AUTOMATYKA • 2010 • Tom 14 • Zeszyt 3/2

Marcin Pietroń*, Paweł Russek**, Kazimierz Wiatr***

Prototyp systemu profilowania pętli kodu źródłowego jako narzędzia analizy kodu w celu efektywnego przyspieszenia obliczeń wielkiej skali

1. Wprowadzenie

Opracowanie przedstawia badania dotyczące akceleracji obliczeń HPC (*High Performance Computing*) za pomocą układów FPGA. Pierwszą aplikacją, na której zaczęto badania jest Gaussian (aplikacja do obliczeń kwantowo-chemicznych). Środowisko sprzętowe stanowi SGI Altix 4700. Najważniejszym zagadnieniem w procesie przyspieszania aplikacji HPC jest analiza kodu w celu jego ekstrakcji do implementacji w układach FPGA. Aplikacje HPC składają się z dużej ilości skomplikowanego kodu źródłowego. Głównym problemem selekcji kodu jest brak odpowiedniego narzędzia wykonującego odpowiednią analizę co utrudnia możliwość optymalnego wykorzystania platform HPRC (*High Performance Reconfigurable Computing*) (*Rekonfigurowalne obliczenia wielkiej skali*). Celem naszych badań jest zbudowanie automatycznego narzędzia analizy kodu HPC pod kątem implementacji w układach FPGA. Jednym z wymagań jest to aby narzędzie miało charakter uniwersalny tzn. mogłoby być zastosowane do jak największej ilości aplikacji HPC i platform. Powstało już wiele publikacji przedstawiających wyniki implementacji pojedynczych algorytmów stosowanych w obliczeniach naukowych. Rezultaty tych badań pokazują, że wybrane algorytmy w HPC mogą być szybciej wykonywane w FPGA szybciej niż w standardowej jednostce CPU [5]. Implementacja tylko wybranych algorytmów względem całości aplikacji HPC nie jest efektywna i nie daje zadowalających wyników. Pokazuje to, że bez automatycznej analizy kodu uzyskanie znaczącego przyspieszenie jest dość trudne. Oprócz automatycznej analizy system wykonuje transformację wybranego kodu i implementację w FPGA. W naszym systemie do automatycznej transformacji i implementacji wykorzystywane są języki HLL (*High Level Language*) (*Języki wysokiego poziomu*) m.in.

* Katedra Elektroniki, Akademia Górniczo-Hutnicza w Krakowie
** Katedra Elektroniki, Akademia Górniczo-Hutnicza w Krakowie, ACK-CYFRONET, Kraków
*** Akademia Górniczo-Hutnicza w Krakowie, ACK-CYFRONET, Kraków

Mitrion-C, Handel-C. Języki HLL zostały stworzone z myślą aby uprościć i skrócić czas implementacji algorytmów w układach FPGA.

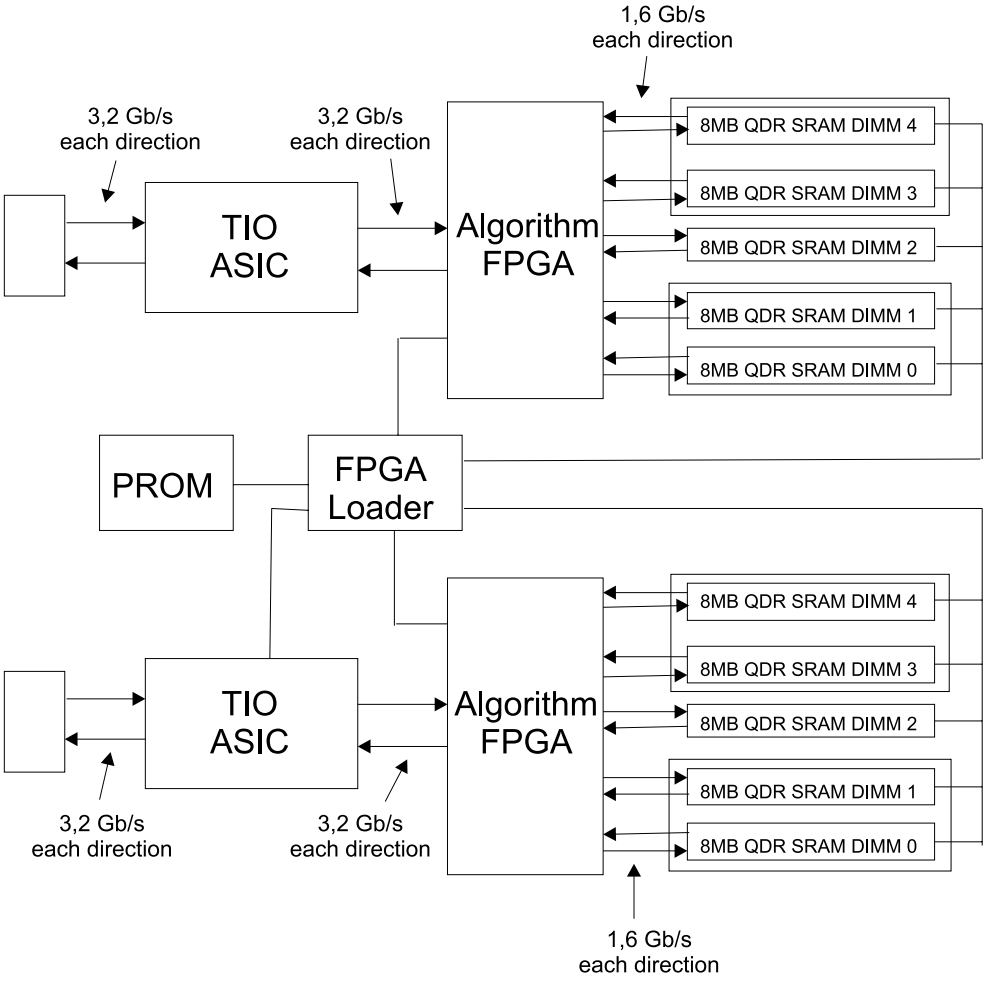
Automatyczna analiza kodu składa się z kilku mechanizmów. Główne z nich to profiler pętl (*Loop Profiler*) (*Profiler pętl*) i generator grafu przepływu danych (*Data Flow Graph Builder*) (*Generator grafu przepływu danych*). Profiler pętli jest jednym z kluczowych modułów, gdyż jak mówią badania 90% czasu aplikacje poświęcają na wykonywanie kodu pętli. Graf przepływu danych wykrywa i wizualizuje zależności między danymi używanymi w trakcie obliczeń.

