ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 2017
Seria: ORGANIZACJA I ZARZĄDZANIE z. 113 Nr kol. 1992

Hubert ŁOPUSIŃSKI, Mateusz ŁOPUSIŃSKI, Ireneusz J. JÓŹWIAK
Politechnika Wrocławska
Wydział Informatyki i Zarządzania
hubert.lopusinski@gmail.com, mateusz.lopusinski2@gmail.com,
ireneusz.jozwiak@pwr.edu.pl

STRATEGIA TWORZENIA APLIKACJI Z WYKORZYSTANIEM PAMIĘCI NIEULOTNYCH

Streszczenie. W artykule omówiono sposób, w jaki należy tworzyć oprogramowanie z wykorzystaniem pamięci nieulotnych. Opisano zagadnienia użycia transakcji, alokacji oraz dealokacji obszarów pamięci, zapewnienia trwałości obiektów oraz danych używanych przez aplikacje. Przedstawiono podstawowe techniki oraz algorytmy użycia biblioteki Non-Volatile Memory.

Słowa kluczowe: pamięć trwała, pamięć nieulotna, programowanie, transakcje, alokacja, non-volatile memory library

STRATEGY OF PROGRAMMING APPLICATIONS USING NON-VOLATILE MEMORIES AND NON-VOLATILE MEMORY LIBRARY

Abstract. The paper presents how to create software using non-volatile memory. Methodology of use of transactions, memory allocation and deallocation, assurance persistence of objects and data used by applications was described. Basic techniques and algorithms for using Non-Volatile Memory were presented.

Keywords: persistent memory, non-volatile memory, programming, transactions, allocation, deallocation, non-volatile memory library

1. Wprowadzenie

Przez wiele lat aplikacje komputerowe dzieliły swoje dane na dwie warstwy: memory i storage, czyli pamięć ulotna i nieultona. W tradycyjnych architekturach pamięć ulotna jest szybko i bezpośrednio adresowalna przez procesor, natomiast nieulotna jest wolna i wymaga

wyspecjalizowanych interfejsów do jej dostępu [5]. Nowe technologie pojawiające się na rynku stwarzają trzecią warstwę: pamieć trwałą, która posiada dostęp jakby była pamięcią ulotną, używa instrukcji load i store procesora, ale z drugiej strony zachowuje swoją zawartość w przypadku utraty zasilania tak jak pamięć trwała. [1]

Tworząc aplikacje z wykorzystaniem pamięci trwałych chcemy mieć bezpośredni dostęp do zapisu oraz odczytu, tak jak ma to miejsce w przypadku użycia tradycyjnych pamięci ulotnych. [5] System plików pamięci trwałej widoczny jest dla systemu operacyjnego jako plik z bezpośrednim dostępem. Pozostanie przy takim sposobie prezentacji pamięci utwierdzalibyśmy aplikacje, że mają bezpośredni dostęp do całej pamięci dynamicznej, której potrzebują, ponieważ system operacyjny udostępnia systemowe wywołanie sbrk(2) czy mmap(2) [4]. Tak jak libc i większość języków programowania udostępniają środowisko wykonawcze, dostarczając przy tym API(Application Interface), takie jak malloc(3) i free(3) [4], które opierają się na surowych interfejsach takich jak sbrk(). Mapowanie w pamięci pliku pamięci daje możliwości dostępu bezpośredniego, jednak chcielibyśmy zapisać go w strukturach danych oraz aktualizować w sposób, który umożliwi spójność w obrębie przerwań działania systemu(restart lub utrata zasilania) [1]. Użycie standardowych interfejsów malloc() oraz free() nie zapewnia trwałości pamięci [4]. Jeśli program będzie chciał zaalokować obszar w pamięci z użyciem malloc() i zostanie przerwany zanim operacja zostanie dokonana, otrzymamy wyciek pamięci trwałej, której dany obszar później nie będzie dostępny. [1] W przypadku pamięci ulotnej nie jest to problem, gdyż jest ona czyszczona przy każdym uruchomieniu programu.

Powyższe problemy rozwiązuje biblioteka NVM [2]. Jest ona dostępna w warstwie użytkownika i może być wykorzystywana w programowaniu z użyciem każdej pamięci nieulotnej, nie tylko pamięci trwałej.

2. Założenia używania biblioteki NVM

Biblioteka udostępnia użytkownikowi funkcjonalne API [2]. Posiada zestaw funkcji, których umiejętne użycie zapewnia aplikacji trwałość danych w przypadku utraty zasilania, natomiast z drugiej strony posiada bezpośredni dostęp do pamięci oraz instrukcji load i store CPU(Central Processing Unit) [5]. Jedną z bibliotek udostępnionych przez autorów NVML(Non-Volatile Memory Library) jest libpmemobj C++, która oparta jest na libpmemobj C API [2]. Wszystkie klasy oraz funkcje znajdują się w namespace nvml::obj. Za pomocą tej biblioteki jesteśmy w stanie zmodyfikować aplikacje ulotne, skupiając się na strukturach danych a nie na kodzie programu. Główny cel zmian w aplikacji polega na przebudowaniu struktur oraz klas z ich niewielkimi modyfikacjami. Podczas modernizacji naszego kodu możemy napotkać wiele problemów [1] spowodowanych tym, że:

- referencje jako część perzystentych struktur lub klas nie są wspierane przez funkcje biblioteki,
- polimorficzne perzystentne klasy są zabronione,
- autorzy biblioteki nie zalecają używania statycznych zmiennych składowych.

