacalne np. alokowanie kolejnego procesora tego samego typu, a dla niezależnych zadań alokacja drugiego procesora

może przyspieszyć system, poprzez zrównoleglenie ich wykonywania przez procesory. Zatem najwięcej możliwości modyfikacji architektury (i tym samym najwięcej kroków) powinno być w przypadku niezależnego wykonywania zadań, tym samym ścieżka poszukiwań najlepszego rozwiązania będzie najdłuższa. Dla dalszych rozważań przyjęto wobec tego, że zadania wykonywane są niezależnie.

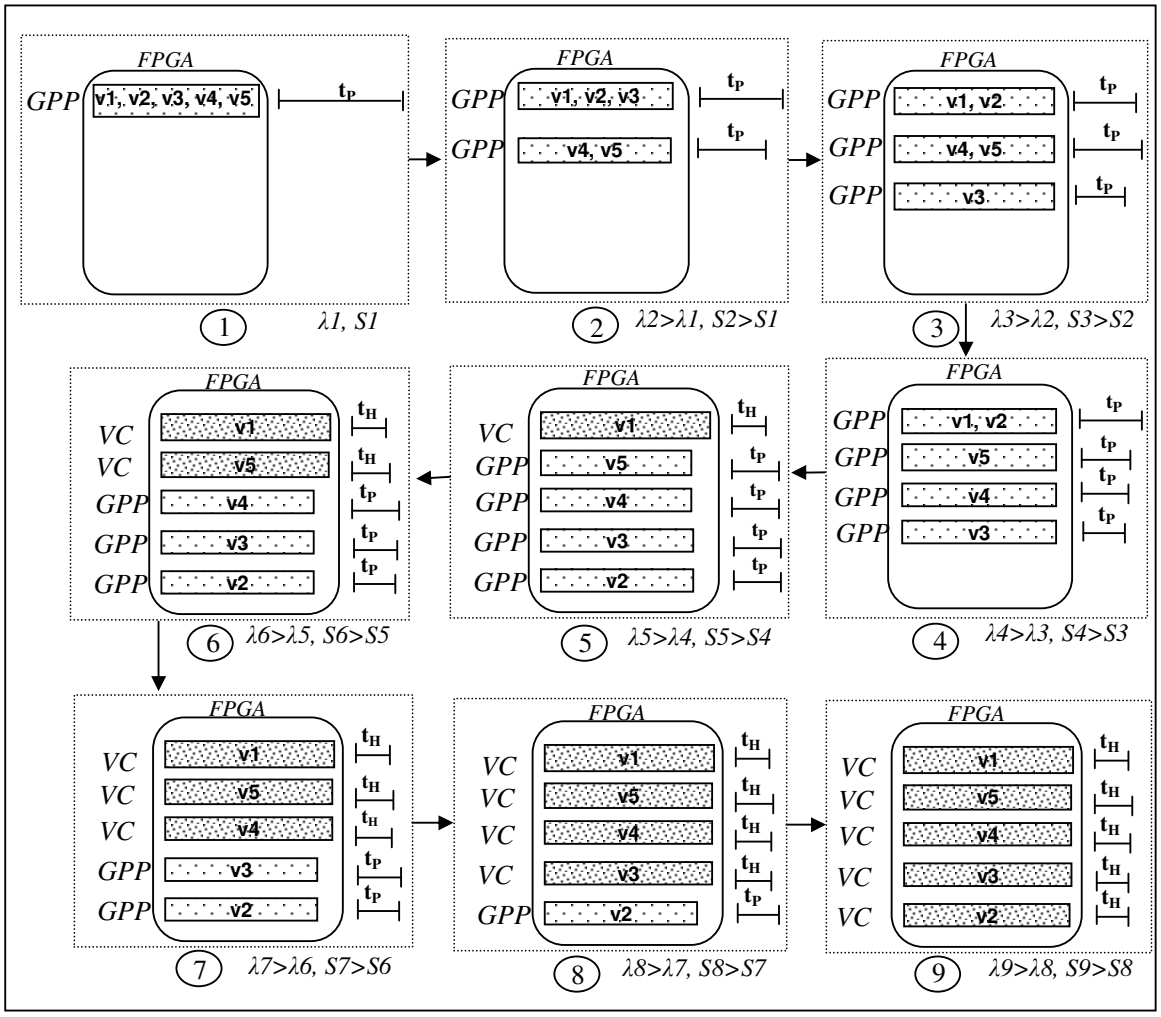
Ponieważ liczba kroków zależy od parametrów zadań, należy rozważyć zachowanie algorytmu dla kilku przypadków ze skrajnie różnymi stosunkami parametrów (szybkości λ i powierzchni S) dla *GPP* i *VC*.

Przypadek 1.

Założenia:

1) Na początku przyjęto, że w bibliotece komponentów jest jeden typ *GPP* i po jednym typie *VC* dla każdego zadania.
2) Dla każdego zadania v_i: $\lambda_P(v_i) < \lambda_H(v_i)$, tzn. wszystkie zadania wykonywane są przez procesor wolniej niż przez *VC*.
3) Dla każdego zadania v_i: $S_P(v_i) < S_H(v_i)$, tzn. powierzchnia modułu procesora jest mniejsza od powierzchni *VC* (dla każdego *VC*).

W celu zobrazowania analizy teoretycznej zostanie przedstawiony prosty przykład dla pięciu zadań o stosunkach parametrów jak wyżej (Rys. 6.1-7). Zakłada się, że zadania *v1-v5* uszeregowane są dla *VC* odpowiednio od największej wartości stosunku λ/S do najmniejszej.



Rysunek 6.1-7 Przebieg rafinacji rozwiązań dla przykładowego systemu z 5 zadaniami i 2 typami zasobów.

Rysunek 6.1-7 pokazuje maksymalną liczbę kroków dla parametrów, jakie zostały podane wcześniej. Liczba kroków (L_k) w takim przypadku wynosi 9. Algorytm w pierwszych krokach alokuje *GPP* (najtańsze), a przesunięcie do nich kilku zadań na raz powoduje zwiększenie szybkości systemu (równoległe wykonanie zadań). Zysk ΔE jest wtedy największy, bo wzrost szybkości $\Delta\lambda$ jest największy, przy minimalnym możliwym zwiększeniu powierzchni. Liczba takich kroków związanych z alokacją *GPP* na początku ścieżki poszukiwań wynosi 4 (wliczając krok pierwszy, czyli rozwiązanie początkowe). Liczba tych kroków wynika z równomiernego rozmieszczania zadań pomiędzy procesorami w architekturze, tak, aby szybkość była największa. W momencie, gdy wszystkie zadania przydzielone są do osobnych *GPP* (z wyjątkiem jednego procesora z dwoma najszybszymi zadaniami – krok 4), algorytm stara się wybrać komponenty droższe, które jeszcze przyspieszą system. W kolejnych krokach następuje zatem stopniowo wymiana *GPP* na *VC*. Takich kroków związanych z wymianą zasobów jest w tym przypadku 5. Łatwo zauważyć, że przedstawiony

przebieg rafinacji jest przypadkiem najgorszym dla w/w założeń. W ogólnym przypadku liczba kroków będzie zależna od liczby zadań i od liczby dostępnych zasobów.

Zależność liczby kroków L_k od *n*.

Jeśli zwiększymy liczbę zadań 2 razy, czyli w tym przypadku do 10-ciu zadań, wówczas w przypadku jednego zasobu w bibliotece będzie konieczne wykonanie maksymalnie 10-ciu kroków (stopniowe dodawanie zasobów, tak, aby ostatecznie każde zadanie było w innym zasobie). Jeżeli w bibliotece są 2 zasoby, wówczas w pierwszych 9-ciu krokach alokowane są *GPP*, a następnie *GPP* wymieniane są na szybsze zasoby – w 10-ciu krokach, wtedy L_k=19. Podobną analizę można przeprowadzić dla większej liczby zadań w grafie. W pierwszych krokach *n-1* razy alokowane są najtańsze *GPP*, w następnych *n*-krokach następuje wymiana wolniejszych zasobów na szybsze.

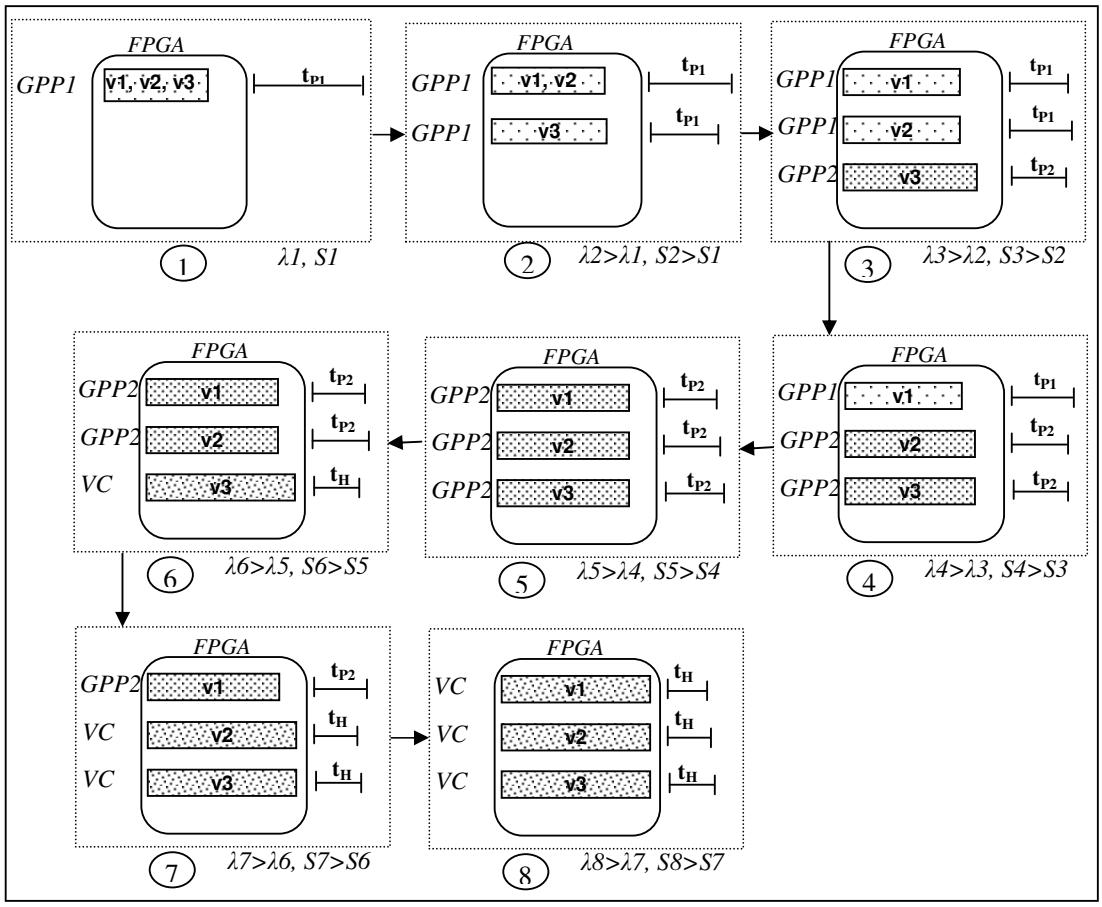
OBSERWACJA 6.1-3

Jeżeli dla każdego zadania v_i: $\lambda_P(v_i) < \lambda_H(v_i)$ i $S_P(v_i) < S_H(v_i)$, to maksymalna liczba kroków jest proporcjonalna do liczby węzłów (***L_k~n***). W przypadku dwóch zasobów (*GPP* i *VC*): ***L_k=2n-1***.

Zależność liczby kroków L_k od *r*.

Zgodnie z wcześniejszymi rozważaniami, jeśli w bibliotece zasobów byłby tylko jeden typ zasobu, czyli np. jeden *GPP*, wówczas algorytm kończyłby działanie po przydzieleniu każdego zadania do osobnego zasobu, czyli po *n*-krokach. Maksymalna liczba kroków w przypadku 2 zasobów (*GPP* i *VC*) jest proporcjonalna do ***2n***. Należy jednak jeszcze sprawdzić jak zachowuje się algorytm w przypadku większej liczby dostępnych zasobów. Przyjmijmy podobne założenie jak poprzednio odnośnie parametrów komponentów, ale dla większej liczby komponentów. Dla uproszczenia analizy w przykładzie przyjęto również 3 zadania w grafie. Na rys.6.1-8 przedstawiono kolejne rozwiązania wybierane przez algorytm jako najlepsze w kolejnych krokach, przy następujących założeniach:

1) W bibliotece są 2 *GPP* i 1 *VC*,
2) Dla każdego zadania v_i: $\lambda_{P1}(v_i) < \lambda_{P2}(v_i) < \lambda_H(v_i)$ i $S_{P1}(v_i) < S_{P2}(v_i) < S_H(v_i)$.



Rysunek 6.1-8 Przebieg rafinacji rozwiązań dla przykładowego systemu z 3 zadaniami i 3 typami zasobów.

Algorytm w pierwszych dwóch krokach alokuje najtańsze zasoby – procesory *GPP1*, następnie w kolejnych krokach alokuje inne zasoby od najwolniejszych, ale najtańszych, do najszybszych (w zależności od współczynnika λ/S). W najgorszym przypadku algorytm rozpatruje wszystkie możliwe zasoby, a więc w przypadku 3 zasobów: 2 kroki dla alokacji wolniejszych procesorów, następnie 3 kolejne kroki dla alokacji szybszych zasobów i 3 kroki dla najszybszych zasobów. W rezultacie dla 3 zasobów algorytm wykona maksymalnie 8 kroków. Dla większej liczby zasobów, np. czterech: w pierwszych dwóch krokach alokowane są najtańsze zasoby, w następnych trzech droższe, ale szybsze, później dla dwóch kolejnych typów szybszych zasobów wykonywane są jeszcze maksymalnie po 3 kroki, a więc L_k=2+3+3+3=11 kroków. W ogólnym przypadku dla każdego z *r*-zasobów maksymalnie zostanie wykonane *n*-kroków, a więc $L_k=r*n-1$.

OBSERWACJA 6.1-4

Jeżeli dla każdego zadania v_i: $\lambda_{R1}(v_i) < \lambda_{R2}(v_i) < .. < \lambda_{Rr}(v_i)$ i $S_{R1}(v_i) < S_{R2}(v_i) < ... < S_{Rr}(v_i)$, to dla ***r*** zasobów i ***n*** zadań maksymalna liczba kroków algorytmu wynosi $\boldsymbol{L_k = r*n-1}$.

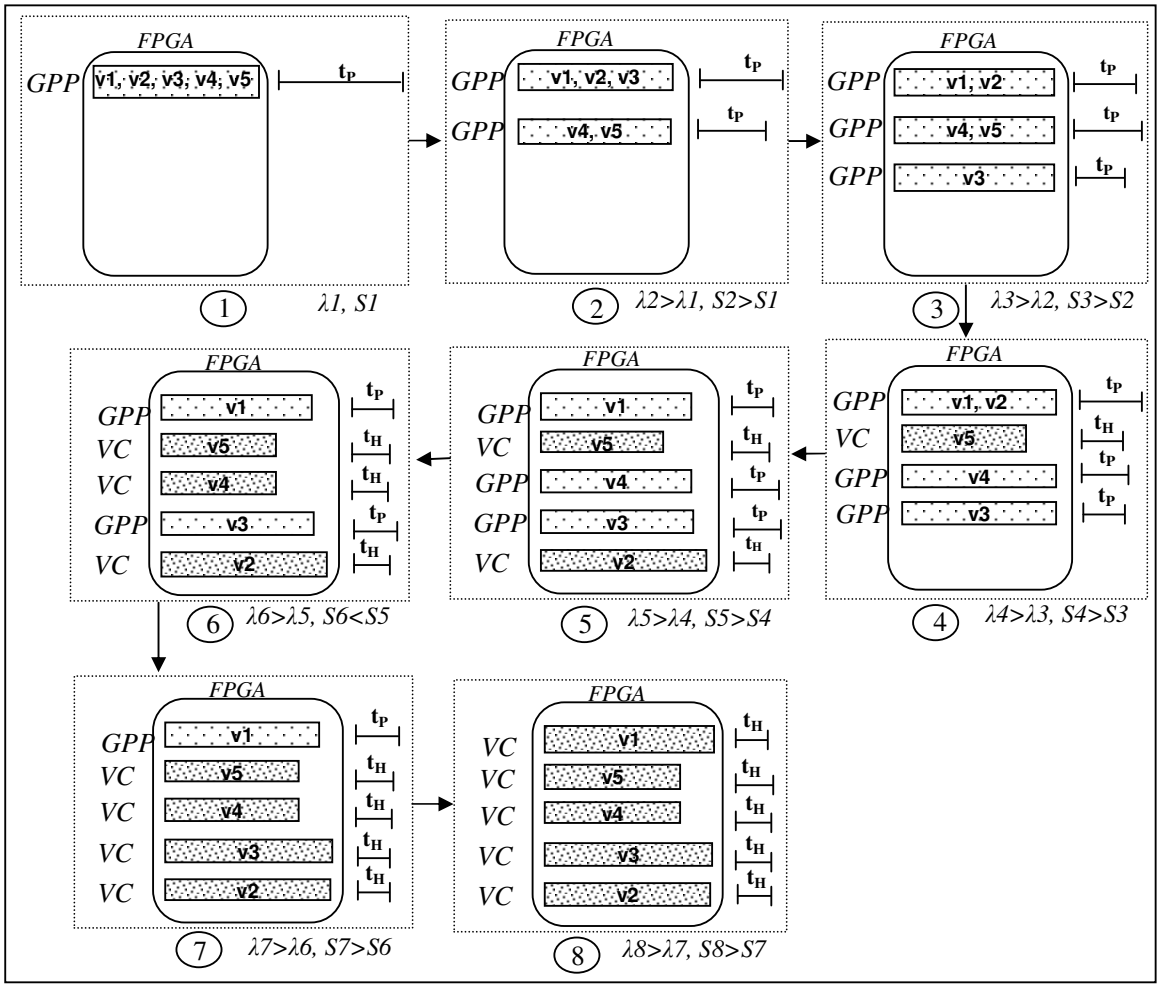
W celu wykazania, że przedstawiona wyżej analiza dotyczy najgorszych przypadków, poniżej zostanie pokazane, że dla każdej innej kombinacji wartości parametrów liczba kroków nie będzie większa od *r*n-1*. W kolejnych przykładach dla uproszczenia przyjęto 2 typy zasobów (*GPP* i *VC*).

Przypadek 2.

Założenia:

1) Niech dla każdego zadania v_i: $\lambda_P(v_i) < \lambda_H(v_i)$,
2) Powierzchnie niektórych *VC* będą mniejsze od powierzchni *GPP*, a pozostałych większe.

Załóżmy, że dla grafu z 5-cioma zadaniami: dla i=1..3: $S_P(v_i) < S_H(v_i)$, dla i=4..5: $S_P(v_i) > S_H(v_i)$.
W takim przypadku niektóre komponenty *VC* o mniejszej powierzchni, ale szybsze od procesorów, zostaną alokowane szybciej niż w poprzednich przykładach (zgodnie z funkcją zysku ΔE), a zatem zmniejszy się też liczba kroków algorytmu. Im więcej takich zadań, dla których $S_{GPP} > S_{VC}$, tym algorytm szybciej zbiega do rozwiązania końcowego. Przykład został zilustrowany na rys. 6.1-9.



Rysunek 6.1-9 Przebieg rafinacji dla przykładowego systemu ze zróżnicowanym stosunkiem powierzchni modułów *GPP* i *VC*.

Liczba kroków dla rozważanego przykładu wynosi 8. Jest mniejsza niż w przypadku 1, ponieważ w kroku czwartym od razu został alokowany zasób najszybszy, bo był jednocześnie najtańszy. Jeśli w bibliotece komponentów będzie więcej takich szybkich i jednocześnie tanich zasobów, to będą one szybko alokowane, co zmniejszy maksymalną liczbę kroków. Analogicznie algorytm będzie zachowywał się dla większej liczby zadań i większej liczby zasobów, więc $L_k<r*n$.

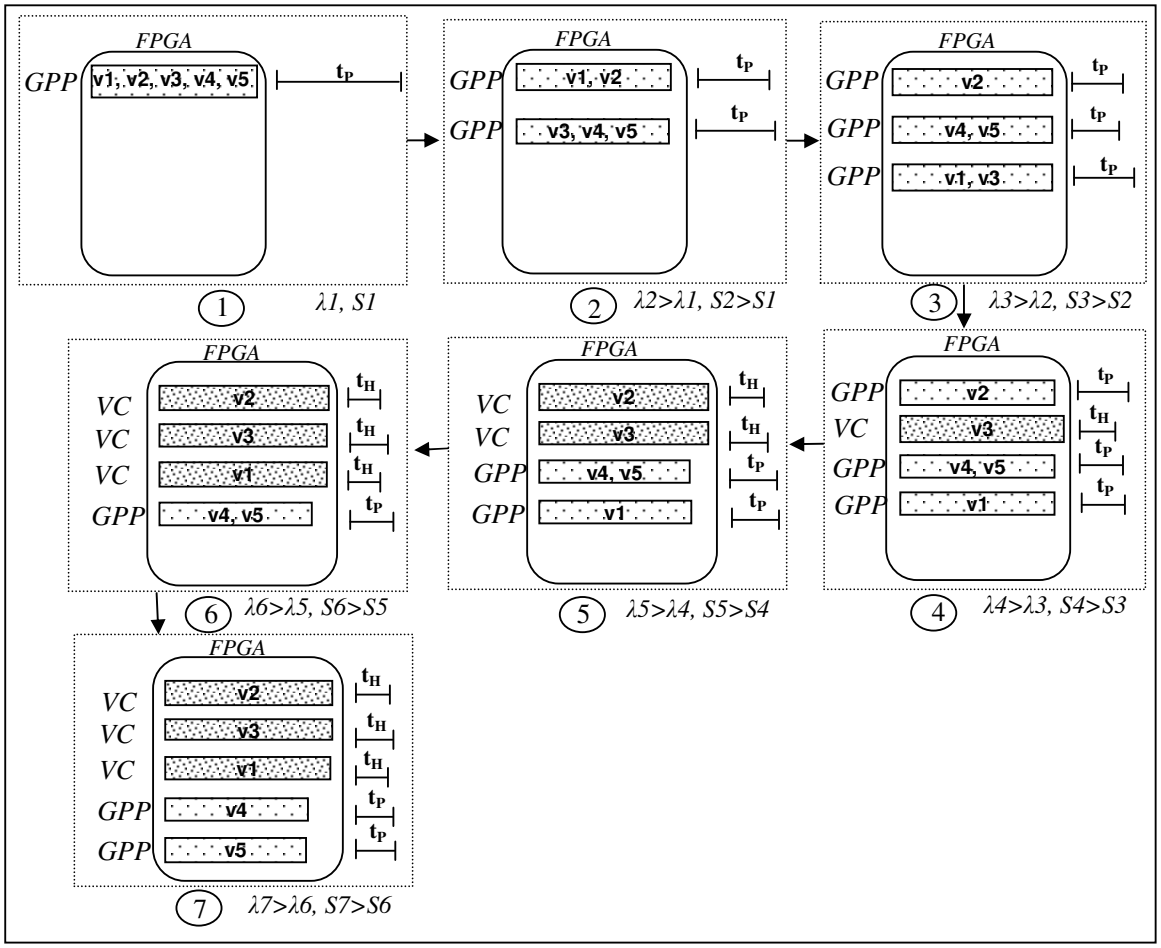
OBSERWACJA 6.1-5

Jeżeli dla każdego zadania v_i: $\lambda_P(v_i)< \lambda_H(v_i)$, a powierzchnie niektórych modułów *VC* będą mniejsze od powierzchni *GPP*, to $L_k<r*n$.

Przypadek 3.

Założenia:

1) Zróżnicowane wartości szybkości zadań na różnych zasobach, tak, aby część zadań wykonywana była szybciej przez procesory (teoretyczny przypadek) niż w sprzęcie.
2) Dla każdego zadania v_i: $S_P(v_i)< S_H(v_i)$.



Rysunek 6.1-10 Rafinacja rozwiązań dla zadań ze zróżnicowanymi szybkościami dla *GPP* i *VC*

Załóżmy, że dla i =1..3: $\lambda_P(v_i)<\lambda_H(v_i)$, a dla i=4..5: $\lambda_P(v_i)>\lambda_H(v_i)$. Na rys. 6.1-10 przedstawiono przebieg rafinacji rozwiązań dla pięciu zadań i dwóch zasobów oraz zależności pomiędzy parametrami jak w założeniach do przypadku 3. Jeśli niektóre zadania wykonywane są szybciej przez *GPP*, to również liczba kroków może być mniejsza. Algorytm w przykładzie kończy działanie po 7 krokach, pozostawiając zadania *v4* i *v5* do wykonania przez procesory, zamiast alokować kolejne zasoby (*VC*). Podobnie dla większej liczby zadań liczba kroków może być tym mniejsza, im więcej zadań jest wykonywanych szybciej przez tańsze *GPP* niż przez inne typy zasobów (te zadania pozostaną na procesorach). Analogicznie, dla tak określonych parametrów algorytm będzie zachowywał się dla większej liczby typów zasobów *r*.

OBSERWACJA 6.1-6

Jeżeli dla każdego zadania v_i: $S_P(v_i)< S_H(v_i)$, a szybkości niektórych zadań wykonywanych przez *GPP* będą większe od szybkości *VC*, to $\boldsymbol{L_k<r*n}$.

Można zauważyć, że w przypadku, gdy wszystkie *GPP* są szybsze od *VC*, to rafinacja skróci się, gdy każde zostanie alokowane do innego *GPP*.

OBSERWACJA 6.1-7

Gdyby dla każdego zadania v_i: $\lambda_P(v_i)>\lambda_H(v_i)$ i $S_P(v_i)<S_H(v_i)$, to $\boldsymbol{L_k\sim n}$.

Pozostał przypadek, gdy powierzchnie niektórych modułów *VC* będą mniejsze od *GPP*. Wtedy, gdy *λ/S* będzie większe dla *VC*, zadanie zostanie alokowane do najszybszego *VC* i pozostanie tam do końca.

OBSERWACJA 6.1-8

Jeśli powierzchnia niektórych *VC* będzie mniejsza od powierzchni *GPP*, to też odpowiadające im zadania, jeśli będą przez te komponenty wykonywane szybciej, to już w pierwszych krokach zostaną alokowane do *VC* (pozostaną do końca w tych komponentach), zmniejszając tym samym liczbę kroków.

WNIOSEK 6.1-1

Na podstawie obserwacji zachowania algorytmu, dla skrajnych przypadków z różnymi wartościami parametrów, najdłuższa droga poszukiwań przez algorytm rozwiązania docelowego (w liczbie kroków) może być oszacowana przez $\boldsymbol{O(r*n)}$.