2. Platforma sprzętowa

Najwięksi producenci HPC m.in. SGI, Cray stworzyli kilka platform HPRC. Ponadto pojawiło się kilka firm specjalizujących się w rozwiązaniach HPRC takich jak SRC Computers, Nallatech Ld. Używanym w projekcie rozwiązaniem jest platforma firmy SGI serii Altix.

Altix 4700 (rys. 1) jest systemem wieloprocesorowym z rozproszoną dzieloną pamięcią (od 8 do 512 jednostek CPU). Każdy procesor ma własną lokalna pamięć oraz zdolność do szybkiego dostępu do pamięci innych procesorów poprzez magistralę NUMALink. NUMALink pozwala na przesyłanie danych z przepustowością 6,4 GB/s. SGI RASC RC100 Blade składa się z dwóch układów FPGA typu Virtex-4 LX 200, z 40 MB pamięci SRAM, podzielonej logicznie na dwa bloki po 16 MB i jeden blok 8 MB. Każdy blok pamięci QDR SRAM transferuje 128 bitów danych w każdym cyklu zegara (200 MHz), w trakcie odczytu i zapisu. Moduł TIO (układ ASIC)i jednostka SSP stanowią moduł komunikacyjny pomiędzy FPGA a reszta systemu Altix. RC100 Blade natomiast dzięki połączeniu NUMALink może wymieniać dane z jednostką sterującą systemu (CPU) z przepustowością 3,2 GB/s.

Altix 4700 posiada własną wbudowana platformę (*RASC API*), która umożliwia pisanie programów na procesor główny, które mogą wykonywać skompilowany kod w języku VHDL w układzie FPGA. Drugim sposobem na pisanie programów z użyciem zasobów RC100 Blade jest użycie języków HLL (*High Level Language*). Jednym z takich języków jest Mitrion-C. Kompilator Mitrion-C generuje kod VHDL z kodu źródłowego. Następnie projekt przy użyciu narzędzi firmy Xilinx jest syntezowany i implementowany do układu FPGA. Oprócz generacji *bitstreamu*, są tworzone pliki konfiguracyjne, które pozwalają na odpowiednia komunikacje z układem FPGA. Implementacja i wywoływanie algorytmu składa się z kilku funkcji, które rezerwują zasoby, ustawiają tryb przesyłania danych oraz wiele innych zadań, które muszą być wykonane w celu poprawnego wykonania algorytmu. Skutkiem tego jest to, że optymalna struktura przyspieszonego programu powinna być zoptymalizowana pod kątem minimalizowania inicjalizacji oraz przesyłania danych do układu FPGA.



Rys. 1. SGI Altix 4700

3. Architektura systemu

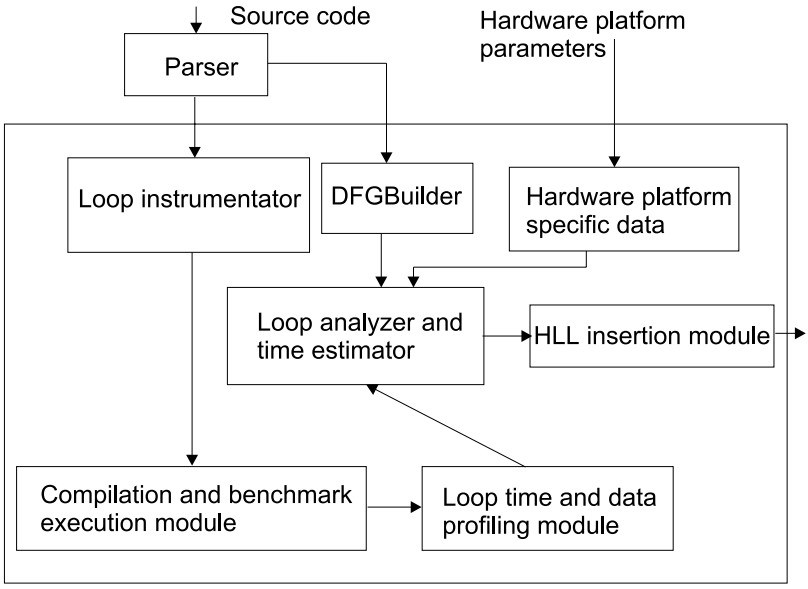
Rozdział ten prezentuje ogólną budowę systemu. System w chwili obecnej pracuje na aplikacjach HPC napisanych w językach: Fortran77 i Fortran95. System napisany jest w całości w języku Java. Rysunki 2 i 3 przedstawiają główne moduły systemu oraz najważniejsze klasy.

