Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa inżynierska

Implementacja maszyny wirtualnej dla funkcyjnych języków programowania wspierających przetwarzanie współbieżne.

Implementation of a virtual machine for functional programming languages with support for concurrent computing.

Autor:	*Kajetan Rzepecki*
Kierunek studiów:	*Informatyka*
Opiekun pracy:	*dr inż. Piotr Matyasik*

Kraków, 2013

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

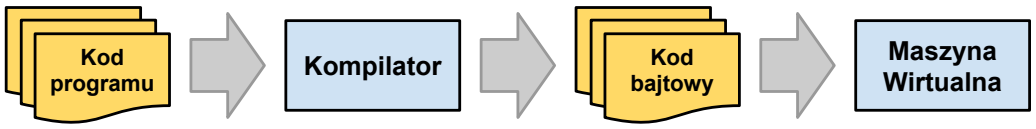
Serdecznie dziękuję opiekunowi pracy za wsparcie merytoryczne oraz dobre rady edytorskie pomocne w tworzeniu pracy.

Spis treści

1. Wstęp

Tematem pracy jest implementacja *maszyny wirtualnej* dla funkcyjnych języków programowania wspierających *przetwarzanie współbieżne*.

Maszyna wirtualna jest warstwą abstrakcji leżącą pomiędzy programem a rzeczywistym sprzętem, która pozwala uniezależnić ów program od rozbieżności w działaniu różnych architektur komputerów. Wystarczy zaimplementować maszynę wirtualną dla danej architektury rzeczywistego sprzętu by umożliwić uruchamianie na niej wszystkich kompatybilnych z programów. Rysunek 1.1 prezentuje uproszczony schemat takiego rozwiązania - programy docelowe zostają skompilowane do *kodu bajtowego* akceptowanego przez maszynę wirtualną a dopiero ów bajtkod jest przez nią uruchamiany.



Rysunek 1.1: Schemat interakcji z Maszyną Wirtualną.

Przetwarzanie współbieżne opiera się o współistnienie wielu procesów, które konkurują o dostęp do współdzielonych zasobów. W kontekście pracy, przetwarzanie współbieżne jest rozumiane jako asynchroniczne przekazywanie wiadomości pomiędzy działającymi, autonomicznymi procesami, czyli jako Model Aktorowy [1, 2].

Celem pracy jest stworzenie interpretera kodu bajtowego zdolnego uruchamiać kod skompilowanych programów, kolektora obiektów nieosiągalnych umożliwiającego automatyczne zarządzanie pamięcią oraz architektury symetrycznego multiprocesora (SMP) zapewniającego rzeczywistą współbieżność uruchamianych programów w oparciu o Model Aktorowy. Językiem implementacji projektu jest język `D` (w wersji 2.0 opisanej w [3]), stosunkowo nowoczesny, kompilowany do kodu maszynowego następca języka `C++`.

1.1. Motywacja pracy

Motywacją powstania pracy są problemy napotkane podczas użytkowania języka `Erlang` [4], dotyczące wydajności przesyłania wiadomości średniego rozmiaru w obecnej, standardowej jego implementacji. Problemy owe zilustrowano na listingu 1.

Zaprezentowany fragment kodu odczytuje plik w formacie JSON, który następnie jest dekodowany do wewnętrznej reprezentacji posiadającej skomplikowaną strukturę, by ostatecznie został on wysłany do dużej liczby współbieżnie działających procesów celem dalszego przetwarzania (linia 8). Rozwiązanie takie powoduje znaczący spadek wydajności.

```
1  start() ->
2      Data = file:read("file.json"),     %% <<"Dane ...">>
3      transmogrify(Data).
4
5  transmogrify(Data) ->
6      Pids = framework:spawn_bajilion_procs(fun do_stuff/1),
7      JSON = json:decode(Data),          %% {[Dane ...]}
8      framework:map_reduce(Pids, JSON). %% !#&%^@
9
10 do_stuff(JSON) ->
11     %% Operacje na danych.
12     result.
```

Listing 1: Fragment kodu prezentujący problem występujący w języku `Erlang`.

Język `Erlang` wykorzystuje skomplikowaną architekturę pamięci, która w różny sposób traktuje obiekty różnego typu. Większość obiektów, w szczególności skomplikowana strukturalnie reprezentacja danych w formacie JSON, przechowywana jest w prywatnych stertach każdego procesu i musi być kopiowana podczas przesyłania jej w wiadomościach pomiędzy nimi. Reguła ta nie dotyczy danych binarnych, w szczególności danych odczytanych z pliku, ponieważ te korzystają z innych algorytmów nie wymagających kopiowania kosztem większego zużycia pamięci.

W związku z tym, aby zaradzić problemowi opisanemu powyżej, wystarczy przenieść operację dekodowania danych odczytanych z pliku bezpośrednio do procesów na nich operujących (listing 2). W nowej wersji procesy przesyłają jedynie dane binarne, które nie wymagają kopiowania pamięci, a narzut wydajności spowodowany wielokrotnym ich dekodowaniem jest niższy niż ten spowodowany nadmiernym kopiowianiem. W efekcie, kod działa wydajniej, kosztem logiki przepływu danych i organizacji modułów.

Celem pracy jest uniknięcie problemu nadmiernego kopiowania pamięci przez wybranie odpowiedniego modelu pamięci i implementację algorytmów kolekcji obiektow nieosiągal-

```
1 transmogrify(Data) ->
2     Pids = framework:spawn_bajilion_procs(fun do_stuff/1),
3     framework:map_reduce(Pids, Data).
4
5 do_stuff(Data) ->              %% <<"Dane ...">>
6     JSON = json:decode(Data), %% {[Dane ...]} * bazylion
7     %% Operacje na danych.
8     result.
```

Listing 2: Suboptymalne rozwiązanie problemu w języku `Erlang`.

nych, które umożliwiają przesyłanie wiadomości pomiędzy procesami bez konieczności kopiowania ich zawartości.

1.2. Zawartość pracy

W skład pracy wchodzi implementacja interpretera kodu bajtowego, kolektora obiektów nieosiągalnych oraz symetrycznego multiprocesora (SMP).

Sekcja 1 opisuje cele, motywację, zakres oraz zawartość pracy.

Sekcja 2 przybliża architekturę maszyny wirtualnej ThesisVM zaimplementowanej w ramach pracy, zaczynając od reprezentacji pośredniej programów (TVMIR) i jej kompilacji do kodu bajtowego, przez interpretację kodu bajtowego i zarządzanie pamięcią do projektu przetwarzania współbieżnego.

Sekcja 3 szczegółowo opisuje implementację interpretera kodu bajtowego maszyny wirtualnej ThesisVM. Zaprezentowane zostają reprezentacje różnych obiektów, na których operuje maszyna, implementacja wpudowanych operatorów i funkcji prymitywnych oraz reprezentacja i generowanie kodu bajtowego akceptowanego przez interpreter.

Sekcja 4 szczegółowo prezentuje implementację wybranego modelu pamięci, alokatora nowych obiektów oraz kolektora obiektów nieosiągalnych.

Sekcja 5 szczegółowo opisuje implementację asynchronicznego przekazywania wiadomości i symetrycznego multiprocesora w maszynie ThesisVM. Zaprezentowana zostaje implementacja Modelu Aktorowego i harmonogramowania procesów.

Sekcja 6 zawiera podsumowanie pracy oraz zarys możliwych kierunków dalszego rozwoju projektu.