Uwzględniając liczbę rozwiązań rozpatrywanych w jednym kroku algorytmu $L_{kl}\leq r*n$ oraz wniosek 6.1-1:

WNIOSEK 6.1-2

W najgorszym przypadku liczba rozwiązań rozpatrywanych przez algorytm COSYSOPC jest oszacowana przez:

$$\underline{\boldsymbol{O(r^2n^2)}}.$$

Każde rozwiązanie tworzone jest w czasie proporcjonalnym do *n*. W związku z tym złożoność w zależności od ilości obliczeń jest oszacowana przez:

$$\underline{\boldsymbol{O(r^2n^3)}}.$$

Dla algorytmu COSYSOPC nie jest możliwe dokładne oszacowanie złożoności obliczeniowej, gdyż liczba wykonywanych przez algorytm kroków zależy nie tylko od *n* i *r*, ale także od parametrów modułów bibliotecznych i struktury grafu zadań. Możliwe jest natomiast oszacowanie złożoności na najgorszy przypadek. Podczas analizy złożoności obliczeniowej nie uwzględniano powierzchni układu FPGA, która ogranicza liczbę rozpatrywanych rozwiązań. Ponadto wiele analizowanych teoretycznie przypadków w rzeczywistości nigdy nie wystąpi, np. rzadko zasoby sprzętowe są wolniejsze od procesorów. Dlatego w praktyce liczba rozpatrywanych rozwiązań powinna być znacznie mniejsza niż ta wynikająca z analizy teoretycznej, co zostanie potwierdzone w następnym podrozdziale.

6.1.7 Wyniki wykonanych eksperymentów

W celu zaprezentowania efektywności stosowanych metod kosyntezy dla systemów SOPC, wykonano eksperymenty dla systemów opisanych grafami zadań o różnej wielkości (10-200 węzłów). Grafy zostały wygenerowane losowo. Dokonano porównania wyników syntezy algorytmu COSYSOPC z innymi metodami znanymi z literatury dla systemów SOPC: algorytmem konstrukcyjnym PA^2 [BAP98] oraz z algorytmem rafinacyjnym Yen-Wolf [YW98]. Algorytmy te zostały zmodyfikowane w ten sposób, aby umożliwiały maksymalizację szybkości przy zadanych ograniczeniach kosztu. W bibliotece komponentów dostępne były parametry dla dwóch typów *GPP*, parametry dla dwóch typów komponentów sprzętowych *VC*, jeden typ łącza komunikacyjnego, czasy wykonania zadań i ich powierzchnia w FPGA (dla *VC*) i zajętość pamięci (dla *GPP*). Powierzchnia docelowego układu FPGA dla kolejnych systemów była zwiększana proporcjonalnie wraz ze wzrostem liczby węzłów grafu zadań. Kolejne kolumny tabeli wyników (Tabela 6.1-1) zawierają: nazwę grafu, liczbę zadań grafu, ograniczenie powierzchni w układzie FPGA, a następnie kolejno dla algorytmów PA^2, Yen-Wolf i COSYSOPC: czas wykonania zadań i powierzchnię systemu otrzymanego w wyniku kosyntezy, czas syntezy w [s]. W Tabeli 6.1-2 przedstawiono liczbę rozwiązań analizowanych przez algorytm COSYSOPC oraz liczbę wykonanych kroków w procesie rafinacji. Obliczenia dla algorytmu COSYSOPC zostały wykonane na komputerze z procesorem AMD Athlon 850MHz, a dla pozostałych algorytmów na komputerze z procesorem Pentium III 450 MHz.

Graf	N	Ograniczenie powierzchni	Czas minimalny	PA2			Yen-Wolf			COSYSOPC		
				Czas	Powierzchnia	CPU [s]	Czas	Powierzchnia	CPU [s]	Czas	Powierzchnia	CPU [s]
G1	10	3000	183	881	2574	0.0	988	2516	0.0	531	2943	0.0
G2	30	4000	259	5357	2983	0.0	1800	3787	1.1	1585	3989	1.9
G3	50	5000	248	7801	4986	0.2	3271	4158	7.4	2561	4989	14.0
G4	70	6000	300	10879	6000	0.8	4449	5082	38.8	3111	5983	57.9
G5	90	7000	437	26278	7000	1.5	6644	6295	96.2	5395	6977	88.0
G6	110	8000	377	38288	8000	2.8	5668	7874	290.1	5025	7969	279.9
G7	130	9000	349	18955	8995	7.1	6880	8970	584.1	4399	8966	518.9
G8	150	10000	441	41724	10000	10.4	7319	9883	1172.3	5388	9463	1335.9
G9	170	11000	410	24444	11000	18.6	6324	10995	2717.3	11090	10993	628.9
G10	200	12000	532	40058	12000	26.9	8235	11949	4360.9	10917	11974	2490.0

TABELA 6.1-1. PORÓWNANIE WYNIKÓW ALGORYTMÓW KOSYNTEZY DLA SYSETMÓW SOPC

		COSYSOPC			
Graf	n	liczba rozwiązań l	liczba kroków k	$(CPU/n^3)*10^4$	l/n^2
G1	10	104	7	0	1.04
G2	30	1940	36	0.7	2.15
G3	50	2996	37	1.1	1.19
G4	70	7036	61	1.6	1.43
G5	90	11376	70	1.2	1.40
G6	110	13012	89	2.1	1.07
G7	130	22540	125	2.3	1.33
G8	150	54056	195	3.9	2.40
G9	170	32420	125	1.2	1.12
G10	200	39152	149	3.1	0.97

TABELA 6.1-2. PORÓWNANIE LICZBY ROZWIĄZAŃ I KROKÓW GENEROWANYCH PRZEZ ALGORYTM COSYSOPC

Z otrzymanych wyników można zaobserwować, że algorytm konstrukcyjny PA^2 jest bardzo szybki, ale wyniki są niskiej jakości. Algorytm COSYSOPC w porównaniu z algorytmem Yen-Wolf znajduje w porównywalnym czasie, prawie zawsze, rozwiązania o większej szybkości. Oznacza to, że zastosowane metody rafinacji są skuteczniejsze, a algorytm ma podobną złożoność obliczeniową. Na podstawie czasów obliczeń dla losowych grafów możliwe było określenie praktycznej złożoności obliczeniowej algorytmu kosyntezy. Jeśli n oznacza liczbę węzłów w grafie, to praktyczna złożoność obliczeniowa algorytmu COSYSOPC w zależności od liczby rozwiązań jest porównywalna, a nawet mniejsza, niż ta oszacowana w wyniku rozważań teoretycznych, czyli $O(r^2n^2)$, co potwierdza te rozważania. Praktyczna złożoność w zależności od liczby obliczeń jest również porównywalna z tą oszacowaną w wyniku rozważań teoretycznych, czyli rzędu $O(r^2n^3)$. Natomiast zależność liczby kroków algorytmu od wielkości grafu jest prawie liniowa ($k \approx n$). Cechy algorytmu COSYSOPC takie jak: nieduża złożoność obliczeniowa i dobre uzyskiwane wyniki dla grafów zadań o różnej złożoności, pozwalają stwierdzić, że algorytm ten jest efektywny.

6.1.8 Podsumowanie

W rozdziale 6.1 przedstawiono nowy algorytm kosyntezy maksymalizujący szybkość projektowanych systemów SOPC. Wyniki eksperymentów oraz analiza teoretyczna złożoności

obliczeniowej pokazują, że zastosowanie prostych modyfikacji polegających na usunięciu i dodaniu zasobu w jednym kroku rafinacji, wraz z odpowiednio dobraną funkcją zysku, pozwala na uzyskiwanie lepszych wyników niż za pomocą metod znanych z literatury. Algorytm COSYSOPC charakteryzuje się ponadto niedużą złożonością, jest zbieżny i potrafi wydobywać się z lokalnych maksimów szybkości. Wyżej wymienione cechy świadczą o tym, iż metody zastosowane w algorytmie COSYSOPC są skuteczne i mogą stanowić podstawę do opracowania algorytmu dla systemów dynamicznie rekonfigurowalnych.

W przyszłości można dodatkowo wprowadzić do algorytmu ograniczenie na minimalną wymaganą szybkość systemu λ_{min}. Czasami zakłada się dwa rodzaje ograniczeń na szybkość: łagodne (ang. soft deadline) oraz sztywne (ang. hard deadline) [DJ98b]. Ograniczenia łagodne są to ograniczenia zalecane, które powinny być spełnione przez system, natomiast ograniczenia sztywne nie mogą być przekroczone (mogą istnieć również ograniczenia na pojedyncze ścieżki w grafie). Ograniczenie na powierzchnię układu jest zawsze sztywne, ponieważ w przeciwnym przypadku nie będzie możliwa implementacja systemu w docelowym FPGA.

6.2 Kosynteza systemów SRSOPC specyfikowanych za pomocą klasycznego grafu zadań

W projektowaniu systemów SRSOPC należy uwzględnić wiele różnic w stosunku do systemów SOPC, w istotny sposób utrudniających opracowanie efektywnych metod kosyntezy. Podstawowe problemy, które należy wziąć pod uwagę są następujące:

- **P1:** *wpływ czasu reprogramowania układu FPGA na szybkość systemu*. Współczesne układy częściowo reprogramowane są coraz tańsze i charakteryzują się szybkimi czasami reprogramowania, ale każde reprogramowanie, nawet fragmentu układu, powoduje dodatkowe opóźnienia w działaniu systemu. W szczególnym przypadku może się zdarzyć, że zamiast przyspieszenia systemu przez zastosowanie dynamicznej rekonfiguracji, system zostanie spowolniony (np. gdy czas reprogramowania jest większy od czasu obliczeń i nie można zrównoleglić rekonfiguracji z obliczeniami). Dlatego konieczna jest minimalizacja wpływu opóźnień, spowodowanych tymi dodatkowymi czasami reprogramowania, na szybkość całego systemu SRSOPC, poprzez właściwe rekonfigurowanie systemu.
- **P2:** *zarządzanie reprogramowalnymi fragmentami FPGA*. Dynamiczne zarządzanie zmieniającymi się w czasie reprogramowania fragmentami układu powoduje, że ich efektywne wykorzystanie jest złożone. Podczas reprogramowania mogą bowiem powstawać niespójne fragmenty w układzie (np. gdy w miejsce danego komponentu alokowany jest jeden lub kilka nowych komponentów, o innej powierzchni). Wówczas wykorzystanie wolnych

przestrzeni może okazać się niemożliwe, jeśli żaden z nowych komponentów (po rekonfiguracji) nie zmieści się w tych wolnych fragmentach. W ten sposób dostępna powierzchnia FPGA nie jest do końca wykorzystana dla nowych komponentów, których alokacja mogłaby przyspieszyć działanie systemu SRSOPC.

- **P3**: *zmienność implementacji w czasie działania systemu*. W czasie działania systemu SRSOPC implementacja sprzętowa systemu nie jest stała. Problem polega na właściwym uszeregowaniu zadań implementowanych sprzętowo w tych samych fragmentach układu, tak, aby dodatkowe rekonfiguracje nie zmniejszały szybkości całego systemu, a powodowały jego przyspieszenie. W związku z tym analiza zmieniających się w czasie działania systemu funkcjonalności realizowanych przez *VC* jest złożona.
- **P4**: *fizyczne ograniczenia układów częściowo reprogramowalnych*. Istnieją ograniczenia fizyczne częściowo reprogramowalnych układów FPGA, które zostały określone warunkami 5.4-1 i 5.4-2. Ograniczenia te również wpływają na proces projektowania systemów SRSOPC.
- **P5**: *organizacja dynamicznej rekonfiguracji*. Aby projektowany system SOPC był samorekonfigurowalny, a więc całkowicie niezależny od zewnętrznych układów sterujących rekonfiguracją, konieczne jest zastosowanie modułu lub modułów, które będą zajmowały się sterowaniem rekonfiguracją. System SRSOPC może być rekonfigurowany przez jeden procesor, który będzie zajmował się tylko rekonfiguracją lub przez procesory, które będą mogły, oprócz wykonywania normalnych obliczeń, zajmować się także sterowaniem rekonfiguracją systemu wbudowanego. Wykorzystanie kilku procesorów współdzielących normalne obliczenia ze sterowaniem rekonfiguracją może wprowadzić dodatkowe problemy z szeregowaniem tych operacji.

Wyżej wymienione problemy powodują konieczność opracowania wyspecjalizowanego algorytmu kosyntezy systemów SRSOPC, który będzie je uwzględniał. W kolejnych podrozdziałach zostanie przedstawiony algorytm COSEDYRES, będący rozszerzeniem algorytmu COSYSOPC na architekturę dynamicznie rekonfigurowalną.

6.2.1 Inicjalizacja

Na etapie inicjalizacji odbywa się wyznaczanie dostępnych rozmiarów sektorów oraz generacja rozwiązania początkowego.

Dynamiczna analiza zmieniających się w czasie działania systemu reprogramowanych powierzchni układu FPGA jest stosunkowo złożona (przy różnych powierzchniach modułów *GPP* i *VC*). W celu zmniejszenia złożoności obliczeniowej algorytmu wprowadza się pojęcie sektora dynamicznie rekonfigurowalnego *RS* (Definicja 5.4-2) - fragmentu układu FPGA o stałej wielkości, w której może być implementowanych kilka *VC*. W związku z koniecznością uwzględnienia sektorów *RS* w algorytmie, należy rozbudować etap inicjalizacji o generację dostępnych sektorów, oraz zmienić

metody rafinacji i uszeregowanie zadań. Odpowiednio dobrane wielkości sektorów pozwolą na rozwiązanie problemu P2. Zakłada się, że do jednego sektora *RS* można alokować kilka komponentów *VC* działających równolegle. Wtedy zamiast wykonywać operacje dodawania lub usuwania *VC* algorytm będzie wykonywał analogiczne operacje na całych sektorach. Dzięki temu sektor *RS* może być traktowany w podobny sposób jak procesor ogólnego przeznaczenia (wykonując przydzielone do niego zadania), ale zadania w *RS* mogą być wykonywane równolegle.

Na początku, na podstawie wielkości dostępnych komponentów *VC*, obliczane są możliwe wielkości powierzchni sektorów *RS*. Obliczane są sumy powierzchni tych komponentów. Największy sektor nie może być mniejszy od powierzchni największego *VC* (warunek 5.4-3). Wielkości sektorów muszą być również zgodne z warunkiem 5.4-1. Najlepsze wielkości sektorów to takie, które mogą pomieścić jak najwięcej różnych grup zadań (komponentów sprzętowych *VC*). W celu uzyskania większej elastyczności sektorów (możliwość pomieszczenia różnych modułów *VC*) powinny być dostępne jak najbardziej zróżnicowane wielkości sektorów *RS*. Niech *r* oznacza liczbę wszystkich możliwych *VC* dostępnych w bibliotece komponentów, *l* będzie maksymalną liczbą stałych sektorów *RS*, które należy wygenerować, $C_r^i(VC)$ oznacza kombinację *i*-elementową ze zbioru *r* komponentów *VC*. Zarys algorytmu generacji sektorów przedstawiony jest na Rys. 6.2-1.

Algorytm 6.2-1

i=1;
powtarzaj {
  dla *każdej kombinacji* $C_r^i(VC)$ ***wykonuj {***
    oblicz $S = \sum_{j \in C_r^i(VC)} S(VC_j)$ *;*
    jeśli $S \le \max(S(VC_1),...,S(VC_r))$ ***to*** *zapamiętaj S;*
  }
  i++;
} dopóki *i*≤*r* ***i*** $\exists S \le \max(S(VC_1),...,S(VC_r))$ *;*
dla *każdego k=1* ***do*** *l* ***wykonuj {***
  znajdź S_k *wśród S, które najczęściej się powtarza* ***i*** *ilość* $(S_k)>1$ ***i*** *nie było jeszcze takie* S_k *wybrane;*
  jeśli *nie ma takiego* S_k ***to*** *znajdź S, które mają zbliżoną wielkość* ***i*** *najczęściej się powtarzają* ***i*** *nie były jeszcze wybrane;*
  zapamiętaj S_k*;*
}
dla *każdego k=1* ***do*** *l wykonuj {*
  zaokrąglij w górę S_k *do najbliższego dozwolonego rozmiaru RS;*
}

Rysunek 6.2-1 Algorytm generacji dostępnych rozmiarów sektorów.

Najpierw obliczane są sumy powierzchni wszystkich możliwych grup komponentów sprzętowych *VC*. Następnie spośród wszystkich sum wybierane są sumy, które najczęściej się powtarzają lub mają zbliżoną wielkość i nie przekraczają wielkości największej powierzchni *VC*. Po zaokrągleniu w górę

wielkości znalezionych sektorów do najbliższego, dozwolonego rozmiaru reprogramowalnego bloku, spełniającego zasady 5.4-1, przyjmuje się te wielkości jako dostępne powierzchnie sektorów.

W rozwiązaniu początkowym architektura systemu składa się tylko z jednego rdzenia procesora uniwersalnego *GPP*. Wykonane eksperymenty dla algorytmu COSYSOPC wykazały, że taki wybór rozwiązania początkowego zapewnia możliwość otrzymania najlepszych wyników. Zatem można się spodziewać, że również okaże się to skuteczne dla systemów SRSOPC. Takie rozwiązanie zostawia najwięcej wolnego miejsca dla sektorów *RS*. Algorytm może dzięki temu dojść do każdego maksimum szybkości z jednakowym prawdopodobieństwem (wszystkie zadania traktowane są jednakowo).

6.2.2 Miara jakości rozwiązania

Dla systemów SRSOPC w funkcji powierzchni projektowanego systemu uwzględnia się powierzchnie sektorów *RS* (zamiast powierzchni poszczególnych *VC*), procesorów implementowanych w postaci modułów, oraz połączeń. Sterowanie rekonfiguracją odbywa się przez wbudowany sterownik rekonfiguracji, zatem należy do funkcji powierzchni dodać powierzchnią modułu takiego sterownika.

Niech architektura systemu składa się z procesorów GPP_i (i=1, …, p), modułu sterującego rekonfiguracją GPP_r o powierzchni Su_r, sektorów RS_j (i=p+1,…, r) i łączy komunikacyjnych CL_j (j=1,…, c).

DEFINICJA 6.2-1 POWIERZCHNIA SYSTEMU SRSOPC(S).

Powierzchnia całkowita systemu SRSOPC jest zdefiniowana następująco:

$$S = \sum_{i=1}^{p} Su_i + \sum_{i=p+1}^{r} S_{RS\,i} + \sum_{j=1}^{c} Sc_j + Su_r \qquad (6.2.1)$$

Ze względu na to, że układ FPGA podzielony jest na reprogramowalne sektory *RS*, sektory te będą traktowane jak zasoby wykonujące zadania sprzętowo. Zatem zadania w takim zasobie będą mogły być wykonywane równolegle. Dodatkowo dzięki możliwości dynamicznej rekonfiguracji *RS* i wykorzystaniu tej samej powierzchni *RS* do implementacji różnych *VC* zmieniających się w czasie, konieczny jest podział sektora na tzw. ***konteksty czasowe***. Pozwoli to na rozwiązanie problemu P3. W jednym kontekście wykonywanych może być kilka zadań realizowanych przez *VC* mieszczące się w powierzchni sektora *RS*. Jeżeli powierzchnia *VC* przydzielonych do *RS* jest większa niż powierzchnia tego *RS*, to przed wykonaniem kolejnych zadań konieczne jest reprogramowanie sektora. Wówczas do czasu wykonania zadań w sektorze należy dodać czas reprogramowania sektora t_{res} (Def. 5.4-4).

Niech architektura systemu składa się z modułów procesorów GPP_i (j=1,…, p), sektorów RS_i (j=p+1,…, r) i łączy komunikacyjnych CL_j (j=1,…, c) i niech czas wykonania zadań systemu zdefiniowany jest następująco:

$$T=max(max(t_k(GPP_1),\ldots, t_k(GPP_p)), max(t_k(RS_{p+1}),\ldots, t_k(RS_r)), max(t_k(CL_1),\ldots, t_k(CL_c))) \quad (6.2.2)$$

gdzie $t_k(PE_j)$ jest to czas zakończenia wykonywania zadania, uszeregowanego jako ostatnie, przez zasób PE_j (RS_j lub GPP_j), a $t_k(CL_j)$ jest to czas zakończenia transmisji uszeregowanej jako ostatniej na łączu komunikacyjnym CL_j. Czas wykonania zadań w sektorze RS_j obliczany jest następująco:

$$T_k(RS_j)= \begin{cases} t_k(kontekst1(RS_j))+t_{c/r1}+t_{res}(RS_j)+ t(kontekst2(RS_j)) +t_{c/r2}+t_{res}(RS_j)+\ldots+ t(kontekst\_n(RS_j)) \\ \quad \text{,gdy konieczne są rekonfiguracje sektora } RS_j \\ t_k(RS_j) \quad \text{,gdy sektor } RS_j \text{ nie jest rekonfigurowany dynamicznie w czasie działania systemu} \end{cases}$$

$t_{c/rn}$ jest to czas uwzględniający dodatkowe czasy komunikacji i ewentualne rekonfiguracje innych sektorów. (6.2.3)

DEFINICJA 6.2-2 SZYBKOŚĆ SYSTEMU SRSOPC(λ).

Szybkość systemu SRSOPC, mierzona w liczbie wykonań grafu na sekundę [G/s], zdefiniowana jest wzorem:

$$\lambda=1/T \quad (6.2.4)$$

Globalny współczynnik zysku ΔE dla systemów SRSOPC obliczany jest zgodnie ze wzorem 6.1.5. Podczas przenoszenia zadań nie zmienia się powierzchnia zajmowana przez system, a zatem nie jest konieczne uwzględnianie parametru $\Delta\alpha$, który jest zawsze równy 0. Dlatego w tym przypadku stosuje się zmodyfikowaną funkcję zysku określoną przez parametr $\Delta\varepsilon$. Często przeniesienie jednego zadania v_k nie prowadzi do zwiększenia szybkości, ale może zwiększyć prawdopodobieństwo uzyskania szybszych architektur w kolejnych krokach rafinacji, np. poprzez większe zrównoleglenie wykonywania zadań. Prawdopodobieństwo to może być większe, jeśli zadanie v_k będzie przeniesione z implementacji softwarowej do sprzętowej oraz, gdy czas reprogramowania t_{res} sektora, na który jest przenoszone zadanie, będzie mniejszy od czasu wykonania v_k przez moduł *GPP* (sytuacji tej odpowiada warunek $\Delta\lambda=0$). Jeśli nie ma możliwości poprawy szybkości, to $\Delta\varepsilon$ <0 (dla takiego przypadku można przyjąć stałą wartość ujemną).

DEFINICJA 6.2-3 LOKALNY WSPÓŁCZYNNIK ZYSKU (Δε).

Lokalny współczynnik zysku $\Delta\varepsilon$ jest zdefiniowany następująco:

$$\Delta\varepsilon = \begin{cases} \Delta\lambda & \text{,gdy } \Delta\lambda >0 \\ (t_i(v_k)- t_{res(RSj)})/t_j(v_k) & \text{,gdy } \Delta\lambda=0 \text{ i } t_i(v_k)> t_j(v_k) \text{ i } t_{res(PEj)}< t_i(v_k), \\ -1 & \text{, w pozostałych przypadkach} \end{cases} \quad (6.2.5)$$

gdzie $t_i(v_k)$ jest czasem wykonania zadania v_k przez GPP_i, z którego to zadanie jest przenoszone, $t_j(v_k)$ jest czasem wykonania zadania v_k przez komponent, do którego zadanie jest przenoszone, $t_{res(RSj)}$ jest czasem reprogramowania sektora RS_j. Przeniesienie zadania v_k z procesora do sprzętu jest opłacalne tylko wtedy, gdy czas wykonania zadania v_k przez procesor jest większy niż czas wykonania v_k w sprzęcie. Po drugie należy uwzględnić czas reprogramowania sektora, gdyż nawet, jeśli $t_i(v_k) > t_j(v_k)$, to zbyt duży czas reprogramowania sektora zniweluje korzyść z przeniesienia zadania v_k do sprzętu. Zatem im czas reprogramowania sektora będzie mniejszy oraz czas wykonania zadania v_k w sprzęcie będzie mniejszy od czasu wykonania v_k przez *GPP*, tym przeniesienie zadania jest bardziej opłacalne (drugie kryterium przyjmuje większą wartość). Drugi przypadek hamuje wobec tego zachłanność algorytmu polegającą na tym, iż odrzucane są wszystkie rozwiązania, poza takimi, które dają od razu wzrost szybkości całego systemu. Uwzględnienie tego przypadku może pozwolić w dalszych krokach na otrzymanie szybszego rozwiązania.

6.2.3 Metody rafinacji systemu

Biorąc pod uwagę konieczność uwzględnienia problemów P1-P5, różnice pomiędzy metodami rafinacji zastosowanymi w algorytmie kosyntezy COSEDYRES, a tymi w algorytmie COSYSOPC będą następujące:

- Zamiast wykonywać operacje dodawania lub usuwania komponentów sprzętowych *VC* wykonywane są analogiczne operacje na całych sektorach. Zadania realizowane przez *VC* są przydzielane i szeregowane w sektorze i mogą być w nim wykonywane równolegle. Rafinacja polega więc na modyfikacji aktualnego rozwiązania poprzez dodanie lub usunięcie modułu procesora *GPP* lub sektora *RS*. W momencie dodawania sektora do architektury dodawany jest z sektorem tylko jeden kontekst czasowy.
- W celu rozwiązania problemu P3 wprowadzono dodatkowe metody - dodawanie kontekstów czasowych sektora *RS* i usuwanie kontekstów sektora. Takie metody rafinacji pozwolą na znalezienie jak najlepszej liczby koniecznych przeprogramowań fragmentów układu, aby uzyskać jak największe przyspieszenie systemu.
- W celu rozwiązania problemu P4 w algorytmie COSEDYRES uwzględniane są dodatkowo ograniczenia w rozmieszczeniu reprogramowalnych sektorów [X04]. Sektory *RS* w każdej architekturze rozmieszczone są w sposób liniowy w ten sposób, że każdy sektor zajmuje sąsiednie kolumny CLB. Takie ograniczenia eliminują pewną grupę rozwiązań, które mogłyby być optymalne, ale niemożliwa będzie ich fizyczna realizacja w układzie FPGA, ze względu na ograniczenia określone warunkiem 5.4-2. W każdym kroku wykonywana jest procedura rozmieszczania sektorów (Rys. 6.2-3) opisana w podrozdziale 6.2.5.

- Ze względu na konieczność uwzględnienia czasów rekonfiguracji, w inny sposób obliczany jest lokalny współczynnik zysku $\Delta\varepsilon$ sterujący przenoszeniem zadań pomiędzy zasobami. Pozwala to na rozwiązanie problemu P1.