Szkielet systemu został podzielony na trzy główne bloki funkcjonalne:

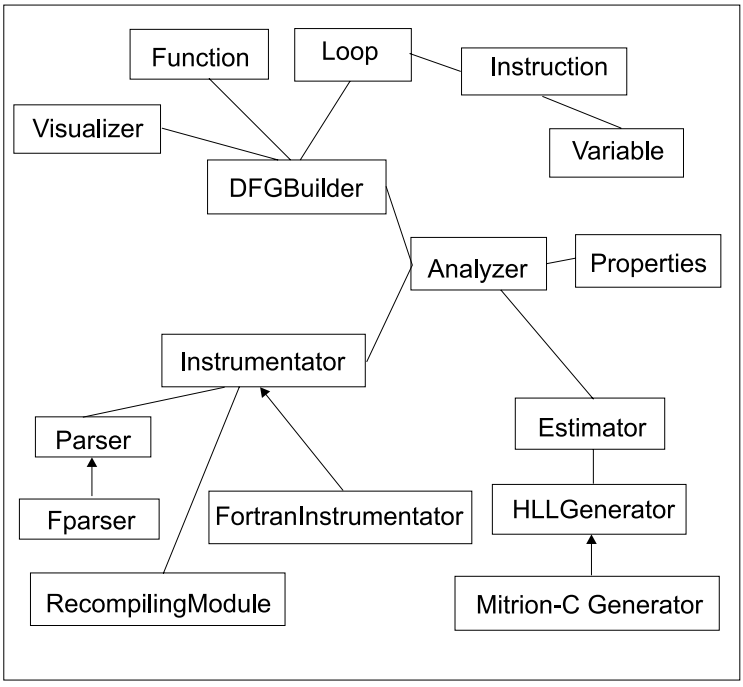
Front-end – parsuje kod źródłowy i wykonuje instrumentacje, pobiera parametry danej platformy sprzętowej,

Engine – analizuje dane, generuje graf przepływu, analizuje wykonanie pętli itd.,

Back-end – generuje nowy kod HPC z wstawioną implementacją w HLL.



Rys. 2. Schemat funkcjonalny systemu



Rys. 3. Diagram klas

Wejściem do systemu jest kod źródłowy aplikacji oraz dane sprzętowe platformy, na której ma być ona przyspieszona. To stanowi wejście do modułu *Front-End*. Ten moduł odpowiedzialny jest parsowanie kodu źródłowego oraz instrumentacje w celu dalszego profilingu aplikacji. Automatyczna instrumentacja jest potrzebna do mierzenia czasu wykonania pętli, pomiaru liczby iteracji pętli oraz szacowania ilości danych używanych w trakcie realizacji poszczególnych pętli programu. W trakcie parsowania klasa Parser tworzy klasy typu *Loop*, *Instruction* oraz *Variable* kiedy tylko rozpozna kod pętli.

Klasy te tworzone są w celu przechowania danych o danej pętli. Klasa *Loop* zawiera listę instrukcji (kolekcja klas typu *Instruction*), a te z kolei zawierają zmienne, które są używane w trakcie ich wykonania. Na podstawie utworzonych klas i przechowywanych w nich informacji generowany jest później graf przepływu danych. Parser równolegle podczas rozpoznania pętli wywołuje także klasę *Instrumentator*, której metody odpowiedzialne są za instrumentację kodu źródłowego w celu uzyskania informacji niezbędnych do wykrycia części kodu, który mógłby być zaimplementowany w FPGA. Parsowanie daje następujące informacje:

- ekstrakcja kodu pętli,
- zmienne iteracyjne pętli,
- listy instrukcji w pętlach,
- zbiory danych używanych w obliczeniach w pętlach.

Po zebraniu wyżej wymienionych informacji zebrane dane przesyłane są do generatora grafu przepływu danych (*DFGBuilder*) i modułu wykonującego pomiary za pomocą instrumentowanego kodu (*RecompilingModule*). Moduły są silnikiem systemu.

Loop DFG Builder, *Loop Execution Time Profiler*, *Loop Analyzer* i *Time Estimator* stanowią jądro systemu, które odpowiedzialne jest za zbieranie kluczowych danych, na podstawie których system może wyodrębnić części kodu do implementacji. *DFG Builder* tworzy graf przepływu danych. Przedstawia dependencje między pętlami, a także wspomaga moduł *Loop Analyer* do analizy zależności między danymi w poszczególnych pętlach. Rozdział 4 dokładnie opisuje funkcjonalność budowę modułu. *Loop Analyzer* jest klasą systemu odpowiedzialną za analizę danych zebranych przez moduły: *DFG Builder*, *Loop Time Profiler* i *Data Profiler*. Analizator bada możliwość zrównoleglenia pętli i pomaga w optymalizacji kodu pętli. *Time Estimator* estymuje czas wykonania pętli w FPGA. Czas ten jest mierzony jako suma następujących składników:

- wysyłanie i ładowanie bitstream'u do układu FPGA (stworzonego przez HLL),
- wysyłanie danych wejściowych z hosta do układu FPGA,
- czas wykonania algorytmu w FPGA (czas uzyskany z symulatora Mitriona),
- wysyłanie danych wyjściowych z algorytmu z FPGA do hosta.