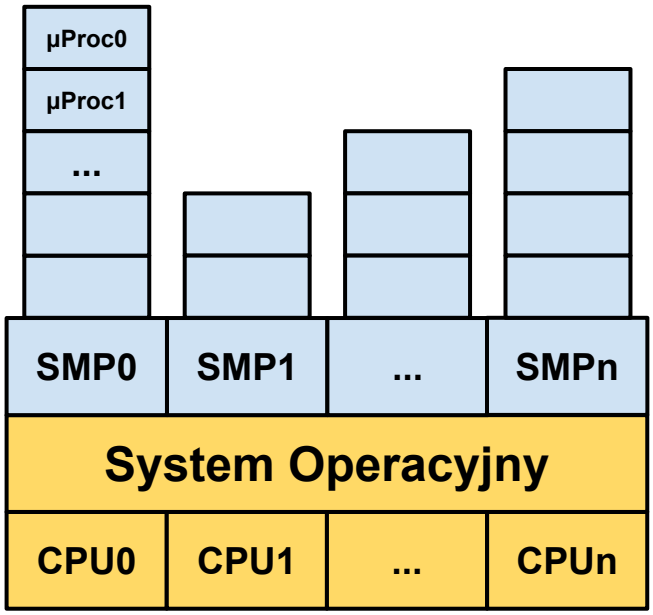
Dodatki A, B i C zawierają odpowiednio przykładowe programy gotowe do uruchomienia na maszynie wirtualnej ThesisVM, spis wbudowanych operatorów i funkcji prymitywnych oraz spisy rysunków, tablic i fragmentów kodu znajdujących się w tekście pracy.

2. Architektura ThesisVM

Ninejsza sekcja opisuje architekturę maszyny wirtualnej ThesisVM powstałej na potrzeby pracy oraz języka przez nią akceptowanego.

Rysunek 2.1 zawiera schematyczną reprezentację maszyny wirtualnej ThesisVM uwzględniającą architekturę procesora sprzętu, na którym działa system operacyjny oraz sama maszyna wirtualna. Na schemacie widać poszczególne podsystemy ThesisVM, takie jak autonomiczne procesy (zwane dalej *mikroprocesami*, `µProcN`), czy symetryczne multiprocesory (zwane dalej `SMPn`).

Mikroprocesy są przypisane do symetrycznych multiprocesorów w stosunku wiele-do-jednego, to znaczy każdy mikroproces jest przypisany do dokładnie jednego symetrycznego multiprocesora, który natomiast może zarządzać zbiorem wielu mikroprocesów.



Rysunek 2.1: Architektura maszyny wirtualnej ThesisVM.

Każdy symetryczny multiprocesor działa w osobnym wątku procesora sprzętowego, zapewniając rzeczywistą współbieżność. Wszystkie `SMPn` są takie same i wykonują ta-

kie same zadania, czyli harmonogramowanie i wywłaszczanie mikroprocesów, a różni je jedynie stan, w którym się znajdują oraz zbiór procesów, którymi zarządzają. Na schemacie widnieje mapowanie jeden-do-jednego pomiędzy rdzeniami procesora (`CPUn`) a poszczególnymi `SMPn`, nie jest to jednak wymóg konieczny i zależy od konfiguracji maszyny wirtualnej. Konfigurowalna ilość równocześnie działających SMP pomaga osiągnąć lepszą skalowalność maszyny wirtualnej i może być zmieniana dynamicznie wedle potrzeb.

Pozostając w zgodzie ze schematem przedstawionym na rysunku 1.1, interakcja z maszyną ThesisVM przebiega w analogiczny sposób. Kod programów w reprezentacji pośredniej (TVMIR) jest kompilowany do kodu bajtowego akceptowanego przez maszynę wirtualną, która następnie go ładuje i wykonuje umożliwiając zrównoleglenie obliczeń poprzez tworzenie nowych procesów i przesyłanie pomiędzy nimi wiadomości.

2.1. Reprezentacja pośrednia programów

ThesisVM wykorzystuje prostą reprezentację pośrednią programów w postaci TVMIR - języka lisp'owego z rodziny `Scheme` [5], który jest dostatecznie ekspresywny, by można w nim było zapisać nietrywialne algorytmy, a jednocześnie na tyle prosty, by ułatwić jego późniejszą kompilację do kodu bajtowego akceptowanego przez maszynę wirtualną.

Języki pośrednie reprezentacji programów są często stosowane w implementacjach wielu maszyn wirtualnych, takich jak ParrotVM, czy CoreVM [6], a także w implementacjach kompilatorów kodu maszynowego wielu języków programowania (na przykład GCC, LLVM). Reprezentacje pośrednie mają wiele zalet, począwszy od ułatwienia wsparcia dla szerszej gamy języków wysokiego poziomu, na możliwości tworzenia wygodnych założeń dodatkowych kończąc.

Na listingu 3 spisana w formacie BNF została gramatyka języka reprezentacji pośredniej wykorzystanego w maszynie wirtualnej ThesisVM. Gramatyka ta jest nieskomplikowana i w dużej mierze przypomina gramatiki różnych dialektów języka `Lisp`.

Języki z rodziny `Lisp` są bardzo wygodnym medium dla pośredniej reprezentacji programów ponieważ przedstawiają one drzewo syntaktyczne analizowanego kodu programu i nie wymagają skomplikowanego algorytmu parsowania. Dodatkowo, homoikoniczność tych języków może pomóc w tworzeniu narzędzi służących do przetwarzania kodu rozpatrywanego języka (w szczególności kompilatorów) bezpośrednio w rozpatrywanym języku. Temat ten został dogłębnie zbadany w [5]. Dodatek A zawiera przykłady kodu w języku pośredniej reprezentacji programów TVMIR.

Język reprezentacji pośredniej przedstawiony w pracy wymaga stworzenia kilku założeń dodatkowych dotyczących transformacji kodu. Najważniejszym z nich jest konieczność przeprowadzenia operacji lambda-unoszenia (ang. *lambda lifting*), opisanej bardzo dokładnie w [6], której efekt zaprezentowano na listingu 4.

```
<program>           ::= <definitions>
<definitions>       ::= <definition> <definitions> | ''
<definition>        ::= '(' 'define' <argument-list>
                                     <function-body> ')'
<argument-list>     ::= '(' <function-name> <arguments> ')'
<arguments>         ::= <argument-name> <arguments> | ''
<argument-name>     ::= <symbol>
<function-name>     ::= <symbol>
<function-body>     ::= <expression>
<expression>        ::= <simple-expression> | <application>
                      | <conditional> | <quote>
<simple-expression> ::= <list> | <symbol> | <number>
<application>       ::= '(' <expression> <expressions> ')'
<expressions>       ::= <exrpession> <exrpessions> | ''
<conditional>       ::= '(' 'if' <expression>
                                 <expression>
                                 <expression> ')'
<quote>             ::= ''' <expression>
<list>              ::= '(' <expression> ')'
<symbol>            ::= <literal-string> | <atom>
<literal-string>    ::= '"' "Dowolny literał znakowy." '"'
<atom>              ::= "Dowolny literał znakowy bez znaków białych."
<number>            ::= "Dowolny literał liczbowy."
```

Listing 3: Gramatyka języka TVMIR.

Lambda-unoszenie polega na transformacji ciał funkcji w taki sposób, by tworzone w nich funkcje anonimowe zostały przeniesione na poziom główny zasięgu nazw (ang. *top-level scope*) dzięki czemu do ich implementacji wystarczy jedynie częściowa aplikacja funkcji. Na drugiej części listingu 4 funkcja `make-adder` zwracająca anonimową funkcję została transformowana na dwie funkcje, z których `make-adder` pozostaje funkcją unarną, która korzysta z częściowej aplikacji funkcji binarnej `__make-adder_lambda0` wykonującej operację dodawania.