6.2.4 Algorytm kosyntezy systemów SRSOPC

Niech $PE(RT_i)$ będzie dostępną w bibliotece jednostką wykonawczą, czyli procesorem GPP_i lub sektorem RS_i. Zarys algorytmu kosyntezy systemów SRSOPC przedstawiono na Rys. 6.2-2.

Algorytm 6.2-2

Utwórz architekturę początkową A;
Oblicz powierzchnię S_{akt}; Oblicz szybkość λ systemu A;
powtarzaj
Z=0;
dla *każdego $PE(RT_i)$* ***wykonuj {***
jeżeli *(ilość zasobów w A)≥2)* ***to {***
dla *każdego $PE_j \in A$* ***wykonuj {***
A'=A- PE_j;
dla *każdego $v_k \in PE_j$* ***wykonuj {***
Znajdź $PE_l \in A'$ dające największą wartość $\Delta\varepsilon$ po przydzieleniu do niego zad v_k;
Przyporządkuj v_k do PE_l
}
jeżeli *$\Delta E>Z$* ***to {***
$Z=\Delta E$; $A^{best}=A'$;
}
}
}
dla każdego *RS_i* ***wykonuj {***
dla *każdego kontekstu(RS_i)* ***wykonuj {***
A'' = A'- kontekst(RS_i);
dla *każdego $v_k \in$ kontekst(RS_i)* ***wykonuj {***
Znajdź $PE_l \in A''$ dające największą wartość $\Delta\varepsilon$ po przydzieleniu do niego zad v_k;
Przyporządkuj v_k do PE_l
}
}

```
    jeżeli ΔE>Z to {
       Z=ΔE;  A^best=A'';
    }
  }
A'''=A'' ∪PE(RT_i);
    powtarzaj
      Znajdź zadanie v_k o największym współczynniku Δε po przydzieleniu go doPE(RT_i);
      jeżeli  Δε >0 to przyporządkuj v_k do PE(RT_i)
    dopóki nie ma już zadań po przeniesieniu których Δε >0 ;
    dla każdego RS_i wykonuj {
          dla każdego k=0 do ilość_kontekstów(RS_i) wykonuj {
                    A'''= A''' ∪kontekst_k(RS_i); Przesuń pozostałe konteksty RS_i
                    Powtarzaj
                       Znajdź zadanie v_k o największym współczynniku Δε po przydzieleniu go do
                       kontekst_k(RS_i);
                       jeżeli  Δε >0 to przyporządkuj v_k do  kontekst_k(RS_i)
                    dopóki nie ma już zadań po przeniesieniu których Δε >0 ;
                                                          }
                                 }
    A'''=A'''- PEs bez przydzielonych  zadań;
    jeżeli ΔE>Z to {
        Z=ΔE;  A^best=A''';
    }
  Rozmieść sektory w FPGA;
  }
  jeżeli Z>0 to A=A^best;
dopóki Z>0;
```

Rysunek 6.2-2 Zarys algorytmu kosyntezy systemów SRSOPC

Szkielet algorytmu 6.2-2 wygląda podobnie do algorytmu 6.1-2. Różnice polegają głównie na tym, iż zamiast operować pojedynczymi komponentami sprzętowymi *VC* jako zasobami, stosowane są całe sektory *RS*. Konieczne jest rozpatrywanie kontekstów czasowych sektorów, w związku z tym wprowadzono dodawanie i usuwanie kontekstów sektora. W momencie dodawania sektora *RS* do architektury dodawany jest jeden kontekst czasowy. Dodatkowo w każdym kroku rafinacji wykonywane jest rozmieszczenie sektorów zgodnie z warunkiem 5.4-2. Do tak utworzonej architektury dodawany jest sterownik rekonfiguracji (wprowadzenie modułu zajmującego się tylko rekonfiguracją sektorów, zamiast współdzielenia obliczeń z rekonfiguracją, pozwoliło na zmniejszenie złożoności algorytmu i rozwiązanie problemu P5). Część powierzchni układu FPGA jest

zarezerwowana na moduł sterownika rekonfiguracji jeszcze przed wykonaniem syntezy systemu. Główne różnice w szkieletach algorytmów COSEDYRES i COSYSOPC zaznaczono na Rys. 6.2-2 zacienionym tłem. Jest jeszcze wiele różnic, takich jak inny sposób szeregowania zadań, traktowania zasobów (sektorów) przy ich dodawaniu/usuwaniu, przenoszeniu zadań, itp. Ostatni krok generacji architektury to rozmieszczenie sektorów. Szczegóły tej procedury omówione zostaną w podrozdziale 6.2.5. W wewnętrznych pętlach, przy przenoszeniu zadań obliczany jest lokalny współczynnik zysku $\Delta\varepsilon$, a rafinacja jest sterowana globalną miarą jakości ΔE.

6.2.5 Rozmieszczenie sektorów

Każdy sektor *RS* musi spełniać dwa podstawowe ograniczenia: dopuszczalne wielkości sektorów (warunek 5.4-1) i zasadę poprawności fizycznej implementacji SRSOPC (warunek 5.4-2). W celu znalezienia najlepszego rozmieszczenia sektorów algorytm sprawdza różne położenia sektorów w układzie FPGA. Dla każdego rozmieszczenia wykonywane jest szeregowanie zadań i komunikacji, podczas którego uwzględnia się wymagania częściowej rekonfiguracji. Spośród analizowanych położeń sektorów wybierane jest rozwiązanie najszybsze A^{best}. Zarys algorytmu rozmieszczenia sektorów przedstawiono na Rys. 6.2-3.

Algorytm 6.2-3

Znajdź wszystkie permutacje sektorów dla A^{best};
dla *każdej permutacji* A^p **wykonuj {**
Uszereguj zadania, komunikacje i rekonfiguracje;
dla *każdego* RS_k **wykonuj{**
jeśli *są transmisje przez* RS_k *podczas konfiguracji* RS_k
to{
zmodyfikuj uszeregowanie przez zmianę czasów rozpoczęcia rekonfiguracji i/lub czasów rozpoczęcia transmisji;
}
}
oblicz $\lambda(A^p)$;
jeśli $\lambda(A^p)>\lambda(A^{pbest})$ **to** $A^{pbest}=A^p$;
}

Rysunek 6.2-3 Algorytm rozmieszczenia sektorów

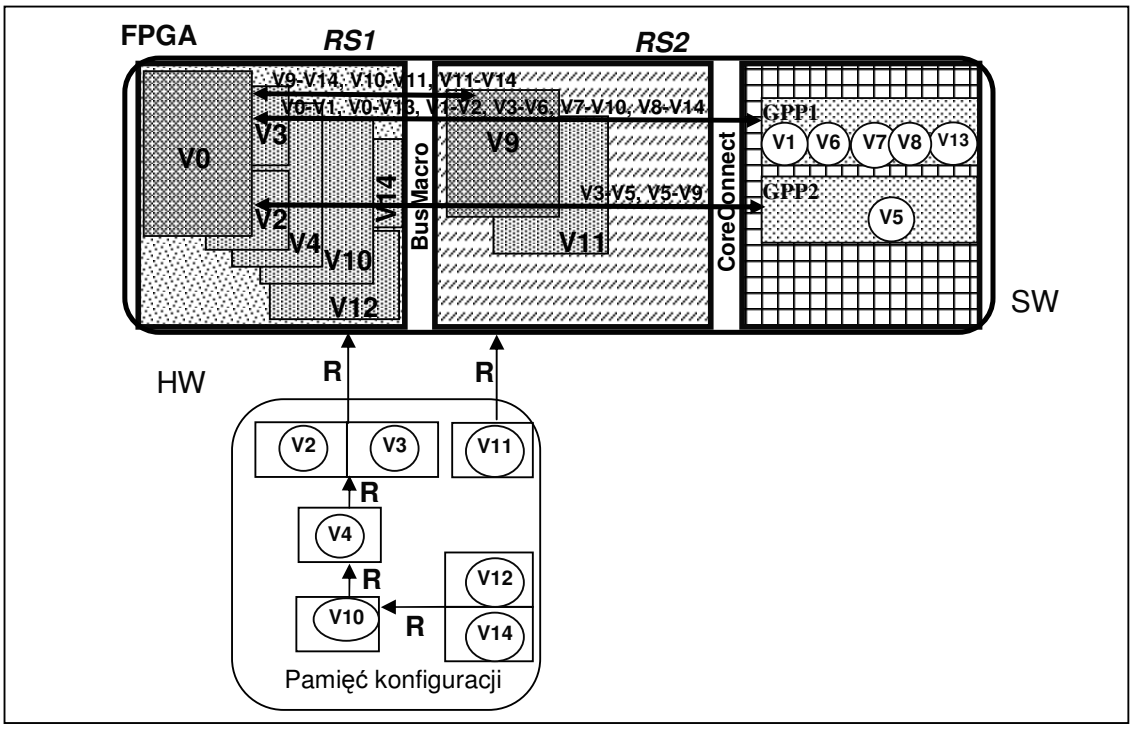
Dla każdej permutacji algorytm sprawdza, czy czas reprogramowania sektora RS_k nie pokrywa się w czasie z transmisją danych pomiędzy dwoma komunikującymi się zasobami położonymi po obu stronach RS_k. Jeśli taki konflikt nastąpi, wówczas konieczna jest zmiana uszeregowania poprzez przesunięcie chwil rozpoczęcia rekonfiguracji i transmisji danych. Wybierane jest rozwiązanie najlepsze pod względem szybkości A^{pbest}.

Teoretycznie maksymalna ilość permutacji sektorów wynosi *n!*, gdzie *n* jest liczbą zadań w grafie (gdyby każde zadanie umieszczone było w innym sektorze). Pomimo konieczności uwzględniania wszystkich permutacji sektorów, złożoność algorytmu w praktyce nie jest duża, nawet dla złożonych

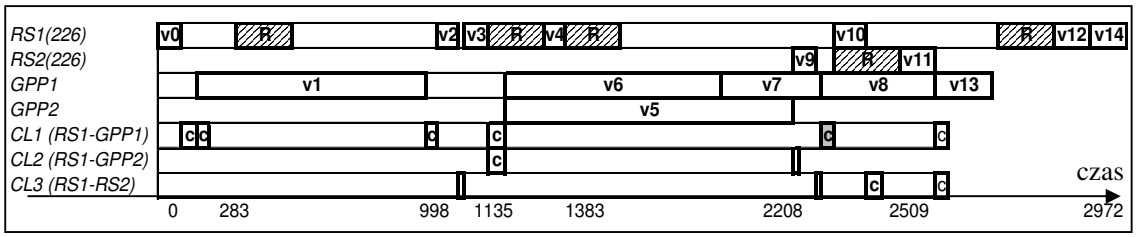
systemów, gdyż liczba sektorów w architekturze jest zwykle nieduża (w eksperymentach nie przekracza kilku sektorów), to pozwala przeanalizować wszystkie permutacje w satysfakcjonującym czasie.

PRZYKŁAD 6.2-1.

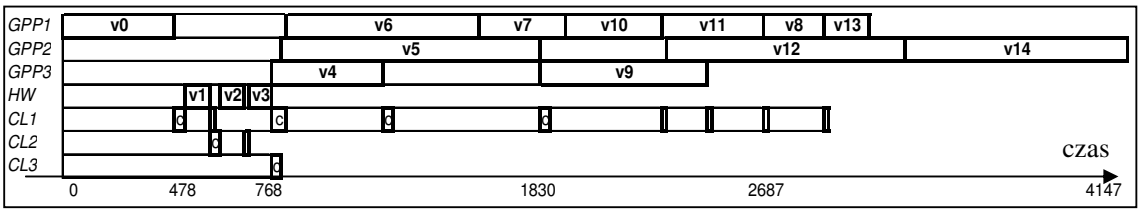
Na rysunku 6.2-4 przedstawiono architekturę otrzymaną w wyniku kosyntezy systemu SRSOPC opisanego grafem z przykładu 5.2-1a, po uwzględnieniu rozmieszczenia sektorów. Rysunek 6.2-5 pokazuje uszeregowanie zadań i rekonfiguracji dla docelowej architektury.



Rysunek 6.2-4 Rozmieszczenie sektorów w FPGA po uwzględnieniu warunków 5.4-1 i 5.4-2



Rysunek 6.2-5 Wykres Gantta dla architektury otrzymanej w wyniku kosyntezy dla dynamicznej rekonfiguracji



Rysunek 6.2-6 Wykres Gantta dla architektury otrzymanej w wyniku kosyntezy bez dynamicznej rekonfiguracji

W architekturze docelowej systemu dynamicznie samorekonfigurowalnego znajdują się trzy sektory. Dwa sektory są dynamicznie rekonfigurowane (*RS1*, *RS2*), a w pozostałej części układu zaimplementowano procesory ogólnego przeznaczenia (*GPP1* i *GPP2*) oraz procesor zajmujący się reprogramowaniem sektorów *GPPr*. Wszystkie zadania przydzielone do sektorów *RS1÷RS2* są implementowane jako komponenty sprzętowe *VC*. Dynamiczna rekonfiguracja pozwala na wykonanie większej liczby zadań, implementowanych w tym samym obszarze FPGA, aniżeli pozwala na to dostępna powierzchnia sektora. Strzałki z oznaczeniami *R* pokazują sekwencję rekonfiguracji poszczególnych sektorów. Pomiędzy sektorami wykonywanych jest wiele transmisji danych, ale tylko transmisja od zadania *v7* do *v10*, pomiędzy sektorem *RS1* a procesorami implementowanymi z prawej strony układu, mogą powodować konflikt związany z reprogramowaniem sektora *RS2*. W celu uniknięcia takiego konfliktu reprogramowanie sektora *RS2* nie może rozpocząć się zaraz po zakończeniu wykonywania zadania *v9*.

Korzyść ze stosowania architektur SRSOPC jest widoczna po porównanniu architektury, otrzymanej w wyniku zastosowania algorytmu COSEDYRES i COSYSOPC dla tych samych wartości parametrów i systemu specyfikowanego grafem z przykładu 5.2-1a. Ograniczenie powierzchni dla obu przypadków przyjęto takie samo i wynosiło 1100 CLB. Dla systemu samorekonfigurowalnego czas wykonania wszystkich zadań systemu wynosi 2972μs przy powierzchni 1087CLB (wliczając powierzchnię modułu *GPPr* sterującego rekonfiguracją), natomiast bez dynamicznej rekonfiguracji czas ten wynosi 4147μs a powierzchnia 1030 CLB. Pomimo pewnych ograniczeń w fizycznym rozmieszczeniu reprogramowalnych modułów i narzutów czasowych związanych z częściową rekonfiguracją dla systemu z możliwością dynamicznej rekonfiguracji udało się otrzymać architekturę, która jest o około 29% szybsza, niż odpowiednia architektura bez dynamicznej rekonfiguracji. Wynik taki został uzyskany dzięki wielokrotnemu wykorzystaniu tych samych powierzchni układu FPGA dla kilku różnych funkcjonalności.

6.2.6 Wyniki wykonanych eksperymentów

W celu zaprezentowania korzyści wynikających z zastosowania dynamicznej rekonfiguracji systemów wbudowanych i oceny efektywności algorytmu COSEDYRES, wykonano eksperymenty dla systemów opisanych grafami zadań o różnej wielkości (10-150 węzłów). Grafy zostały wygenerowane losowo. Dokonano porównania wyników syntezy tych grafów dla implementacji SRSOPC oraz dla implementacji SOPC. W bibliotece komponentów dostępne były parametry dla jednego typu procesora uniwersalnego *GPP* (implementowanego w postaci modułu IP), parametry komponentów sprzętowych *VC*, jeden typ łącza komunikacyjnego, czasy wykonania zadań i

powierzchnia *VC* realizujących te zadania oraz zajętość pamięci dla implementacji softwarowej. Dostępna powierzchnia dla kolejnych systemów była zwiększana proporcjonalnie wraz ze wzrostem liczby węzłów grafu.

Kolejne kolumny tabeli wyników 6.2-1 zawierają: liczbę zadań grafu, ograniczenie powierzchni dla systemu wbudowanego, dla każdego algorytmu: czas wykonania zadań i powierzchnię zajmowaną przez system otrzymany w wyniku kosyntezy, opis znalezionej architektury (*p* – ilość *GPP*, *s* – ilość *RS*, *hw* – liczbę wszystkich zadań wykonywanych w sprzęcie), czas syntezy w [s] (procesor Intel Core 2 Duo 1,86 GHz). Ostatnia kolumna zawiera procentowy wzrost szybkości systemów SRSOPC w stosunku do szybkości systemów SOPC. Do liczby wykorzystanych procesorów uniwersalnych zamieszczonych w opisie architektury należy dodać w każdym przypadku jeden procesor zajmujący się wyłącznie sterowaniem rekonfiguracją systemu.

graf	ograniczenie powierzchni	COSEDYRES			CPU [s]	COSYSOPC			przyrost szybkości
		czas	pow.	opis		czas	pow.	opis	[%]
10	1100	**2509**	933	2p,1s,5hw	0.20	**4078**	927	2p, 3hw	62%
30	1500	**4302**	1498	3p,2s,18hw	2.14	**6734**	1425	4p, 2hw	57%
50	2000	**5926**	1786	3p,3s,29hw	6.39	**8352**	1825	6p, 3hw	41%
70	2500	**6435**	2480	5p,2s,30hw	30.69	**9605**	2407	7p, 5hw	49%
90	3000	**8508**	2865	4p,4s,43hw	177.26	**9707**	2658	8p, 6hw	14%
110	3500	**8360**	2708	5p,5s,50hw	265.86	**8859**	3353	11p, 7hw	6%
130	4000	**8815**	2661	6p,3s,56hw	428.97	**9564**	3995	12p, 13hw	9%
150	4500	**10891**	3332	7p,4s,60hw	727.26	**13037**	4295	9p, 13hw	20%

TABELA 6.2-1. PORÓWNANIE WYNIKÓW DLA ALGORYTMÓW COSEDYRES I COSYSOPC

		COSEDYRES			
Graf	**n**	**liczba rozwiązań l**	**liczba kroków k**	**$(CPU/n^3)*10^4$**	**l/n^2**
G1	10	16	4	2.04	0.16
G2	30	90	8	0.79	0.10
G3	50	115	8	0.51	0.05
G4	70	212	10	0.89	0.04
G5	90	463	15	2.43	0.01
G6	110	386	12	1.99	0.03
G7	130	531	10	1.95	0.03
G8	150	807	14	2.15	0.03

TABELA 6.2-2. PORÓWNANIE LICZBY ROZWIĄZAŃ I KROKÓW GENEROWANYCH PRZEZ ALGORYTM COSEDYRES

Na podstawie uzyskanych wyników, można stwierdzić, że wykorzystanie dynamicznej rekonfiguracji dla przeważającej większości systemów wbudowanych jest korzystne. Rozwiązania otrzymane w wyniku zastosowania algorytmu COSEDYRES dla systemów SRSOPC wykazały średnio 30% wzrost szybkości w stosunku do architektur uzyskiwanych dla algorytmu COSYSOPC. Dla systemów SRSOPC można zauważyć dużo lepsze wykorzystanie zasobów sprzętowych, w których zadania wykonywane są szybciej. Jest to możliwe dzięki kilkukrotnemu użyciu tych samych fragmentów układu FPGA dla różnych zadań. Dynamiczna rekonfiguracja sektorów układu wiąże się z

dodatkowymi czasami reprogramowania. Wpływ tych czasów na szybkość systemu SOPC został zmniejszony dzięki odpowiedniemu uszeregowaniu zadań.

Wyniki eksperymentów pokazują, że liczba sektorów, na jakie podzielony jest układ, w docelowym rozwiązaniu, nie przekracza kilku (dla losowych grafów z tabeli 6.2-1 jest nie więcej niż 5 sektorów *RS*). Tak więc złożoność związana z rozmieszczaniem sektorów (ilość permutacji) jest stosunkowo nieduża. Praktyczna złożoność obliczeniowa algorytmu COSEDYRES w zależności od liczby obliczeń jest rzędu $O(n^3)$. Mimo konieczności wprowadzenia dodatkowych metod, związanych z dodawaniem/ usuwaniem kontekstów czasowych i sprawdzaniem w każdym kroku położenia sektorów w układzie, praktyczna złożoność nie uległa zmianie w stosunku do algorytmu COSYSOPC. W algorytmie COSEDYRES w jednym kroku sprawdzanych jest więcej rozwiązań. Praktyczna złożoność tego algorytmu w zależności od liczby rozwiązań jest rzędu $O(n^2)$, czyli również porównywalna z algorytmem COSYSOPC.

6.2.7 Podsumowanie

Technologia współczesnych częściowo reprogramowalnych układów FPGA pozwala na projektowanie dynamicznie rekonfigurowalnych systemów wbudowanych. W starszych układach czas reprogramowania był zbyt duży (rzędu 100ms) i tym samym dynamiczna rekonfiguracja nie mogła być efektywnie wykorzystywana, natomiast układy o krótkim czasie reprogramowania były zbyt drogie, aby opłacało się wykorzystać zalety dynamicznej rekonfiguracji. Teraz, dzięki krótkim czasom reprogramowania, można w znacznym stopniu przyspieszyć działanie projektowanych systemów SRSOPC. Wyniki eksperymentów pokazują, że poprzez zastosowanie właściwych metody rafinacji w algorytmie COSEDYRES i zrównoleglenie obliczeń z rekonfiguracją sektorów, można uzyskać kilkudziesięcioprocentowe przyspieszenie systemu wbudowanego z wykorzystaniem dynamicznej rekonfiguracji. Chociaż, jeśli czas rekonfiguracji byłby większy od czasów obliczeń i nie dałoby się zrównoleglić obliczeń z rekonfiguracją, wówczas uzyskanie lepszych wyników mogłoby nie być możliwe, lub nawet system mógłby zostać spowolniony (ale w takim przypadku przyjmowana jest implementacja SOPC bez rekonfiguracji). Jednocześnie algorytm COSEDYRES charakteryzuje się niedużą złożonością.

W przyszłości w algorytmie kosyntezy systemów SRSOPC można rozważyć inne metody rekonfiguracji systemów wbudowanych, np. poprzez zastosowanie kilku wbudowanych procesorów, które oprócz sterowania rekonfiguracją, zajmowałyby się także normalnymi obliczeniami. Jednak problem ten może być bardziej złożony, zwiększając tym samym złożoność całego algorytmu. Można również zastosować inne metody generacji sektorów i metody ich rozmieszczenia. Wreszcie system może być opisany w dokładniejszy sposób, poprzez uwzględnienie wzajemnie wykluczających się zadań, co zostanie opisane w kolejnym podrozdziale. W ten sposób będzie można w jeszcze większym stopniu wykorzystać zalety dynamicznej rekonfiguracji systemów wbudowanych.

6.3 Kosynteza systemów SRSOPC reprezentowanych przez warunkowe grafy zadań

Warunkowe grafy zadań umożliwiają specyfikację wzajemnie się wykluczających zadań (ZWW). Mając informacje o zadaniach wykonywanych warunkowo i wykorzystując dynamiczną rekonfigurację systemu możliwe jest przyporządkowanie takich zadań do tego samego sektora w tym samym kontekście czasowym. Wtedy reprogramowanie sektora będzie się odbywać w zależności od aktualnej wartości warunku. W ten sposób w znacznym stopniu można zmniejszyć powierzchnię zajmowaną przez system wbudowany (alokowane są tylko te zadania, dla których spełnione są w danym czasie odpowiednie warunki). Podczas działania systemu wykorzystywane są te same zasoby sprzętowe dla różnych zadań ZWW.

W celu wykorzystania w algorytmie kosyntezy systemów SRSOPC reprezentacji systemu w formie warunkowych grafów zadań, konieczne jest uwzględnienie kilku problemów nie występujących w algorytmie COSEDYRES:

- W metodach rafinacji należy uwzględnić fakt istnienia wzajemnie się wykluczających zadań, podczas przenoszenia zadań pomiędzy zasobami. Może być bardziej opłacalne alokowanie jednocześnie kilku wykluczających się nawzajem zadań do jednego zasobu.
- Należy w inny sposób obliczać powierzchnię: w przypadku przydzielenia zadań ZWW do jednego sektora, w funkcji kosztu należy uwzględnić maksimum z powierzchni zajmowanych przez te zadania (zamiast sumy tych powierzchni).
- Należy zmienić sposób szeregowania zadań.
- Inaczej również obliczana jest szybkość systemu, gdzie w przypadku przyporządkowania zadań ZWW do jednego procesora, będą one uszeregowane równolegle (w danym czasie będzie wykonywane tylko jedno takie zadanie, w zależności od warunku).

W algorytmie COSEDYRES-CTG wprowadzony zostanie m.in. etap etykietowania warunkowego grafu zadań (opisany w kolejnym podrozdziale), nowy algorytm szeregowania zadań, zmienione zostaną metody rafinacji (w celu uwzględnienia zadań ZWW).

6.3.1 Algorytm etykietowania warunkowego grafu zadań

Znalezienie w warunkowym grafie zadań wszystkich zadań ZWW może być stosunkowo trudne, gdy istnieje wiele różnych warunków i gdy te warunki są dodatkowo zagnieżdżone. W celu identyfikacji wszystkich wykluczających się wzajemnie zadań w grafie można zastosować algorytm etykietowania grafu zadań. Wykonanie etykietowania, jako pierwszego etapu w algorytmie COSEDYRES-CTG, ułatwia analizę zadań ZWW w kolejnych krokach algorytmu. Do wyznaczenia zadań ZWW definiuje się dodatkowe parametry zadań w grafie CTG:

DEFINICJA 6.3-1 PARAMETR $C_K(V_J)$.

Parametr $c_k(v_j)$= *cond* jest określony dla węzła v_j odpowiadającemu zadaniu wykonywanemu w zależności od warunku *cond*.

Wartość *cond* jest kolejnym numerem warunku.

DEFINICJA 6.3-2 PARAMETR LEVEL(V_J).

Parametr $level(v_j)$ określa poziom zadania v_j w hierarchii kolejnych zagnieżdżonych ścieżek warunkowych S_{Wk}.

DEFINICJA 6.3-3 PARAMETR V_J->FORK[LEVEL(V_J)].

Parametr v_j->$fork[level(v_j)]$ oznacza zadanie, w którym sprawdzany jest warunek na poziomie $level(v_j)$, w hierarchii kolejnych zagnieżdżonych ścieżek warunkowych S_{Wk},, gdzie $v_j \in S_{Wk}$.

Rekurencyjny algorytm etykietowania krawędzi grafu CTG jest pokazany na rysunku 6.3-1.