Następnie wybrane części kodu stanowią dane wejściowe dla modułu *HLL Generator*, który jest ostatnim funkcjonalnym blokiem w systemie. Moduł ten dokonuje transformacji

oryginalnego kodu aplikacji na implementację w HLL. W naszym przypadku jest to generacja kodu w Mitrion-C (podrozdz. 6.2). Jak zostało wcześniej napisane, jednym z głównych celów było stworzenie systemu, który byłby łatwo adaptowalny na wiele platform sprzętowych. Podzielenie architektury systemu na trzy główne części umożliwia przy niewielkiej zmianie modułów (*front-end* i *back-end*) dostosowanie systemu do wielu aplikacji HPC (przez wymianę parsera kodu) oraz do różnych platform HPRC (przez wymianę HLL generatora).

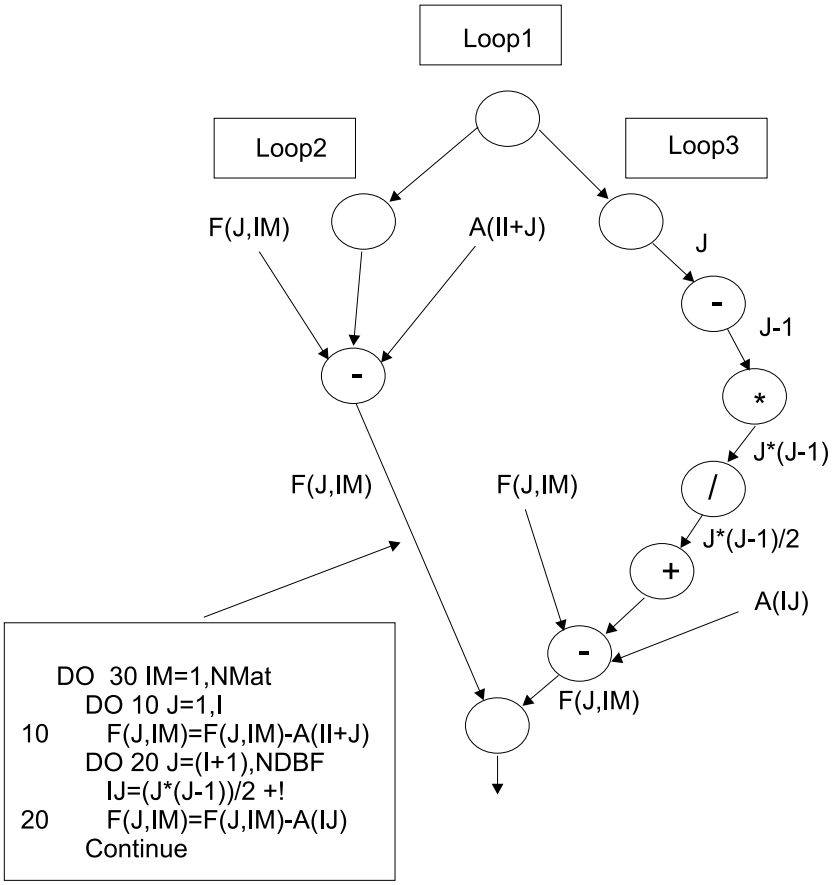
Głównymi klasami systemu są (rys. 3):

- *Parser* – parsuje kod źródłowy aplikacji.
- *DFG Builder* – generuje graf przepływu danych.
- *Instrumentator* – klasa wykonująca instrumentacje kodu m.in. instrumentacja pętli w celu pomiaru czasu jej wykonania, pomiar rozmiaru danych używanych w obliczeniach, pomiar ilości iteracji pętli itd.
- *Loop* – analizuje zależności między danymi i instrukcjami wewnątrz pętli (*Loop DFG Builder*, *Loop Data Profiler*) oraz zbiera dane dotyczące czasu wykonania pętli (*Loop Execution Time Profiler*).
- *Instruction* – przechowuje informacje o instrukcjach wewnątrz pętli m.in. rodzaj operacji w nich użyty, typ danych na których operuje, rozmiar danych.
- *Estimator* – estymuje za pomocą środowiska języka HLL czas wykonania i zajmowane zasoby algorytmu który jest wybrany przez analizator jako możliwy do przyspieszenia.
- *Visualizer* – wizualizuje graf przepływu danych.
- *Parameter* – zawiera parametry platformy sprzętowej, które podawane są przez użytkownika (przepustowość magistrali pomiędzy hostem i FPGA, wielkość pamięci SRAM itd.).
- *HLL Generator* – implementuje wybrane części kodu źródłowego w HLL.

4. Generator grafu przepływu danych (*DFG Builder*)

Głównym celem generatora jest uzyskanie informacji o zależnościach między pętlami i wspomaganie analizatora pętli w badaniach dependencji między danymi wewnątrz pętli. Przykład działania generatora jest przedstawiony na rysunku 4. W naszym grafie przepływu danych (*DFG*) węzły są pojedynczymi operacjami a krawędzie to dane wejściowe operacji. *DFG Builder* otrzymuje dane wejściowe od parsera kodu. Dane zawierają m.in. zbiór wykrytych pętli oraz listy instrukcji w nich zawartych. Ekstrakcja zależności między pętlami i wewnątrz pętli jest jednym z najważniejszych zagadnień, gdyż jest niezbędna do wykorzystania mechanizmów języków HLL (Mitrion-C). Rysunek 4 pokazuje tylko część danych zanalizowanych i zwizualizowanych w postaci wygenerowanego grafu (*Visualizer*).