Pełna i poprawna implementacja operacji lambda-unoszenia jest skomplikowana, toteż nie została zawarta w dołączonym do projektu kompilatorze kodu bajtowego i musi zostać wykonana ręcznie.

Język pośredniej reprezentacji programów zastosowany w maszynie wirtualnej ThesisVM jest bardzo podobny do języka `Core Lang` wykorzystywanego w [6], jednak nie

```
;; Przed lambda-unoszeniem:
(define (make-adder n)
  (lambda (x)
    (+ x n)))
```

```
;; Po lambda-unoszeniu:
(define (__make-adder_lambda0 n x)
  (+ x n))

(define (make-adder n)
  (__make-adder_lambda n))
```

Listing 4: Fragmenty kodu prezentujące operację lambda-unoszenia.

wspiera on niektórych jego konstrukcji, takich jak `let(rec)`, czy definicje dowolnych obiektów złożonych. Z drugiej strony wspiera on konstrukcje związane z Modelem Aktorowym (`receive`, `send` oraz `spawn`) oraz jest w stanie emulować brakujące konstrukcje odpowiednio przez wykorzystanie transformacji kodu połączonej z lambda-unoszeniem (listing 5) oraz "tagowania" list (przechowywania informacji o typie obiektu w pierwszym elemencie listy enkodującej ten obiekt).

```
;; Przed transformacją:
(define (function x)
  (let ((value (* 2 x)))
    (* value value)))

;; Po transformacji:
(define (function x)
  ((lambda (value)
     (* value value))
   (* 2 x)))
```

```
;; Po lambda-unoszeniu:
(define (__function_lambda0 value)
  (* value value))

(define (function x)
  (__function_lambda0 (* 2 x)))
```

Listing 5: Ograniczona implementacja konstrukcji `let`.

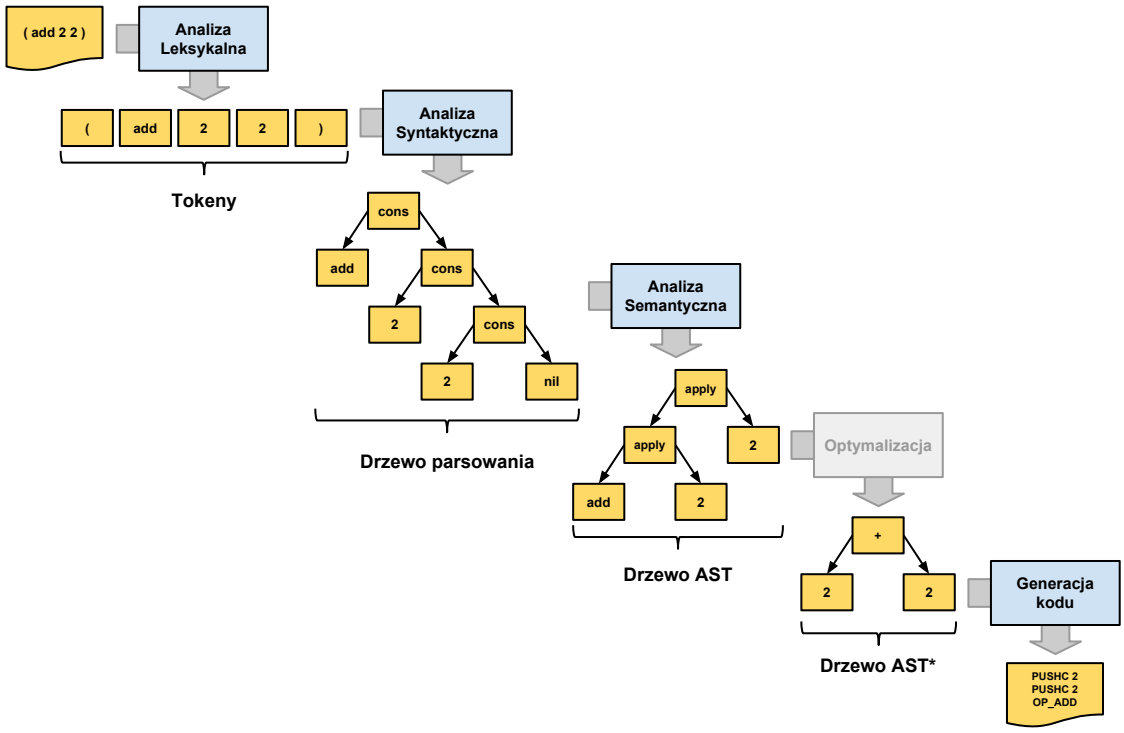
Kolejnym podobnym językiem reprezentacji pośredniej jest `Core Erlang` [7] wykorzystywany w standardowej implementacji języka `Erlang`. TVMIR jest bardzo okrojoną wersją języka `Core Erlang`, pozbawioną elementów dopasowywania wzorców, która jednak wspiera pozostałe ważne jego elementy, takie jak konstrukcje odpowiedzialne za tworzenie procesów oraz przesyłanie i odbieranie wiadomości. Istnieje możliwość rozszerzenia funkcjonalności TVMIR celem wsparcia pełnej specyfikacji `Core Erlang` [8], jednak jest to poza zakresem pracy. Więcej informacji o przyszłych kierunkach rozwoju projektu zostało zawarte w sekcji 6.3.

2.2. Kompilacja kodu bajtowego

Język pośredniej reprezentacji programów jest wygodnym medium do zapisu algorytmów, jednak wymaga on uprzedniego skompilowania do kodu bajtowego, który jest akceptowany przez maszynę wirtualną ThesisVM.

Ponieważ kompilacja kodu nie jest *stricte* tematem pracy, mniej ważne szczegóły implementacji zostały pominięte, a niniejsza sekcja zarysowuje poszczególne fazy kompilacji kodu bajtowego ThesisVM.

Rysunek 2.2 zawiera schemat działania kompilatora kodu bajtowego ThesisVM wraz z przykładami pośrednich reprezentacji kompilowanego kodu w poszczególnych fazach kompilacji.



Rysunek 2.2: Schemat potokowega działania kompilatora kodu bajtowego ThesisVM wraz ze przykładami reprezentacji danych poszczególnych faz kompilacji.

Kompilator został zaimplementowany w sposób *potokowy*, to znaczy poszczególne fazy są logicznie odseparowane od siebie i wykonywane jedna po drugiej. Dzięki zastosowaniu leniwych konstrukcji języka `D` [3] wszystkie te fazy odbywają się *jednocześnie* i *na rządanie* a w przypadku wykrycia błędu w danej fazie poprzednie fazy natychmiastowo się kończą, bez konieczności przetworzenia całego zestawu danych, które otrzymały na wejściu.

Pierwszą fazą jest faza analizy leksykalnej, której zadaniem jest przetworzenie *strumienia znaków* kodu źródłowego programu w pośredniej reprezentacji TVMIR do *strumienia tokenów*, czyli elementarnych ciągów znaków będących leksemami języka. Faza ta przeprowadza także walidację składni na poziomie tokenów oraz filtrację niepotrzebnych tokenów (takich jak znaki białe, które nie mają znaczenia w TVMIR).