Algorytm 6.3-1

```
etykietowanie (vj) {
   Jeżeli level(vj)=0, to ck(vj)=0; cond=0;
   Dla każdego następnika vi wykonuj {
        Jeżeli rozejście bezwarunkowe i ck(vj)=0, to {
                level(vi)= level(vj); ck(vi)=0;
        }
        w przeciwnym przypadku {
                jeżeli rozejście warunkowe, to {
                 level(vi)= level(vj)+1;
                cond++;
                ck(vi)=cond;
                }
                w przeciwnym przypadku jeżeli vi≠vjoin to{
                ck(vi)= ck(vj); level(vi)= level(vj);
                }
                w przeciwnym przypadku jeżeli vi=vjoin to{
                level(vi)= level(vj)-1; ck(vi)=ck(vi->fork[level(vi)]);
                }
                dla każdego m= level(vi), m≥1, m-- wykonuj
                        vi->fork[m]=vj->fork[m];
        }
        etykietowanie(vi);
}
```

Rysunek 6.3-1 Algorytm etykietowania krawędzi

Rozejścia warunkowe mogą być hierarchiczne, tzn. na jednej ścieżce może być sprawdzanych kilka warunków. W związku z tym dla każdego węzła v_j określa się dodatkowo parametry takie jak: warunek $c_k(v_j)$, poziom zadania $level(v_j)$ w hierarchii rozejść warunkowych; numery poprzedników, w

których były sprawdzane warunki na tej samej ścieżce (*branch fork tasks*). Jeśli zadanie v_j nie jest wykonywane warunkowo, to $level(v_j)$=0, a $c_k(v_j)$ i *fork* są nieokreślone. Jeśli w ścieżce warunkowej S_{Wk} rozpoczynającej się od zadania v_{fork}, do węzła łączącego v_{join} znajduje się kilka zadań wykonywanych tylko po spełnieniu danego warunku, to wszystkie węzły, odpowiadające tym zadaniom, mają przypisane takie same wartości warunku $c_k(v_i)$ i ten sam poziom $level(v_i)$, jak zadanie wykonywane bezpośrednio po zadaniu v_{fork}.

TWIERDZENIE 6.3-1

Niech istnieją zadania $v_i, v_j \in CTG$ i parametry zadań będą przypisane zgodnie z alg.6.3-1, niech E_{ci} będzie zbiorem krawędzi warunkowych na ścieżkach S_W od v_i->*fork[1]* do v_i i niech E_{cj} będzie zbiorem krawędzi warunkowych na ścieżkach S_W od v_j->*fork[1]* do v_j.

Zadania v_i i v_j są wzajemnie się wykluczającymi <=> spełnione są następujące warunki:

1) v_j->*fork[1]*=v_i->*fork[1]* i
2) $E_{ci} \not\subset E_{cj} \wedge E_{cj} \not\subset E_{ci}$

Dowód:

Ad 1)

Załóżmy, że v_j->*fork[1]*≠v_i->*fork[1]*. Wtedy:

~ ($\exists S_{Wi}, S_{Wj} : v_i \in S_{Wi} \wedge v_j \in S_{Wj} \wedge v_i \rightarrow fork[1] \in S_{Wi} \wedge v_j \rightarrow fork[1] \in S_{Wj}$), czyli zadania v_i i v_j nie są ZWW. A zatem warunkiem koniecznym, aby zadania v_i i v_j były ZWW jest warunek:

v_j->*fork[1]*=v_i->*fork[1]*.

Ad 2)

Załóżmy, że $E_{ci} \subset E_{cj} \vee E_{cj} \subset E_{ci}$. Wtedy v_i i v_j są na tej samej ścieżce S_W, czyli v_i i v_j nie są ZWW. A zatem warunkiem wystarczającym, aby v_i i v_j były ZWW jest warunek:

$\exists e_{ci} \in E_{ci} \wedge e_{cj} \in E_{cj} : c(e_{ci}) \wedge c(e_{cj}) = 0$ ■

Algorytm etykietowania identyfikuje zatem wszystkie zadania wzajemnie się wykluczające.

6.3.2 Podział i szeregowanie zadań wzajemnie się wykluczających

Jeśli zadania wzajemnie się wykluczające będą przydzielone do tego samego sektora, będzie zmniejszona powierzchnia implementowanego systemu. Powierzchnia zajmowana przez nie wynosi $max(S_{vc}(v_i),\ldots, S_{vc}(v_{i+n}))$, gdzie $S_{vc}(v_i)$ jest powierzchnią zajmowaną przez komponent sprzętowy realizujący zadanie v_i. Opłacalne może być przydzielenie kilku zadań wykluczających się wzajemnie

do jednego sektora, gdyż, aby wykonać jedno z takich zadań nie jest konieczna rekonfiguracja, dopiero po zmianie warunku inne zadania mogą być alokowane w tych samych sektorach układu w wyniku reprogramowania. W przypadku przydzielenia zadań ZWW do jednego *GPP* zadania takie nie muszą być szeregowane sekwencyjnie, mogą być uszeregowane równolegle (wykonywane w zależności od warunku). Tym samym przydzielając zadania ZWW do jednego sektora zmniejsza się powierzchnię systemu, a poprzez przydział takich zadań do *GPP* również czas obliczeń przez ten procesor może być mniejszy, niż w przypadku nie uwzględnienia informacji o warunkowym wykonaniu zadań. Aby to wykorzystać konieczne jest uwzględnienie tych informacji w algorytmie szeregowania. Szkic algorytmu szeregującego zadania ZWW przedstawiony jest na Rysunku 6.3-2.

Algorytm 6.3-2

Jeżeli $c_k(v_i)\neq 0$ *i* $v_i \in PE$, ***to***
Znajdź zadania $v_s \in PE$ *wykluczające się z* v_i*;*
Jeżeli $PE \in GPP$, ***to*** *uszereguj* v_i *najwcześniej jak możliwe i* ***jeżeli*** $T_{pk}(v_i) \subseteq T_{pk}(v_s)$, ***to*** *uszereguj* v_i *i* v_s *równolegle;*
Jeżeli $PE \in RS$, ***to*** $S(v_i) = max(S_{vc}(v_s))$*;*

T_{pk} – czas wykonania zadania w zasobie, p –początek wykonania, k – zakończenie wykonania

Rysunek 6.3-2 Algorytm szeregowania zadań wzajemnie się wykluczających

6.3.3 Miara jakości rozwiązań

W przypadku algorytmu kosyntezy, w którym system wbudowany jest reprezentowany przez warunkowy graf zadań, w funkcji kosztu należy zastosować inny sposób obliczania powierzchni, uwzględniający zadania ZWW przydzielone do jednego zasobu.

Niech $\{v_i,..,v_n\},\ldots,\{v_{n+1},\ldots, v_k\}$ oznaczają zbiory, w których zadania nawzajem się wykluczają i są przydzielone do sektora *RS*, zbiór $\{v_j,..,v_{j+m}\}$ oznacza pozostałe zadania przydzielone do *RS*.

DEFINICJA 6.3-4 POWIERZCHNIA KOMPONENTÓW W SEKTORZE RS.

Powierzchnia zajmowana przez komponenty wykonujące zadania w sektorze *RS* określona jest następująco:

$$S(RS) = \max(S(v_i),...,S(v_n)) + ... + \max(S(v_{n+1}),...,S(v_k)) + \sum_{l=j}^{j+m} S(v_l) \qquad (6.3.1)$$

Powierzchnia systemu wbudowanego obliczana jest wg wzoru (6.2.1). Jednak dzięki przydzieleniu zadań ZWW do jednego sektora liczba sektorów w układzie może być mniejsza, bądź wielkości sektorów mogą być mniejsze i tym samym mniejsza jest powierzchnia systemu wbudowanego.

Podczas obliczania szybkości systemu należy uwzględnić istnienie wzajemnie się wykluczających zadań, jeśli takie są przydzielone do tego samego procesora. Czas wykonania wszystkich zadań przez procesor *GPP* obliczany jest zgodnie z definicją 6.3-5. Niech

$\{v_i,..,v_n\},\ldots,\{v_{n+1},\ldots,\ v_k\}$ oznaczają zbiory zadań , w których zadania nawzajem się wykluczają i są przydzielone do procesora *GPP* , $v_j,..,v_{j+m}$ oznaczają pozostałe zadania przydzielone do *GPP*.

DEFINICJA 6.3-5 CZAS WYKONANIA ZADAŃ PRZEZ PROCESOR GPP.

Czas wykonania wszystkich zadań przydzielonych do *GPP* obliczany jest wg wzoru:

$$T(GPP)= \max(t(v_i),...,t(v_n)) + ... + \max(t(v_{n+1}),...,t(v_k)) + \sum_{l=j}^{j+m} t(v_l) + t_{rest} \qquad (6.3.2)$$

gdzie t_{rest} jest pozostałym czasem związanym z oczekiwaniem procesora na dokończenie wykonywania innych zależnych zadań, wykonywanych w pozostałych zasobach, oraz z czasami transmisji pomiędzy zasobami.

Dla tak zdefiniowanego czasu wykonania wszystkich zadań przez procesor całkowita szybkość systemu obliczana jest wg wzoru (6.2.4).

Parametr α oznaczający stopień wykorzystania powierzchni układu uwzględnia już ewentualne zmniejszenie powierzchni systemu związane z wykonaniem zadań ZWW w tych samych zasobach, tak więc można przyjąć ten sam sposób obliczania globalnego zysku ΔE, jak we wzorze (6.1.5). Również niezmieniony został zysk lokalny $\Delta\varepsilon$ (zgodnie z wzorem (6.2.5)).

6.3.4 Algorytm kosyntezy systemów SRSOPC reprezentowanych przez warunkowe grafy zadań

Na rysunku 6.3-3 przedstawiono szkic algorytmu kosyntezy systemów SRSOPC opisanych grafem CTG: COSEDYRES-CTG, z zaznaczeniem różnic (ciemnym kolorem) w stosunku do algorytmu COSEDYRES. Grupę zadań wzajemnie wykluczających się z v_i oznaczono V_w.

Algorytm 6.3-3

Utwórz architekturę początkową A;
Oblicz powierzchnię S_{akt}; Oblicz szybkość λ systemu A;
powtarzaj
  Z=0;
  dla *każdego PE(RT_i)* ***wykonuj {***
    jeżeli *(ilość zasobów w A)≥2)* ***to {***
      dla *każdego procesora/sektora $PE_j \in A$* ***wykonuj {***
        A'=A- PE_j;
          dla *każdego $v_k \in PE_j$* ***wykonuj {***
            Znajdź $PE_l \in A'$ dające największą wartość $\Delta\varepsilon$ po przydzieleniu do niego zad v_k/V_w;
            Przyporządkuj v_k/V_w do PE_l
          }

```
        jeżeli ΔE>Z to {
              Z=ΔE;  A^best=A';
            }
          }
        }
    dla każdego RS_i  wykonuj {
       dla każdego kontekstu(RS_i) wykonuj {
        A'' = A'- kontekst(RS_i);
        dla każdego v_k∈ kontekst(RS_i) wykonuj {
         Znajdź PE_l∈A'' dające największą wartość Δε po przydzieleniu do niego zad v_k/V_w;
         Przyporządkuj  v_k/V_w do PE_l
      }
    }
    jeżeli ΔE>Z to {
      Z=ΔE;  A^best=A'';
   }
  }
A'''=A'' ∪ PE(RT_i);
    powtarzaj
      Znajdź zadanie v_k/V_w  o największym współczynniku Δε po przydzieleniu go doPE(RT_i);
      jeżeli  Δε >0 to przyporządkuj v_k/V_w  do PE(RT_i);
    dopóki nie ma już zadań po przeniesieniu których Δε >0 ;
    dla każdego RS_i wykonuj {
        dla każdego k=0 do ilość_kontekstów(RS_i) wykonuj {
              A'''= A''' ∪ kontekst_k(RS_i); Przesuń pozostałe konteksty RS_i
              Powtarzaj
                Znajdź zadanie v_k/V_w  o największym współczynniku Δε po przydzieleniu go do
                kontekst_k(RS_i);
                jeżeli  Δε >0 to przyporządkuj v_k/V_w  do  kontekst_k(RS_i);
              dopóki nie ma już zadań po przeniesieniu których Δε >0 ;
                                                    }
                          }
    A'''=A'''- PEs bez przydzielonych  zadań;
    jeżeli ΔE>Z to {
       Z=ΔE;  A^best=A''';
    }
 Rozmieść sektory w FPGA;
 }
 jeżeli Z>0 to A=A^best;
dopóki Z>0;
```

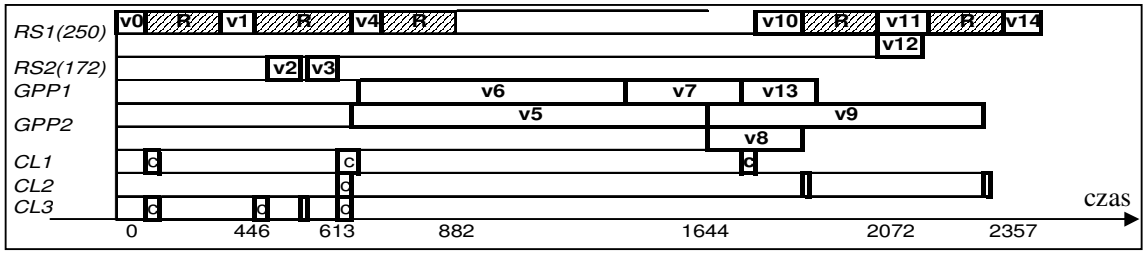
Rysunek 6.3-3 Zarys algorytmu kosyntezy dla systemów SRSOPC opisanych grafem CTG

W stosowanych metodach rafinacji w algorytmie COSEDYRES-CTG różnica w stosunku do algorytmu COSEDYRES występuje tylko podczas lokalnych zmian w systemie, tzn. podczas przenoszenia zadań (w wewnętrznych pętlach). Dla przenoszonego zadania v_i algorytm sprawdza, czy nie występują jeszcze inne zadania, które wzajemnie wykluczają się z v_i. Jeśli jest kilka takich zadań, to następuje próba przeniesienia równocześnie wszystkich takich zadań do dodawanego zasobu,

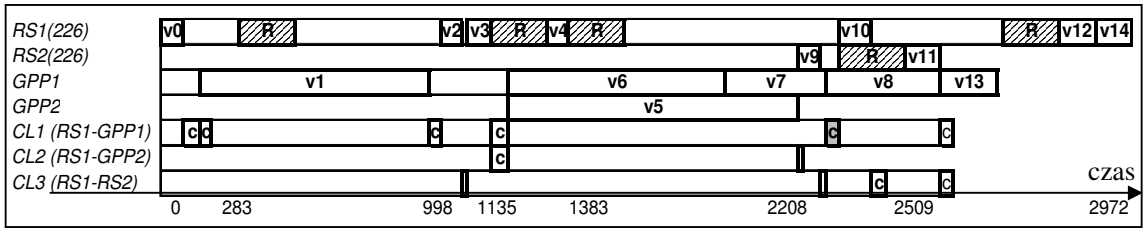
natomiast przy usuwaniu zasobu – również przesunięcia zadań wykluczających się nawzajem do jednego zasobu. Przenoszone jest tyle zadań wzajemnie się wykluczających ile jest możliwe, bo może nie dać się przenieść wszystkich takich zadań, np. ze względu na ograniczoną powierzchnię sektora *RS*. Przenosząc taką grupę zadań jest bardzo duże prawdopodobieństwo zmniejszenia kosztu systemu, a co za tym idzie w dalszych krokach zwiększenie szybkości.

PRZYKŁAD 6.3-1.

Na rysunku 6.3-4 przedstawiono uszeregowanie zadań i rekonfiguracji w wyniku zastosowania algorytmu COSEDYRES-CTG, dla systemu SRSOPC specyfikowanego grafem z przykładu 5.2-1b (warunkowy graf zadań). Rysunek 6.3-5 pokazuje uszeregowanie zadań i rekonfiguracji po zastosowaniu algorytmu COSEDYRES , w którym nie uwzględniono informacji o wzajemnie się wykluczających zadaniach (dla grafu z przykładu 5.2-1a).



Rysunek 6.3-4 Wykres Gantta dla systemu SRSOPC reprezentowanego przez warunkowy graf zadań



Rysunek 6.3-5 Wykres Gantta dla systemu SRSOPC optymalnego bez uwzględnienia informacji o wzajemnie się wykluczających zadaniach

W obu przypadkach przyjęto jednakowe ograniczenie powierzchni: 1100CLB. Wszystkie parametry w bibliotece komponentów również były takie same. Grafy zadań mają tą samą strukturę, różnią się tylko uwzględnieniem informacji o zadaniach wykonywanych warunkowo w grafie 5.2-1b. W wyniku kosyntezy systemu reprezentowanego przez warunkowy graf zadań otrzymano rozwiązanie o czasie wykonania wszystkich zadań 2357μs i powierzchni 1082 CLB, podczas gdy bez uwzględnienia informacji o zadaniach ZWW czas wykonania wszystkich zadań systemu SRSOPC wynosi 2972 μs, przy podobnej powierzchni: 1087CLB. Wzrost szybkości o 21% jest spowodowany m.in. lepszym wykorzystaniem powierzchni sektora *RS1*. Zadania *v11* i *v12* zajmują tą samą powierzchnię układu i są wykonywane w zależności od warunku, który wystąpił (uszeregowane równolegle). Dwa inne wzajemnie się wykluczające zadania *v8* i *v9* przydzielone zostały do wykonania przez procesor, a ponieważ jest to ten sam procesor, zostały one uszeregowane równolegle (wykonywane tylko jedno z

nich, po spełnieniu warunku). Zadania te nie zostały alokowane w jednym sektorze, ze względu na bardzo dużą powierzchnię zajmowaną przez *VC* realizujący zadanie *v8* (482 CLB). Reprogramowanie tak dużej powierzchni nie byłoby opłacalne, bo wymagałoby przesunięcia w czasie wykonania pozostałych zadań alokowanych w sprzęcie, tak, aby rekonfiguracje poszczególnych sektorów nie występowały w tym samym czasie.

6.3.5 Wyniki wykonanych eksperymentów

W celu pokazania dodatkowych możliwości zwiększenia jakości projektowanych systemów SRSOPC, w przypadku reprezentowania systemu przez warunkowy graf zadań, wykonano eksperymenty dla losowych grafów. Porównano wyniki uzyskane dla systemu reprezentowanego przez warunkowe grafy zadań z wynikami dla systemów, w których nie uwzględniono informacji o wzajemnie się wykluczających zadaniach. W obu przypadkach wszystkie parametry w bibliotece komponentów były takie same, również taka sama była struktura grafu zadań. W bibliotece komponentów dostępne były: jeden typ procesora *GPP*, po jednym typie *VC* dla każdego zadania, jeden typ łącza komunikacyjnego. Dostępna powierzchnia była zwiększana proporcjonalnie wraz ze wzrostem wielkości grafu. Wraz ze wzrostem liczby węzłów w grafie zwiększano również proporcjonalnie liczbę wzajemnie się wykluczających zadań (od jednej pary dla najmniejszego grafu). Kolejne kolumny tabeli wyników 6.3-1 zawierają: liczbę zadań grafu, ograniczenie powierzchni dla systemu wbudowanego, dla każdego z algorytmów kolejno: czas wykonania zadań i powierzchnię zajmowana przez system otrzymany w wyniku kosyntezy, liczbę zadań wykonywanych sprzętowo (HW). Dwie ostatnie kolumny to: liczba wzajemnie się wykluczających zadań w grafie CTG i procentowy wzrost szybkości systemów SRSOPC reprezentowanych przez grafy CTG, w stosunku do szybkości systemów reprezentowanych przez grafy TG.

Graf	ograniczenie powierzchni	COSEDYRES			COSEDYRES-CTG			liczba par zadań ZWW	przyrost szybkości [%]
		czas	powierzchnia	HW	czas	powierzchnia	HW		
10	1500	**941**	1498	4	**804**	1490	5	1	**15**
30	2000	**4642**	1863	13	**3448**	1986	15	2	**26**
50	2500	**7054**	2303	6	**6531**	2373	13	3	**8**
70	3000	**9174**	2935	21	**8396**	2964	35	4	**9**

TABELA 6.3-1. PORÓWNANIE WYNIKÓW DLA ALGORYTMÓW COSEDYRES I COSEDYRES-CTG

Wyniki eksperymentów pokazują, że uwzględnienie informacji o wzajemnie się wykluczających zadaniach w grafie zadań prowadzi do zwiększenia szybkości systemu SRSOPC otrzymanego w wyniku kosyntezy wykonanej algorytmem COSEDYRES-CTG. Dzięki dynamicznej rekonfiguracji w wyniku przydzielenia zadań ZWW do jednego sektora zwiększa się powierzchnia dla zadań wykonywanych sprzętowo, gdyż zadania ZWW zajmują wtedy tę samą powierzchnię układu FPGA (są wykonywane w zależności od warunku, który wystąpił). Wzrost szybkości zależy jednak od liczby zadań ZWW w grafie CTG. Jeśli liczba zadań ZWW, w stosunku do liczby wszystkich zadań w grafie, jest niewielka, to również wzrost szybkości nie będzie duży, bo korzyść wynikająca z równoległego

uszeregowania zadań ZWW w tym samym zasobie zostanie zniwelowana przez pozostałe zadania. Również istotne jest to, gdzie zadania ZWW są usytuowane w warunkowym grafie zadań. Jeżeli znajdują się na ścieżce krytycznej, to korzyść z umieszczenia ich w jednym zasobie będzie większa, niż gdyby były wykonywane na ścieżkach w mniejszym stopniu decydujących o szybkości całego systemu wbudowanego (w eksperymentach zadania ZWW zostały rozmieszczone losowo).

6.4 Skuteczność stosowania dynamicznej rekonfiguracji w systemach wbudowanych

W tym podrozdziale zostanie przeprowadzona analiza i ocena, mająca określić czy stosowanie dynamicznej rekonfiguracji w systemach wbudowanych jest zawsze skuteczne, czy też są pewne przypadki, dla których może się ona jednak okazać nieskuteczna. W tym celu na początku podana zostanie definicja skuteczności dynamicznej rekonfiguracji.

Niech A_{SOPC}^{best} będzie rozwiązaniem otrzymanym w wyniku kosyntezy systemu SOPC, a A_{DRSOPC}^{best} będzie rozwiązaniem otrzymanym w wyniku kosyntezy systemu DRSOPC.

DEFINICJA 6.4-1 SKUTECZNOŚĆ DYNAMICZNEJ REKONFIGURACJI

Dynamiczna rekonfiguracja jest skuteczna, gdy $\Delta E(A_{SOPC}^{best}, A_{DRSOPC}^{best}) > 0$.

Dynamiczna rekonfiguracja jest zatem skuteczna, gdy:

1) Prowadzi do uzyskania systemu tańszego przy tej samej lub większej szybkości, lub
2) Prowadzi do uzyskania systemu szybszego, nie przekraczając zadanego ograniczenia na koszt.

6.4.1 Minimalizacja kosztu

Dynamiczna rekonfiguracja prowadząca do uzyskania systemu tańszego będzie skuteczna wówczas, jeśli spełniony będzie układ nierówności:

$$\begin{cases} S_{SOPC} > S_{DRSOPC} \\ \lambda_{SOPC} \leq \lambda_{DRSOPC} \end{cases} \quad (6.4\text{-}1)$$

gdzie S_{SOPC} jest powierzchnią systemu SOPC obliczaną zgodnie ze wzorem 6.1.1, S_{DRSOPC} jest powierzchnią systemu DRSOPC obliczaną zgodnie ze wzorem 6.2.1, λ_{SOPC} jest szybkością systemu SOPC obliczaną zgodnie ze wzorem 6.1.3, a λ_{DRSOPC} jest szybkością systemu DRSOPC obliczaną zgodnie ze wzorem 6.2.4.

Dynamiczna rekonfiguracja nie będzie skuteczna, gdy w najszybszym możliwym systemie SOPC, ze względu na ograniczenie S_{max}, poza najtańszym modułem *GPP*, alokowany jest tylko jeden

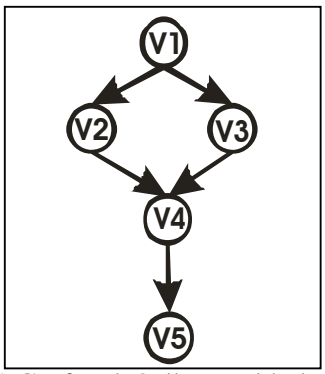
najtańszy moduł VC_k, a alokowanie każdego większego modułu *VC* przekracza powierzchnię S_{max}. Jeśli VC_k jest komponentem o najmniejszej powierzchni w bibliotece dostępnych komponentów, to nigdy nie znajdą się takie zadania, które wykonywane sprzętowo w wyniku dynamicznej rekonfiguracji spowodują zmniejszenie powierzchni systemu (zakładając, że w specyfikacji systemu znajdują się przynajmniej dwa zadania i konieczność istnienia przynajmniej jednego *GPP* w architekturze). Czasami zmniejszenie powierzchni w wyniku dynamicznej rekonfiguracji, może wiązać się z koniecznością zmniejszenia szybkości projektowanego systemu, spowodowaną czasem reprogramowania. Wówczas również dynamiczna rekonfiguracja nie jest skuteczna. Taki przypadek ma miejsce, gdy czasy reprogramowania są na tyle duże, że nie można w tym samym czasie wykonać innych obliczeń na pozostałych zasobach, tak, aby czasy reprogramowania nie zwiększyły całkowitego czasu wykonania zadań systemu. Dynamiczna rekonfiguracja może prowadzić do uzyskania systemu tańszego, o nie mniejszej szybkości, gdy w systemie SOPC mogą być alokowane, poza najtańszym *GPP*, przynajmniej dwa moduły *VC*, ale czasy reprogramowania modułów nie zmniejszają szybkości systemu.

W momencie, gdy dla danej specyfikacji systemu dynamiczna rekonfiguracja nie jest skuteczna, to można zwiększyć granulację zadań systemu. Wówczas przy większej liczbie zadań, niektóre zadania mogą być wykonane równolegle, z mniejszymi czasami i przy mniejszych powierzchniach poszczególnych zadań realizowanych jako *VC*, tak, aby więcej zadań zmieściło się w powierzchni S_{max}. Można również dodać do biblioteki komponentów tańsze moduły *VC* i *GPP*.

Dynamiczna rekonfiguracja stwarza możliwość obniżenia kosztu systemu (nie zmieniając szybkości) wtedy, gdy system SOPC jest zadowalający pod względem szybkości. Zdarzają się przypadki, kiedy nie da się obniżyć kosztu systemu poprzez dynamiczną rekonfigurację, ale są to przypadki rzadko spotykane w praktyce, ponadto można większość z nich wyeliminować poprzez modyfikację specyfikacji lub biblioteki komponentów.

PRZYKŁAD 6.4-1.

Na rys. 6.4-1 przedstawiono przykład systemu opisanego grafem zadań. Parametry zadań podane są w tabeli 6.4-1.