Reszta danych zgromadzonych w trakcie tworzenia grafu przepływu danych jest zapisywana w plikach. Moduł *DFG Builder* tworzy graf wywołań pętli. Z tego grafu uzyskujemy informacje na temat zagnieżdżenia między pętlami. Pozwala to stwierdzić, które grupy pętli mogą być razem zaimplementowane i wywołane w FPGA. Rysunek 5 pokazuje przykładowe rezultaty działania modułu na części jednej z głównych bibliotek *Gaussiana*. 'Zbiór pętl' na osi X oznacza zbiór pętli jednocześnie implementowanych (1 – zbiór z jedną pętlą, 2 – zbiór z dwoma pętlami itd.). Oś Y przedstawia liczbę i rodzaj implementacji pętli (*wide parallel – całkowite zrównoleglenie*, *partially parallel – częściowe zrównoleglenie*, *sequential – sekwencyjne wykonanie*) dla poszczególnych wykrytych zbiorów pętl. Należy jednak zaznaczyć, że w dalszym etapie analizy wykrywane są dalsze zależności między pętlami m.in. wspólne dane wykorzystywane przez kod wielu pętli, które wspiera funkcjonalność tego modułu (rozdz. 5 i podrozdz. 6.1).



Rys. 4. Graf dependencji i przepływu danych

Rys. 5. Przykład analizy jednej z bibliotek przez Loop Profiler

5. Profiler pętli (*Loop profiler*)

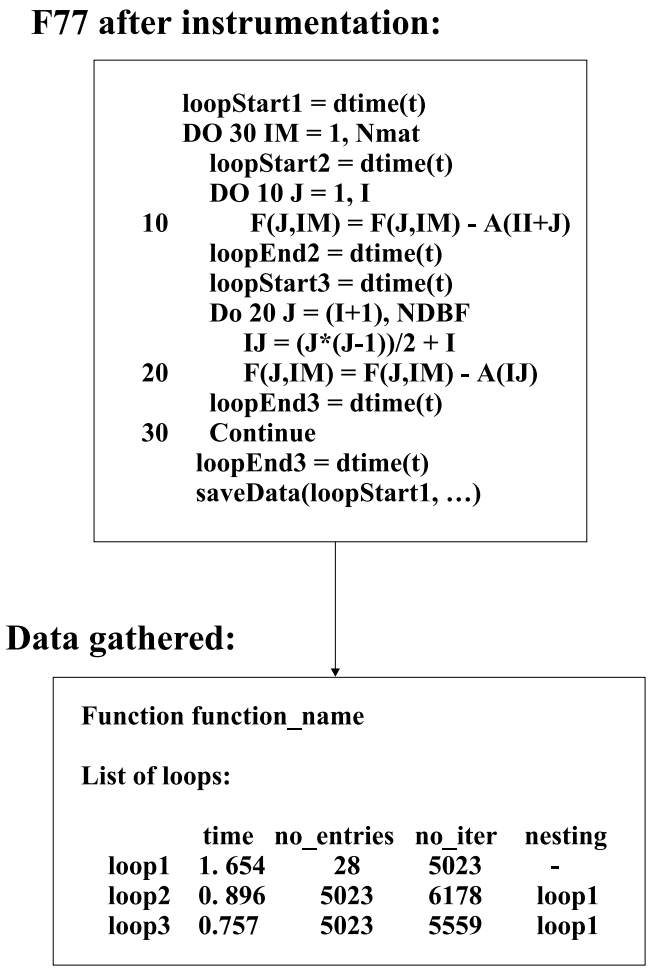
Pierwszym krokiem jaki został wykonany w celu selekcji kodu do przyspieszenia, był standardowy profiling. Ten typ profilingu ogranicza się do raportowania czasu wykonania poszczególnych funkcji [9, 11, 12]. Takie informacje są niewystarczające do znalezienia części aplikacji, które mogłyby być zaimplementowane w FPGA. Z tego powodu muszą być podjęte próby bardziej dokładnej analizy i pomiarów czasowych kodu HPC. Konsekwencją tego jest budowa systemu *hyper-profilingu*, którego głównym elementem jest Profiler pętli (*Loop Profiler*). Profilowanie pętli jest przede wszystkim procesem, który mierzy czas wykonania pętl w programie. Oprócz pomiaru czasu wiele innych danych może być zbierane w trakcie wykonywania tego typu profilingu. Dodatkowymi informacjami są m.in.:liczba iteracji pętli, liczba wykonań kodu pętli. Są dwa typy profilowania pętli: profiling statyczny i profiling dynamiczny. Profiling dynamiczny [8] jest realizowany przez dynamiczną instrumentację kodu binarnego, natomiast profiling statyczny polega na instrumentacji kodu źródłowego.