Drugą fazą jest faza analizy syntaktycznej, której zadaniem jest przetworzenie powstającego leniwie *strumienia tokenów* na *wstępne drzewo parsowania* składające się z prymitywnych konstrukcji języka TVMIR, takich jak listy, symbole i liczby. Faza ta waliduje składnię na poziomie zaawansowanych konstrukcji języka, które dzięki jego homoikoniczności zbudowane są z prymitywniejszych jego konstrukcji.

Trzecią fazą jest faza analizy semantycznej, której zadaniem jest przetworzenie *wstępnego drzewa parsowania* na bardziej abstrakcyjne *drzewo składniowe* (ang. *Abstract Syntax Tree*, *AST*) składające się semantycznie znaczących węzłów, takich jak aplikacja funkcji, wywołania operatorów wbudowanych, czy odwołania do zmiennych. Faza ta waliduje kod na poziomie semantycznym, sprawdzając poprawność wykorzystania różnych konstrukcji języka TVMIR.

Czwartą fazą jest faza optymalizacji, której zadaniem jest transformacja *drzewa składniowego* powstałego w poprzedniej fazie do jego ekwiwalentu działającego szybciej po skompilowaniu. Faza ta obecnie nie wykonuje rzadnych interesujących transformacji, jednak istnieje możliwość rozszerzenia jej funkcjonalności w przyszłości (opisane krótko w sekcji 6.3).

Ostatnią, piątą fazą kompilacji jest faza generacji kodu bajtowego akceptowanego przez ThesisVM. Zadaniem tej fazy jest przetworzenie *drzewa składniowego* do *strumienia kodu bajtowego* za pomocą reguł kompilacji zgodnych z wybranym modelem maszyny wirtualnej (opisane szczegółowo w sekcji 3.6).

2.3. Interpretacja kodu bajtowego

Istnieje wiele różnych modeli maszyn wirtualnych cechujących się różnymi architekturami interpreterów kodu bajtowego, czy nawet stopniem abstrakcyjności (tak zwane maszyny abstrakcyjne).

Pod względem architektury interpretera kodu bajtowego można wyróżnić trzy główne architektury maszyn wirtualnych:

– architekturę **stosową**, korzystającą ekskluzywnie z jednego lub wielu stosów podczas przetwarzania danych, która charakteryzuje się krótkimi, pod względem zajmowanej pamięci, instrukcjami;

– architekturę **rejestrową**, korzystającą ekskluzywnie z wielu rejestrów podczas przetwarzania danych, która charakteryzuje się instrukcjami przyjmującymi wiele argumentów określających adresy rejestrów maszyny;

– architektury **hybrydowe**, łączące dwa powyższe rozwiązania w różnym stopniu.

Pod względem abstrakcyjności maszyny wirtualne można podzielić na dwie główne grupy:

– **niskopoziomowe**, do których należą maszyny implementujące wyżej wymienione architektury; główną cechą maszyn niskopoziomowych jest obecność stosunkowo nieskomplikowanego kodu bajtowego, który jest przez maszynę interpretowany podczas jej działania;

– **wysokopoziomowe**, które wymagają niestandardowego traktowania kodu programów; na przykład maszyna redukcji grafowych G-machine wykorzystująca grafową naturę kodu języków funkcyjnych do zrównoleglenia jego ewaluacji, opisana szczegółowo w [6].

Od wyboru architektury interpretera kodu bajtowego bardzo często zależą dostępne funkcjonalności docelowego języka programowania. W celu wybrania odpowiedniej architektury należy przeprowadzić szczegółową analizę porządanych funkcjonalności implementowanego języka i możliwości ich zrealizowania w poszczególnych modelach maszyny wirtualnej. Szczegółowa analiza wpływu języka na możliwość jego zaimplementowania w danej architekturze została zawarta w [9] wraz z praktycznymi wskazówkami dotyczącymi implementacji maszyn wirtualnych, co okazało się niezastąpionym źródłem wiedzy pomocnym przy implementacji ThesisVM.

Interpreter kodu bajtowego zaimplementowany w ramach pracy wykorzystuje niskopoziomową architekturę stosową wykorzystującą wiele stosów oraz niewielki zbiór rejestrów i jest zmodyfikowaną wersją interpretera opisanego w [6, rozdział 4]. Szczegółowy opis implementacji został zawarty w dedykowanej mu sekcji 3 pracy.

2.4. Zarządzanie pamięcią

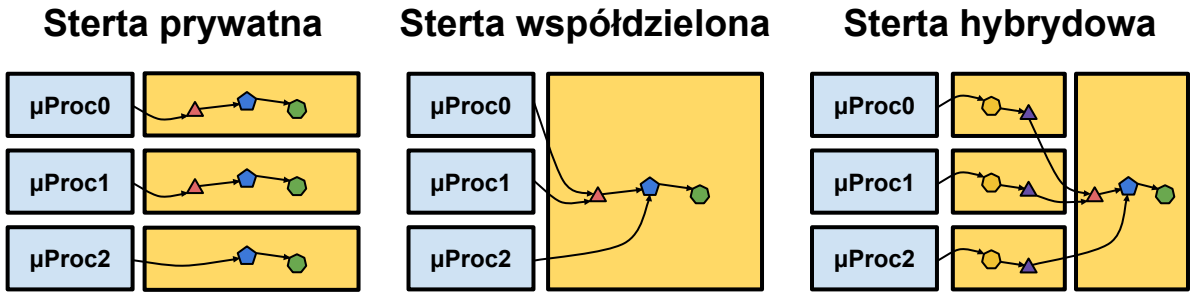
Ważnym aspektem architektury maszyny wirtualnej jest sposób w jaki wykorzystuje ona pamięć operacyjną i rozdziela ją pomiędzy procesy w niej działające, czyli architektura wykorzystania sterty (ang. *heap architecture*).

Rysunek 2.3 przedstawia trzy główne architektury wykorzystania sterty w środowisku wielo-procesowym, gdzie wiele autonomicznych procesów konkuruje o zasób jakim jest pamięć:

– architektura **sterty prywatnej**, charakteryzująca się zupełną separacją pamięci poszczególnych procesów, co prowadzi do konieczności kopiowania obiektów składających się na wiadomości przesyłane pomiędzy nimi;

– architektura **sterty współdzielonej**, charakteryzująca się współdzieleniem jednego obszaru pamięci pomiędzy wszystkie procesy, dzięki czemu wiadomości (a także ich części) mogą być współdzielone przez procesy bez konieczności ich kopiowania;

– architektura **hybrydowa**, mająca za zadanie połączenie zalet obu powyższych rozwiązań przez separację danych lokalnych procesów i współdzielenie danych wiadomości przesyłanych pomiędzy procesami; rozwiązanie to wymaga skomplikowanej, statycznej analizy kodu programów, która nie zawsze może być przeprowadzona.



Rysunek 2.3: Różne modele wykorzystania pamięci maszyn wirtualnych.

Szczegółowa analiza wydajności architektur przedstawionych na rysunku 2.3 w kontekście języka `Erlang`, do semantyki którego ThesisVM jest bardzo zbliżona, została zawarta w [10]. Na podstawie tej analizy zdecydowano się zastosować architekturę sterty współdzielonej, która minimalizuje problem kopiowania pamięci (*ergo*, spełnia nieformalny cel pracy sformułowany w sekcji 1.1) oraz nie wymaga skomplikowanej statycznej analizy kodu programów. Implementacja pozostawia jednak możliwość późniejszej modyfikacji architektury wykorzystania sterty.