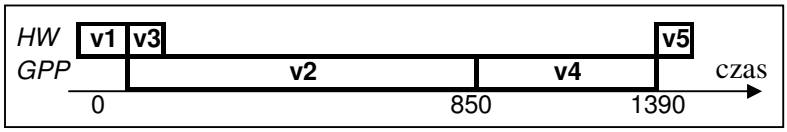


Rysunek 6.4-1 Graf zadań dla przykładowego systemu

Zadanie	SW		HW	
	t	CM	t	S
v1	1000	1500	50	200
v2	800	1000	10	400
v3	1200	1100	30	150
v4	500	2000	8	500
v5	1000	1500	40	200

TABELA 6.4-1. BIBLIOTEKA KOMPONENTÓW DLA SYSTEMU OPISANEGO GRAFEM Z RYS.6.4-1

Załóżmy, że powierzchnia modułu procesora *GPP* wynosi 200 CLB. Dla tak opisanego systemu przyjęto ograniczenie powierzchni: 800 CLB. Uzyskano architekturę systemu SOPC z jednym procesorem *GPP* oraz trzema *VC*. Uszeregowanie dla systemu SOPC pokazano na rys. 6.4-2.

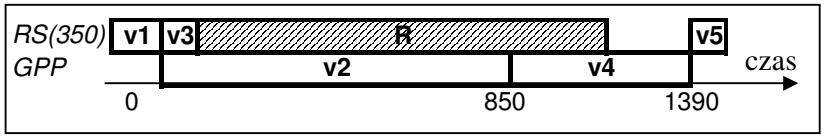


Rysunek 6.4-2 Uszeregowanie zadań dla systemu SOPC

Czas wykonania wszystkich zadań takiego systemu wynosi 1390 μs, a powierzchnia 750 CLB. Jest to najszybsze rozwiązanie w postaci systemu SOPC dla przyjętego ograniczenia powierzchni.

Minimalizacja kosztu systemu

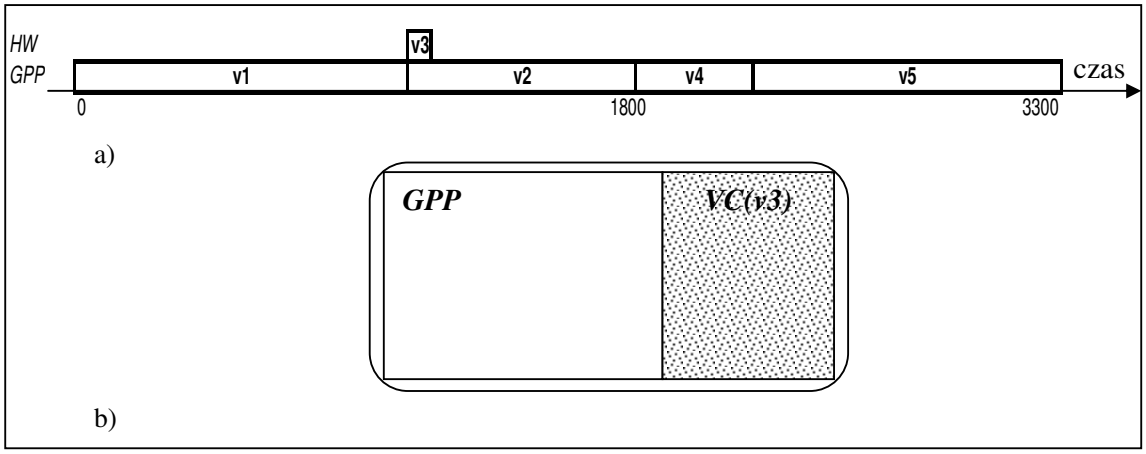
Niech czas reprogramowania wynosi 3,4μs/CLB. Załóżmy, że celem optymalizacji będzie znalezienie tańszej architektury, niż dla systemu SOPC, przy zachowaniu tej samej szybkości. Ze względu na to, iż w systemie SOPC trzy zadania były wykonywane sprzętowo, można stwierdzić, że dynamiczna rekonfiguracja będzie w tym przypadku skuteczna. Otrzymano architekturę z jednym procesorem *GPP* oraz jednym sektorem o powierzchni 350. Sektor ten jest dynamicznie rekonfigurowany w trakcie działania systemu. Uszeregowanie dla systemu DRSOPC pokazano na rys. 6.4-3.



Rysunek 6.4-3 Uszeregowanie zadań dla systemu DRSOPC

Szybkość systemu DRSOPC nie wzrosła (t=1390 μs), ale powierzchnia projektowanego systemu jest mniejsza (550 CLB). Możliwe byłoby uzyskanie jeszcze tańszego systemu poprzez dwukrotną rekonfigurację sektora RS, ale kosztem zmniejszenia szybkości systemu. Nie jest również możliwe, dla takiej specyfikacji systemu, otrzymanie systemu szybszego w wyniku zastosowania dynamicznej rekonfiguracji.

Załóżmy teraz, że dla systemu SOPC mamy ograniczenie powierzchni 350 CLB. W takiej powierzchni zmieści się jeden procesor oraz tylko jeden najmniejszy komponent VC implementujący zadanie *v3*, co zostało pokazane na rys. 6.4-4.



Rysunek 6.4-4 System SOPC przy ograniczeniu S_{max}=350CLB: a) wykres Gantta, b) architektura systemu SOPC

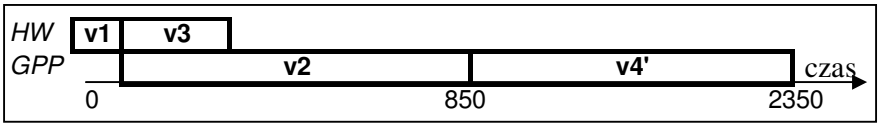
Dla takiego ograniczenia powierzchni nie jest możliwe uzyskanie jeszcze tańszego systemu przy zachowaniu jego szybkości z wykorzystaniem dynamicznej rekonfiguracji, bez zmiany specyfikacji systemu. Nie ma również możliwości alokowania dodatkowych zadań w sprzęcie i tym samym przyspieszenia projektowanego systemu. Dla współczesnych układów FPGA takie przypadki zdarzają się jednak bardzo rzadko.

Załóżmy, że zostanie specyfikacja systemu reprezentowanego przez graf z rys. 6.4-1, w ten sposób, że zadania *v4* i *v5* wykonywane sekwencyjnie zostaną połączone w jedno zadanie *v4'* oraz zostanie zwiększony czas wykonania zadania *v3* w sprzęcie do 200µs. Zmodyfikowaną bibliotekę komponentów uwzględniającą w/w zmiany przedstawiono w tabeli 6.4.2.

Zadanie	SW		HW	
	t	CM	t	S
v1	1000	1500	50	200
v2	800	1000	10	400
v3	1200	1100	200	150
v4'	1500	3500	48	700

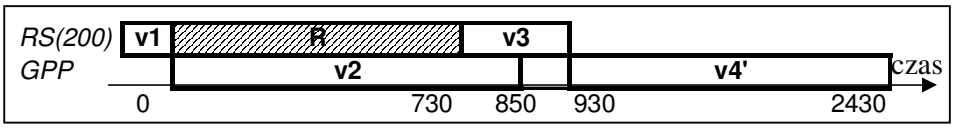
TABELA 6.4-2. ZMODYFIKOWANA BIBLIOTEKA KOMPONENTÓW

Dla tak określonych danych przyjęto ograniczenie powierzchni: 600 CLB. Wówczas uszeregowanie zadań dla systemu implementowanego w postaci SOPC przedstawiono na rys. 6.4-5.



Rysunek 6.4-5 Uszeregowanie zadań dla zmodyfikowanego systemu SOPC

Czas wykonania zadań systemu SOPC wynosi 2350μs, a powierzchnia: 550 CLB. Jest to najszybszy system SOPC, nie przekraczający powierzchni 600 CLB. Możliwe jest zmniejszenie powierzchni tego systemu poprzez dynamiczną rekonfigurację. Zadań *v2* i *v4'* nie opłaca się wykonywać sprzętowo, ze względu na zbyt dużą powierzchnię *VC*, zatem jedyną możliwością zmniejszenia powierzchni systemu (poza implementacją całego w *GPP* – dużo wolniejszą) jest reprogramowanie fragmentu FPGA przed wykonaniem zadania *v3*. Wykres Gantta dla takiego systemu DRSOPC przedstawiono na rys. 6.4-6.



Rysunek 6.4-6 Uszeregowanie zadań dla systemu DRSOPC

Zmniejszenie powierzchni systemu do 400 CLB, poprzez dynamiczną rekonfigurację, spowodowało wzrost czasu wykonania zadań systemu do 2430μs. Wzrost ten jest spowodowany czasem reprogramowania sektora *RS*, który wymusił przesunięcie w czasie wykonania zadania *v4'* (musi być wykonane po *v3*). W takim przypadku dynamiczna rekonfiguracja nie jest skuteczna, bo powoduje obniżenie powierzchni kosztem obniżenia szybkości.

6.4.2 Maksymalizacja szybkości

W celu oceny skuteczności dynamicznej rekonfiguracji prowadzącej do uzyskania systemu szybszego należy uwzględnić, oprócz różnicy czasów wykonania zadań po przeniesieniu ich do sprzętu, także czasy reprogramowania sektorów RS. Zakładając, że $S_{SOPC} \leq S_{max}$ (S_{SOPC} jest powierzchnią systemu SOPC obliczaną zgodnie ze wzorem 6.1.1), dynamiczna rekonfiguracja prowadząca do uzyskania systemu szybszego, który nie przekracza S_{max}, będzie skuteczna wówczas, jeśli spełniony będzie układ nierówności:

$$\begin{cases} \lambda_{SOPC} < \lambda_{DRSOPC} \\ S_{DRSOPC} \leq S_{max} \end{cases} \qquad (6.4\text{-}3)$$

gdzie S_{DRSOPC} jest powierzchnią systemu DRSOPC obliczaną zgodnie ze wzorem 6.2.1, λ_{SOPC} jest szybkością systemu SOPC obliczaną zgodnie ze wzorem 6.1.3, a λ_{DRSOPC} jest szybkością systemu DRSOPC obliczaną zgodnie ze wzorem 6.2.4.

Dynamiczna rekonfiguracja nie prowadzi do uzyskania systemu szybszego (nie przekraczającego S_{max}), wówczas, gdy nie jest możliwe zmniejszenie wpływu czasu reprogramowania sektorów na całkowity czas wykonania wszystkich zadań systemu. Jeśli czasy reprogramowania będą zbyt długie w stosunku do czasów zadań wykonywanych równolegle w pozostałych sektorach lub przez *GPP*, to dynamiczna rekonfiguracja nie będzie skuteczna, w przeciwnym przypadku wpływ czasów reprogramowania na szybkość systemu będzie pomijalny i tym samym skuteczność stosowania dynamicznej rekonfiguracji w celu uzyskania systemu szybszego może być duża.

PRZYKŁAD 6.4-2.

Niech system będzie opisany grafem z rys. 6.4-1, a parametry zadań będą takie jak w tabeli 6.4-1. Najszybszy, uzyskany w wyniku kosyntezy, systemu SOPC nie przekraczający powierzchni 800 CLB przedstawiony jest na rys. 6.4-2 (czas wykonania zadań: 1390 μs). Zadań *v2* i *v4* nie można wykonać sprzętowo, bo przekroczona zostałaby dozwolona powierzchnia. Najbardziej opłacalne, ze względu na szybkość, jest wykonanie pozostałych zadań w sprzęcie.

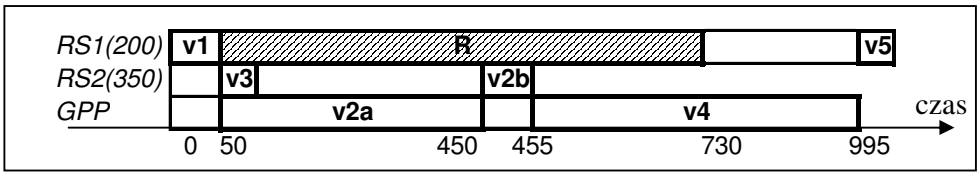
Maksymalizacja szybkości systemu

Załóżmy, że celem optymalizacji będzie znalezienie szybszej architektury, niż dla systemu SOPC, przy ograniczeniach takich samych jak dla systemu SOPC, tak aby system nie przekraczał powierzchni 800 CLB. Tak jak zostało wspomniane w przykładzie 6.4-1 dla ograniczeń powierzchni 800 CLB i 350CLB wzrost szybkości projektowanego systemu poprzez dynamiczną rekonfigurację nie jest możliwy. Dla specyfikacji z tabeli 6.4-1 opłacalne mogą być teoretycznie rekonfiguracje zadań: *v1* (680+50<1000), *v3* (510+30<1200) oraz *v5* (680+40<1000), natomiast ze względu na zbyt duże czasy reprogramowania w stosunku do czasu wykonania zadania przez *GPP* nie jest opłacalna rekonfiguracja zadania *v2* (1360+10>800) i *v4* (1700+8>500). W praktyce okazuje się, że więcej niż jedna rekonfiguracja w trakcie działania systemu jest w tym przypadku nieopłacalna, ponieważ minimalna powierzchnia komponentów sprzętowych przydzielonych do sektora *RS*, który miałby być dynamicznie rekonfigurowany, to 200CLB, czyli sam czas dwóch rekonfiguracji wyniósłby 1360 μs. Natomiast przy jednej rekonfiguracji sektora dynamiczna rekonfiguracja mogłaby być skuteczna, gdyby nieznacznie zmienić specyfikację, tzn. podzielić zadanie *v2* lub *v4* na dwa (lub więcej) mniejsze zadania. Rozważmy wobec tego specyfikację, w której zadanie *v2* jest podzielone na dwa mniejsze *v2a* i *v2b* i wykonywane sekwencyjnie (o takich samych parametrach). Fragment zmienionej specyfikacji przedstawiono w tabeli 6.4-3.

Zadanie	SW		HW	
	t	CM	t	S
v2a	400	500	5	200
v2b	400	500	5	200

TABELA 6.4-3. FRAGMENT BIBLIOTEKI KOMPONENTÓW DLA ZMODYFIKOWANEJ SPECYFIKACJI.

Dla takiej specyfikacji najszybszy system SOPC, który można zaimplementować w powierzchni nie większej niż 800 CLB, jest taki sam jak na rys. 6.4-2, przy czym zadanie *v2* wykonywane jest jako dwa sekwencyjne *v2a* i *v2b*. Natomiast wykorzystując dynamiczną rekonfigurację, tym razem jest ona skuteczna. Uszeregowanie zadań dla systemu DRSOPC ze zmienioną specyfikacją przedstawiono na rys. 6.4-7.



Rysunek 6.4-7 Wykres Gantta dla systemu DRSOPC opisanego zmodyfikowaną specyfikacją

Czas wykonania wszystkich zadań, po zastosowaniu dynamicznej rekonfiguracji, wynosi 995 μs (zysk 40%), przy mniejszej powierzchni: 750CLB. Zatem czasami konieczna jest zmiana specyfikacji projektowanego systemu, aby dynamiczna rekonfiguracja była skuteczna.

Na podstawie powyższych rozważań, poniżej przedstawiono pewne właściwości określające, kiedy dynamiczna rekonfiguracja może nie być skuteczna. Niech v_k będzie zadaniem, które ma być wykonane po rekonfiguracji sektora *RS*, $t_{pk}(v_k)$ oznacza przedział czasu wykonania zadania v_k, $t_{recpk}(v_k)$ oznacza przedział czasu potrzebny na rekonfigurację v_k, $t_{rec}(v_k)$ oznacza czas reprogramowania sektora, w którym ma być wykonywane v_k, $t_{GPP}(v_k)$ i $t_{VC}(v_k)$ oznaczają czasy wykonania v_k odpowiednio przez *GPP* i *VC* Dynamiczna rekonfiguracja może nie być skuteczna, gdy:

1. $t_{GPP}(v_k) \leq t_{rec}(RS_k) + t_{VC}(v_k)$,

   Warunek ten oznacza, że jeśli czas wykonania zadania przez procesor będzie mniejszy od sumarycznego czasu rekonfiguracji i czasu wykonania tego zadania przez *VC*, to rekonfiguracja sektora może nie być opłacalna.

2. $\sim (\exists v_i,...,v_j,v_k : \{v_i,...,v_j\} \notin RS_k \wedge v_k \in RS_k \wedge$

   $(t_{pk}(v_i) \cap t_{recpk}(v_k) \neq 0 \wedge ... \wedge t_{pk}(v_j) \cap t_{recpk}(v_k) \neq 0) \wedge$
   $((t_{pk}(v_i) \cap t_{recpk}(v_k)) + ... + (t_{pk}(v_j) \cap t_{recpk}(v_k)) \geq t_{rec}(v_k))$

   Warunek oznacza, że dynamiczna rekonfiguracja nie jest skuteczna, gdy nie istnieje możliwość zrównoleglenia czasu reprogramowania sektora z innymi obliczeniami wykonywanymi przez procesory, bądź inne sektory.

Im większa jest powierzchnia *VC*, tym dynamiczna rekonfiguracja sektora *RS*, w celu alokacji *VC*, może być mniej skuteczna, bo wydłuża się czas reprogramowania sektora. W takich przypadkach, aby dynamiczna rekonfiguracja była bardziej skuteczna, można zwiększyć granulację zadań, wykonać reorganizację grafu zadań, tak, aby niektóre z zadań podzielić na mniejsze (wykonywane równolegle, jeśli jest to możliwe). Można zmienić komponenty dostępne w bibliotece na tańsze. Wtedy można zastosować więcej sektorów (dla większej liczby *VC*) o mniejszych powierzchniach, a co za tym idzie takie, których rekonfiguracja trwa o wiele krócej.

Przeprowadzone eksperymenty, przedstawione w p. 6.2.7, pokazują, że dynamiczna rekonfiguracja, w zdecydowanej większości przypadków, jest skuteczna. Rzadko zdarzają się sytuacje, gdy dynamiczna rekonfiguracja nie prowadzi do zmniejszenia powierzchni projektowanego systemu lub zwiększenia jego szybkości. Najczęściej takie sytuacje mogą się zdarzyć przy specyfikacji systemu w postaci grafów z małą liczbą zadań i grafów z dużą liczbą zależności pomiędzy zadaniami, jeśli trudno jest zrównoleglić obliczenia z rekonfiguracją sektorów, ale takie przypadki zwykle można wyeliminować poprzez zmianę specyfikacji systemu lub zmianę biblioteki komponentów.

7 PRZYKŁADY

W rozdziale 6 zaprezentowane zostały wyniki kosyntezy wykonanej algorytmem COSEDYRES, uzyskane dla losowych grafów zadań. Losowe przykłady mogą jednak generować nierealistyczne parametry, które w rzeczywistości mogą nigdy nie wystąpić. Dlatego, aby wykazać skuteczność algorytmu COSEDYRES dla praktycznych systemów, w tym rozdziale przedstawione zostaną wyniki wykorzystania tego algorytmu dla głównych dziedzin stosowania systemów wbudowanych, czyli cyfrowego przetwarzania sygnałów (DSP), systemów komunikacyjnych i systemów sieciowych. Pierwszy przykład dotyczy kompresji zdjęć w aparacie cyfrowym, następny to moduł translatora transakcji wchodzącego w skład koncentratora USB2.0, a trzeci to dynamicznie rekonfigurowany serwer internetowy wbudowany w układ FPGA. Wszystkie przykłady zostały zsyntezowanie na komputerze w konfiguracji z procesorem Intel Core 2 Duo 1,86GHz, z pamięcią 1GB RAM.

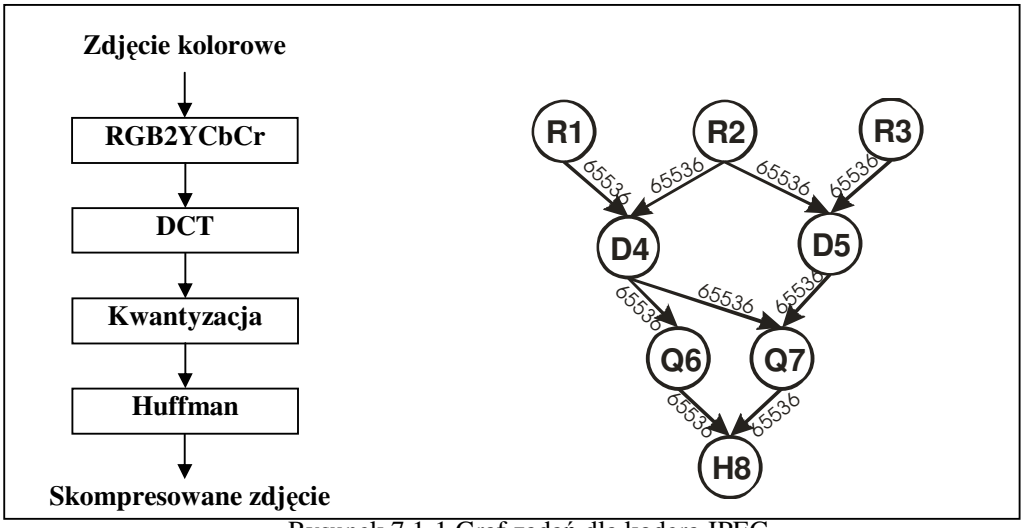
7.1 Fotograficzny aparat cyfrowy

Przykładem systemu z zakresu DSP jest fotograficzny aparat cyfrowy. Działanie aparatu można podzielić na trzy główne funkcje: przetwarzanie obrazu przez element CCD, kompresja obrazu do wybranego formatu (jpeg, png, itp.) oraz zapis skompresowanego zdjęcia do pamięci (wbudowanej lub na kartę pamięci). Pominięte zostały inne funkcje aparatu cyfrowego, które nie wymagają optymalizacji. Przechwytywanie obrazu wykonywane jest sprzętowo, a zapisywanie skompresowanego obrazu w pamięci wykonywane jest programowo. Nie ma tutaj zatem konieczności stosowania kosystezy. Głównym elementem takiego aparatu, który pozwala na zastosowanie możliwości dynamicznie rekonfigurowalnych heterogenicznych systemów wbudowanych SOPC, jest moduł kompresji **JPEG**. Wówczas można wykorzystać kosyntezę do znalezienia architektury pozwalającej na uzyskanie jak najszybszego czasu kompresji zdjęcia, takiej, aby moduł ten zmieścił się w układzie. Algorytm kodera JPEG wraz z odpowiadającym mu grafem zadań przedstawiono na Rys. 7.1-1. Jest on taki sam jak opisywany w pracach [BBD05b] i [FVAGMT07], aby można było porównać te metody z algorytmem COSEDYRES. Algorytm kodera JPEG składa się z czterech procesów wykonywanych sekwencyjnie:

- konwersja kolorów RGB na jasność (luminancje) i 2 kanały barwy (chrominancje) YCbCr,
- dyskretna transformata kosinusowa (DTC),
- kwantyzacja, czyli zastąpienie danych zmiennoprzecinkowych przez liczby całkowite,
- algorytm Huffmana.

Czasy wykonania dla tych czterech procesów przyjęto takie same jak w [FVAGMT07], natomiast ze względu na to, że w żadnej z prac nie opublikowano danych dotyczących wielkości komponentów *VC*, jedynie podano informacje o zajętości układu w przypadku implementacji całego systemu sprzętowo,

wielkości te zostały przeliczone w oparciu o te informacje. W celu zwiększenia możliwości równoległego przetwarzania danych zastosowano kilka instancji tych samych typów zadań kodera JPEG. Parametry poszczególnych zadań zostały umieszczone w Tabeli 7.1-1.



Rysunek 7.1-1 Graf zadań dla kodera JPEG.

Zadanie		*HW*		*SW*
		t [us]	*S [CLB]*	*t [us]*
R1, R2, R3	RGB2YCbCr	1310,82	39	5243,28
D4, D5	DCT	9175,14	65	36700,56
Q6, Q7	Kwantyzacja	1310,98	87	5243,92
H8	Huffman	1310	160	5244

TABELA 7.1-1. PARAMETRY ZADAŃ DLA KODERA JPEG DLA 8-BITOWYCH BLOKÓW DANYCH 256X256 PIKSELI.

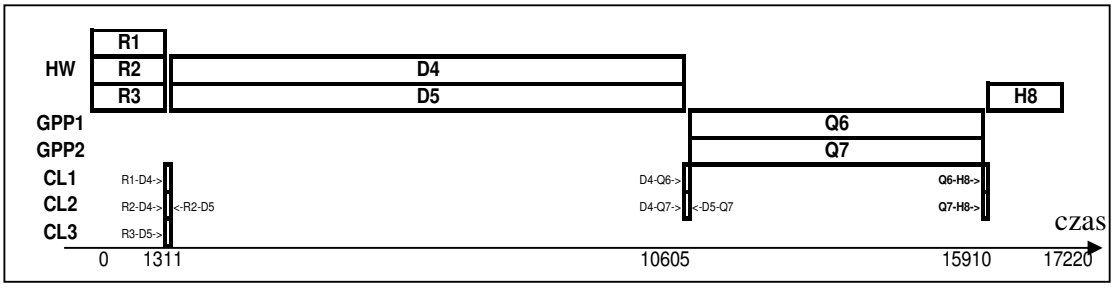
W obu pracach [BBD05b] i [FVAGMT07] przyjęto, że dla implementacji softwarowej dla każdego zadania czasy wykonania są cztery razy mniejsze niż w modułach sprzętowych. Wszystkie dane w Tabeli 7.1-1 dotyczą bloków danych 8-bitowych o rozmiarze 256x256 pikseli. W pracy [BBD05b] obliczono, że gdyby wszystkie zadania, były wykonywane sprzętowo, mogłyby zająć w układzie FPGA Xilinx Virtex 2 XC2V2000 powierzchnię równą 11 kolumnom, więc mniejszą niż powierzchnia tego FPGA. W celu optymalizacji parametrów przyjęto wobec tego ograniczenie: 8 kolumn (**448 CLB**). Przy czym w [BBD05b] i [FVAGMT07] zakłada się, że istnieje tylko jeden procesor - PowerPC, czyli wbudowany na stałe w układzie FPGA. Zatem w pracy [BBD05b] zakłada się, że te 8 kolumn pozostaje tylko dla impelmentacji sprzętowej. Zaimplementowano system w układzie Xilinx Virtex 2 XC2V2000, który posiada 2688 CLB (56 rzędów x 48 kolumn). Układ ma 22 ramki przypadające na jedną kolumnę CLB (1456 ramek w całym układzie). Czas reprogramowania całego układu (dla trybu programowania SelectMap 50 MHz) wynosi 17,01ms. Częstotliwość reprogramowania wynosi 66MHz. Zatem czas reprogramowania kolumny CLB wynosi: t_{rec}=

22/1456*17,01*50/66=190μs. Dostępna jest szyna komunikacyjna PLB (ang. Processor Local Bus) [X03] 64-bitowa pracująca z częstotliwością 133 MHz. Czas transmisji danych bloku 256x256 wynosi: 256*256*8/64 cykle z częstotliwością 133MHz, czyli **60 μs**. Powierzchnia łączy komunikacyjnych wynosi 5 CLB.