Dynamiczny profiling pętli jest zależny od kompilatora w przeciwieństwie (zależy od platformy sprzętowej) do statycznego, który zależy tylko od języka programowania, w którym jest napisana profilowana aplikacja. W obecnej wersji systemu profiling pętli może być wykonany w przypadku języka Fortran w wersji F77 i F95. Instrumentacja wykonywana jest na pętlach: *do*, *do while*. Proces profilingu analizuje kod w celu instrumentacji wyżej wymienionych rodzajów pętli poprzez wstawienie instrukcji mierzących czas wykonania pętli (rys. 6), liczbę wykonywanych iteracji oraz typów i rozmiarów danych, na których

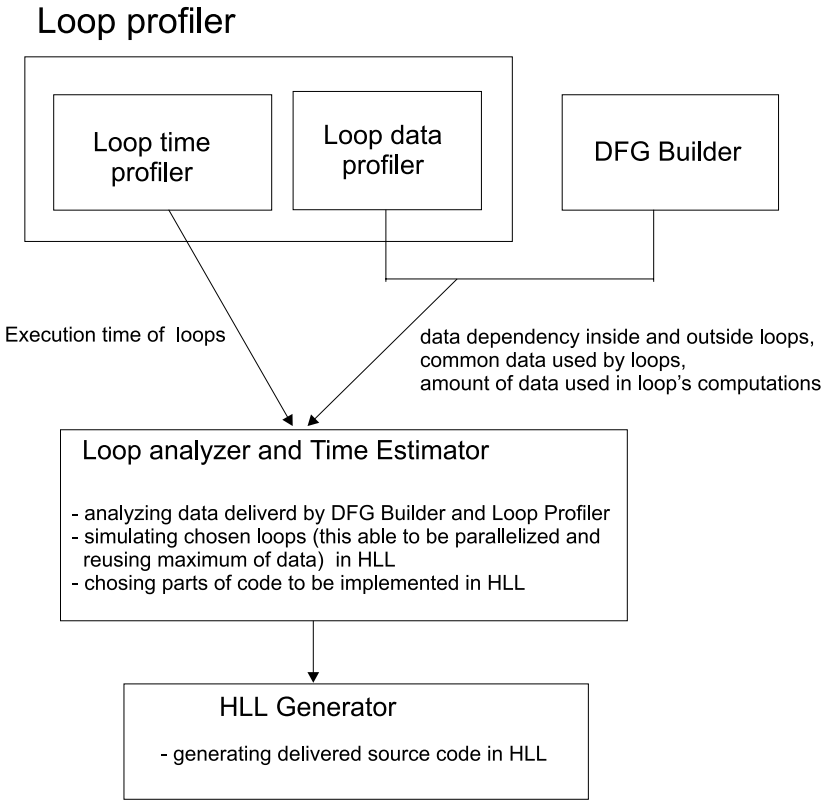
wykonywane są instrukcje pętli (rys. 8). Wyniki instrumentacji zapisywane są w specjalnych plikach (rys. 6) w celu dalszego przetwarzania. Są to między innymi:

- liczba wykonywanych iteracji pętli,
- liczba wywołań ciała pętli,
- czas wykonania pętli,
- rozmiar danych, które są używane w trakcie wykonywania pętli.

Aby dokonać selekcji wybranych części kodu, oprócz profilingu pętli niezbędna jest generacja grafu przepływu danych w celu uzyskania informacji o zależnościach między danymi w pętli oraz między pętlami (rys. 7).



Rys. 6. Loop profiling



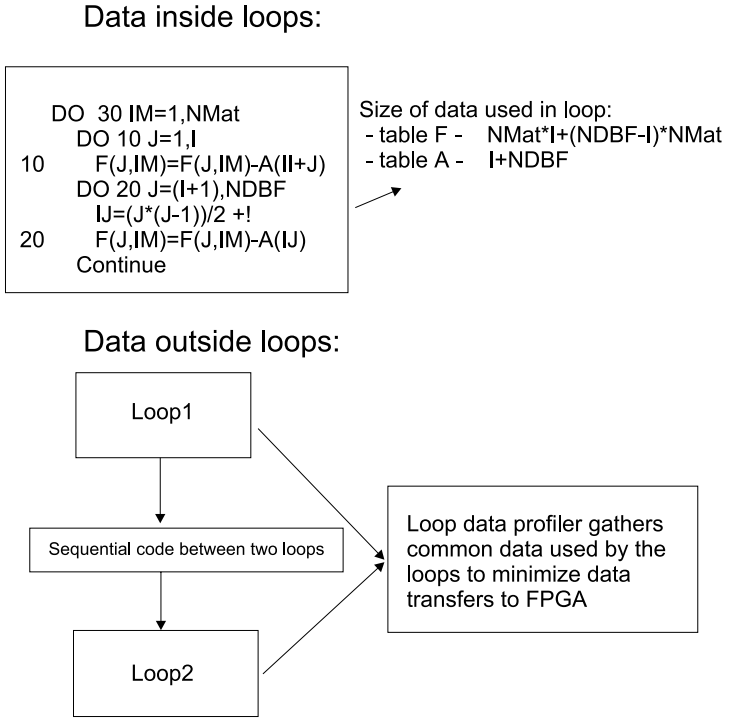
Rys. 7. Loop Analyzer

6. *Loop profiler* jako narzędzie wspomagające przyspieszanie aplikacji HPC

6.1. Profilowanie danych w pętli

Jak zostało wcześniej wspomniane, częstotliwość transferu danych pomiędzy hostem (CPU) i układem FPGA jest krytycznym zagadnieniem ograniczającym przyspieszenie aplikacji HPC na platformach HPRC. Jeżeli dane wybrane części kodu do zaimplementowania używają jak najwięcej wspólnych danych do obliczeń, to minimalizuje się częstotliwość przesyłania danych. Z tego powodu oczywisty staje się fakt monitorowania i mierzenia ilości danych występujących w obliczeniach pętli oraz ekstrakcji danych, które są wykorzystywane przez wiele algorytmów. Konsekwencją tego jest stworzenie w systemie modułu, który analizuje dane używane w obliczeniach pętli. Przede wszystkim analizowane i zbierane są następujące informacje: rozmiary i ilość danych występujące w obliczeniach, analiza danych między pętlami pod kątem ich reużycia.