Z problemem architektury wykorzystania sterty ściśle związany jest problem wyboru algorytmu alokacji pamięci. W [11] zawarto obszerne zestawienie algorytmów alokacji pamięci, na podstawie, którego zdecydowano się wykorzystać alokatory kaskadowe, *cache*'ujące pamięć zwolnionych obiektów w celu optymalizacji alokacji. Implementacja zastosowanego alokatora została zawarta w sekcji 4.

Ostatnim aspektem zarządzania pamięci maszyny wirtualnej jest kolekcja pamięci obiektow nieosiągalnych. Kolektory obiektów nieosiągalnych można podzielić na dwa typy, ze względu na dane, które analizują:

– kolektory **śledzące** (ang. *tracing-GC*), które okresowo trawersują zbiór obiektów bazowych (ang. *root-set*) celem oznaczenia wszystkich obiektów *osiągalnych* w danej chwili w systemie;

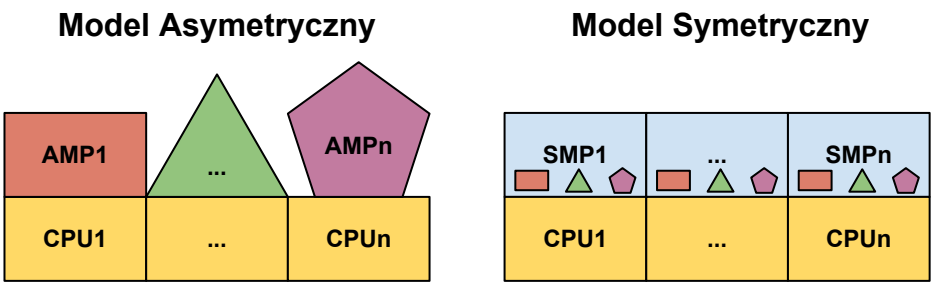
– kolektory **zliczające** (ang. *reference-counting-GC*), które na bieżąco zliczają ilość aktywnych referencji do każdego obiektu i natychmiastowo usuwają obiekty, których licznik referencji osiąga zero, co oznacza, że dany obiekt jest *nieosiągalny*.

Kolektory różnych typów mają bardzo różne charakterystyki wydajnościowe w zależności od architektury wykorzystania sterty zastosowanej w maszynie wirtualnej. Kolektory śledzące przeważnie generują długie pauzy w architekturach współdzielonych, natomiast kolektory zliczające prezentują stały narzut obliczeniowy związany z ciągłą modifykacją liczników referencji. Oczywiście istnieją dobrze poznane metody optymalizacji obu typu algorytmów [12, 13], które zacierają wszelkie różnice w ich charakterystykach wydajnościowych.

W implementacji ThesisVM zdecydowano się wykorzystać mechanizm automatycznej kolekcji "śmieci", oparty o *leniwe zliczanie referencji*, na podstawie wnikliwej analizy zawartej w [13] oraz w związku z wykorzystaniem podobnych algorytmów kolekcji danych binarnych w standardowej implementacji języka `Erlang`. Rozwiązanie to zostało szczegółowo opisane w sekcji 4, a implementacja umożliwia późniejsze jej rozszerzenie o dodatkowe optymalizacje.

2.5. Przetwarzanie współbieżne

Systemy współbieżne często realizują model symetrycznego multiprocessingu (*SMP*), którego cechą szczególną jest istnienie wielu identycznych jednostek operacyjnych wykonujących jednakowe operacje na różnych zbiorach danych (`SMPn` na rysunku 2.4).



Rysunek 2.4: Różne modele przetwarzania współbieżnego.

Alternatywnym rozwiązaniem jest model asymetrycznego multiprocessingu (*AMP*) (`AMPn` na rysunku 2.4), gdzie dla różnych typów zadań istnieją dedykowane, wyspecjalizowane jednostki operacyjne, takie jak wątki, lub procesy systemu operacyjnego.

Rozwiązania asymetryczne są interesujące ze względu na zupełnie nowe klasy algorytmów, których implementację umożliwiają (na przykład algorytm zarządzania pamięcią VCGC [14] wykorzystujący trzy asymetryczne wątki), jednak charakteryzują się skomplikowaniem interakcji poszczególnych jednostek operacyjnych a niejednokrotnie także słabą skalowalnością całego rozwiązania.

Model przetwarzania współbieżnego został już przybliżony przy okazji ogólnego opisu architektury ThesisVM na początku rozdziału. Wybrany został model SMP, który w kontekście maszyny wirtualnej ThesisVM polega na zrównolegleniu wielu interpreterów kodu bajtowego operujących na różnych kontekstach procesów (zbiorach rejestrów i danych znajdujących się na ich stosach) w celu osiągnięcia realnej współbieżności interpretowanego kodu.

Dodatkową zaletą modelu SMP jest jego kompatybilność z Modelem Aktorowym [1], którego głównym założeniem jest istnienie autonomicznych aktorów, którzy reagując na zmiany otoczenia dążą do swoich celów porozumiewając się z innymi aktorami za pośrednictwem wysyłania wiadomości [2]. W modelu SMP zastosowanym w maszynie wirtualnej ThesisVM aktorami są poszczególne procesy, które porozumiewają się za pomocą asynchronicznych wiadomości przesyłanych poprzez nieblokujące kolejki FIFO (ang. *First In Last Out*).

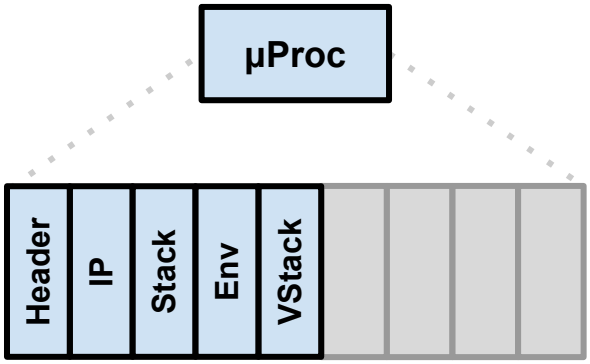
Szczegółowy opis implementacji symetrycznego multiprocesora i realizacja Modelu Aktorowego za jego pomocą zostały zawarte w sekcji 5.

3. Interpreter kodu bajtowego

– Opisać wybrany model Three Instruction Machine. [15, 6], [16]

– Opisać krótko działanie TIM, zwrócić uwagę na leniwość. [15, 6], [16]

– Opisać modyfikacje modelu TIM.



Rysunek 3.1: Schemat stanu maszyny wirtualnej.

– Opisać wykorzystywane rejestry.

– Opisać krótko alternatywne rozwiązania (SECD, TRSECD, SICP machine). [17, 18, 5, 9], [16]

3.1. Implementacja obiektów prostych

– Opisać implementację atomów ($\leq$ 8 bajtów).

– Opisać metodę tagowania atomów (dolne trzy bity) [19], [20]

– Opisąć optymalizacje/trejdofy wybranego sposobu tagowania. [19], [20]

3.2. Implementacja obiektów złożonych

– Opisać implementację obiektów złożonych ($\geq$ 8 bajtów - pary, funkcje/domknięcia, procesy).

– Opisać metodę tagowania (dolne dwa bajty + górne 48 bitów zarezerwowane dla GC). [19], [20]

– Opisać komponenty par.

– Opisać poszczególne komponenty obiektów funkcyjnych.

– Opisać reprezentację obiektów procesów (gołe rejestry).

– Opisać relację pomiędzy zbiorem rejestrów a reprezentacją procesu.

3.3. Implementacja wbudowanych operatorów

– Opisać wykorzystanie VStack.