Powyższe problemy wymuszają często na nas modyfikacje nie tylko w niektórych klasach lub strukturach naszej aplikacji, lecz całej jej architekturze.

3. Strategia używania „pooli”

Pamięć trwała jest przedstawiana przez system operacyjny jako plik zmapowany w pamięci. Takie pliki nazywane są pool’ami. Ta abstrakcja znajduje również odzwierciedlenie w szablonach klas biblioteki libpmemobj C++. Jest to szablon klasy, którego jedynym parametrem jest typ root’a. Klasa pool [2] umożliwia nam łatwe zarządzanie plikami zmapowanymi w pamięci z poziomu użytkownika. Nie musimy własnoręcznie używać funkcji mmap() [4].

Mamy do dyspozycji trzy podstawowe operacje, które możemy dokonywać za pomocą klasy pool[1],[2]:

- open – operacja ta otwiera istniejącą poolę,
- create – operacja ta tworzy nową poolę,
- close – operacja ta zamyka aktualnie otwartą lub utworzoną poolę.

Klasa pool posiada również zestaw funkcji do niskopoziomowego zarządzania pamięcią trwałą takich jak: flush, persist, drain, memcpy_persist oraz memset_persist [1],[2].

Strategia użycia klasy pool:

1) sprawdź za pomocą metody check(), czy dana poola nadaje się do użycia,
2) jeśli dana poola nadaje się do użycia, możesz ją otworzyć używając metody open(). W przeciwnym przypadku opracuj nową poolę używając metody create(),
3) jeśli posiadasz otwartą lub opracowaną poolę gotową do użycia, pobierz za pomocą metody get_root() root danej pooli,
4) po dokonaniu operacji na pooli, zamknij poolę używając metody close().

W powyższym algorytmie jest jedna rzecz, o której trzeba wspomnieć. Konstruktor obiektu root nie zostanie wywołany podczas użycia metody get_root() [1]. Jest to decyzja podjęta przez autorów spowodowana tym, że obiekt root może być realokowany. Takie zachowanie może powodować nierozwiązywalne problemy i posiada duże prawdopodobieństwo wystąpienia wycieków w pamięci nieulotnej. Oznacza to, że odpowiedzialność prawidłowego inicjowania obiektu root pozostawiona jest użytkownikowi.

Jeśli aplikacja tworzona jest wykorzystując programowanie zorientowane obiektowo oraz posiada tylko jedną poolę, to rozsądnym rozwiązaniem jest zastosowanie wzorca

obiektowego Singleton, aby zapewnić jej unikalność oraz łatwe zarządzanie poolą(m.in. tworzenie lub otwieranie pooli, zamykanie pooli, prawidłowe inicjowanie obiektu root).

4. Transakcje

Samym sercem biblioteki libpmemobj są transakcje [1][2]. Jeśli chcemy, aby nasze zmienne posiadały cechy trwałości, to zmiany na nich muszą być dokonywane przy użyciu transakcji. Transakcje są [1]:

1) niepodzielne – transakcja wykonywana jest całkowicie albo nie jest wykonywana wcale, tzn. w przypdaku jej niepowodzenia przywracany jest stan zmiennych sprzed rozpoczęcia transakcji,
2) trwałe – po pomyślnym zakończeniu transakcji zmiany dokonane wewnątrz niej zachowane są na stałe,
3) spójne – jeśli w danej chwili transakcja wprowadza zmiany na danych, to dla każdej innej transakcji te dane będą widoczne przed lub po zmianie(nie nastąpi sytuacja, w której część danych jest przed zmianą a część po zmianie)
4) izolowane – dwie różne transakcje nie mogą wykonywać jednocześnie zmian na tych samych danych.

Transakcje jako argument przyjmują poole oraz niektóre z nich blokady, służące do programowania wielowątkowego.

Jednym z typów transakcji są transakcje closure-like [1],[2]. Jest to typ transakcji rekomendowany przez autorów biblioteki NVM, ponieważ automatycznie przechwytuje wszystkie zdarzenia commit/abort tak, że użytkownik nie musi robić tego manualnie. Najlepszą postacią tej transakcji jest funkcja lambda, lecz użytkownik nie musi jej używać, jeśli nie chce, gdyż akceptuje ona obiekt std::function<void>() [3]. Typ transakcji closure-like może również przyjmować jako argument rezydentne blokady pamięci trwałej, które dostępne są w bibliotece libpmemobj C++ [2]. W przypadku, gdy wewnątrz transakcji program wyrzuci wyjątek, transakcja jest przerywana i oryginalny wyjątek jest wyrzucony ponownie [1]. W ten sposób wyjątki naszego programu nie zostaną utracone, a wyjątki tworzone przez błędy transakcji (np. nvml::transaction_error) są poprawnie przechwytywane przez bibliotekę.