Rysunek 8 przedstawia działanie modułu *Loop Data Profiler* (profilowanie danych w pętli i między pętlami). Zebrane w ten sposób dane są przekazywane estymatorowi, który na ich podstawie jest w stanie oszacować częstotliwość i rozmiar danych potrzebnych do realizacji algorytmu w FPGA.

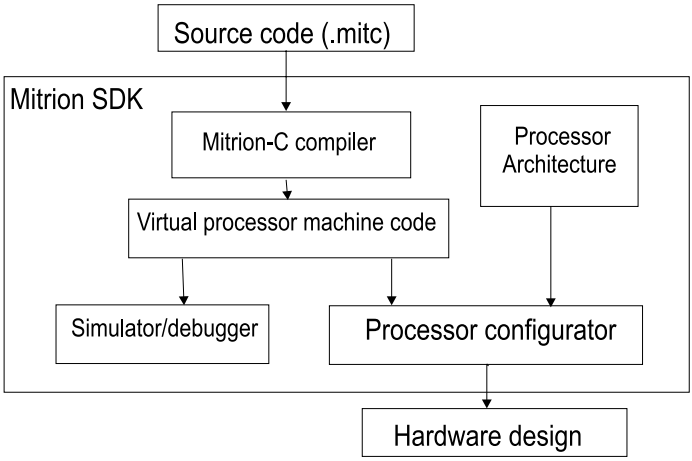


Rys. 8. Analiza danych

6.2. Automatyczna implementacja kodu aplikacji HPC w HLL (*High Level Language*)

Układy FPGA są głównie programowane poprzez języki opisu sprzętu takie jak VHDL i Verilog. Oprócz standardowych języków opisu sprzętu istnieją języki wysokiego poziomu (*High Level Languages*), które także umożliwiają bezpośrednią kompilację i implementacje do układu FPGA. Takimi językami są między innymi: Handel-C, Impulse-C i Mitrion-C. W naszym systemie aktualnie jest to tylko Mitrion-C. Mitrion-C jest kompilowany przez środowisko programistyczne na pośrednią reprezentacje dla wirtualnego procesora Mitriona. Następnie ta postać skompilowanego programu jest implementowana na wybraną platformę sprzętową. Na środowisko programistyczne Mitriona (rys. 9) składa się kompilator języka, generator i narzędzie do wizualizacji grafu przepływu danych, symulator, biblioteka MITHAL (*Mitrion Host Abstarction Layer* – biblioteka wspomagająca

wywołanie implementacji Mitriona z aplikacji napisanych w językach C/C++ i Fortran) oraz konfiguratora procesora Mitrion (konfiguracja procesora pod kątem platformy sprzętowej). Z pośredniej reprezentacji skompilowanego kodu środowisko umożliwia symulacje (symulacja czasu wykonania algorytmu, szacowanie zajętości układu) oraz debugowanie programu. Mitrion-C jest językiem, który poprzez wbudowane mechanizmy posiada możliwość równoległego wykonywania programu w układzie FPGA. Umożliwiają to przede wszystkim wbudowane typy danych (Listy, Wektory) oraz specjalne konstrukcje języka takie jak pętle. Najistotniejszymi konstrukcjami umożliwiającymi równoległość obliczeń są pętle *foreach* i *for*. Ich zastosowanie powoduje równoległe bądź potokowe wykonanie instrukcji programu. Związek pomiędzy typami pętli i typami danych na których odbywają się obliczenia przedstawia tabela 1.



Rys. 9. Środowisko Mitrion

Tabela 1
Pętle w Mitrion-C

	Vector	List
Foreach	wide parallel	pipelined
For	unrolled	sequential

Jak zostało wymienione wyżej, Mitrion-C posiada specjalną bibliotekę, która umożliwia wywoływanie skompilowanego kodu Mitrion-C na układzie FPGA z poziomu aplikacji napisanej w C/C++ lub w Fortranie. Umożliwia to zastępowanie wybranych części kodu implementacją w Mitrion-C. Biblioteka ta umożliwia wykorzystanie wyników profilowania aplikacji i automatyczną generację wybranego kodu w Mitrion-C. Moduł automatycznego

mapowania i transformacji kodu aplikacji jest jeszcze w fazie badań (*HLL Generator*). Jedną z pierwszych publikacji porównujących czasy wykonania wybranego algorytmu w Mitrion-C i języku C zostały przedstawione w pracy [2, 3].