– Opisać dostępne operacje prymitywne (LispKit). [5]

– Skonfrontować dostępne operacje prymitywne z `Core Erlang`. [8]

– Opisać optymalizacje operacji arytmetycznych. [19]

3.4. Ewaluacja argumentów i aplikacja funkcji

– Opisać działanie interpretera kodu bajtowego ThesisVM. [15, 6]

– Opisać leniwą ewaluację argumentów.

– Opisać aplikację funkcji.

– Opisać aplikację operacji prymitywnych.

3.5. Reprezentacja kodu bajtowego ThesisVM

– Opisać reprezentację kodu bajtowego (listy opkodów).

– Opisać optymalizacje TVMBC (wykorzystanie górnych dwóch bajtów słowa, 0 = pushc, threading, itd).

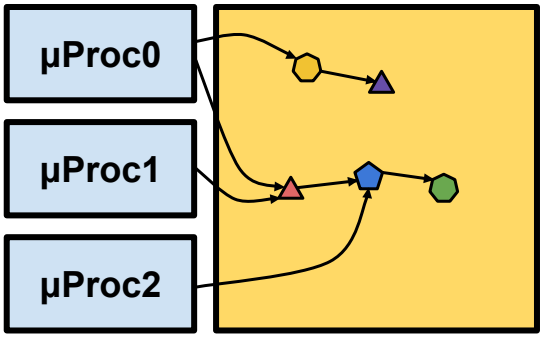
– Opisać dostępne opkody kodu bajtowego. [15, 6]

3.6. Generacja kodu bajtowego ThesisVM

– Opisać szczegółowo generację kodu bajtowego. [6]

4. Model zarządzania pamięcią

– Opisać krótko architekturę wspólnej sterty. [10]



Rysunek 4.1: Model wspólnej pamięci ThesisVM.

– Opisać strategie zarządzania pamięcią (alokator i GC). [13]

4.1. Architektura wspólnej sterty

– Opisać szczegółowo wybraną architekturę.

– Wspomnieć o problemach wybranej architektury (duży root-set, długie kolekcje). [10]

– Skonfrontować publiczną stertę z architekturą prywatnej sterty. [10]

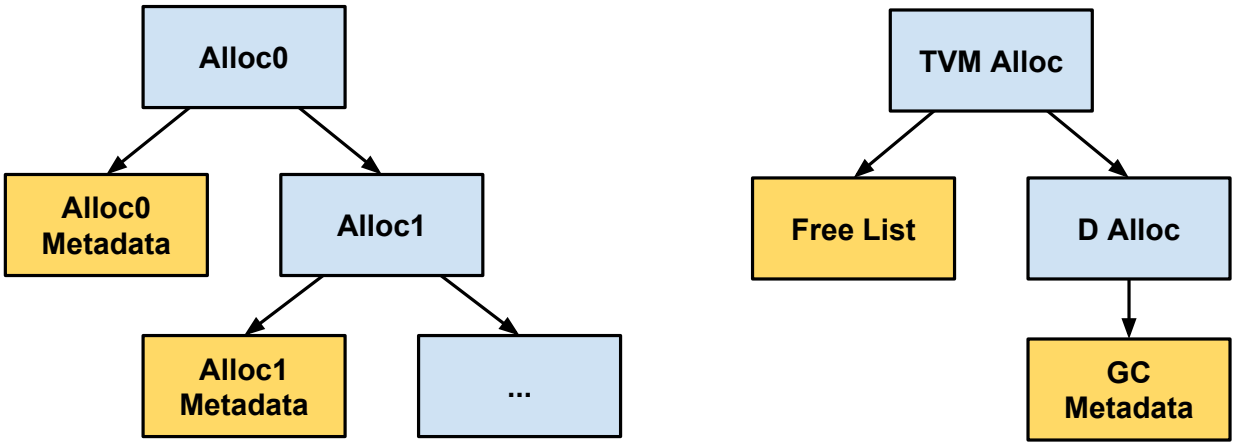
– Wspomnieć o problemach prywatnej sterty (powolne przekazywanie wiadomości przez kopiowanie). [10]

– Wspomnieć o istnieniu rozwiązań hybrydowych. [10]

– Wspomnięć o problemach rozwiązań hybrydowych (usunięte z `Erlang/OTP` R15B02).

4.2. Implementacja alokatora obiektów

– Opisać działanie kaskadowego alokatora. [11]



Rysunek 4.2: Schemat kaskadowych alokatorów wykorzystanych w ThesisVM.

– Opisać implementację wykorzystanego alokatora.

– Opisać optymalizacje alokatora (wykorzystanie free listy).

– Opisać zmiany wprowadzone w stanie maszyny wirtualnej (dodatkowe rejestry).

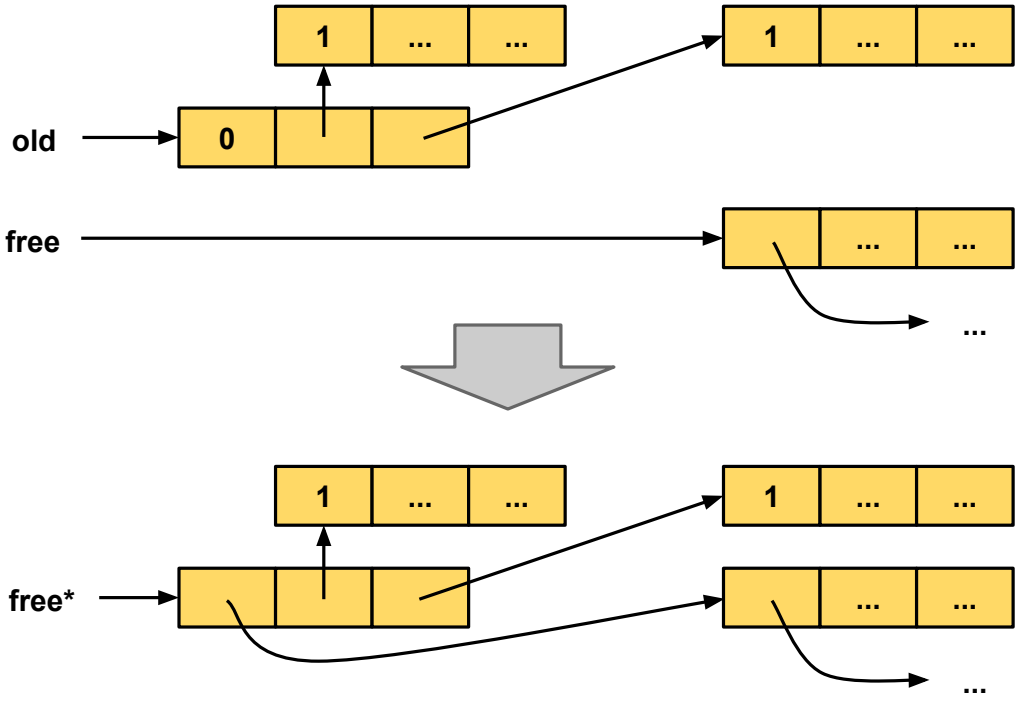
– Opisać krótko alternatywne rozwiązania (mallocator, etc). [11]

4.3. Kolekcja nieosiągalnych obiektów

– Opisać leniwe zliczanie referencji. [21]

– Opisać implementację algorytmu leniwego zliczania referencji. [13]

– Opisać konieczność wykorzystania operacji atomowych i barier pamięci (liczniki referencji).



Rysunek 4.3: Schemat działania zwalniania pamięci obiektów.

– Opisać zmiany wprowadzone w stanie maszyny wirtualnej (dodatkowe rejestry).