W celu porównania efektywności algorytmów COSYSOPC i COSEDYRES z metodami znanymi z literatury, przyjęto początkowo ograniczenia takie, jak w [BBD05b]. Czyli ograniczono liczbę dostępnych procesorów do jednego (PowerPC) i przyjęto, że taki procesor jest na stałe w architekturze Ponieważ w algorytmach COSYSOPC i COSEDYRES przyjmuje się fizyczne ograniczenie powierzchni, a powierzchnia fizyczna zajmowana przez procesor PowerPC jest duża, to mogłoby się okazać, że algorytm zamiast alokować procesor wybrałby rozwiązanie najszybsze, czyli cały system wykonywany sprzętowo. Wobec powyższego założono, że na koszt systemu składa się 8 kolumn oraz pewien stały koszt dla impelmentacji procesora. Przy takich założeniach oraz parametrach zadań i układu FPGA jak podano wcześniej najszybsza znaleziona architektura systemu implementowanego w postaci SOPC ma czas wykonania wszystkich zadań wynoszący: 17,22 ms i powierzchnię 436 CLB dla komponentów sprzętowych (w których wykonywane sa zadania: R1, R2, R3, D4, D5, Q6, Q7) oraz dodatkowo powierzchnię zajmowaną przez procesor w FPGA. Przez procesor wykonywane jest tylko jedno zadanie: H8, ponieważ zajmuje ono największą powierzchnię w sprzęcie i bardziej opłacalne jest wykonanie pozostałych zadań sprzętowo, ze względu na mniejsze czasy ich wykonania i większe możliwości zrównoleglenia obliczeń. Dla wcześniej przyjętych ograniczeń i parametrów uzyskany system jest najszybszym systemem, nie przekraczającym zadanych ograniczeń. Dla porównania w algorytmie [BBD05b] czas obliczeń systemu SOPC wyniósł 16,74 ms, a więc jest niewiele mniejszy. Jednak, ponieważ nie zostały opublikowane parametry poszczególnych zadań, różnice w uzyskiwanych wynikach prawdopodobnie spowodowane są rozbieżnościami pomiędzy parametrami zadań przyjętymi w literaturze, a parametrami przyjętymi w przedstawianym przykładzie. Taka architektura jest jednak droga (dodatkowa, duża powierzchnia dla procesora PowerPC). Metody [BBD05b] i [FVAGMT07] mają ograniczone możliwości generacji architektur. Nie umożliwiają generacji architektur wieloprocesorowych. Okazuje się, że można uzyskać system tańszy stosując właśnie architektury wieloprocesorowe i tańsze procesory, np. 8-bitowe PicoBlaze. Takie możliwości daje algorytm COSYSOPC.

Przyjęto tym razem, że w bibliotece komponentów jest tańszy procesor PicoBlaze o powierzchni **35 CLB** i nie ma ograniczenia na liczbę implementowanych procesorów (poza ograniczeniem powierzchni). Przyjęto ograniczenie dla całego systemu: 448 CLB. Pozostałe parametry (w tym parametry zadań) pozostawiono bez zmian, tak aby można było porównać wyniki kosyntezy. Zastosowano algorytm COSYSOPC dla powyższych założeń**.** W rozwiązaniu początkowym wszystkie zadania zostały przydzielone do jednego procesora. Czas wykonania wszystkich zadań w takim rozwiązaniu to: 104,859 ms, a powierzchnia 35 CLB. W wyniku rafinacji architektury otrzymano system o całkowitym czasie wykonania wszystkich zadań 17,2ms i

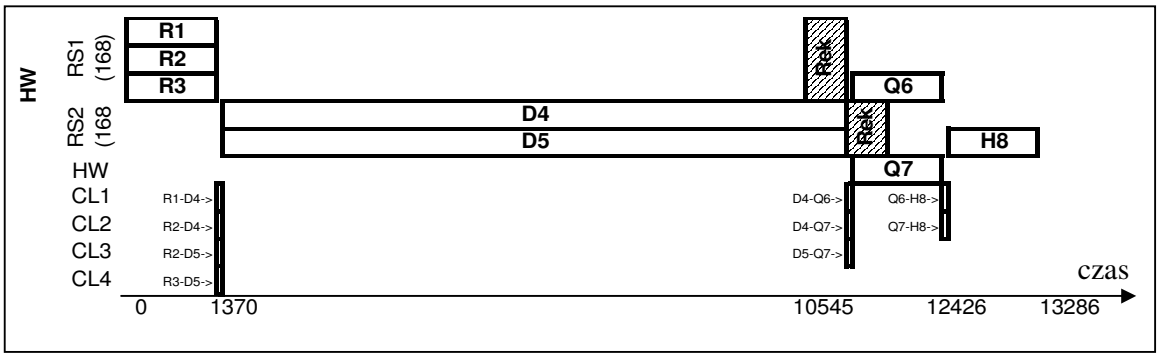
zajmowanej powierzchni 448 CLB. Wygenerowana architektura zawiera 2 *GPP* wykonujące po jednym zadaniu, pozostałe zadania wykonywane są sprzętowo. Wykres Gantta z uszeregowaniem zadań dla kodera JPEG pokazano na Rys. 7.1-2.



Rysunek 7.1-2 Wykres Gantta dla architektury kodera JPEG.

Okazuje się, że możliwe było uzyskanie dużo tańszego systemu SOPC, stosując architekturę wieloprocesorową, z tańszymi procesorami, a praktycznie czas wykonania zadań nie uległ zmianie. Dlatego dużą zaletą algorytmu COSYSOPC jest to, że potrafi generować na drodze rafinacji architektury wuieloprocesorowe, a takiej możliwości nie mają porównywane metody znane z literatury.

Po zastosowaniu algorytmu COSEDYRES, przy tym samym ograniczeniu powierzchni (448 CLB), wygenerowane zostały dostępne wielkości sektorów: 56, 112 i 168 CLB. W rozwiązaniu docelowym otrzymano system o całkowitym czasie wykonania wszystkich zadań 13,29ms i zajmowanej powierzchni 443 CLB. W otrzymanej architekturze wszystkie zadania wykonywane są sprzętowo. Wykorzystuje się dwa dynamicznie rekonfigurowane sektory. Z tego względu nie było konieczności specjalnego rozmieszczenia sektorów w układzie w celu zapewnienia zasad rekonfiguracji modułowej (żadna transmisja nie jest wykonywana przez reprogramowany sektor). Dzięki możliwości dynamicznej rekonfiguracji wszystkie zadania mogły być wykonane sprzętowo w FPGA i tym samym szybkość systemu wzrosła o 29% przy porównywalnej powierzchni. Wykres Gantta z uszeregowaniem zadań systemu dynamicznie rekonfigurowalnego dla kodera JPEG pokazano na Rys. 7.1-3.



Rysunek 7.1-3 Wykres Gantta dla architektury dynamicznie rekonfigurowalnej kodera JPEG.

Uzyskana przez algorytm COSEDYRES architektura jest najszybszą jaką można było otrzymać przy ograniczeniu powierzchni 448 CLB, bo wszystkie zadania wykonywane są sprzętowo, czasy rekonfiguracji nie wydłużają całkowitego czasu obliczeń (są zrównoleglone z innymi obliczeniami) i maksymalnie zrównoleglono wszystkie zadania zachowując zadane ograniczenia kolejnościowe.

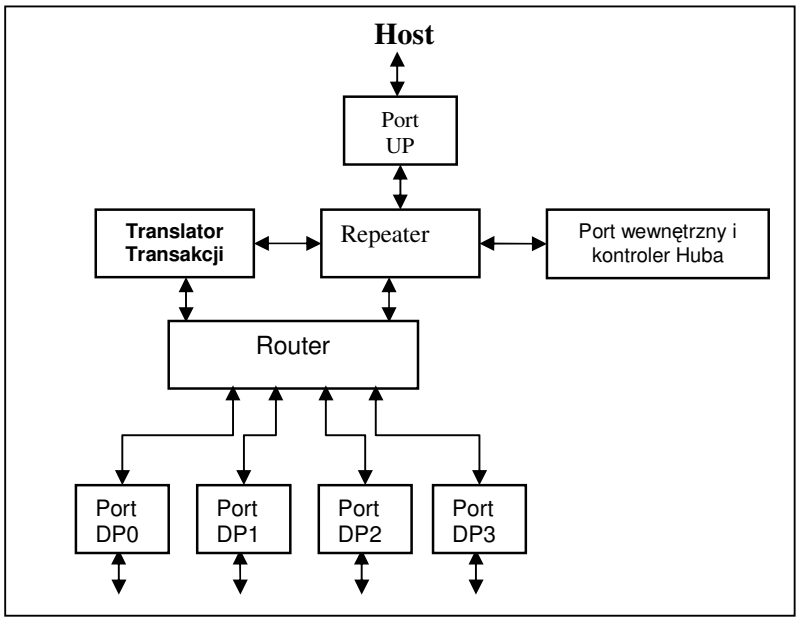
Dla porównania w algorytmie [BBD05b] czas wykonania wszystkich zadań systemu z możliwością dynamicznej rekonfiguracji wynosił 9,9ms. Różnice w szybkości również mogą wynikać z odmiennych, przyjętych parametrów. Podobny wynik można by uzyskać przy zastosowaniu algorytmu COSEDYRES po zmianie specyfikacji systemu. Architektura otrzymana w [BBD05b] będzie miała dużo większą powierzchnię, bo zawsze uwzględnia powierzchnię procesora wbudowanego na stałe w FPGA. Algorytm COSEDYRES można zastosować do różnych układów FPGA, także takich, które nie posiadają procesora PowerPC. Możliwe jest zatem uzyskanie architektur wieloprocesorowych dla różnych typów procesorów, również w postaci modułów *IP* (np. PicoBlaze, MicroBlaze). W ten sposób często można zmniejszyć powierzchnię systemu, jednocześnie nie zmniejszając jego szybkości.

Porównanie wyników uzyskanych z zastosowaniem algorytmów COSYSOPC i COSEDYRES, z wynikami z literatury jest stosunkowo trudne, gdyż nie są publikowane wszystkie parametry zadań, więc mogą one różnić się na tyle, że wyniki również będą nieco inne. Przykład JPEG jest stosunkowo prostym przykładem, dlatego też trudniej jest porównać efektywność algorytmów (dla takich systemów przeważnie otrzymywane są najlepsze możliwe architektury). Ważne jest to, iż przy takich samych założeniach można było uzyskać również najlepsze systemy dla przyjętych parametrów, więc opracowane algorytmy są skuteczne. Ponadto algorytmy COSYSOPC i COSEDYRES mają większe możliwości generacji architektur (wieloprocesorowe, różne procesory). Algorytmy te są również efektywne pod względem szybkości obliczeń (dla tego samego przykładu JPEG algorytm COSEDYRES znalazł rozwiązanie w czasie 203ms, podczas, gdy np. dla algorytmu [FVAGMT07] ten czas wynosił 1,29s).

7.2 Koncentrator USB 2.0

Kolejnym przykładem systemu wbudowanego jest czteroportowy koncentrator USB (ang. *Hub*). Szyna USB umożliwia dołączenie 127 urządzeń do sterownika (ang. *Host*). Fizycznie, szyna USB ma strukturę drzewa, w którym korzeniem jest *Host*, liśćmi są urządzenia a węzłami są koncentratory. Transmisja wykonywana jest szeregowo w sposób asynchroniczny w jednej z 3 następujących prędkości: LS (ang. *Low Speed*) - 1.5 MB/s, FS (ang. *Full Speed*) - 12 MB/s i HS (ang *High Speed*) - 480 MB/s. *Host* zawsze wysyła dane do wszystkich urządzeń, natomiast transmisje z

urządzeń wykonywane są tylko do *Hosta*. Hub USB pośredniczy w transmisjach pomiędzy *Hostem* a urządzeniami. Przesyłane dane pogrupowane są w pakiety. Każda transmisja jest inicjowana przez *Host*. Wyróżnia się 4 rodzaje transmisji: transmisje sterujące, transmisje blokowe, transmisje przerwań, transmisje izochroniczne. Transmisje zorganizowane są w tzw. transakcje (składające się zwykle z przesłania kilku pakietów). Schemat blokowy 4-portowego Huba jest przedstawiony na Rys.7.2-1.



Rysunek 7.2-1. Schemat blokowy 4-portowego koncentratora USB

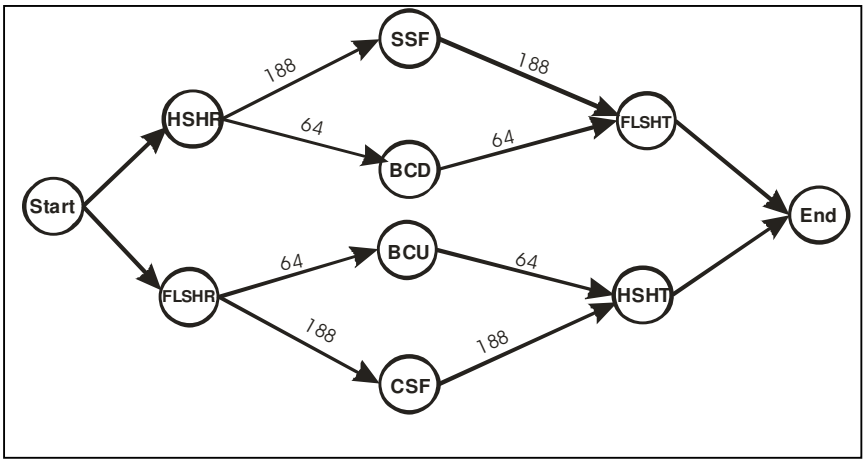
Transmisja może się odbywać tylko z portu UP jednocześnie do wszystkich portów DP0, ..., DP3, lub z jednego z portów DP0, ..., DP3 do portu UP. Jeśli szyna USB pracuje z szybkością FS to wszystkie transmisje odbywają się bezpośrednio przez Repeater, natomiast jeśli szyna USB pracuje z szybkością HS to transmisje do/z urządzeń HS odbywają się bezpośrednio przez Repeater a do/z urządzeń FS/LS przez Translator Transakcji (TT), który buforuje transmisje i dokonuje konwersji z HS na FS/LS. Transmisja z wykorzystaniem TT wygląda następująco:

- Host wysyła pakiet danych do TT,
- TT wysyła pakiet do urządzeń (w tym czasie Host może wykonywać inne transmisje),
- urządzenie wysyła do TT odpowiedź,
- Host odbiera od TT odpowiedź.

W specyfikacji koncentratora USB w [D05] wyróżniono 27 procesów. Jednym z bardziej złożonych modułów jest translator transakcji. Moduł ten można zaimplementować w postaci rozproszonej. Dlatego do minimalizacji całkowitego czasu wykonania wszystkich procesów modułu TT można wykorzystać algorytm kosyntezy COSEDYRES. Specyfikacja TT składa się z następujących procesów:

- High-Speed Handler (HSH) – proces implementujący komunikację pomiędzy TT i Hostem. Przy czym w procesie HSH można wyróżnić funkcje odpierania danych z Hosta i przesyłania danych do Hosta. Obie funkcje nie są wykonywane jednocześnie, zatem proces HSH można podzielić na dwa: HSHR i HSHT.
- Full/Low-Speed Handler (FLSH) - proces implementujący komunikację pomiędzy TT i urządzeniami. Podobnie jak w przypadku HSH, FLSH można podzielić na dwa procesy: FLSHR i FLSHT.
- Isoch/Int Start-split (SSF) – proces buforujący transmisje okresowe kierowane od Hosta do urządzeń.
- Isoch/Int Comp-split (CSF) – proces buforujący transmisje okresowe kierowane z urządzeń do Hosta,
- B/C In/Out (BC) - proces buforujący transmisje blokowe i sterujące. Proces BC można również podzielić na dwa: BCU i BCD.

Na podstawie specyfikacji [D05] można uzyskać 2 rozłączne grafy zadań, jeden opisujący transmisję z Hosta do urządzeń, drugi transmisję w przeciwnym kierunku. Graf zadań opisujący TT przedstawiono na Rys. 7.2-2. W celu uzyskania jednego spójnego grafu zadań dodano na początku i na końcu zadania (o parametrach t=0 i S=0) łączące dwa grafy. Konieczne jest zapewnienie spójności grafu, gdyż dwa grafy reprezentujące transmisje są wykonywane tyle samo razy i nie mogą być rozpatrywane niezależnie. Procesy SSF i BCD oraz BCU i CSF wzajemnie się wykluczają. Informacja o zadaniach ZWW w grafie CTG może pozwolić na lepsze wykorzystanie zasobów sprzętowych (poprzez dynamiczną rekonfigurację) w przypadku przydzielenia zadań ZWW do jednego sektora.



Rysunek 7.2-2. Graf zadań dla Translatora Transakcji

W tabeli 7.2-1 podano parametry wszystkich zadań wchodzących w skład TT dla implementacji sprzętowej i softwarowej. Podobnie jak w przykładzie 7.1 koncentrator USB zaimplementowano w układzie FPGA Xilinx Virtex 2 XC2V2000 o powierzchni 2688 CLB. Dostępny jest jeden typ

procesora 8-bitowego 10MHz, który może być zaimplementowany w postaci modułu o powierzchni 100CLB. Powierzchnia łączy komunikacyjnych wynosi 34 CLB.

Zadanie	SW	HW	
	t [µs]	t [µs]	S [CLB]
HSHR	2400	20	400
HSHT	2400	20	400
SSF	1800	12	100
CSF	2000	12	135
BCD	1600	12	120
BCU	1600	12	120
FLSHR	1400	8	170
FLSHT	1400	8	170

TABELA 7.2-1. PARAMETRY ZADAŃ DLA TRANSLATORA TRANSAKCJI

Wielkości transmitowanych danych w TT zaznaczono w grafie z Rys. 7.2-2 jako etykiety odpowiadających im krawędzi. Czasy transmisji wynoszą odpowiednio:

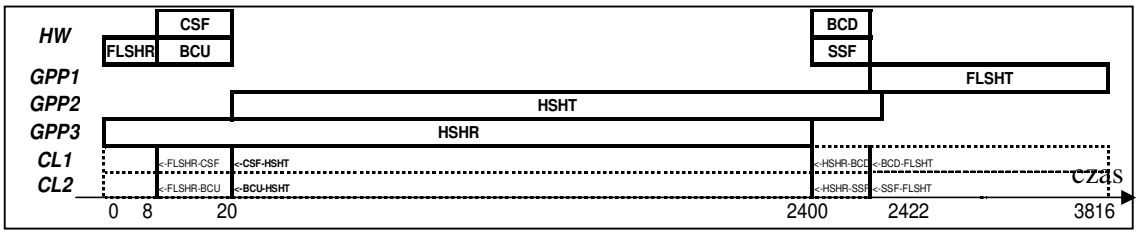
- przy przesyłaniu 188 bajtów: 188*8/64 cykle z częstotliwością 10 MHz, czyli **2350ns**,
- przy przesyłaniu 64 bajtów: 64*8/64 cykle z częstotliwością 10 MHz, czyli **800ns**.

Ze względu na to, że implementacja całego huba USB nie powinna przekroczyć powierzchni zastosowanego układu FPGA, dla potrzeb tego przykładu przyjęto ograniczenie powierzchni dla modułu TT: **1100 CLB**. Bez takiego ograniczenia wszystkie zadania można wykonać w sprzęcie, a więc najszybciej, a co za tym idzie nie byłaby konieczna optymalizacja. W przykładzie przedstawione zostaną trzy możliwe rozwiązania otrzymane w wyniku kosyntezy z zastosowaniem opracowanych algorytmów: COSYSOPC, COSEDYRES i COSEDYRES-CTG.

W rozwiązaniu początkowym wszystkie zadania wykonywane są przez jeden procesor. Całkowity czas wykonania wszystkich zadań wynosi:**14600µs**, a powierzchnia: 100CLB. W wyniku zastosowania kosyntezy chcemy uzyskać architekturę jak najszybszą, która nie przekroczy założonej powierzchni układu FPGA.

System SOPC:

W wyniku rafinacji dla implementacji SOPC otrzymano architekturę składającą się z 3 procesorów (każdy wykonuje inne zadanie) i 5 komponentów sprzętowych. Uszeregowanie dla takiego systemu przedstawiono na Rys. 7.2-3.

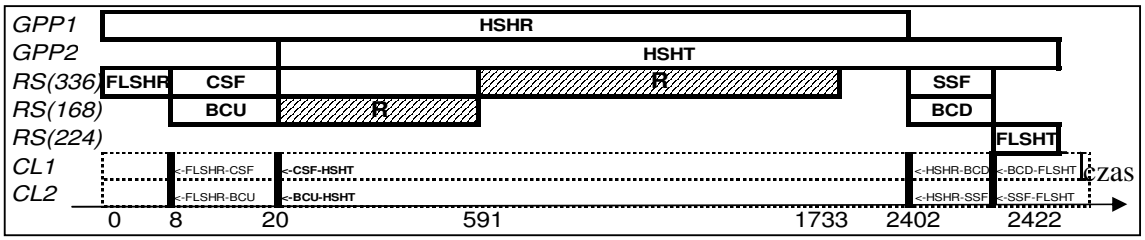


Rysunek 7.2-3 Wykres Gantta dla Translatora Transakcji

Całkowity czas wykonania wszystkich zadań wynosi: **3816µs**, a zajmowana powierzchnia: 1013 CLB. Jest to jednocześnie najszybsza architektura SOPC nie przekraczająca zadanego ograniczenia powierzchni. Wszystkich zadań nie da się jednocześnie zaimplementować w sprzęcie, a procesy HSHR, HSHT zajmują największą powierzchnię, implementowane jako *VC*, i bardziej opłacalne jest wykonanie innych zadań przez *VC*. FLSHT ma najkrótszy czas wykonania przez *GPP*, a ponieważ $S_{VC}(FLSHT)<S_{GPP}$ i jedno z zadań musi być wykonane softwarowo, ze względu na ograniczenie powierzchni, to FLSHT przydzielono do *GPP*.

System SRSOPC:

Dla takiego samego ograniczenia, czyli 1100CLB, wykonano syntezę systemu DRSOPC. Wygenerowane zostały następujące wielkości sektorów możliwe do wykorzystania w kosyntezie: 112, 168, 224, 280, 336, 392, 448. Wszystkie sektory są wielokrotnością jednej kolumny CLB (każda kolumna posiada 56 CLB). Otrzymano architekturę składającą się z 2 procesorów i 3 sektorów (o wielkościach 168, 224 i 336 CLB). Dwa z nich są dynamicznie rekonfigurowane w trakcie działania TT. Na rys. 7.2-4 pokazano uszeregowanie dla takiego systemu, które spełnia warunki 5.4-1 i 5.4-2.

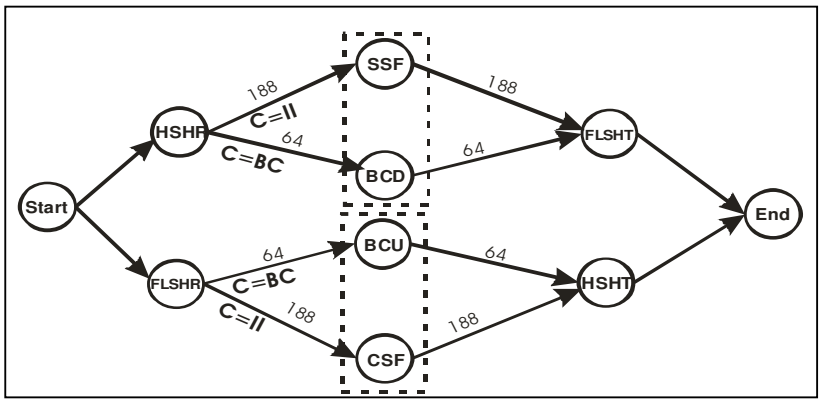


Rysunek 7.2-4 Wykres Gantta dla Translatora Transakcji zaimplementowanego z wykorzystaniem dynamicznej rekonfiguracji

Pomimo, że tylko jedno zadanie więcej jest wykonywane w sprzęcie udało się uzyskać system szybszy o **37%**. Całkowity czas wykonania wszystkich zadań wynosi bowiem: **2422µs**, przy zajmowanej powierzchni **1096 CLB** (wliczając moduł sterujący rekonfiguracją o powierzchni 100 CLB).

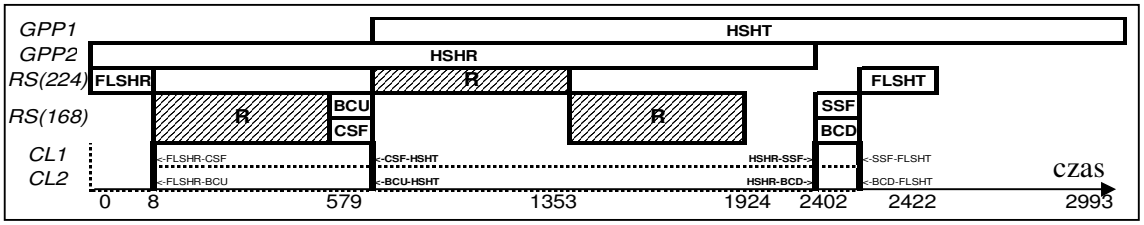
System SRSOPC reprezentowany przez warunkowy graf zadań:

Moduł Translatora Transakcji można opisać uwzględniając wzajemne wykluczanie się niektórych zadań: SSF z BCD i CSF z BCU. Graf opisujący TT z uwzględnieniem rozejść warunkowych przedstawiono na Rys. 7.2-5. Sprawdzany jest warunek C. Jeśli C=II, czyli w przypadku transmisji okresowych, dane przesyłane są do procesów SSF lub CSF. Jeśli C=BC, czyli dla transmisji blokowych i sterujących, transmisje są buforowane przez procesy BCD i BCU.



Rysunek 7.2-5. Warunkowy graf zadań dla Translatora Transakcji

Możliwe jest uzyskanie tańszego systemu po uwzględnieniu informacji o wzajemnie się wykluczających zadaniach. Zmniejszono zatem ograniczenie do 800 CLB. W tym przypadku bez uwzględnienia rozejść warunkowych całkowity czas wykonania zadań wynosi **4408µs** przy powierzchni **741CLB** (3 zadania wykonywane są sprzętowo bez dynamicznej rekonfiguracji). W wyniku kosyntezy dla grafu z rozejściami warunkowymi jak na Rys.7.2-5 otrzymano architekturę składającą się z 2 procesorów oraz dwóch sektorów dynamicznie rekonfigurowalnych. Zadania wzajemnie się wykluczające zostały umieszczone w tym samym zasobie sprzętowym (sektorze). Dzięki temu nie jest konieczne jednoczesna obecność takich zadań, można je alokować w systemie w zależności od potrzeb (warunków). Tym samym powierzchnia zajmowana przez nie zmniejsza się dwukrotnie i zmniejsza się powierzchnia całego systemu. Uszeregowanie zadań dla takiej implementacji pokazano na Rys. 7.2-6.