7. Wnioski

Dotychczasowe wyniki badań pokazują, ze proces przyspieszania aplikacji HPC w układach FPGA jest problemem dość złożonym. Profilowanie z użyciem standardowych profilerów nie daje satysfakcjonujących efektów. Profilery te dają tylko informacje o czasie wykonania funkcji. Jest to niewystarczająca informacja do wyselekcjonowania kodu do implementacji w układzie FPGA. Jak pokazują badania, próba przyspieszenia aplikacji HPC poprzez implementacje funkcji elementarnych np. funkcja eksponenty [10] jest też nieefektywna. To powoduje, że automatyczna analiza kodu źródłowego HPC jest niezbędna w celu optymalnego oszacowania możliwości przyspieszenia aplikacji. Miejscem, które należy monitorować, są przede wszystkie pętle kodu źródłowego. Pierwszym z powodów jest to, że programy większość czasu spędzają na ich wykonaniu. Drugim czynnikiem jest to, iż są to części kodu, które są możliwe do zrównoleglenia. Z tego powodu główną częścią systemu stał się profiler pętli (*Loop profiler*). Należy nadmienić, że profilowanie pętli bez dodatkowej analizy m.in. zależności między danymi oraz rozmiaru danych używanych w obliczeniach, jest w większości przypadków bezużyteczne (*DFG Builder*).

Następnymi krokami, które należy podjąć w celu udoskonalenia systemu, jest optymalizacja predykcji czasu wykonania i zajętości zasobów wybranych części kodu w układach FPGA. Kolejny zagadnieniem, które należy wykonać, jest próba automatycznej analizy zakresu danych używanych w trakcie obliczeń. Dzięki temu możliwe byłoby w niektórych przypadkach zmniejszenie reprezentacji danych. Skrócenie reprezentacji danych wpływa znacznie na szybkość wykonywanych obliczeń w FPGA oraz zmniejsza też narzut czasowy związany z przesyłaniem danych do układu FPGA.

System, który jest realizowany, jest innowacyjnym przedsięwzięciem, gdyż nie istnieje jeszcze narzędzie do automatycznej analizy kodu źródłowego pod katem jego przyspieszenia w FPGA. Ponadto wykorzystanie języków HLL nie jest tak wielkie w dziedzinie HPC, gdyż wymaga znajdywania i przepisywanie ogromnej ilości kodu. Automatyczne narzędzie znacznie wspomaga proces wykorzystania takich języków w istniejących aplikacjach HPC.

Literatura

[1] Bennett D., Dellinger E., Mason J., Sundarajan P., *An FPGA-oriented target language for HLL compilation*. Reconfigurable Systems Summer Institute, RSSI 2006.

[2] Kindratenko V., Brunner R., Myers A., *Mitrion-C. Application Development on SGI Altix 350/ RC100*. International Symposium on Field Programmable Custom Computing Machines, 2007, 239–250.

[3] Kindratenko V., Myers A. , Brunner R., *Using Mitrion-C to implement floating-point arithmetic on a Cray XD1 supercomputer*. 2nd Annual Reconfigurable Systems Summer Institute, 2006.

[4] Liu K., Cameron Ch., Sarkady A., *Using Mitrion-C to implement floating-point arithmetic on a Cray XD1 supercomputer.* DoD HPCMP Users Group Conference, 2008, 391–395.
[5] Deng L., Kim J.,S.,Mangalagiri P., Irick K., Sobti K., Kandemir M., Narayanan V., Chakrabarti Ch., Pitsianis N., Sun X., *An Automated Framework for Accelerating Numerical Algorithms on Reconfigurable Platforms Using Algorithmic/Architectural Optimization.* IEEE Transactions on Computers, vol. 58, Issue 12, 2009.
[6] Messmer P., Bodenner R., *Accelerating Scentific Applications Using FPGAs.* XCell Journal, 2006.
[7] Mohl S., *The Mitrion-C programming language.* Mitrionics Inc., 2006.
[8] Moseley T., Grunwald D., Connors A., Ramanujam R., Tovinkere V., Peri R., *LoopProf: Dynamic Techniques for Loop Detection and Profiling*. Proc. of the 2006 Workshop on Binary Instrumentation and Applications, 2006.
[9] Pietroń M., Wiatr K., Russek P., *Metodyka sprzetowej akceleracji obliczeń w środowisku obliczeniowym komputerow dużej mocy.* Automatyka (półrocznik AGH), 2007.
[10] Pietroń M., Russek P., Wiatr K., Jamro E., Wielgosz M., *Two Electron Integrals calculation accelerated with Double Precision exp*() *Hardware Module.* Reconfigurable Systems Summer Institute, 2007.
[11] Russek P, Wiatr K., *The prospect of computing acceleration using reconfigurable logic technology in huge computational power systems.* Proc. of IFAC Workshop on Programable Devices and Embedded Systems, Brno, 2006.
[12] Russek P., Wiatr K., *Perspektywa przyśpieszania obliczeń instalacjach o wielkich mocach obliczeniowych za pomocą układów logiki rekonfigurowalnej*. Automatyka (półrocznik AGH), t. 9, z. 3, Kraków, 2005.
[13] Gasper P., Herbst C., McCough J., Rickett C., Stubbendieck G., *Automatic Parallelization of Sequential C Code.* Midwest Instruction and Computing Symposium, Duluth, MN, 2003.
[14] Gong W., Wang G., Kastner R., *A High Performance Application Representation for Reconfigurable Systems*. Conference on Engineering of Reconfigurable Systems and Algorithms, 2004.
[15] Memik S.O., Bozorgzadeh G., Kastner R., Sarrafzadeh M., *A Scheduling Algorithm for Optimization and Planning in High-level Synthesis.* ACM Transactions on Design Automation of Electronic Systems, vol. 10, No. 1, 2005.