– Opisać narzut pamięci związany z licznikiem referencji i leniwością algorytmu. [21, 13]

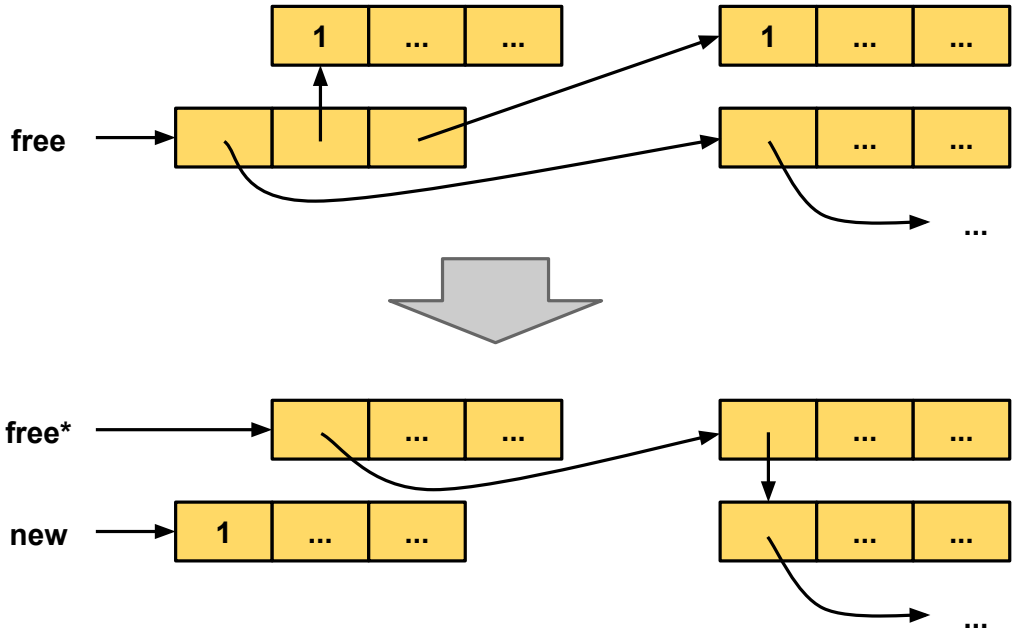
– Opisać krótko wady, możliwe usprawnienia i alternatywne rozwiązania (zaproponowane przez Joe'go oraz VCGC) [22, 14]

4.4. Kolekcja obiektów cyklicznych

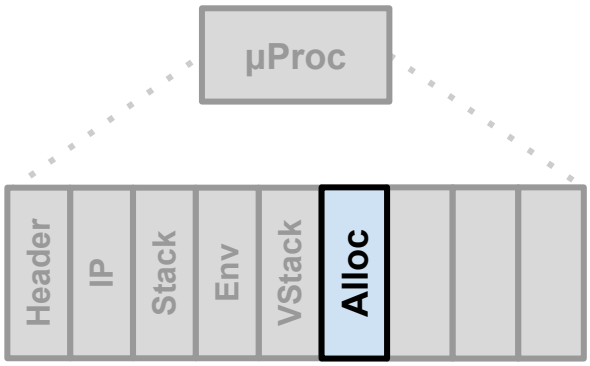
– Opisać, że obiekty cykliczne nie występują.

– Wspomnieć o możliwości zaimplementowania zapasowego stop-the-world GC.

– Wspomnieć o możliwości cyklicznego uruchamiania D'owego GC.



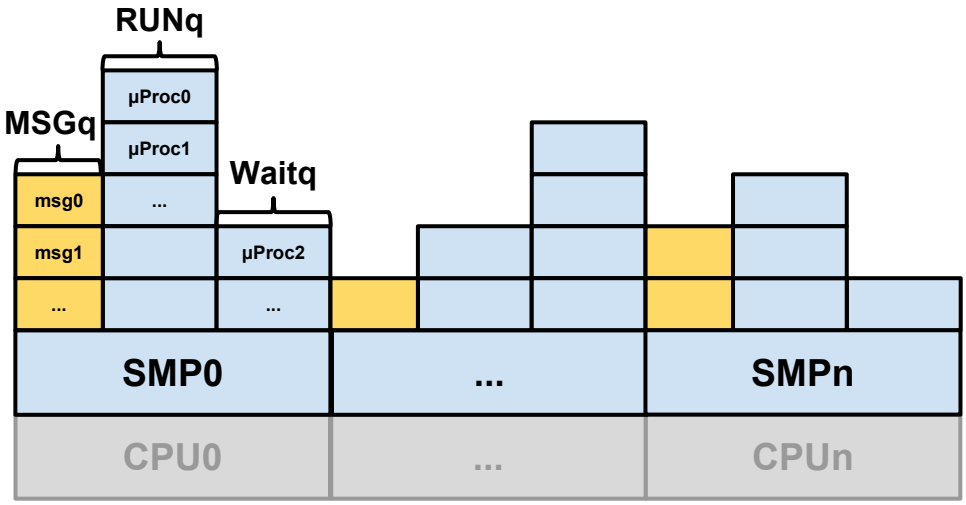
Rysunek 4.4: Schemat działania alokacji pamięci nowych obiektów.



Rysunek 4.5: Schemat rejestrów wymaganych przez implementację kolektora obiektów nieosiągalnych.

5. Model przetwarzania współbieżnego

– Opisać bardziej szczegółowo Model Aktorowy i asynchroniczne przekazywanie wiadomości. [1, 2]



Rysunek 5.1: Schemat symetrycznego multiprocesora ThesisVM.

– Opisać bardziej szczegółowo działanie SMP - wiadomości kontrolne oraz RQue.

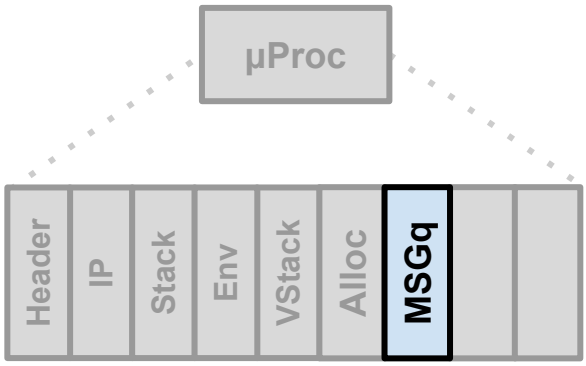
5.1. Implementacja Modelu Aktorowego

– Opisać powstawanie procesów i prymityw `spawn`.

– Opisać logiczną autonomiczność procesów (brak mutacji = inne procesy nie mogą ingerować).

– Opisać sposób porozumiewania się procesów (kolejki nieblokujące). [23, 24]

– Opisać implementację kolejek nieblokujących (+ weryfikacja poprawności). [23, 25]- Opisać wykorzystanie CAS i problem ABA.



Rysunek 5.2: Schemat rejestrów wymaganych przez implementację Modelu Aktorowego.

– Opisać zmiany wprowadzone w stanie maszyny wirtualnej (dodatkowe rejestry).

– Opisać krótko wady i możliwe usprawnienia zastosowanego rozwiązania (dynamic size, wait-free, optimistic FIFO). [24, 26, 27]

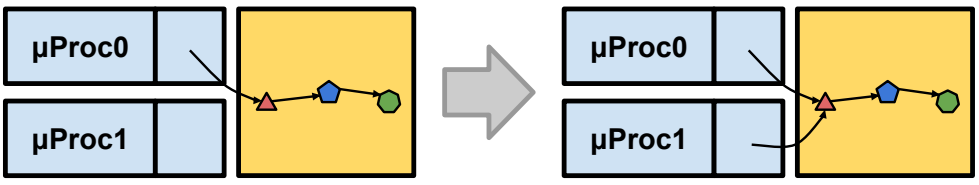
– Opisać krótko alternatywne podejścia (synchroniczne przekazywanie wiadomości - kanały, locki/mutexy/semafory).