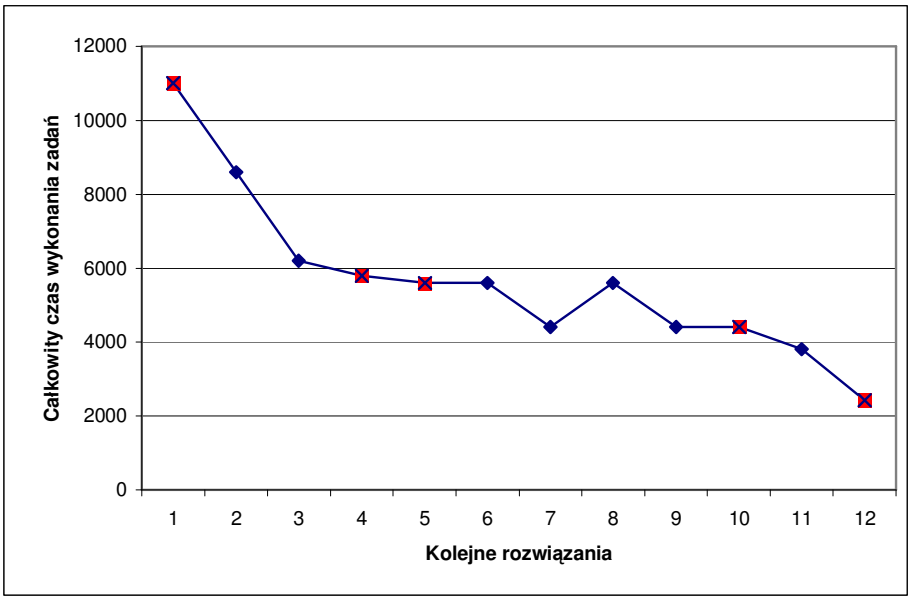


Rysunek 7.2-6 Wykres Gantta dla Translatora Transakcji zaimplementowanego z wykorzystaniem dynamicznej rekonfiguracji, reprezentowanego przez warunkowy graf zadań.

Całkowity czas wykonania wszystkich zadań wynosi: **2993µs**, a zajmowana powierzchnia **760 CLB**. Dla takiego ograniczenia powierzchni (800 CLB) niemożliwe byłoby uzyskanie szybszej architektury, nawet z wykorzystaniem dynamicznej rekonfiguracji, ale bez uwzględnienia rozejść warunkowych. Jak się okazuje można zaimplementować architekturę tańszą o **33%** i szybszą o **27%** w stosunku do implementacji SOPC. Możliwy jest jeszcze większy wzrost szybkości systemu, gdyby zmienić

specyfikację tego systemu. Należałoby wówczas podzielić zadanie HSHT (które jest najwolniej wykonywanym zadaniem przez procesor, a jednocześnie zajmuje największą powierzchnię przy realizacji w postaci *VC* – 400 CLB) na kilka zadań wykonywanych sekwencyjnie, które zajmowałyby mniejszą powierzchnię w sprzęcie.

W celu zademonstrowania sposobu zachowania się algorytmu COSEDYRES na rys. 7.2-7 pokazano rozwiązania znalezione w trakcie procesu rafinacji systemu wbudowanego. Kwadratami zaznaczono rozwiązania wybrane jako najlepsze (dające największy zysk) w każdym z kroków. Algorytm wybiera nie koniecznie rozwiązania o największym przyroście szybkości, ale także takie, dla których jest większe prawdopodobieństwo uzyskania w kolejnych krokach jeszcze lepszych rozwiązań (na rys.7.2-7 przykładem takiego wyboru jest ósme rozwiązanie).



Rysunek 7.2-7 Wykres zbieżności dla TT opisanego grafem z rozejściami warunkowymi

W tabeli 7.2-2 dokonano podsumowania rozwiązań wygenerowanych przez algorytmy COSYSOPC, COSEDYRES i COSEDYRES-CTG dla Translatora Transakcji, który jest reprezentowany grafem zadań z rys. 7.2.2 (SOSYSOPC i COSEDYRES) i z rys.7.2-5 (COSEDYRES-CTG) oraz parametrami zadań jak w tabeli 7.2-1.

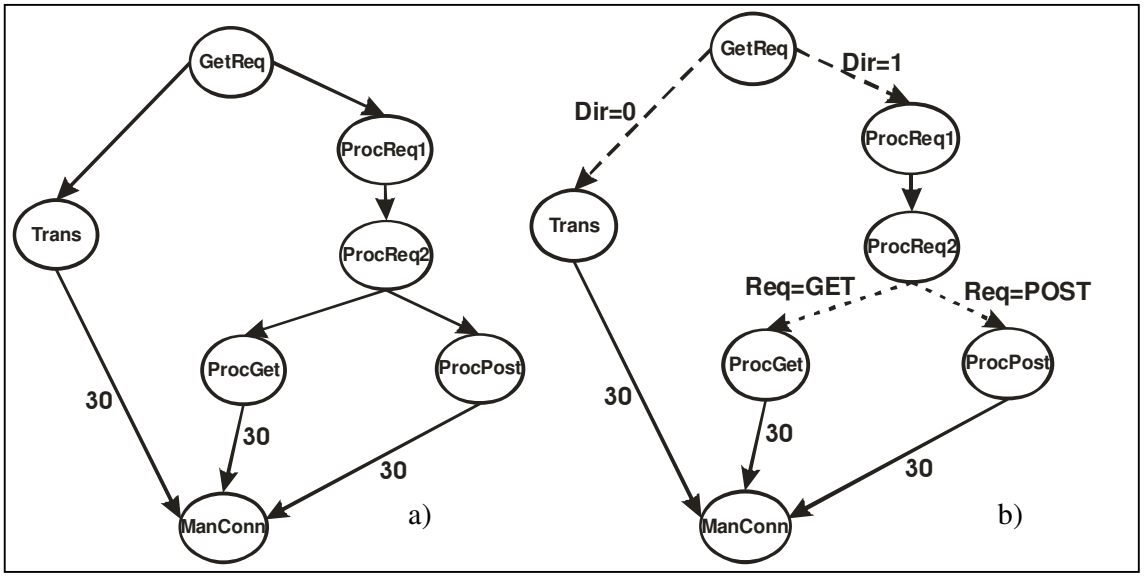
	COSYSOPC		COSEDYRES		COSEDYRES-CTG	
ograniczenie powierzchni [CLB]	S [CLB]	T [us]	S [CLB]	T [us]	S [CLB]	T [us]
1100	1013	3816	1096	2422	X	X
800	741	4408	741	4408	760	2993

TABELA 7.2-2. PODSUMOWANIE WYNIKÓW UZYSKANYCH DLA RÓŻNYCH IMPLEMENTACJI TRANSLATORA TRANSAKCJI

7.3 Wbudowany serwer internetowy

Obecnie coraz więcej urządzeń obsługiwanych jest przez Internet przy pomocy protokołu HTTP. Przykładem mogą być routery, punkty dostępowe, itp. Do obsługi takich urządzeń wbudowanych wystarczą tylko wybrane polecenia protokołu HTTP (GET i POST). W tym podrozdziale przedstawiono przykład wbudowanego serwera internetowego obsługującego te polecenia. Struktura specyfikacji serwera w postaci grafu zadań jest przedstawiona na rysunku 7.3-1 i składa się z następujących zadań:

- **GetReq**: proces oczekujący na zgłoszenia pojawiające się na porcie 80. Wszystkie odebrane żądania są przekazywane do *ProcReq*, natomiast po każdym opróżnieniu bufora nadajnika przekazywana jest informacja do procesu *Trans*.
- **ProcReq**: proces, który aktywowany jest po otrzymaniu komunikatu od *GetReq*, następnie odczytuje pakiety danych do bufora. W przypadku, gdy w buforze jest już kompletne żądanie HTTP, w zależności od jego typu wysyłane są informacje do procesów *ProcGet* lub *ProcPost*. Zadanie *ProcReq*, ze względu na złożoność, zostało podzielone na dwa zadania wykonywane sekwencyjnie *ProcReq1* i *ProcReq2*.
- **ProcGet:** proces przetwarzający polecenie GET.
- **ProcPost:** proces przetwarzający polecenie POST.
- **Trans:** proces, który transmituje kolejne strony HTML.
- **ManCon:** proces określający status danego połączenia. W przypadku, gdy dane połączenie przekroczyło limit czasu lub, gdy wystąpił błąd przy przetwarzaniu polecenia, następuje wyzerowanie buforów i zamknięcie połączenia.



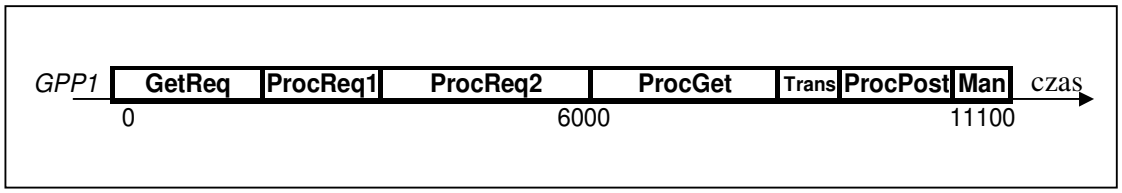
Rysunek 7.3-1. Specyfikacja wbudowanego serwera internetowego w postaci:
a) grafu zadań, b) warunkowego grafu zadań

Zadanie	SW		HW	
	t[µs]	S[B]	t[µs]	S[CLB]
GetReq	2000	600	400	250
ProcReq1	1200	1500	650	300
ProcReq2	2800	500	850	200
ProcGet	2500	1300	1000	300
ProcPost	1500	1200	300	50
Trans	500	400	150	150
ManConn	600	500	200	200

TABELA 7.3-1. PARAMETRY ZADAŃ DLA WBUDOWANEGO SERWERA INTERNETOWEGO

Rysunek 7.3-1a przedstawia graf zadań, natomiast rys. 7.3-1b przedstawia warunkowy graf zadań reprezentujący wbudowany serwer internetowy. W tym drugim grafie dzięki dodatkowym informacjom o warunkowym wykonaniu niektórych zadań, algorytm kosyntezy COSEDYRES-CTG będzie mógł uwzględnić fakt, że zadania *Trans* i *ProcReq* (*ProcReq1, ProcReq2*) nigdy nie będą wykonywane jednocześnie, a w zależności od wartości warunku *Dir*. Jeśli *Dir*=1, to po wykonaniu *ProcReq*, w zależności od warunku *Req* wykonane zostanie zadanie *ProcGet* lub *ProcPost*. W przykładzie tym występują więc warunki zagnieżdżone. Tabela 7.3-1 zawiera parametry wszystkich zadań wchodzących w skład wbudowanego serwera internetowego dla implementacji sprzętowej i softwarowej. Podobnie jak we wcześniejszych przykładach serwer internetowy zaimplementowano w układzie FPGA Xilinx Virtex 2 XC2V2000. Dostępny jest jeden typ procesora 8-bitowego 50MHz, który może być zaimplementowany w postaci modułu o powierzchni 200CLB. Czas transmisji od *Trans*, *ProcGet* i *ProcPost* do *ManConn* wynosi: 30*8/64 cykle z częstotliwością 50 MHz, czyli **75ns**. Przyjęto ograniczenie powierzchni dla serwera: **1000 CLB**. Poniżej przedstawiono trzy możliwe rozwiązania otrzymane w wyniku zastosowania algorytmów, kolejno: COSYSOPC, COSEDYRES i COSEDYRES-CTG.

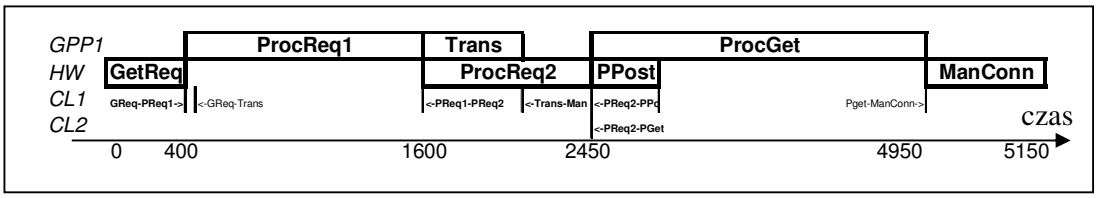
W rozwiązaniu początkowym wszystkie zadania wykonywane są przez jeden procesor. Całkowity czas wykonania wszystkich zadań wynosi:**11100µs**, a powierzchnia: 200CLB. Uszeregowanie zadań dla rozwiązania początkowego przedstawiono na rys. 7.3-2.



Rysunek 7.3-2. Wykres Gantta dla rozwiązania początkowego

System SOPC:

W wyniku zastosowania algorytmu COSYSOPC do kosyntezy systemu reprezentowanego przez graf z rys.7.3-1a otrzymano architekturę składającą się z 1 procesora *GPP* wykonującego 3 zadania oraz z czterech komponentów *VC*. Uszeregowanie dla takiego systemu przedstawiono na Rys. 7.3-3.

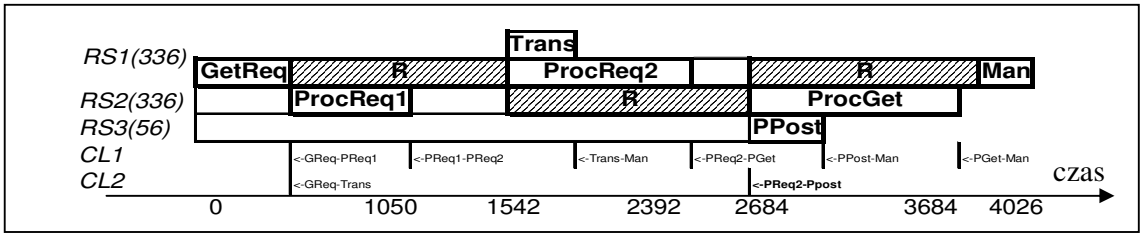


Rysunek 7.3-3. Uszeregowanie zadań dla implementacji SOPC wbudowanego serwera internetowego

Całkowity czas wykonania wszystkich zadań dla systemu SOPC wynosi: **T=5150µs**, przy S=968 CLB. Dla implementacji w formie systemu SOPC jest to równocześnie najszybsze rozwiązanie.

System SRSOPC:

W wyniku zastosowania algorytmu COSEDYRES dla systemu SRSOPC reprezentowanego przez graf z rys.7.3-1a na początku zostały wygenerowane dostępne rozmiary sektorów o wielkościach: 56, 168, 280, 336 CLB. Otrzymano docelową architekturę, w której wszystkie zadania wykonywane są sprzętowo. Było to możliwe dzięki dynamicznej rekonfiguracji i w związku z tym kilkukrotnemu wykorzystaniu tych samych powierzchni układu dla różnych zadań. Komponenty sprzętowe zostały przydzielone do trzech sektorów o powierzchniach 56, 336 i 336 CLB. Uszeregowanie zadań dla docelowej implementacji SRSOPC przedstawiono na rys. 7.3-4.

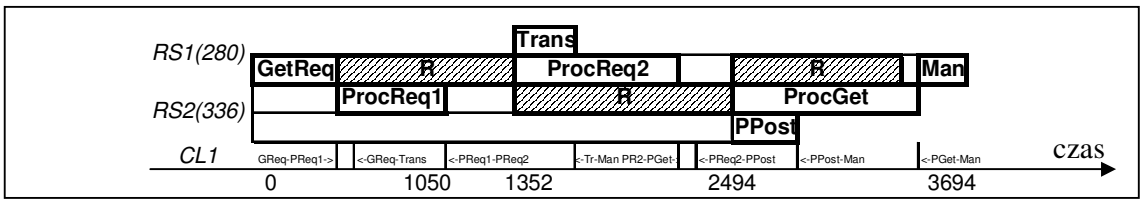


Rysunek 7.3-4. Uszeregowanie zadań dla systemu SRSOPC reprezentowanego przez graf z rys.7.3-1a

Całkowity czas wykonania wszystkich zadań dla systemu SRSOPC reprezentowanego przez graf z rys.7.3-1a wynosi: **T=4026µs**, przy S=996CLB (wliczając moduł sterujący rekonfiguracją o powierzchni 200 CLB). Wzrost szybkości w stosunku do implementacji SOPC wynosi zatem **22%**. Rekonfiguracja sektora *RS1* została zrównoleglona z obliczeniami wykonywanymi przez komponenty w sektorze *RS2* i *RS3*, a rekonfiguracja *RS2* z obliczeniami w sektorze *RS1*.

System SRSOPC reprezentowany przez warunkowy graf zadań:

W kolejnym eksperymencie wykonano syntezę systemu SRSOPC reprezentowanego przez warunkowy graf zadań z rys. 7.3-1b. W grafie tym występuje kilka par zadań wzajemnie się wykluczających: ZWW={(*Trans*, *ProcReq1*), (*Trans*, *ProcReq2*), (*ProcGet*, *ProcPost*), (*Trans*, *ProcGet*), (*Trans*, *ProcPost*)}. W otrzymanej architekturze docelowej wykorzystane zostały dwa sektory o powierzchniach: 280 i 336CLB. Na rys. 7.3-5 przedstawiono uszeregowanie zadań dla docelowej implementacji SRSOPC.



Rysunek 7.3-5. Uszeregowanie zadań dla systemu SRSOPC reprezentowanego przez graf z rys.7.3-1b

W porównaniu z systemem SRSOPC opisanym grafem TG, w przypadku uwzględnienia informacji o zadaniach ZWW, możliwe było lepsze wykorzystanie powierzchni układu. Zadania ZWW: *ProcReq2* i *Trans* zostały umieszczone w tym samym sektorze *RS1* i uszeregowane równolegle (są wykonywane w zależności od wartości warunku *Dir*) i zajmują tą samą powierzchnię układu (200CLB), podobnie zadania *ProcGet* i *ProcPost* zostały umieszczone w tym samym sektorze *RS2* i wykonywane są w zależności od wartości warunku *Req* (S=300CLB). Zadania takie nie muszą być jednocześnie alokowane w układzie (poprzez dynamiczną rekonfigurację będą mogły być alokowane w zależności od warunku). Uzyskano następujące wyniki: **T=3694μs** i S=850CLB (wliczając moduł sterujący rekonfiguracją o powierzchni 200 CLB). Zatem wzrost szybkości w stosunku do systemu SRSOPC opisanego grafem TG wyniósł **8%**, przy jednoczesnym zmniejszeniu powierzchni zajmowanej przez wbudowany serwer w układzie FPGA (o 15%). Jeszcze lepsze wykorzystanie powierzchni dla zadań realizowanych sprzętowo (mniejsze wielkości sektorów) spowodowało, ze czasy reprogramowania sektorów były mniejsze, co przyczyniło się do zmniejszenia całkowitego czasu wykonania wszystkich zadań wyspecyfikowanych w grafie.

W tabeli 7.3-2 dokonano podsumowania rozwiązań wygenerowanych przez algorytmy COSYSOPC, COSEDYRES i COSEDYRES-CTG dla wbudowanego serwera internetowego, który jest reprezentowany grafem zadań z rys. 7.3.1a (SOSYSOPC i COSEDYRES) i z rys.7.3-1b (COSEDYRES-CTG) oraz parametrami zadań jak w tabeli 7.3-1.

	COSYSOPC		COSEDYRES		COSEDYRES-CTG	
ograniczenie powierzchni [CLB]	S [CLB]	T [us]	S [CLB]	T [us]	S [CLB]	T [us]
1000	968	5150	996	4026	850	3694

TABELA 7.3-2. PODSUMOWANIE WYNIKÓW UZYSKANYCH DLA RÓŻNYCH IMPLEMENTACJI WBUDOWANEGO SERWERA INTERNETOWEGO

Przykłady pokazują, że dynamiczna rekonfiguracja systemów implementowanych w częściowo reprogramowalnych układach FPGA pozwala na uzyskanie szybszych systemów, jeśli właściwie zostaną zrównoleglone obliczenia z rekonfiguracją. Dodatkowo uwzględnienie informacji o wzajemnie wykluczających się zadaniach w warunkowym grafie zadań może pozwolić na jeszcze lepsze wykorzystanie dynamicznej rekonfiguracji, w celu uzyskania szybszych systemów, jeśli zadania ZWW przydzielone są do tego samego sektora.

8 PODSUMOWANIE

Celem pracy było opracowanie wyspecjalizowanego algorytmu kosyntezy systemów dynamicznie rekonfigurowalnych SRSOPC, który będzie maksymalizował szybkość projektowanego systemu przy zadanym ograniczeniu powierzchni układu FPGA. Metoda miała uwzględniać rzeczywiste ograniczenia współczesnych częściowo reprogramowalnych układów FPGA znanych producentów. Postawiona została teza, że wykorzystanie dynamicznej rekonfiguracji w projektowaniu szerokiej klasy systemów wbudowanych prowadzi do uzyskania implementacji szybszych niż w przypadku nie stosowania tej techniki, dla tego samego docelowego układu FPGA.

Opracowano i zaimplementowano trzy wersje algorytmu kosyntezy systemów SOPC, które stanowiły ewolucję zmierzającą do otrzymania wyspecjalizowanego algorytmu dla systemów SRSOPC. Zastosowane metody kosyntezy zostały porównane z innymi metodami znanymi z literatury i wykazana została duża efektywność prezentowanych w pracy nowych algorytmów. Algorytm COSEDYRES charakteryzuje się stosunkowo niedużą złożonością, co zostało wykazane poprzez praktyczne przykłady i eksperymenty dla losowych grafów zadań, a także w wyniku analizy teoretycznej. Algorytm jest szybki i pozwala na uzyskanie wyników lepszych od innych algorytmów kosyntezy. Zastosowano proste metody rafinacji rozwiązań, ale umożliwiające wykonanie istotnych modyfikacji architektury systemu. Dobrano również odpowiednią funkcję zysku sterującą rafinacją rozwiązań, która uwzględnia nie tylko wzrost szybkości, ale także prawdopodobieństwo poprawy rozwiązań w następnych krokach. Te cechy algorytmu prowadzą do zmniejszenia prawdopodobieństwa zatrzymywania się algorytmu w lokalnych maksimach szybkości (co jest częstą wadą metod znanych z literatury).

Algorytm COSEDYRES jest jednym z pierwszych algorytmów kosyntezy systemów dynamicznie rekonfigurowalnych, uwzględniającym rzeczywiste ograniczenia dotyczące zasad rekonfiguracji modułowej, występujących we współczesnych częściowo reprogramowalnych FPGA. Prace nad algorytmem kosyntezy systemów DRSOPC, uwzględniającym te ograniczenia, są prowadzone jednocześnie przez zespół badawczy pod kierunkiem S. Banerjee, na Uniwersytecie Kalifornijskim [BBD05a].

Prezentowany w tej pracy algorytm jest przeznaczony dla wieloprocesorowych systemów SRSOPC. Architektury wieloprocesorowe stanowią obecnie i będą, zgodnie z przewidywaniami [H07a], także w najbliższych latach stanowić jeden z głównych trendów badań i rozwoju architektur. W szczególności badania będą skierowane w stronę systemów wbudowanych, ze względu na szybki rozwój coraz bardziej złożonych urządzeń integrujących wiele różnych funkcjonalności (m.in. urządzeń mobilnych). W celu pełnej integralności systemu w jednym układzie FPGA w niniejszej pracy wprowadzono wbudowany sterownik rekonfiguracji, w postaci dedykowanego modułu (logicznego lub procesora *GPP*), który zajmuje się tylko rekonfiguracją sektorów układu. Dynamiczna

rekonfiguracja systemu przez wbudowany sterownik jest możliwa dzięki istnieniu we współczesnych układach FPGA (firmy Xilinx) wbudowanego portu konfiguracji ICAP. Algorytm COSEDYRES jest pierwszym algorytmem kosyntezy wieloprocesorowych, dynamicznie samorekonfigurowalnych systemów wbudowanych.

W celu udowodnienia tezy rozprawy wykonano szereg eksperymentów dla losowych grafów o różnej liczbie zadań, jak i dla praktycznych przykładów systemów wbudowanych. Porównano wyniki otrzymane poprzez zastosowanie algorytmów COSYSOPC i COSEDEDYRES. Eksperymenty potwierdziły, że zastosowanie dynamicznej rekonfiguracji pozwala, dla szerokiej klasy systemów, na uzyskanie systemów szybszych, niż bez stosowania tej techniki, dla tego samego układu FPGA. Wzrost szybkości systemu wbudowanego jest możliwy dzięki wielokrotnemu wykorzystaniu tych samych fragmentów układu FPGA do alokacji komponentów sprzętowych realizujących różne zadania, w trakcie działania tego systemu. Wówczas, ze względu, iż większa liczba zadań jest wykonywana sprzętowo (czyli przeważnie szybciej niż softwarowo), to szybkość całego systemu też zwykle jest większa. Jednak, aby dynamiczna rekonfiguracja była skuteczna konieczne jest spełnienie kilku warunków. Po pierwsze czas reprogramowania sektora w celu wykonania zadania przez komponent sprzętowy nie powinien być większy niż czas obliczeń tego zadania przez procesor uniwersalny. Konieczne jest również odpowiednie zrównoleglenie przeprogramowań sektora z obliczeniami w innych sektorach lub z obliczeniami wykonywanymi przez procesory. Jeśli nie jest możliwe takie zrównoleglenie, wówczas narzut spowodowany czasem rekonfiguracji może nawet spowolnić system. Istotne jest wtedy, aby takich rekonfiguracji nie było zbyt wiele, w szczególności, jeśli czas rekonfiguracji jest duży.

W większości istniejących prac dotyczących kosyntezy system reprezentowany jest przez graf zadań. Do tej pory nie zaprezentowano jeszcze żadnej metody kosyntezy systemów DRSOPC, w której system byłby opisany warunkowym grafem zadań. W niniejszej rozprawie przedstawiono pierwszy algorytm kosyntezy, który umożliwia syntezę systemów SRSOPC reprezentowanych przez CTG. Opis systemu w postaci grafu CTG daje większe możliwości wykorzystania dynamicznej rekonfiguracji systemów wbudowanych w celu uzyskania jeszcze szybszego systemu. Mając bowiem informacje o wzajemnie się wykluczających zadaniach, zadania te, jeśli są alokowane w tym samym sektorze układu FPGA, mogą być przydzielone do tego samego fragmentu układu, zajmując tym samym mniejszą powierzchnię i zostawiając więcej miejsca dla innych zadań realizowanych sprzętowo. Spośród zadań wzajemnie się wykluczających, w trakcie jednego cyklu wykonania zadań w grafie jest wykonywane tylko jedno z nich (w zależności od wartości warunku). Dla innych wartości warunków alokowane są inne zadania (spośród wzajemnie się wykluczających) poprzez dynamiczną rekonfigurację systemu. Po każdym sprawdzeniu warunku konieczne jest reprogramowanie sektora w celu implementacji jednego z zadań wykonywanych warunkowo. To może przyczynić się do powstania dodatkowych narzutów czasowych spowodowanych rekonfiguracją, ale dzięki mniejszej powierzchni sektorów, czas ten również jest mniejszy i przy właściwym zrównolegleniu

rekonfiguracji z obliczeniami może w rezultacie prowadzić do przyspieszenia systemu. Wyniki eksperymentów dla losowych grafów i wyniki uzyskane po zastosowaniu algorytmu COSEDYRES-CTG dla praktycznych przykładów, pozwoliły na potwierdzenie, iż zwykle informacje o zadaniach ZWW pozwalają na lepsze wykorzystanie dynamicznej rekonfiguracji w celu uzyskania szybszego systemu, niż w przypadku reprezentacji systemu przez graf TG, przy tej samej powierzchni układu.