5. Sposoby alokacji i dealokacji

Jedną z najważniejszych cech biblioteki libpmemobj C++ jest szablon inteligentnego wskaźnika persistent_ptr [1],[2]. Tak jak w przypadku standardowych odwpowiedników języka C++ [3] potrzebny jest mechanizm alokacji z odpowiednim konstruowaniem obiektu. Każdy kto miał do czynienia z inteligentymi wskaźnikami C++(np. std::shared_ptr) [3] zna ideę ich używania. Persistent_ptr [1],[2] działa w ten sam sposób oraz udostępnia użytkownikowi operatory *, ->, []. Prawdopodobnie najczęstszym zastosowaniem funkcji alokacji jest transakcja. Najłatwiejszym sposobem alokowania obiektu i przypisania go do inteligentnego wskaźnika jest użycie funkcji transaction::exec_tx na otwartej aktualnie pooli oraz użycia metody make_persistent() [1],[2] z danym konstruktorem klasy alokowanego obiektu. Algorytm wygląda następująco:

1) opracuj lub otwórz poole,
2) utwórz transakcję transaction::exec_tx na danej pooli ,
3) przypisz do inteligentnego wskaźnika persistent_ptr zaolokowany obiekt przy użyciu funkcji make_persistent z użyciem konstruktora klasy alokowanego obiektu.

Uważa się, że w powyższym algorytmie najrozsądniej jest użyć konstruktora domyślnego klasy obiektu. Spowodowane jest to tym, że chcąc w transakcji alokować tablicę obiektów, nie możemy używać konstruktorów parametryzowanych. Obiekty w tablicy muszą być alokowane przy użycia konstruktora domyślnego [3].

Jeśli w aplikacji dany obiekt nie jest już potrzebny lub po prostu chcemy go usunąć, należy go dealokować przy użyciu funkcji delete_persistent [1][2]. W przypadku destrukcji obiektów transakcyjnych biblioteka wywołuje destruktor obiektu. Nie jest to jednak przypadek alokacji atomowych, gdzie nie ma możliwości atomowego niszczenia i dealokacji obiektu. Przydziały transakcyjne są najbardziej dogodnym sposobem na stworzenie trwałych obiektów, zwłaszcza jeśli alokacja jest jedną z sekwencji operacji, która musi być dokonana atomowo pod względem trwałości.

Istnieje jednak inny sposób tworzenia obiektów. Jeśli chcemy, aby obiekt był tylko zaalokowany atomowo to nie musimy rozpoczynać transakcji, aby tego dokonać. W tym celu możemy używać szablonu funkcji make_persistent_atomic [1],[2]. Atomowe usuwanie tak zaalokowanych obiektów odbywa się przy użyciu szablonu funkcji delete_persistent_atomic.
Jednak są problemy związane z alokacjami i dealokacjami, o których trzeba pamiętać:

1) nie należy nigdy łączyć alokacji atomowych oraz transakcji – może doprowadzić to do wycieku pamięci, niezdefiniowanego zachowania się obiektu lub błędu segmentacji pamięci,
2) w przypadku dealokacji atomowych pamięć jest zwalniana, lecz destruktor obiektu nie jest wywoływań,

3) wersja transakcyjna może być używana tylko i wyłącznie wewnątrz transakcji, używanie poza transakcją spowoduje wyrzucenie wyjątku przez program.

Powyższy sposób dotyczy alokacji dynamicznej i służy do alokowania klas obiektów. Dla typów prostych alokowanych statycznie biblioteka posiada zmienne rezydentne p<> [1],[2]. Powinniśmy je stosować, jeśli chcemy, aby nasze zmienne typów prostych były trwałe. Zmienne te mają nadpisany operator= , więc zapewnienie nieulotności w czasie trwania aplikacji jest bardzo proste.

Należy pamiętać, że każdą zmianę wartości zmiennych alokowanych dynamicznie oraz statycznie należy dokonywać w operacji transakcji.

6. Podsumowanie

Przy tworzeniu oprogramowania z wykorzystaniem pamięci nieulotnych istotna jest podstawowa znajomość tradycyjnej architektury komputera oraz technik programowania z użyciem biblioteki Non-Volatile Memory. Projektowanie kodu nowej aplikacji oraz modyfikowanie istniejącej może odbywać w oparciu o sporządzone algorytmy. Zaproponowana strategia programowania z wykorzystaniem pamięci nieulotnych oraz biblioteki Non-Volatile Memory zapewni trwałość oraz nieulotność danych w przypadku restartu systemu lub nieoczekiwanej utraty zasilania.

Bibliografia

1. Bjarne S.: Język C++. Kompendium wiedzy, Helion, Wydanie IV, Gliwice 2013.
2. Love R.: Linux. Programowanie Systemowe, Helion, Wydanie II, Gliwice 2013
3. Null L., Lobur J.: Struktura organizacyjna i architektura systemów komputerowych, Helion 2004.
4. Intel Corporation Open-Sourced License, http://pmem.io
5. Intel Corporation Open-Sourced License, https://github.com/pmem/nvml/