5.2. Implementacja przesyłania wiadomości

– Opisać implementację prymitywów `send` oraz `receive`.

– Zwrócić uwagę na konieczność wykorzystania operacji atomowych oraz barier pamięci.

– Snippet kodu przesyłającego wiadomość.



Rysunek 5.3: Schemat działania przesyłania wiadomości.

– Opisać co dzieje się podczas wysyłania wiadomości.

– Opisać sposób pobierania wiadomości z kolejki.

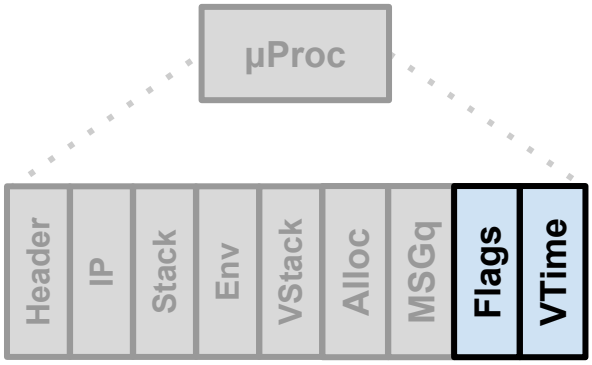
– Zwrócić uwagę na fakt, że problem kopiowania został zniwelowany kosztem lekkich barier pamięci.

5.3. Harmonogramowanie procesów

– Opisać sposób harmonogramowania procesów (brak load-balancingu, losowy spawn). [28]

– Opisać implementację prymitywu `sleep` oraz sleep-table.

– Opisać wiadomości kontrolne.



Rysunek 5.4: Schemat rejestrów wymaganych przez usprawnienia hanmonogramowania SMP.

– Opisać możliwe usprawnienia (load-balancing i dzielenie zużycia).

6. Podsumowanie

– Opisać co udało się zrobić.

– Opisać czego nie udało się zrobić (+ możliwe usprawnienia).

6.1. Leniwe zliczanie referencji

– Przeanalizować szybkość, pauzy, zużycie pamięci.

6.2. Przesyłanie wiadomości

– Przeanalizować szybkość przesyłania wiadomości/konieczność czekania procesów, wielkość kolejek wiadomości.

6.3. Kierunki przyszłego rozwoju

– Opisać plany na przyszły rozwój projektu (priorytet procesów, load balancing SMP, wsparcie dla `Core Erlang`, bytecode threading, przebiegi optymalizacyjne podczas kompilacji, umożliwienie dystrybucji na wiele maszyn, zapasowy kolektor śmieci cyklicznych, opcja wykorzystania sterty prywatnej i autonomicznego alokatora, natywna kompilacja JIT, wektory, data-level parallelism, optymalizacja wykorzystania stosu, hardłerowa implementacja interpretera kodu bajtowego).

Bibliografia

[1] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[2] W. D. Clinger, "Foundations of actor semantics," tech. rep., Cambridge, MA, USA, 1981.

[3] A. Alexandrescu, *The D Programming Language*. Pearson Education, 2010.

[4] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG (2Nd Ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[5] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MA, USA: MIT Press, 2nd ed., 1996.

[6] S. P. Jones and D. Lester, *Implementing functional languages: a tutorial*. Prentice Hall, 1992. Free online version.

[7] R. Carlsson, "An introduction to Core Erlang," in *In Proceedings of the PLI'01 Erlang Workshop*, 2001.

[8] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding, "Core Erlang 1.0.3 language specification," tech. rep., Department of Information Technology, Uppsala University, Nov. 2004.

[9] G. L. Steele Jr and G. J. Sussman, "The art of the interpreter of the modularity complex (parts zero, one, and two)," 1978.

[10] J. Wilhelmsson, *Efficient Memory Management for Message-Passing Concurrency — part I: Single-threaded execution*. Licentiate thesis, Department of Information Technology, Uppsala University, May 2005.

[11] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," 1995.

[12] R. Shahriyar, S. M. Blackburn, and D. Frampton, "Down for the count? getting reference counting back in the ring," in *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, (New York, NY, USA), pp. 73–84, ACM, 2012.

[13] D. F. Bacon, P. Cheng, and V. T. Rajan, "A unified theory of garbage collection," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, (New York, NY, USA), pp. 50–68, ACM, 2004.

[14] L. Huelsbergen and P. Winterbottom, "Very concurrent mark-&-sweep garbage collection without fine-grain synchronization," in *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, (New York, NY, USA), pp. 166–175, ACM, 1998.

[15] J. Fairbairn and S. Wray, "TIM: A simple, lazy abstract machine to execute supercombinators," in *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, (London, UK, UK), pp. 34–45, Springer-Verlag, 1987.

[16] O. Kaser, S. Pawagi, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar, "Fast parallel implementation of lazy languages - the equals experience," in *Journal of Functional Programming*, pp. 335–344, ACM, 1992.

[17] D. Van Horn and M. Might, "Abstracting abstract machines," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, (New York, NY, USA), pp. 51–62, ACM, 2010.

[18] J. D. Ramsdell, "The Tail-Recursive SECD Machine," *Journal of Automated Reasoning*, vol. 23, no. 1, pp. 43–62, 1999.

[19] D. Gudeman, "Representing type information in dynamically typed languages," 1993.

[20] W. R. Cook, "Anatomy of programming languages." Free online version.

[21] H.-J. Boehm, "The space cost of lazy reference counting," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, (New York, NY, USA), pp. 210–219, ACM, 2004.

[22] J. Armstrong and R. Virding, "One pass real-time generational mark-sweep garbage collection," in *IN INTERNATIONAL WORKSHOP ON MEMORY MANAGEMENT*, pp. 313–322, Springer-Verlag, 1995.

[23] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, (New York, NY, USA), pp. 267–275, ACM, 1996.

[24] M. Herlihy, V. Luchangco, P. Martin, and M. Moir, "Dynamic-sized lock-free data structures," tech. rep., 2002.

[25] L. Groves, "Verifying Michael and Scott's lock-free queue algorithm using trace reduction," in *Proceedings of the Fourteenth Symposium on Computing: The Australasian Theory - Volume 77*, CATS '08, (Darlinghurst, Australia, Australia), pp. 133–142, Australian Computer Society, Inc., 2008.

[26] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), pp. 223–234, ACM, 2011.

[27] E. Ladan-Mozes and N. Shavit, "An optimistic approach to lock-free FIFO queues," 2004.

[28] C. S. Pabla, "Completely fair scheduler," *Linux J.*, vol. 2009, Aug. 2009.

A. Przykładowe programy

- Opisać sposób uruchamiania maszyny wirtualnej.
- Hello world.
- Factorial.
- Fibonacci.
- Concurrent Hello world.
- Map-reduce.

B. Spisy wbudowanych funkcji i operatorów

Spis funkcji wbudowanych

– Wylistować funkcje wbudowane.

Spis operatorów wbudowanych

– Wylistować operacje prymitywne.

C. Spisy rysunków, fragmentów kodu i tablic

Spis rysunków

Spis listingów

Spis tablic