Prace nad efektywnymi algorytmami kosyntezy wbudowanych systemów rozproszonych są kontynuowane. Badane będą inne możliwości rekonfiguracji systemu, np. zamiast wykorzystania procesora *GPP* zajmującego się tylko rekonfiguracją, sterowaniem rekonfiguracją sektorów mogą zajmować się procesory, które będą też wykonywały inne zadania. Badane będą również inne sposoby reprezentacji systemu niż skierowany i acykliczny graf zadań. Prace będą też skierowane na dalsze udoskonalenie metod rafinacji systemów SRSOPC.

BIBLIOGRAFIA

[A04] Atmel Corporation, "Performing Dynamic Reconfiguration in FPSLIC™ Devices--A Scrolling Message Display", *Atmel Application Note*, 2004.

[BAP98] L. Bianco, M.Auguin, A. Pegatoquet, "A Path Analysis Based Partitioning for Time Constrained Embedded Systems", *Proc. of the 6th International Workshop on Hardware/Software Codesign*. *IEEE Computer Society Press*, pp. 85-89, 1998.

[BBD05a] S. Banerjee, E. Bozorgzadeh, N. Dutt, "Physically-aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration", *Proc. DAC'05*, pp. 335-340, 2005.

[BBD05b] S. Banerjee, E. Bozorgzadeh, N. Dutt, "HW-SW Partitioning for Architectures with Partial Dynamic Reconfiguration", *Technical Report CECS-TR-05-02, UC Irvine*, 2005.

[BM02] L. Benini, G. De Micheli , "Networks on Chips: a New SoC Paradigm", *Computer ,* Vol. 35, Iss. 1, pp. 70 -78, 2002.

[BM03] B. Blodget, S. McMillan, "A Lightweight Approach for Embedded Reconfiguration of FPGA", *Proc. DATE'03*, pp. 399-400, 2003.

[BNH05] F. Berthelot, F. Nouvel, D. Houzet, "Design Methodology for Runtime Reconfigurable FPGA: From High Level Specification Down to Implementation", *IEEE Workshop on SPSDI 2005*, pp. 497-502, 2005.

[BRK03] B. Blodget, P.J. Roxby, E. Keller, "A Self-reconfiguring Platform", *Proc. FPL '03*, pp. 565-574, 2003.

[CA02] K. B. Chehida, M. Auguin, "HW/SW Partitioning Approach for Reconfigurable System Design", *Proc. CASES 2002*, pp. 247-251, 2002.

[CCBM04] E. Carvalho, N. Calazans, E. Briao, F. Moraes, "PaDReH – a Framework for the Design and Implementation of Dynamically and Partially Reconfigurable Systems", *Proc. SBCCI'04*, pp. 10-15, 2004.

[CD06] R. Czarnecki, S.Deniziak, „Kosynteza dynamicznie rekonfigurowalnych systemów SOPC”, *Pomiary. Automatyka. Kontrola nr 7 bis*, str. 34-36, 2006.

[CD07a] R. Czarnecki, S. Deniziak, “Resource Constrained Co-synthesis of Self-reconfigurable SOPCs”, *Proc. DDECS’07*, pp. 49-54, 2007.

[CD07b] R. Czarnecki, S. Deniziak, "Kosynteza samorekonfigurowalnych systemów SOPC", *Czasopismo Techniczne Politechniki Krakowskiej, Informatyka*, z.1-I/2007, Zeszyt 7, str. 3-16, 2007.

[CDHG07] J.Cui, Q.Deng, X.He, Z.Gu, „An Efficient Algorithm for Online Management of 2D Area of Partially Reconfigurable FPGAs”, *Proc. DATE’07*, pp. 129-134, 2007.

[CDS02] R. Czarnecki, S. Deniziak, K. Sapiecha, "Rafinacyjny algorytm kosyntezy heterogenicznych systemów SOPC”, *materiały V Krajowej Konferencji Naukowej RUC’02*, str. 121-128, 2002.

[CDS03] R. Czarnecki, S. Deniziak, K. Sapiecha, "Zastosowanie wielokontekstowych układów FPGA w kosyntezie systemów cyfrowych”, *materiały VI Krajowej Konferencji Naukowej RUC’03*, str. 63-70, 2003

[CDS03] R. Czarnecki, S. Deniziak, K. Sapiecha, “An Iterative Improvement Co-synthesis Algorithm for Optimization of SOPC Architecture with Dynamically Reconfigurable FPGAs”, *Proc. EUROMICRO DSD’03*, pp. 443-446, 2003.

[CH02] K. Compton, S. Hauck, ”Reconfigurable Computing: a Survey of Systems and Software”, *ACM Computing Surveys,* Vol 34, No.2, pp. 171-210, 2002.

[CV99] K. S. Chatha , R. Vemuri, “Hardware-Software Codesign for Dynamically Reconfigurable Architectures”, *Proc. FPL’99*, pp. 175-184, 1999.

[CV00] K. S. Chatha, R. Vemuri, ”An Iterative Algorithm for Hardware-Software Partitioning, Hardware Design Space Exploration, and Scheduling”, *Jrnl Design Automation for Embedded Systems,* Vol 5, pp. 281-293, 2000.

[CV01] K. S. Chatha , R. Vemuri, “MAGELLAN: Multiway Hardware-Software Partitioning and Scheduling for Latency Minimization of Hierarchical Control-Dataflow Task Graphs”, *Proc. of the 9th International Symposium on Hardware/Software Codesign*, pp. 42-47, 2001

[CZMS07] C. Claus, J. Zeppenfeld, F. Müller, W. Stechele, “Using Partial-Run-Time Reconfigurable Hardware to Accelerate Video Processing in Driver Assistance System”, *Proc. DATE’07*, pp. 498-503, 2007.

[D98] G. De Michelli, „Synteza i optymalizacja układów cyfrowych.”, *WNT, Warszawa*, 1999.

[D02] R.P. Dick, “Multiobjective Synthesis of Low-power Real-time Distributed Embedded Systems”, *PhD dissertation*, 2002.

[D04] S. Deniziak, "Cost-Efficient Synthesis of Multiprocessor Heterogeneous Systems", *Control and Cybernetics*, Vol.33, No.2, pp. 341-355, 2004.

[D05] S. Deniziak, "Metodologia szybkiego prototypowania systemów cyfrowych", *Wydawnictwo Politechniki Krakowskiej*, 2005.

[DE98] A. Daboli, P. Eles, “Scheduling Under Data and Control Dependencies for Heterogeneous Architectures”, *Proc. of the International Conference on Computer Design 1998*, pp. 602-608, 1998.

[DG97] G. De Michelli, R.K. Gupta, ”Hardware/Software Co-design”, *Proc. of the IEEE,* Vol. 85, Issue: 3, pp. 349-365, 1997.

[DH94] J.G. D'Ambrosio, X. Hu, “Configuration-Level Hardware/Software Partitioning for Real-time Embedded Systems”, *Proc. of the 3rd International Workshop on Hardware/Software Codesign*, pp. 34-41, 1994.

[DJ97] R.P. Dick, N.K. Jha, “MOGAC: A Multiobjective Genetic Algorithm for the Co-Synthesis of Hardware-Software Embedded Systems, *Proc. of the 1997 International Conference on Computer-Aided Design*, pp. 522-529, 1997.

[DJ98a] B.P. Dave, N.K. Jha, “CASPER: Concurrent Hardware-Software Co-Synthesis of Hard Real-Time Aperiodic and Periodic Specifications of Embedded Systems”, *Proc. DATE’98*, pp. 118-124, 1998.

[DJ98b] R. P. Dick , N. K. Jha, “CORDS: Hardware-Software Co-synthesis of Reconfigurable Real-time Distributed Embedded Systems”, *Proc. of the 1998 International Conference on Computer-Aided Design*, pp. 62-67, 1998.

[DJ99] R. P. Dick , N. K. Jha, “MOCSYN: Multiobjective Core-Based Single-Chip System Synthesis”, *Proc. DATE’99*, pp. 263-270, 1999.

[DLJ97] B. Dave, G. Lakshminarayana, N. Jha, “COSYN: Hardware-Software Co-synthesis of Embedded Systems”, *Proc. DAC’97,* pp. 703-708, 1997.

[DRW98] R. P. Dick , D. L. Rhodes , W. Wolf, “TGFF: Task Graphs for Free”, *Proc. of the 6th International Workshop on Hardware/Software Codesign*, pp. 97-101, 1998.

[DWXHY05] Q. Deng, S. Wei, H. Xu, Y. Han, G. Yu, “A reconfigurable RTOS with HW/SW Co-scheduling for SOPC”, *Proc. of the Second ICESS’05*, pp. 116-121, 2005.

[E3S] “E3S: The Embedded System Synthesis Benchmarks Suite”, *http://www.eecs.northwestern.edu/dickrp*.

[EHB93] R. Ernst, J. Henkel, and T. Benner, “ Hardware-Software Cosynthesis for Microcontrollers,” *IEEE Design and Test of Computers*, vol.10, no.4, pp. 64-75, 1993.

[EPKD97] P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, “System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search”, *Design Automation for Embedded Systems,* V.2, N1, pp. 5-32, 1997.

[EP02] M. Eisenring, M. Platzner, “A Framework for Run-time Reconfigurable Systems”, *The Journal of Supercomputing,* v.21, pp. 145–159, 2002

[FMCOZ07] B. Fiethe, H. Michalik, C. Dierker, B. Osterloh, G. Zhou, “Reconfigurable System-on-Chip Data Processing Units for Space Imaging Instruments”, *Proc. DATE’07*, pp. 977-982, 2007.

[FSS05] F. Ferrandi, M.D. Santabrogio, D. Sciuto, “A Design Methodology for Dynamic Reconfiguration: The Coronte Architecture”, *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS’05) – Workshop 3*, pp. 163-166, 2005.

[FVAGMT07] S. Fekete, J. van der Veen, J. Angermeier, D. Göhringer, M. Majer, J. Teich., ”Scheduling and Communication-aware Mapping of HW-SW Modules for Dynamically and Partially Reconfigurable SoC Architectures”, *Proc. of the Dynamically Reconfigurable Systems Workshop (DRS 2007)*, 2007.

[G89] D. E. Goldberg, “Genetic Algorithms in Search, Optimization and Machine Learning”, *Addison-Wesley, Reading, MA*, 1989.

[GASF03] M. Gericota, G. Alves, M. Silva, J. Ferreira, “Run-Time Management of Logic Resources on Reconfigurable Systems”, *Proc. DATE’03*, pp. 974-979, 2003.

[GCSBF06] P. Garcia, K. Compton, M. Schulte, E. Blem, W. Fu, “An Overview of Reconfigurable Hardware in Embedded Systems”, *EURASIP Journal on Embedded Systems*, Vol. 2006, pp. 1–19, 2006.

[GM93] R. Gupta, G. De Micheli, “Hardware-Software Cosynthesis for Digital Systems,” *IEEE Design and Test of Computers*, vol.10, no.3, pp. 29-41, 1993.

[GREAMB01] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, „MiBench: A Free, Commercially Representative Embedded Benchmark Suite”, *Proc. of Workload Characterization 2001*, pp. 3-14, 2001.

[GTD93] F. Glover, E. Taillard, D. De Werra, ”A User’s Guide to Tabu Search”, *Annals of Operations Research*, Vol. 41, pp. 3-28, 1993.

[GV00] S. Ganesan and R. Vemuri, “An Integrated Temporal Partioning and Partial Reconfiguration Technique for Design Latency Improvement”, *Proc. DATE’00*, pp. 320-325, 2000.

[H01] R. Hartenstein, “A Decade of Reconfigurable Computing: A Visionary Retrospective”, *Proc. DATE’01*, pp. 642-649, 2001.

[H07a] K. De Bosschere, W. Luk, X. Martorell, N. Navarro, M. O’Boyle, D. Pnevmatikatos, A. Ramirez, P. Sainrat, A. Seznec, P. Stenstrom, O. Temam, “High-Performance Embedded Architecture and Compilation Roadmap“, *Transactions on HiPEAC I,* LNCS 4050, pp. 5–29, 2007.

[H07b] S. Ha, “Model-based Programming Environment of Embedded Software for MPSoC”, *ASP-DAC '07*, pp. 330-335, 2007.

[HCG07] A. Hansson, M. Coenen, K. Goossens, “Undisrupted Quality-of-Service during Reconfiguration of Multiple Applications in Networks on Chip”, *Proc. DATE’07*, pp. 954-959, 2007.

[HE98] J. Henkel, R. Ernst, “High-level Estimation Techniques for Usage in Hardware/Software Co-design”, *Proc. ASP-DAC’98*, pp. 353-360, 1998.

[HXVKI05] W-L. Hung, Y. Xie, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, “Thermal-Aware Task Allocation and Scheduling for Embedded Systems”, *Proc DATE’05*, pp. 898-899, 2005.

[HW96] J. Hou, W. Wolf, “Process Partitioning for Distributed Embedded Systems”, *Proc. of the 4th International Workshop on Hardware/Software Codesign*, pp. 70-76, 1996.

[HV05] R. Huang, R. Vemuri, “On-Line Synthesis for Partially Reconfigurable FPGAs”, *Proc. of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID’05)*, pp. 663-668, 2005.

[HVG07] M.C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, D. DiSabello, „Achieving High Performance with FPGA-Based Computing”, *Computer*, Vol. 40, No. 3, pp. 50-57, 2007.

[HZ07] M. Z. Hasan, S. G. Ziavras, ”Runtime Partial Reconfiguration for Embedded Vector Processors”, *International Conference on Information Technology (ITNG'07)*, pp. 983-988, 2007.

[HZG04] T. Hollstein, H. Zimmer, M. Glesner, “Dynamic Hardware/Software Co-design Based on a Communication-centric Hyper-Platform”, *Proc. of the 16th ICM*, pp. 355-358, 2004.

[I00] K. Ito, “A Scheduling and Allocation Method to Reduce Data Transfer Time by Dynamic Reconfiguration”, *Proc. ASP-DAC '00*, pp. 323-328, 2000.

[IMJ07] I. Hiroaki, M. Edahiro, J. Sakai, “Towards Scalable and Secure Execution Platform for Embedded Systems”, *Proc. ASP-DAC '07*, pp. 350-354, 2007.

[ITM05] E. Ilavarasan, P. Thambidurai, R. Mahilmannan, ”Performance Effective Task Scheduling Algorithm for Heterogeneous Computing System”, *The 4th International Symposium on Parallel and Distributed Computing*, pp. 28-38, 2005.

[JKH05] A. Jhumka, S. Klaus, S. A. Huss, “A Dependability-Driven System-Level Design Approach for Embedded Systems”, *Proc DATE’05*, Vol.1, pp. 372-377, 2005.

[JOT04] J. Jantsch, J. Oberg, H. Tenhunen, "Special Issue on Networks on Chip", *Journal of System Architecture*, Vol. 50, Issues 2-3, pp. 61-63, 2004.

[JYLC00] B. Jeong, S. Yoo, S. Lee, K. Choi, "Hardware-Software Cosynthesis for Run-time Incrementally Reconfigurable FPGAs", *Proc. of ASP-DAC 2000*, pp. 169-174, 2000.

[KGV83] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, "Optimization by Simulated Annealing", *Science, Vol. 220, No. 4598*, pp. 671-680, 1983.

[KHHC07] A. Kumar, A. Hansson, J. Huisken, H. Corporaal, "An FPGA Design Flow for Reconfigurable Network-Based Multi-Processor Systems on Chip", *Proc. DATE'07, 2007*, pp. 117-122, 2007.

[KKS01] S.A Khayam, S.A. Khan, S Sadiq, "A Generic Integer Programming Approach to Hardware/Software Codesign", *Proc. of IEEE International Multi Topic Conference IEEE INMIC 2001. Technology for the 21st Century*, pp.6-9, 2001.

[KLVBR02] H. Kalte, D. Langen, E. Vonnahme, A. Brinkmann, U. Ruckert, "Dynamically Reconfigurable System-on-Programmable-Chip", *Proc. Euromicro'02 PDP*, pp. 235-242, 2002.

[LBMYB06] P. Lysaght, B. Blodget, J. Mason, J. Young , B. Bridgford, "Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs", *Proc. FPL'06*, pp. 1-6, 2006.

[LCDHKS00] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, J. Stockwood, "Hardware-Software Co-design of Embedded Reconfigurable Architectures", *Proc. DAC'00*, pp. 507-512, 2000.

[LH05] C. Lee, S. Ha, "Hardware-Software Cosynthesis of Multitask MPSoCs with Real-time Constraints", *Proc of the 6th International Conference On ASIC*, pp. 919-924, 2005.

[LYC02] S. Lee, S. Yoo, K. Choi, "Reconfigurable SoC Design with Hierarchical FSM and Synchronous Dataflow Model", *Proc. CODES'02*, pp. 199-204, 2002.

[LW99] Y. Li and W. Wolf, "Hardware/Software Co-synthesis with Memory Hierarchies," *IEEE Transactions on CAD*, vol. 18, no.10, pp.1405-1417, 1999.

[M05a] Y. Meng, “An Agent-based Reconfigurable System-on-Chip Architecture for Real-time Systems”, *Second ICEES’05*, pp. 166-173, 2005.

[M05b] K. Morris, “Prime-Time Processing. Are Embedded Systems on FPGA Ready?”, *FPGA and Programmable Logic Journal*, www.fpgajournal.com, 2005.

[MD05] B. Miramond, J.M. Delosme, “Design Space Exploration for Dynamically Reconfigurable Architectures”, *Proc. DATE‘05*, Vol1, pp. 366-371, 2005.

[MFKBS00] R. Maestre , M. Fernandez , F. J. Kurdahi , N. Bagherzadeh , H. Singh, “Configuration Management in Multi-context Reconfigurable Systems for Simultaneous Performance and Power Optimizations”, *Proc. of the 13th International Symposium on System Synthesis*, pp. 107-113, 2000.

[MMTDM07] P. Manet, D. Maufroid, L. Tosi, M. Di Ciano, O. Mulertt, Y. Gabriel, J.-D. Legat, D. Aulagnier, C. Gamrat, R. Liberati, V. La Barba, ”RECOPS: Reconfiguring Programmable Devices for Military Hardware”, *Proc. DATE ‘07*, pp. 994-999, 2007.

[MSV00] B. Mei, P. Schaumont, S. Vernalde, ”A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems”, *ProRisc workshop on Ckts, Systems and Signal processing*, 2000.

[NB01] J. Noguera , R. Badia, “A HW/SW Partitioning Algorithm for Dynamically Reconfigurable Architectures”, *Proc. DATE’01*, pp. 729-734, 2001.

[NM96] R. Niemann, P. Marwedel, "Harware/Software Partitioning using Integer Programming”, *Proc. European Design & Test Conference*, pp. 473-479, 1996.

[OCP05] J. Ou, S. B. Choi, V. K. Prasanna, „Energy-Efficient Hardware/Software Co-synthesis for a Class of Applications on Reconfigurable SoCs”, *International Journal of Embedded Systems 2005* - Vol. 1, No.1/2, pp. 91 - 102, 2005.

[OH02] H. Oh S. Ha, “Hardware-Software Cosynthesis of Multi-Mode Multi-Task Embedded Systems with Real-Time Constraints”, *Proc. of the 10th International Symposium on Hardware/Software Codesign*, pp. 133-138, 2002.

[OJ97] M. O’Nils, A. Jantsch, “Communication in Hardware/Software Embedded Systems - A Taxonomy and Problem Formulation”, *Proc. of the 15th NORCHIP Conference*, pp. 67-74, 1997.

[P02] C. H. Papadimitriou, "Złożoność obliczeniowa", *Wydawnictwa Naukowo-Techniczne*, 2002.

[PB99] K. M. Gajjala Purna , D. Bhatia, "Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers", *IEEE Transactions on Computers,* v.48 n.6, pp. 579-590, 1999.

[PMW04] J. H. Pan, T. Mitra, W.F. Wong, "Configuration Bitstream Compression for Dynamically Reconfigurable FPGAs", *Proc. ICCAD'04*, pp. 766-773, 2004.

[PP92] S. Prakash, A. Parker, "SOS: Synthesis of Aplication-Specific Heterogeneous Multiprocessor Systems", *Journal of Parallel and Distrib. Comp.,* 16, pp. 338-351, 1992.

[QSN06] Y. Qu, J.-P. Soininen, J. Nurmi, "A Parallel Configuration Model for Reducing the Run-time Reconfiguration Overhead", *Proc. DATE'06*, pp. 965-969, 2006.

[QTS05] Y. Qu, K. Tiensyrja, J.P. Soininen, "SystemC-based Design Methodology for Reconfigurable System-on-Chip", *8th Euromicro DSD'05*, pp.364-371, 2005.

[RV02] D. N. Rakhmatov, S. B.K. Vrudhula, " Hardware-Software Bipartitioning for Dynamically Reconfigurable Systems", *Proc. of the 10th International Symposium on Hardware/Software Codesign*, pp. 145-150, 2002.

[SB94] S. Singh, P. Bellec, "Virtual Hardware for Graphics Applications using FPGAs", *Proc. FCCM'94*, pp. 49-58, 1994.

[SC05] L. Shannon, P. Chow, "Leveraging Reconfigurability in the Hardware/Software Codesign Process", *Proc. FPL'05*, pp. 731-732, 2005.

[SDJ07] L. Shang, R. P. Dick, N. K. Jha, " SLOPES: Hardware–Software Cosynthesis of Low-Power Real-Time Distributed Embedded Systems With Dynamically Reconfigurable FPGAs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* pp. 508-526, 2007.

[SFMHBK02] M. Sanches-Elez, M. Fernandez, R. Maestre, R. Hemida, N. Bagherzadeh, F. J. Kurdahi, "A Complete Data Scheduler for Multi-Context Reconfigurable Architectures", *Proc. DATE'02*, pp. 547-552, 2002.

[SHE05] M.T. Schmitz, B.M. Al-Hashimi, P. Eles, “Cosynthesis of Energy-efficient Multimode Embedded Systems with Consideration of Mode-execution Probabilities”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp.153-169, 2005.

[SHT05] T.Streichert, Ch. Haubelt, J. Teich, “Online Hardware/Software Partitioning in Networked Embedded Systems”, *Proc. ASP-DAC’05*, pp. 982-985, 2005.

[SJ02] L. Shang, N.K. Jha, “Hardware-Software Co-synthesis of Low Power Real-time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs”, *Proc. ASP-DAC’02*, pp. 345-352, 2002.

[SK03] D. Shin, J. Kim, “Power-Aware Scheduling of Conditional Task Graphs in Real-Time Multiprocessor Systems”, *Proc. of the 2003 International Symposium on Low Power Electronics and Design*, pp. 408-413, 2003.

[SW97] J. Staunstrup, W. Wolf , “Hardware/Software Co-Design: Principles and Practice”, *Kluwer Academic Publishers, Boston, MA*, 1997.

[T05] S. Trimberger, “FPGA Futures: Trends, Challenges and Roadmap”, *Keynote Presentation from IEE FPGA Developers Forum,* 2005.

[T07] Tensilica Inc., "Configurable Processors: What, Why, How?", *http://www.tensilica.com/products/WP_config.htm*, 2007.

[TCJW97] S. Trimberger, D. Carberry, A. Johnson, J. Wong, “A Time-Multiplexed FPGA”, *Proc. FCCM’97*, pp. 22-28, 1997.

[VJ03] K.S. Vallerio, N.K. Jha, “Task Graph Extraction for Embedded System Synthesis”, *Proc. of the 16th International Conference on VLSI Design*, pp.480-486, 2003.

[W94] W. Wolf, “Hardware/software Co-design of Embedded Systems,” *IEEE Micro,* Vol. 14, Issue 4, pp. 26-36, 1994.

[WHE03] D. Wu, B. Al-Hashimi, P. Eles, “Scheduling and Mapping of Conditional Task Graphs for the Synthesis of Low Power Embedded Systems”, *Proc. DATE’03*, pp. 10090-10095, 2003.

[WKINN07] R. Watanabe, M. Kondo, M. Imai, H. Nakamura, T. Nanya, „Task Scheduling under Performance Constraints for Reducing the Energy Consumption of the GALS Multi-Processor SoC", *Proc. DATE'07*, pp. 797-802, 2007.

[X03] Xilinx Inc., "Processor Local Bus (PLB) v3.4 Data Sheet", *http://www.xilinx.com/support/documentation/ip_documentation/plb_v34.pdf* , 2003.

[X04] Xilinx Inc., "Two flows for Partial Reconfiguration: Module Based or Difference Based", *Xilinx Application Note XAPP290*, v.1.2, 2004.

[XLKVI04] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, "Reliability-aware Co-synthesis for Embedded Systems", *Proc. Of the 15th IEEE International Conference on Architectures and Processors*, pp. 41-50, 2004.

[XW00] Y. Xie, W. Wolf, "Co-synthesis with Custom ASICs", *Proc. ASP-DAC'2000*, pp. 129-135, 2000.

[XW01] Y. Xie, W.Wolf, "Allocation and Scheduling of Conditional Task Graph in Hardware/Software Co-synthesis", *Proc. DATE'01*, pp. 620-625, 2001.

[YT06] K. Young-Jun, K. Taewhan, "HW/SW Partitioning Techniques for Multi-mode Multi-task Embedded Applications", *Proc Of the 16th ACM Great Lakes Symposium on VLSI*, pp. 25-30, 2006.

[YW95] T.-Y. Yen, W.H. Wolf, "Sensitivity-Driven Co-Synthesis of Distributed Embedded Systems", *Proc of International Symposium on System Synthesis*, pp. 4-9, 1995.

[YW98] T.-Y. Yen, W.H. Wolf, "Performance Estimation for Real-Time Distributed Embedded Systems", *IEEE Transactions on Parallel And Distributed Systems*, vol. 9, No. 11, pp. 1125-1136, 1998.

[ZN00] X-J. Zhang, K-W. Ng, "An Effective High-level Synthesis Approach for Dynamically Reconfigurable Systems", *The 4th International Conference on High-Performance Computing in the Asia-Pacific Region-Volume 1*, pp. 343-348, 2000.

[ZX06] J.-Y. Zhan, G.-Z. Xiong, "Optimal Hardware/Software co-synthesis for core-based SoC Designs", *Jrnl of Systems Engineering and Electronics*, pp. 402-409, 2006.